icebreakeR

Andrew Robinson
Department of Mathematics and Statistics
University of Melbourne
Parkville, Vic. 3010
A.Robinson@ms.unimelb.edu.au

February 17, 2013

Contents

List of Figures

Chapter 1

Introduction

1.1 R

R is a computer language for data analysis and modeling. R can be used as an object-oriented programming language, or as a statistical environment within which lists of instructions can be performed sequentially without user input.

We can interact with R by typing or pasting commands into a command line in the console, or by writing them in a text editor[1] and submitting them to the console, using the `source()` command. The command–line version of R provides us with two prompts, that is, two signals that it is ready for input. They are: the caret

```
>
```

which means that R is ready for a new command, and the plus sign:

```
+
```

which means that R is waiting for you to finish the current command.

The exclusive use of a command-line interface (CLI) does make our learning curve steeper. However, it also allows us to store collections of commands and run them without intervention. This simplifies many operations and tasks, for example, communicating an analysis to another person, or making templates for often-used graphics or reports. A principal advantage of the CLI is that it simplifies the development and use of scripts. These allow us to keep a permanent, documented text record of the steps that we have done.

Some time later, when you look back to your notes to figure out again how you did an analysis in this workshop, you will be glad that you wrote scripts to perform them, instead of pointing and clicking in menus. If for no other reason, you will be protected against change: when software changes, menus change. The engines that read scripts change very slowly, if at all.

Some interaction with R can be done through menus for some operating systems, but this interaction is mainly administrative. Users have written GUIs for R[2], but we do not consider them here.

You can find and freely download the executables and source code at: http://www.r-project.org.

1.2 Hello World

The `Hello world` function is a touchstone of computer language exposition. It simply prints "Hello World." In R:

```
> hi.there <- function() {
+    cat("Hello World!\n")
+ }
```

[1] I use emacs; other options are RStudio, WinEDT, vi, or R's own text editor. I recommend that you do not use a word processor! The specific problem with using a word-processor to *write* scripts is that they make all sorts of assumptions about grammar and spelling that are not true for scripts. Some very clever word processors will quietly convert your ordinary boring quotes into so-called "smart-quotes", which are much better, unless the interpreter that you're writing for really does prefer ordinary ones.

[2] More can be learned about these undertakings at http://www.r-project.org/GUI

This input has a few points worth discussing. Firstly, if the command is executed correctly, then all you will see is another prompt. *You won't see any other feedback.* This can be confusing to the novice!

Secondly, `function()` creates a new object in the random-access computer memory. The object is called `hi.there`, and it contains the result of creating the function that we have defined.

- `hi.there` is the name of the new object; hereafter, if we wish to manipulate this object we can do so by the name "hi.there",
- `<-` is the assignment operator. This is how we tell R to name (or rename) the object, and
- `function()` tells R to create a function that will run the commands in the curly braces that follow.

```
> hi.there()

Hello World!
```

Our inclusion of `Hello World` in the icebreakeR is at least partially ironical, as the program bears very little resemblence to the real functionality of R[3]. Maybe the following is more appropriate.

```
> 1 + 2

[1] 3
```

1.3 Why R?

1. R runs on Windows, Mac-OS, and Unix variants (eg the BSD and Linux flavors)
2. R provides a vast number of useful statistical tools, many of which have been painstakingly tested.
3. R produces publication-quality graphics in a variety of formats, including JPEG, postscript, eps, pdf, and bmp, from a flexible and easily enhanced interface.
4. R plays well with LaTeX via the `Sweave` package.
5. R plays well with FORTRAN, C, and shell scripts.
6. R scales, making it useful for small and large projects.
7. R eschews the GUI.

Anecdote: I was telecommuting from New Haven (CT) to Moscow (ID). I developed and trialled simulation code on my laptop, *ssh*-ed into a FreeBSD server in Moscow and ran the full code inside a screen[4] session. Any time I wanted to monitor progress I could log in to the session remotely, from anywhere. When it had concluded, R sent me an email to let me know.

1.4 Why not R?

1. R cannot do everything.
2. R will not hold your hand.
3. The documentation can be opaque.
4. R can drive you crazy, or age you prematurely.
5. The contributed packages have been exposed to varying degrees of testing and analysis. Some are probably unreliable.
6. There is no guarantee that it is worth more than you paid for it.
7. R eschews the GUI.

[3] This is also true for most other languages. So, I'm being precious.
[4] *Screen* is a very useful open-source terminal multiplexor for Unix–like systems. Try it today!

Anecdote: I was developing a relatively large-scale data analysis that required numerous steps. One of them was to run a third-party forest growth model (ORGANON) 180 times, each with a different configuration, then scoop up the results and assimilate them. I had written code for R to do all this: produce 180 configuration files, run a pre-processor 180 times, save the results, run the model 180 times, import the simulations, and manipulate them as dataframes. As this had to run numerous times, and required identifiable temporary files, I decided to include intermediate cleaning-up steps, which involved deleting certain files. If you run such code as a script, and a command fails, then the script stops. This is a feature. If you run such code by copying it from your document and then pasting it as as group of commands to the console, and a command fails, the interpreter goes to the next command anyway. The failed command was a change of directory. My code obediently deleted itself, and all its companion scripts. This was a classic case of the power of one's tools being exceeded by one's incompetence. R is very powerful.

1.5 The Open Source Ideal

R is free, as in: you can download the executables and the source code at no cost. However, this is not the most important kind of freedom at stake in computer programming. There is also freedom as in lack of constraints.

The phrase most commonly used to describe this particular principle of freedom is: think of free speech, not free beer. You can make any alterations with only one obligation: any further distribution must be under the identical license and must include the source code. But, you can still be charge or be charged a reasonable cost for its dissemination.

There are numerous flavors of "Open Source", which are commonly bundled together, mistakenly. The label refers to the license under which the code is released, or not, to the public domain. There are at least three distinct licensing approaches to open source materials: GNU, BSD, and OSI.

R is released under the GNU license, which, loosely speaking, says that the source code is freely available, and that you can modify it as you like, but if you *distribute* any modifications then they must be accompanied by the modified source code and also distributed under the same license. A nominal fee may be charged. This license means that R will always be available, and will always be open source.

It also means that if you make changes but do not distribute them, then you are under no obligation to share the source code with anyone.

1.6 Using this Document

This document is in `pdf` format. The Acrobat Reader gives you the ability to copy any text to the clipboard, by choosing the Select Text Tool, marking the text you want, and pressing `Ctrl-c`. You can then paste it to the R console (in MS Windows) by right-clicking in the console and selecting the "Paste commands only" option. This will save you a lot of tedious typing.

The R commands are printed in a slanting typewriter font. It can be a little misleading, for example when the vertical bar | is used and appears to be a slash. Copying and pasting is safer than typing for yourself. Also, commands that span more than one line are connected by means of a `+` sign. Do not type this in if you are transcribing the code; it will create errors. The "Paste commands only" option takes care of this detail; it also cleverly ignores any non-commands, for example, this paragraph.

Also, any text in this document can be copied and pasted into text documents.

If you do choose to type the commands in to the console, please note that I have been rather lazy about an important convention: always conclude your input using the `Return` key.

The datasets quoted in this document can be found at the following link:
`http://www.ms.unimelb.edu.au/~andrewpr/r-users/`

Sometimes during this workshop you will come across labels or phrases that are unfamiliar to you. Please use your local resources to find out more about them[5]. You only need to learn enough to satisfy yourself that your undestanding is suitable for the context. If I want you to know more, then I will tell you so at the time.

[5] *e.g.* `http://www.google.com`

1.7 Being Organized

We will now set up the directory structure on our hard drive that we will use for the workshop. Create a single directory, called, for example `icebreakeR`. This will be the directory that contains all the workshop materials. Within this directory, create the following directories:

- `data`
- `graphics`
- `images`
- `library`
- `notes`
- `output`
- `scripts`
- `src`
- `sweave`

We won't necessarily use all of them.

The advantage of this structure is that R permits relative directory labelling, so I know that from the script directory, the data are always in `../data`, the graphics will be in `../graphics` and the images always in `../images`. This structure makes writing reports straightforward as well.

File structure is ordinarily a matter of taste and convenience. I prefer to devote a single directory to each project.

1.8 Getting Started

Different operating systems interact with R in different ways. Some will require that you double–click the appropriate icon, some that you type R and press enter.

R has a huge number of add–on packages that offer different statistical and graphical tools. A small handful of these packages are loaded into memory upon startup. We'll go into this at greater length in section A, but for the moment, it is enough to know that a reasonable amount of functionality is available, and when we need more we can get it.

1.9 Stopping

Use `Ctrl-C` or `Esc` to stop processing. This will work just fine most of the time. You will rarely need to ask your operating system to intercede.

Quit R altogether by typing the command `q()`.

When you do so, R will ask you if you want to save your workspace. The workspace is described in more detail in Section 4.3. Briefly, it contains all the objects that you created and did not delete in this session, as well as any that you loaded from the last session, if there are any.

Generally, whether or not you choose to save the workspace depends on your workflow. Personally, I almost never do it. I prefer to write scripts, which are a complete record of the analysis performed, and if I need to recreate the analysis, I re–source the scripts.

In any case, if you choose to save the workspace then a compressed image of the objects, called `.RData`, will be saved into your working directory. To easily access these objects again in a future session, use the `load` function. If you have followed the workshop's convention (and were once again using the `scripts` directory as the working directory), you would type

```
> load(".RData")
```

1.10 Working Directory

The working directory is the location to and from which R writes and reads files by default.

On Windows, there is a menu item that allows for selecting the working directory, as well as the command line. In the CLI versions of R one can use only the command line.

```
> getwd()                 # What is the working directory?
> setwd("C:/Temp")        # Set this to be the working directory.
```

This latter command has a few points worth discussing.

- `setwd()` is a *function*; we use functions to tell R what we want it to do, and
- `"C:/Temp"` is an *argument* for the function. Arguments are the main way that we tell R what to do the function upon. In this case, we use the argument to tell R that we would like to use `C:/Temp` as the working directory.

The forward slashes are used regardless of the underlying operating system. This is distracting in Windows, but it is again a feature, as it provides a level of portability for our code.

If one wants to read or write to a different location, one must explicitly say so. Life is therefore much easier if all the data and scripts for an analysis are kept in a single (frequently backed up!) location.

1.11 Conventions

There are many different ways to do things in R. There are no official conventions on how the language should be used, but the following thoughts may prove useful in communicating with long-time R users.

1. Although the equals sign "`=`" does work for assignment, it is also used for other things, for example in passing arguments. The arrow "`<-`" is only used for assignment. Please use it for assignment, especially if you want to be able to ask experienced R users to read your code.
2. Spaces are cheap. Use spaces liberally between arguments and between objects and arithmetic operators.
3. Call your objects useful names. Don't call your model `model`, or your dataframe `data`.
4. You can terminate your lines with semi-colons, but don't.

The following code is hard to read and understand. We don't know what role the constant is playing, the semi-colons are hanging around like dags, and the text is dense.

```
> constant=3.2808399;
> x=x*constant;
```

The following code is easier to read and understand. The identities (and the units) of the variables and the constant are obvious by our naming convention. The equations are spaced so that the distinction between operators and variables is easily seen.

```
> feet_per_metre <- 3.2808399
> heights_m <- heights_ft / feet_per_metre
```

Coming back to this code years later, it will be obvious what we did and why we were doing it. Trust me, you won't remember otherwise.

1.12 Acknowledgments

I appreciate the corrections and suggestions for updates from many sources, in particular, Guillaume Therien and Felisa Vazquez–Abad. Also, thanks are due to Kathy Allen, Auro Almeida, Patrick Baker, Hugh Carter, Damien Carthew, Geoff Downes, David Drew, Adam Dziedzic, Stephanie Ewen, Alieta Eyles, Jason Ferris, Rodolfo García–Flores, Toby Gass, Valerie LeMay, Arko Lucieer, Eleanor McWilliams, Jane Medhurst, Craig Mistal, Chris Powell, Audrey Quentin, David Ratkowsky, Neil Sims, Jim Webb, and Charlotte Wickham.

Chapter 2

Quickstart

Here is a list of points to get you going quickly.

- **Interaction**: If you are reading about R from a pdf like this one, you can most easily execute the commands by marking the text (as much as you want), copying it to the clipboard, and pasting to the R console by using `Right-click` then selecting "Paste commands only". R will ignore all the non-R input.
- **Data Input**: getting your data into R from a file is relatively easy. You need to do two things:
  1. be sure that R knows where to find the data. Do this by storing the data in the `data` directory and making the `scripts` directory the working directory. Then you know that the data will be available at `../data`. (Section 5.1.1).
  2. be sure that R knows what type of data they are, by choosing the appropriate function to read them — use `read.csv` for comma–delimited files (e.g. Section 5.1.1).
- **Data Structure**: Use the `str` command to learn about the structure of the data object.
- **Plots**: scatterplots of the data can be constructed using the `plot()` function — see Chapter 8.

Chapter 3

Showcase

The goal of this chapter is to provide a whirlwind tour of some of the capabilities of R, in a relatively coherent workflow. You should study this chapter to get a sense of the kinds of things that R can do, without worrying too much about how it does them.

Copy the code swatches you see here to the R console, and watch what happens. Try to relate what you see to your day-to-day analysis challenges.

Don't worry too much about the nitty-gritty of execution. Some of what you see will be familiar, some will certainly not. Later on we will repeat these steps in a different context, and explain them more carefully.

3.1 Getting Ready

Make sure that your directory structure is as laid out in Section 1.7, and the `ufc.csv` dataset is in the `data` directory. Then, start R in the manner appropriate to your operating system. Finally, make that directory your working directory, as follows. Select the *Change dir...* option from the *File* menu, and navigate to the `scripts` directoty inside the workshop directory. Click `OK`. At this point, the code below should work by `Copy - Paste Commands Only` without further intercession.

3.2 Data Input

First, we read the data in from a comma-delimited file. The data are stand inventory data on a 300 ha parcel of the University of Idaho Experimental Forest, called the Upper Flat Creek stand. The sample design was a systematic sample of 7 m^2/ha BAF variable radius plots; diameter measured for all trees and height for a subsample[1]. In the dataset, some plots are empty. Those plots are flagged by having trees with missing diameters and blank species.

```
> ufc <- read.csv("../data/ufc.csv")
```

Let's also input the sample design parameter and population size for later computation.

```
> ufc_baf <- 7
> ufc_area <- 300
```

3.3 Data Structure

Here we examine the `str()`ucture of the data, check the first few rows, and count the missing values (`na`) by column.

```
> str(ufc)
```

```
'data.frame':        637 obs. of  5 variables:
 $ plot   : int  1 2 2 3 3 3 3 3 3 3 ...
```

[1] If you don't know what this means then don't worry about it. Seriously.

```
 $ tree   : int  1 1 2 1 2 3 4 5 6 7 ...
 $ species: Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6 11 11 11 11 11 ...
 $ dbh    : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ height : int  NA 205 330 NA 300 NA NA 207 NA NA ...
```

```
> head(ufc)
```

```
  plot tree species dbh height
1    1    1          NA     NA
2    2    1      DF 390    205
3    2    2      WL 480    330
4    3    1      WC 150     NA
5    3    2      GF 520    300
6    3    3      WC 310     NA
```

```
> colSums(is.na(ufc))
```

```
   plot    tree species     dbh  height
      0       0       0      10     246
```

3.3.1 Manipulation

Ah, some diameters are missing. Those correspond to empty plots. Also, many heights are missing; the height attributes must have been subsampled. Next, obtain familiar units on our tree measurements (cm for dbh and m for height, respectively).

Note that we append the units to the variable names to be sure that the reader can easily interpret the statistics that we will later compute.

```
> ufc$height_m <- ufc$height/10
> ufc$dbh_cm <- ufc$dbh/10
```

We now examine the structure again, using the `str` function.

```
> str(ufc)
```

```
'data.frame':        637 obs. of  7 variables:
 $ plot    : int  1 2 2 3 3 3 3 3 3 3 ...
 $ tree    : int  1 1 2 1 2 3 4 5 6 7 ...
 $ species : Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6 11 11 11 11 11 ...
 $ dbh     : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ height  : int  NA 205 330 NA 300 NA NA 207 NA NA ...
 $ height_m: num  NA 20.5 33 NA 30 NA NA 20.7 NA NA ...
 $ dbh_cm  : num  NA 39 48 15 52 31 28 36 34 26 ...
```

We have created two more variables, which are now storred in our dataframe.

3.3.2 Data Summaries

Now we obtain some useful snapshots of the data.

```
> range(ufc$dbh_cm)
```

```
[1] NA NA
```

```
> range(ufc$height_m, na.rm = TRUE)
```

```
[1]  0 48
```

Zero height is a problem, because trees are usually taller than 0 m. Let's knock that observation out by flagging it with an official R flag that means a missing value — `NA`.

```
> ufc$height_m[ufc$height_m < 0.1] <- NA
> range(ufc$height_m, na.rm = TRUE)
```

```
[1]  3.4 48.0
```

How do the numbers of trees of each species look?

```
> table(ufc$species)

     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

Here we see the species codes and the numbers of trees of each species. Possibly some of the codes are erroneous. For example, the F(ir) and the FG(?) are probably intended to be GF (Grand fir). Let's make that change.

```
> ufc$species[ufc$species %in% c("F", "FG")] <- "GF"
> ufc$species <- factor(ufc$species)
> table(ufc$species)

     DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3 188   5   7   4  14 251  34  44
```

How many missing heights do we have for each species? We can count the number of trees that have missing heights as follows.

```
> table(ufc$species[is.na(ufc$height_m)])

     DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 10  20   1  70   0   4   2   4 112  12  12
```

These following three summaries are using the standard graphical tools. (Figure 3.1, first three panels).

```
> boxplot(dbh_cm ~ species, data = ufc, ylab = "Dbh (cm)")
> boxplot(height_m ~ species, data = ufc, ylab = "Height (m)")
> scatter.smooth(ufc$dbh_cm, ufc$height_m)
> hist(ufc$dbh_cm, breaks = (0:50) * 2.5, col = "darkseagreen3",
+     main = "")
```

We can also use more developed graphical tools. To do so we need to obtain access to an R package that is installed with R, but not loaded by default: `lattice`.

```
> library(lattice)
```

The following code creates a histogram of tree diameters by species (Figure 3.2).

```
> histogram(~dbh_cm | species, data = ufc)
```

3.3.3 Conditional Estimates

We can easily obtain summary statistics of a variable of interest, conditional on some other variable. For the data that we have, it is interesting to obtain summary statistics of the variables conditional on the tree species. First, we get a tree count by species (compare with the output from the `table` function).

```
> tapply(ufc$dbh_cm, ufc$species, length)

     DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3 188   5   7   4  14 251  34  44
```

Then we get the mean of the diameters at breast height (in cm!) by species.

```
> tapply(ufc$dbh_cm, ufc$species, mean)

               DF       ES       GF       HW       LP
      NA 38.37143 40.33333 35.20106 20.90000 23.28571
      PP       SF       WC       WL       WP
56.85000 13.64286 37.50757 34.00588 31.97273
```

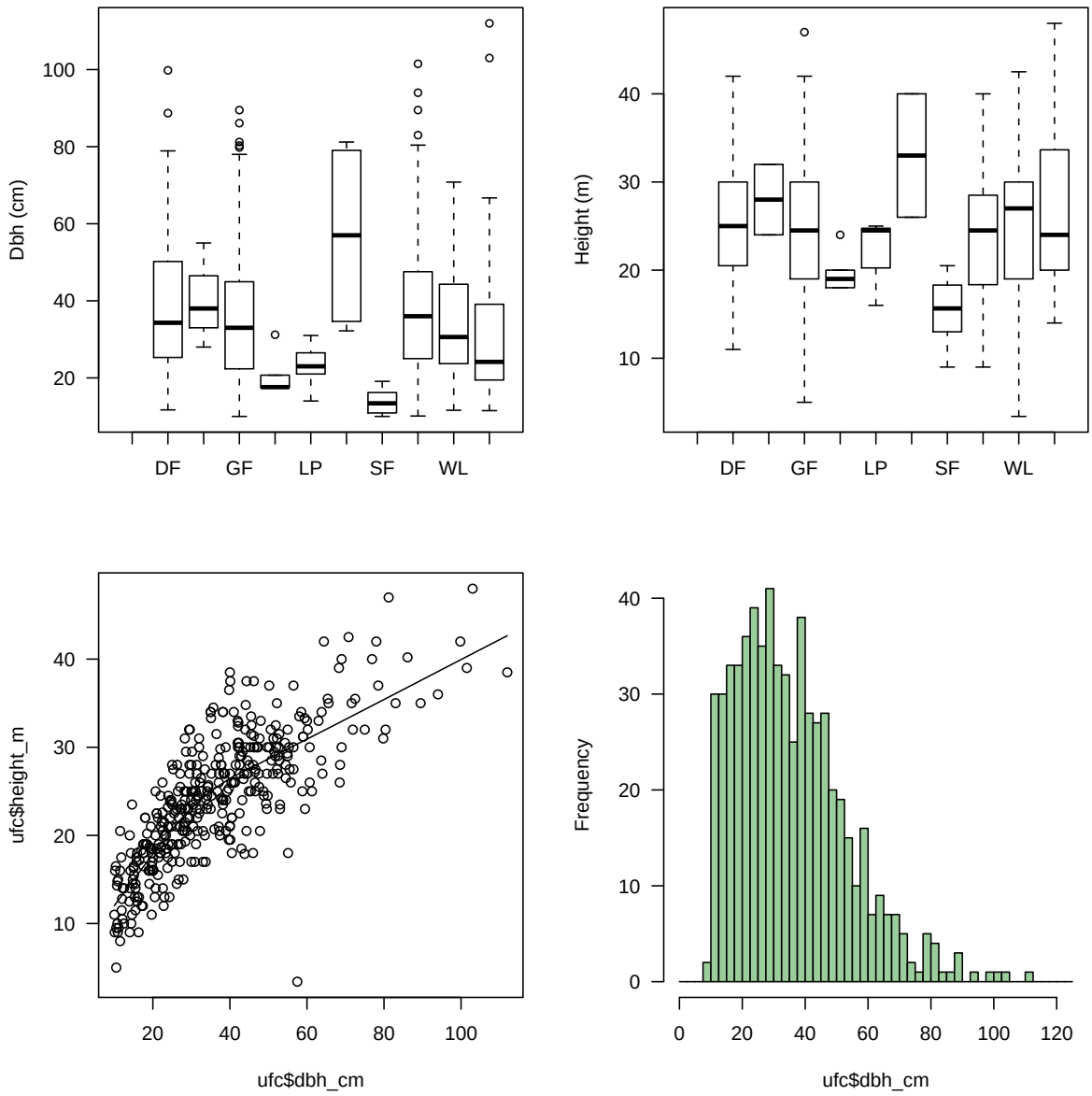

Figure 3.1: Simple summary plots of the UFC dataset.

3.4 Models

Let's take a closer look at the relationship between height and diameter, using some simple modeling tools. We start with linear regression (`lm` fits the linear model).

```
> hd_lm_1 <- lm(height_m ~ dbh_cm, data = ufc)
> summary(hd_lm_1)

Call:
lm(formula = height_m ~ dbh_cm, data = ufc)

Residuals:
      Min        1Q    Median        3Q       Max
-27.50661  -2.81622   0.08051   2.69760  13.20520

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.46781    0.53074   23.49   <2e-16 ***
```

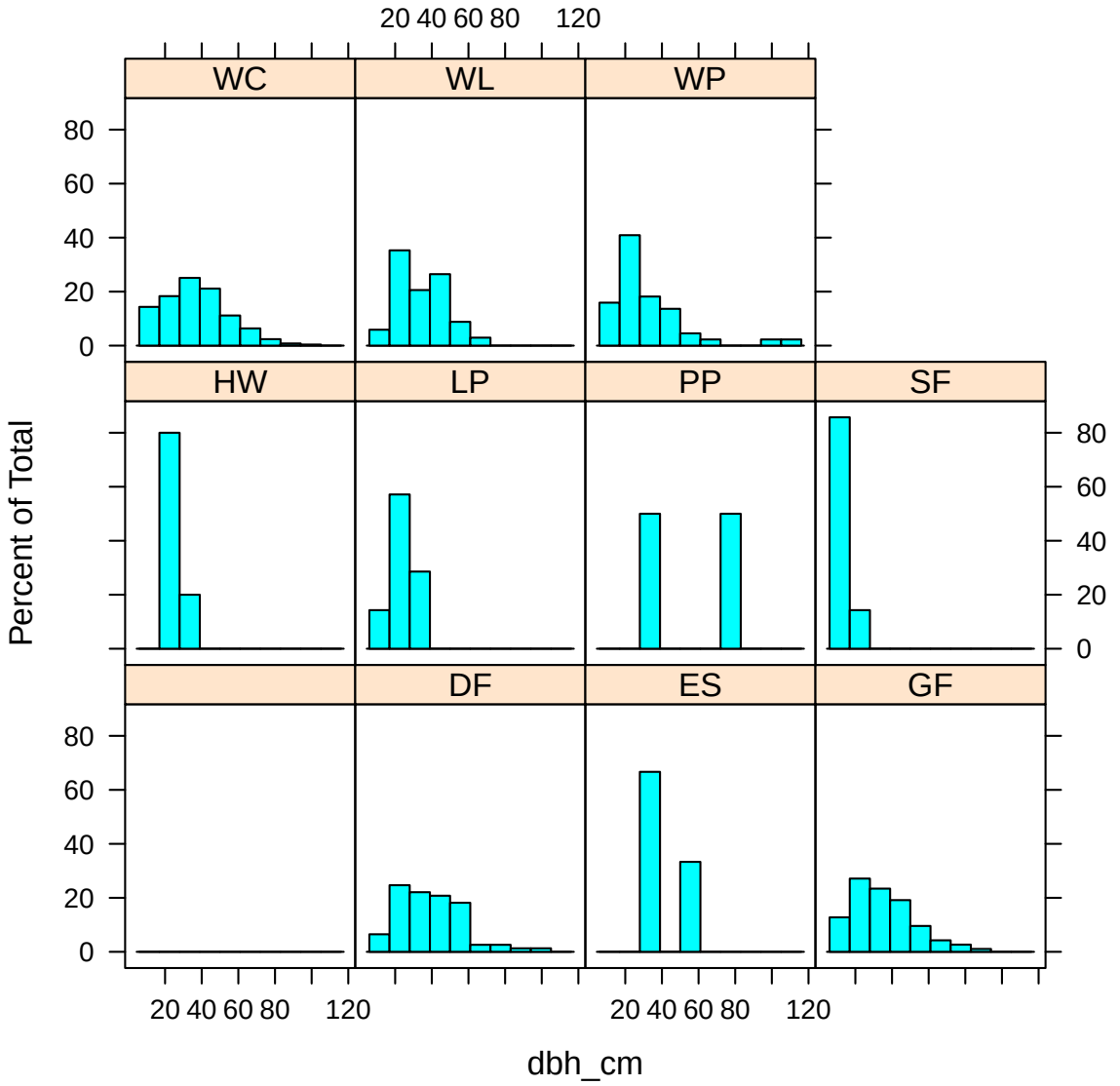

Figure 3.2: A histogram of tree diameters by species

```
dbh_cm       0.32067    0.01309   24.50   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.643 on 388 degrees of freedom
  (247 observations deleted due to missingness)
Multiple R-squared: 0.6075,         Adjusted R-squared: 0.6065
F-statistic: 600.5 on 1 and 388 DF,  p-value: < 2.2e-16
```

The following graph provides a summary of the data and some information about the utility of a straight line for expressing the relationship.

```
> scatter.smooth(ufc$dbh_cm, ufc$height_m)
> abline(hd_lm_1, col = "red")
```

Now let's fit a distinct line for each species, and then graphically compare the regression from the previous model.

```
> hd_lm_2 <- lm(height_m ~ dbh_cm * species, data = ufc)
```

The following complicated code provides insight into the graphical flexibility (Figure 3.3). Examine the graph carefully and try to interpret it.

```
> xyplot(height_m ~ dbh_cm | species, panel = function(x,
+     y, ...) {
+     panel.xyplot(x, y)
+     panel.abline(lm(y ~ x), col = "blue")
+     panel.abline(hd_lm_1, col = "darkgreen")
+     if (sum(!is.na(y)) > 2) {
+         panel.loess(x, y, span = 1, col = "red")
+     }
+ }, subset = species != "", xlab = "Dbh (cm)",
+     ylab = "Height (m)", data = ufc)
```

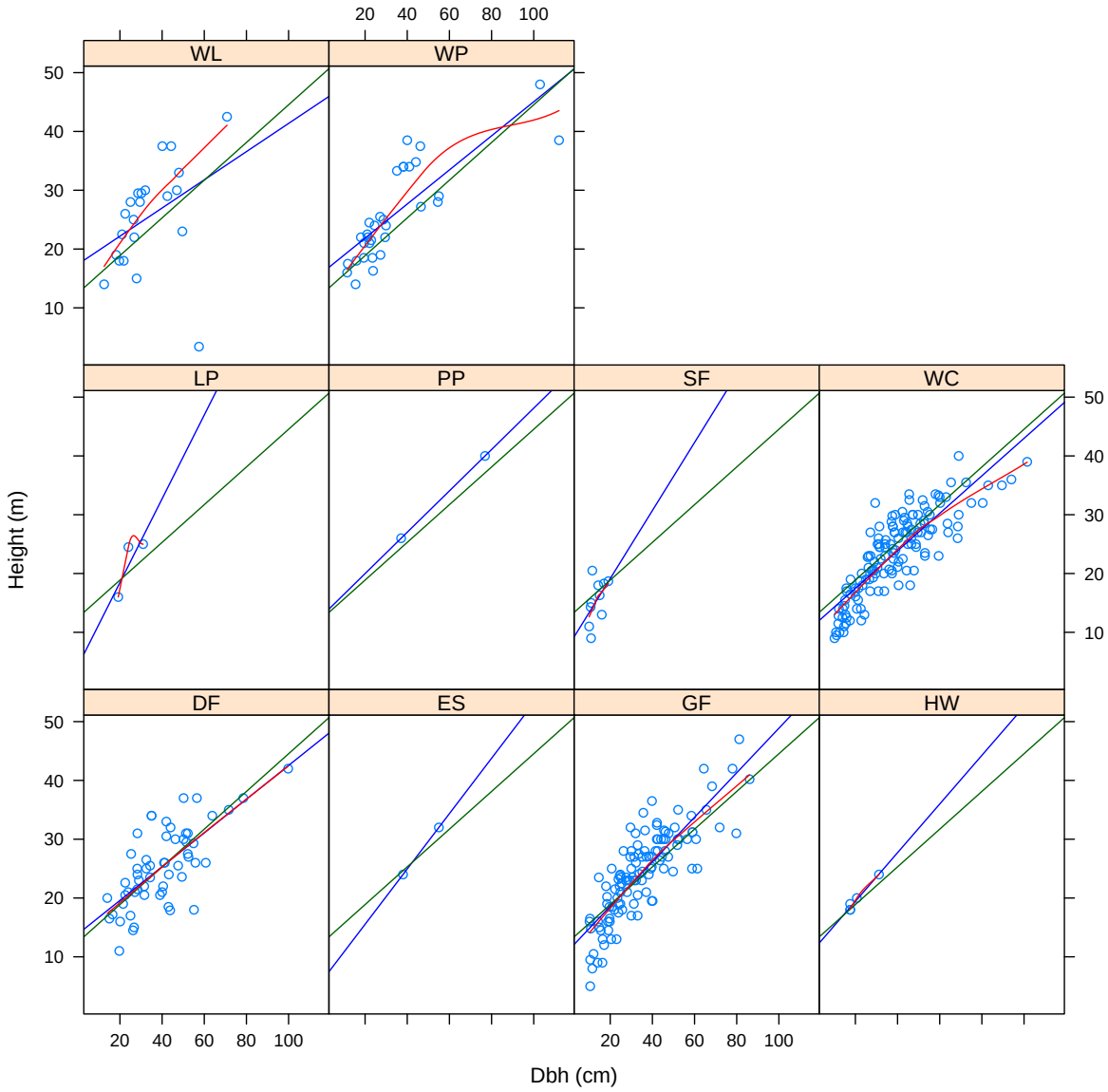

Figure 3.3: Scatterplot of tree heights and diameters by species. The green line is the overall line of best fit, the blue line is the species-specific line of best fit, and the red line is a smoothed curve of best fit.

Recall that many of the trees were missing height measurements. Perhaps we should impute[2] those missing heights. We could use a model for the imputation; let's use a simple allometric[3] mixed-effects[4] model (Robinson and Wykoff, 2004). We need to access the code in another R package, `nlme`.

```
> library(nlme)
> hd_lme <- lme(I(log(height_m)) ~ I(log(dbh_cm)) * species,
+               random = ~ I(log(dbh_cm)) | plot,
+               na.action = na.exclude,
+               data = ufc)
> predicted_log_heights <- predict(hd_lme, na.action=na.exclude, newdata=ufc)
```

The next steps are, firstly, copy the predicted heights, and secondly, copy over the predicted heights with the observed heights where they exist. If you have done this correctly then you will get no output!

```
> ufc$p_height_m[!is.na(ufc$dbh_cm)] <- exp(predicted_log_heights)
> ufc$p_height_m[!is.na(ufc$height_m)] <- ufc$height_m[!is.na(ufc$height_m)]
```

What are the characteristics of the new variable?

```
> summary(ufc$p_height_m)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   3.40   18.51   23.00   23.66   28.51   48.05   10.00
```

The new variable is defined as follows: for trees for which height was measured, their height is used. For trees for which height was missing, the height that was predicted from the model was used. For empty records, the height is missing.

3.5 Functions

Functions exist to compute tree volume from diameter and height, by species (Wykoff et al., 1982), for this geographical area. They look complex, but are easy to use, and efficient. Here we keep the unit conversions outside the function, but they could also be inside.

```
> vol_fvs_ni_bdft <- function(spp, dbh_in, ht_ft){
+   bf_params <-
+     data.frame(
+                species = c("WP", "WL", "DF", "GF", "WH", "WC", "LP", "ES",
+                  "SF", "PP", "HW"),
+                b0_small = c(26.729, 29.790, 25.332, 34.127, 37.314, 10.472,
+                  8.059, 11.851, 11.403, 50.340, 37.314),
+                b1_small = c(0.01189, 0.00997, 0.01003, 0.01293, 0.01203,
+                  0.00878, 0.01208, 0.01149, 0.01011, 0.01201, 0.01203),
+                b0_large = c(32.516, 85.150, 9.522, 10.603, 50.680, 4.064,
+                  14.111, 1.620, 124.425, 298.784, 50.680),
+                b1_large = c(0.01181, 0.00841, 0.01011, 0.01218, 0.01306,
+                  0.00799, 0.01103, 0.01158, 0.00694, 0.01595, 0.01306))
+   dimensions <- data.frame(dbh_in = dbh_in,
+                            ht_ft = ht_ft,
+                            species = as.character(spp),
+                            this_order = 1:length(spp))
+   dimensions <- merge(y=dimensions, x=bf_params, all.y=TRUE, all.x=FALSE)
+   dimensions <- dimensions[order(dimensions$this_order, decreasing=FALSE),]
+   b0 <- with(dimensions, ifelse(dbh_in <= 20.5, b0_small, b0_large))
+   b1 <- with(dimensions, ifelse(dbh_in <= 20.5, b1_small, b1_large))
+   volumes_bdft <- b0 + b1 * dimensions$dbh_in^2 * dimensions$ht_ft
+   return(volumes_bdft)
+ }
```

[2] If you don't know what this means then don't worry about it. Seriously.
[3] If you don't know what this means then don't worry about it. Seriously.
[4] If you don't know what this means then don't worry about it. Seriously.

Having saved the function we can use it in our code like any other function. One small complication is that the function was written for imperial units, so we have to take care of the unit conversion.

```
> cm_to_inches <- 1/2.54
> m_to_feet <- 3.281
> bd_ft_to_m3 <- 0.002359737
```

Now we can use the function that we wrote. What do you think we achieve by using `with`, below?

```
> ufc$vol_m3 <- with(ufc, vol_fvs_ni_bdft(species,
+                                         dbh_cm * cm_to_inches,
+                                         p_height_m * m_to_feet) * bd_ft_to_m3)
```

The following are examples of the construction of new variables. The variables are related to the analysis of forest inventory. Look away if such things disturb you. But, do execute them. If the execution is successful then you will get no feedback.

```
> ufc$g_ma2 <- ufc$dbh_cm^2 * pi/40000
> ufc$tree_factor <- ufc_baf/ufc$g_ma2
> ufc$vol_m3_ha <- ufc$vol_m3 * ufc$tree_factor
```

We now have an estimate of the volume in cubic metres per hectare represented by each tree.

```
> str(ufc)
```

```
'data.frame':        637 obs. of  12 variables:
 $ plot       : int  1 2 2 3 3 3 3 3 3 ...
 $ tree       : int  1 1 2 1 2 3 4 5 6 7 ...
 $ species    : Factor w/ 11 levels "","DF","ES","GF",..: 1 2 10 9 4 9 9 9 9 9 ...
 $ dbh        : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ height     : int  NA 205 330 NA 300 NA NA 207 NA NA ...
 $ height_m   : num  NA 20.5 33 NA 30 NA NA 20.7 NA NA ...
 $ dbh_cm     : num  NA 39 48 15 52 31 28 36 34 26 ...
 $ p_height_m : num  NA 20.5 33 13.8 30 ...
 $ vol_m3     : num  NA 0.4351 0.98 0.0575 1.3392 ...
 $ g_ma2      : num  NA 0.1195 0.181 0.0177 0.2124 ...
 $ tree_factor: num  NA 58.6 38.7 396.1 33 ...
 $ vol_m3_ha  : num  NA 25.5 37.9 22.8 44.1 ...
```

3.6 Plot Level Processing

Our next step is to aggregate the tree–level volume estimates to the plot level. First, we construct a *dataframe* (called `ufc_plot` that has the (known) plot locations. A dataframe will be defined more formally later, for the moment, treat it as a receptacle to keep variables in.

```
> ufc_plot <- as.data.frame(cbind(c(1:144), rep(c(12:1),12),
+                                 rep(c(1:12), rep(12,12))))
> names(ufc_plot) = c("plot","north.n","east.n")
> ufc_plot$north = (ufc_plot$north.n - 0.5) * 134.11
> ufc_plot$east = (ufc_plot$east.n - 0.5) * 167.64
```

Then we can construct plot-level values for any of the measured characteristics, for example, merchantable volume per hectare, by adding up the volumes per hectare of the trees within each plot. We know how to do this.

```
> ufc_plot$vol_m3_ha <- tapply(ufc$vol_m3_ha, ufc$plot,
+     sum, na.rm = TRUE)
```

How does the sample look, spatially (Figure 3.4)?

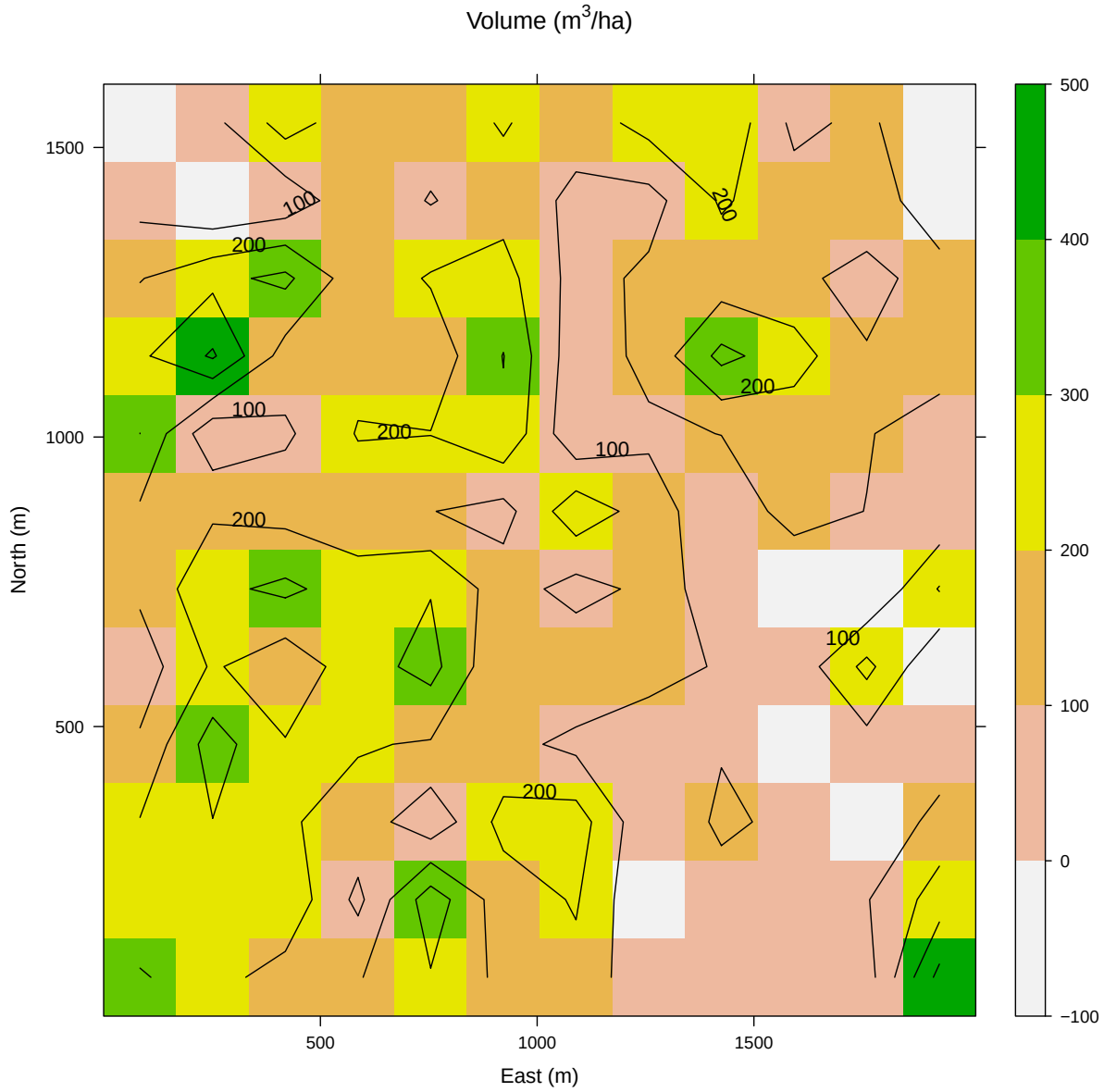

Figure 3.4: Contour plot of plot-level volumes for UFC forest inventory data. The units are cubic meters per hectare. 300 is quite a lot.

```
> contourplot(vol_m3_ha ~ east * north, main = expression(paste("Volume (",
+     m^3, "/ha)", sep = "")), xlab = "East (m)",
+     ylab = "North (m)", region = TRUE, col.regions = terrain.colors(11)[11:1],
+     data = ufc_plot)
```

Pretty patchy. But, let's save the plot data object for later (specifically, Section 8.6.1).

```
> save("ufc_plot", file="../images/ufc_plot.RData")
```

We can also compute total, confidence intervals, and so on.

```
> mean(ufc_plot$vol_m3_ha) * ufc_area
```

```
[1] 44856.21
```

```
> (mean(ufc_plot$vol_m3_ha) + qt(c(0.025,0.975),
+                                df = length(ufc_plot$vol_m3_ha) - 1) *
+   sd(ufc_plot$vol_m3_ha) / sqrt(length(ufc_plot$vol_m3_ha))) * ufc_area
```

```
[1] 39978.39 49734.04
```

Finally, let's estimate the total volume and 95% confidence interval for each species. First, get the plot-level estimates by species. As before, these commands should result in no output.

```
> vol_by_species <- tapply(ufc$vol_m3_ha, list(ufc$plot, ufc$species), sum)
> vol_by_species[is.na(vol_by_species)] <- 0
```

We can examine the distributions as follows:

```
> boxplot(as.data.frame(vol_by_species))
```

Now we can apply the wonders of vectorization!

```
> (totals <- apply(vol_by_species, 2, mean) * ufc_area)
```

```
                  DF          ES          GF          HW
   0.0000  5303.6005   210.4520 17817.4133   479.8852
       LP          PP          SF          WC          WL
 453.5305   499.5319   913.3555 12686.8227  2540.6045
       WP
3951.0148
```

```
> (ci_bars <- apply(vol_by_species, 2, sd) / sqrt(144) * 1.96 * ufc_area)
```

```
                DF        ES        GF        HW        LP
  0.0000 1538.9513  311.4001 3493.9574  645.3071  592.3508
      PP        SF        WC        WL        WP
506.4334  703.1405 2464.3451 1333.0400 1623.2950
```

An augmented boxplot shows the variability in the data and the estimates of the mean volume per hectare by plot (Figure 3.5). We add the overall estimate by species and the 95% confidence interval to aid interpretation. Also note the use of `expression` and `paste` to construct the y-axis. The first level represents the empty plots.

```
> boxplot(as.data.frame(vol_by_species),
+         ylab=expression(paste("Volume (", m^3, ha^-1, ")")))
> lines(1:11, (totals - ci_bars) / ufc_area, col="blue")
> lines(1:11, (totals + ci_bars) / ufc_area, col="blue")
> lines(1:11, totals / ufc_area, col="red")
```

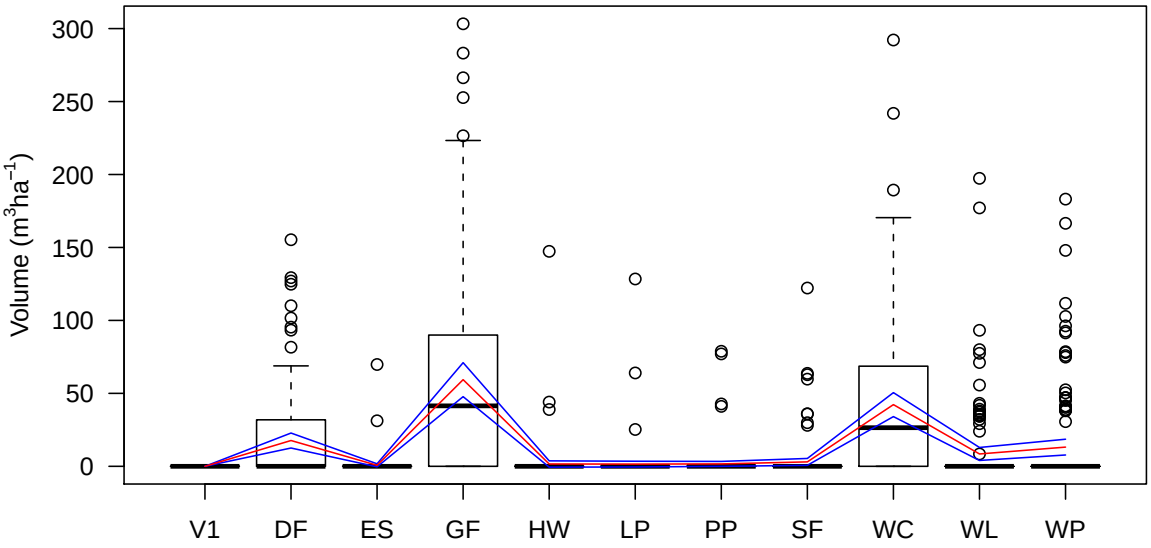

Figure 3.5: Boxplot of plot-level volumes per species.

Chapter 4

Infrastructure

R provides several tools to help us keep organized and efficient. We showcase the simple and important ones below. The complicated and important stuff, such as data analysis, environments, scoping, and so on, will wait.

4.1 Getting Help

There are four main sources of assistance: the help files, the manuals, the R-help archive, and R-help itself.

4.1.1 Locally

While working through this document, it is likely that you will find commands that are used in example code which have been inadequately explained, or perhaps ignored completely. When you find such commands, you should read about them in `help()`.

The help files can be accessed using the `help()` command, which has `?` as a prefix-style shortcut. We can get help on commands this way; for example

```
> ?mean             # Works!  mean() is an R command
> help(mean)        # Works!  mean() is an R command
```

However, we can't get information on concepts unless they have been specifically registered with the help system.

```
> help(regression)  # Fails!  regression() is not an R command
```

This fails because "regression" isn't an R command. If we want to know which commands refer to regression, we use either of:

```
> ??regression
> help.search("regression")
```

The output from this command is long and confusing. We have to read through all the descriptions of the functions that correspond to this call until we find the one that we want.

```
MASS::lm.ridge          Ridge Regression
MASS::lqs               Resistant Regression
MASS::polr              Ordered Logistic or Probit Regression
MASS::rms.curv          Relative Curvature Measures for Non-Linear
                        Regression

...

stats::ksmooth          Kernel Regression Smoother
stats::lm               Fitting Linear Models
stats::lm.influence     Regression Diagnostics

...
```

Now we can try `help(lm)`. It also tells us that the `lm()` function is in the `stats` package. It doesn't tell us that the `stats` package is already loaded, but, it is. In passing, let me recommend the following function, whic hwill tell you what version of the software you are using, and what packages are loaded. It's considered mandatory for reporting problems.

```
> sessionInfo()

R version 2.15.0 (2012-03-30)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

locale:
[1] C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] tools_2.15.0
```

Note that the help command omits the parentheses that normally follow a function call. Also note that the level of help provided about the different commands is variable. If you can think of a clearer way to express something, then you can submit an improvement to the community.

I have found that the best way to get a feel for what information the commands are expecting is to try out the examples that usually appear at the end of the help information. For most help files we can just copy those example commands and paste them to the console, and see what happens. The commands can then be altered to suit our needs. Furthermore, these examples are often miniature data analyses, and provide pointers to other useful functions that we can try.

We can also access the files that are installed with R using a WWW browser. Again inside R, execute the following command.

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

This opens a browser (or a window inside an existing browser) in to which help results will be sent, and within which you can then point, click, and search for keywords. You may need to set the default browser, depending on your operating system. That would be done by, for example,

```
> options(browser="firefox")
```

This browser also provides hyperlinked access to R's manuals, which provide a great deal of very useful information. The manuals are constantly under development, so check back regularly.

4.1.2 Remotely

There is a thriving community of programmers and users who will be happy to answer carefully-worded questions and, in fact, may well have already done so. Questions and answers can be easily found from inside R using the following commands:

```
> RSiteSearch("Bates lmer")              # Douglas Bates on lmer
> RSiteSearch("{logistic regression}")   # matches the exact phrase
```

If you don't find an answer after a solid search, then you should consider asking the community, using the email list R-help. There is a posting guide to help you write questions that are most likely to obtain useful answers - it is essential reading! Point your browser to: http://www.r-project.org/posting-guide.html

Details on joining the email list group can be found at: https://stat.ethz.ch/mailman/listinfo/r-help. I recommend the digest option; emails arrive at a rate of up to 100 per day.

4.2 Writing scripts

Using R effectively is as simple as writing scripts. We can write the scripts in any text editor - Winedt, MS Wordpad, Open Office, vi, emacs, or Xemacs. Some editors, for example emacs and Xemacs, will make our lives easier by providing us with syntactically aware tabbing, text colours, and command completion, which are all delicious, as well as running R as a sub-process, which means that the R output is also a manipulable buffer.

Regardless of our editor, we save the scripts to a known directory, and then either copy and paste them into the R console, or read them in using the one of the following commands:

```
> source(file="C://path/to/filenameworkshop/scripts/file.R", echo=T)
> source(file="../scripts/file.R", echo=T)
> source("file.R", echo=TRUE) # If file.R is in working directory
```

Note the use of forward slashes to separate the directories. Also, the directory names are case-sensitive and are permitted to contain blank spaces.

Writing readable, well-commented scripts is a really good habit to get into early; it just makes life much easier in the future. Large projects are vastly simplified by rigorous script writing.

A key element of good script writing is commentary. In R the comment symbol is the # symbol. Everything on a line after a # is ignored. Some editors will tab comments based on the number of #'s used.

Instructions can be delimited by line feeds or semi-colons. R is syntactically aware, so if you insert a return before your parentheses or brackets are balanced, it will politely wait for the rest of the statement.

Script writing is a very powerful collaborative tool. It's very nice to be able to send your code and a raw data file to a cooperator, and know that they can just `source()` the code and run your analysis on their machine.

4.3 Work Spaces

R uses the concept of *workspaces* to help us keep our objects organized. All the objects that we make, whether directly or indirectly, are stored within a workspace. We can save, load, share, or archive these workspaces. We can also list the contents of our workspace using `ls()`, and clear items from it using `rm()`. For example,

```
> test <- 1:10                          # Create an object
> ls()                                  # List all objects in workspace
> save.image(file="../images/W-Shop.RData")
    # Save all objects to a binary file in the images directory
> save(test, file="../images/test.RData")
    # Save test to a binary file
> rm(test)                              # Delete the "test" object.
> ls()                                  # List the objects that we have
> rm(list=ls())                         # Delete them all
> load(file="../images/W-Shop.RData")   # Load them again
> ls()                                  # List the objects that we have
```

The binary files to which the `save.image()` command writes the objects in our workspace are readable by R under all operating systems. They are considerably compressed when compared with, say, comma delimited files.

Work spaces can be very useful. Data analyses can often be broken into chunks, some of which can be very time-consuming, and others of which can be quite quick. It is inconvenient to repeat the time-consuming chunks unnecessarily. The results of these chunks can be saved as binary images, and loaded at will. For example, you may have a dataset of 20 megabytes that requires considerable cleaning and manipulation before the proper analysis can begin. It would be tedious to repeat that process every time. It's much easier to do it once anmd save the work space in an image.

Work spaces also hide nefarious traps for the unwary. It is possible for one user, let's say a student, to construct a script containing numerous steps of analysis, and try to share it with another user, let's say a supervisor, only for the latter to find that the success of the script requires an object stored and long forgotten in the first user's workspace! This is a common pitfall for collaboration. My students and I will, as much as possible, preface all our code with:

```
> rm(list=ls())
```

This ensures a clean workspace. `rm()` is the function to remove objects, and `ls()` lists the objects in your workspace.

4.4 History

R's history facility keeps track of every command that we type in, even the dumb ones. This can be very helpful, especially as one can access it through the up and down arrows. So, if we mistype a command the remedy is as simple as hitting the up arrow, editing the command, and hitting `Enter`.

This is just fine for small problems; it is quick and effective. It rapidly becomes very inefficient, though, when you want to repeat a number of commands with small alterations. It will drive you insane. Write scripts. See Section 4.2.

It is sometimes useful after a session to save the history to a file, for subsequent conversion to a script.

```
> savehistory(file="History.txt")   # History saved as a text document
> loadhistory(file="History.txt")   # Text document loaded into History
```

We can then use this history as the basis for writing a script (see Section 4.2). Comments can be inserted as we have seen above, and then the whole thing can be read in to R using the `source` command. I almost never do this. I almost never develop scripts inside R. I prefer to use an external editor that is optimized for script development - examples are emacs, Xemacs, and WinEdt.

Chapter 5

Interface

5.1 Importing and Exporting Data

There is no better way to become familiar with a new program than to spend time with it using data that you already know. It's so much easier to learn how to interpret the various outputs when you know what to expect! Importing and exporting data from R can seem a little confronting when you're accustomed to Microsoft Wizards, but it's easy to pick up, and there's much more control and flexibility. Whole manuals have been written on the topic, because there are so many different formats, but we'll focus on the simplest one: comma-delimited files.

Comma-delimited files have become the *lingua franca* of small-scale data communication: pretty much every database can input and output them. They're simply flat files, meaning that they can be represented as a two-dimensional array; rows and columns, with no extra structure. The columns are separated by commas and the rows are separated by line feeds and possibly carriage returns, depending on your operating system. The column names and the row names may or may not be stored in the first row and column respectively. It's always a good idea to check!

```
Plot, Tree, Species, DBH, Height, Damage, Comment
1, 1, PiRa, 35.0, 22, 0,
1, 2, PiRa, 12.0, 120, 0, "Surely this can't be a real tree!"
1, 3, PiRa, 32.0, 20, 0,
```

... and so on.

R provides a limited number of functions to interact with the operating system. I rarely use them, but when I do they are invaluable. The command to obtain a list of the objects in a directory is `list.files()`.

This can be most useful when you are writing a script that requires R to read in a number of files, but at the time of writing you don't know the names of the files

```
> list.files("../data")
```

```
 [1] "DBT_Data_For_A_Robinson_20090702.csv"
 [2] "DBT_Metadata_ec28_20090702.doc"
 [3] "Sawlog_Data_For_A_Robinson_20090702.csv"
 [4] "blowdown.txt"
 [5] "ehc.csv"
 [6] "gutten.csv"
 [7] "hanford.csv"
 [8] "ndvi.csv"
 [9] "rugby.csv"
[10] "stage.csv"
[11] "ufc.csv"
[12] "wind-river.csv"
```

or, indeed, how many there will be.

```
> length(list.files("../data"))
```

```
[1] 12
```

5.1.1 Import

R needs a few things to import data. Firstly, it has to know where they are, secondly, where you want to put them, and finally, whether there are any special characteristics. I always start by examining the data in a spreadsheet - Excel does fine - because I'd like to know several things:

1. Are the rows and columns consistent? Excel spreadsheets can be problematic to import, as users often take advantage of their flexibility to include various things that R won't understand.
2. Are the columns or rows labeled? Will the labels be easily processed?
3. Are any of the data missing? Are missing data explicitly represented in the data? If so, what symbol(s) are used to represent the missing values?
4. Are there any symbols in the database that might make interpretation difficult?

We should also know the location of the file to be added. Then we can tell R how to load the data. Assuming that the data are indeed appropriate and comma-delimited, we use the `read.csv()` command. Note that we have to give R the object name - in this case, `ufc` - in order for it to store the data. Using an absolute path (if we know it) looks like this.

```
> ufc <- read.csv(file="C://path/to/workshop/data/ufc.csv")
```

We can also read the file using a relative address.

```
> ufc <- read.csv(file="../data/ufc.csv")
```

Note the use of forward slashes to separate the directory names. The directory names are case sensitive and are permitted to contain blank spaces. If you use `read.csv()` then R assumes that the first row will be the column names; tell it otherwise by using the option `header=FALSE`. See `?read.csv` for details.

Also note that you can call the object anything you want. An easily-recalled and distinct name will prove to be very useful.

Other commands are useful when the data have a more general structure: `read.fwf()` reads in data of fixed-width format, `read.table()` will accommodate a broader collection of arrays, and `scan()` will read arbitrary text-based data files.

When the data are imported, a few other useful tools help us to check the completeness of the dataset and some of its attributes. Use these regularly to be sure that the data have come in as you had intended.

```
> dim(ufc)          # Reports the number of rows and columns

[1] 637   5

> names(ufc)        # Lists the column names

[1] "plot"    "tree"    "species" "dbh"     "height"
```

I prefer to always work with lower-case variable names. One way to ensure that the names are always lower case is to enforce it.

```
> names(ufc) <- tolower(names(ufc))

> str(ufc)          # Reports the structure of ufc

'data.frame':        637 obs. of  5 variables:
 $ plot   : int  1 2 2 3 3 3 3 3 3 3 ...
 $ tree   : int  1 1 2 1 2 3 4 5 6 7 ...
 $ species: Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6 11 11 11 11 11 ...
 $ dbh    : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ height : int  NA 205 330 NA 300 NA NA 207 NA NA ...

> head(ufc)         # Prints the first 5 rows of ufc
```

```
  plot tree species dbh height
1    1    1          NA     NA
2    2    1      DF 390    205
3    2    2      WL 480    330
4    3    1      WC 150     NA
5    3    2      GF 520    300
6    3    3      WC 310     NA

> ufc[1:5,]        # Prints the first 5 rows of ufc

  plot tree species dbh height
1    1    1          NA     NA
2    2    1      DF 390    205
3    2    2      WL 480    330
4    3    1      WC 150     NA
5    3    2      GF 520    300
```

The last among the commands hints at one of the most useful elements of data manipulation in R: subscripting. We'll cover that in Section 6.3.

5.1.2 Export

Exporting data from R is less commonly done than importing, but fortunately it is just as straightforward. Again we prefer to use the comma–delimited (`csv`) file format unless there is any reason not to.

```
> write.csv(ufc, file="C://path/to/filenameworkshop/output/file.csv")
> write.csv(ufc, file="../output/file.csv")
```

Exporting graphics is every bit as simple. Skipping ahead a little, we will use the `plot()` command to create two-dimensional graphs of variables.

```
> plot(1:10,1:10)  # ... or can do something more sophisticated ...
```

In Windows, we can right-click this graph to copy it to the clipboard, either as a bitmap, or as a windows metafile. The latter file type is extremely useful as we can paste such files in to documents – for example, MS Word – and they can be resized or edited subsequently.

Saving graphics in a more permanent file format is also possible. To save a graph as a pdf, for example, we will write

```
> pdf("../graphics/fileName.pdf")  # Opens a pdf device
> plot(1:10,1:10)      # ... or can do something more sophisticated ...
> dev.off()            # Closes the pdf device and saves the file
```

Similarly simple protocols are available for `postscript()`, `jpeg()` etc. You can learn more via

```
> ?Devices
```

Chapter 6

Data

Strategies for convenient data manipulation are the heart of the R experience. The object orientation ensures that useful and important steps can be taken with small, elegant pieces of code.

6.1 Data: Objects and Classes

So, what does it mean to say that R is object oriented? Simply, it means that all interaction with R is through objects. Data structures are objects, as are functions, as are scripts. This seems obscure right now, but as you become more familiar with it you'll realize that this allows great flexibility and intuitiveness in communication with R, and also is occasionally a royal pain in the bum.

For the most part, object orientation will not change the way that we do things, except that it will sometimes make them easier than we expect. However, an important consequence of object orientation is that all objects are members of one or more classes. We will use this facility much later (in Section 15.5, to be precise.)

We create objects and assign names to them using the left arrow: "<-". R will guess what class we want the object to be, which affects what we can do with it. We can change the class if we disagree with R's choice.

```
> a <- 1                  # Create an object "a" and
>                         #    assign to it the value 1.
> a <- 1.5                # Wipe out the 1 and make it 1.5 instead.
> class(a)                # What class is it?

[1] "numeric"

> class(a) <- "character" # Make it a character
> class(a)                # What class is it now?

[1] "character"

> a

[1] "1.5"

> a <- "Andrew"           # Wipe out the 1.5 and make it "Andrew" instead.
> b <- a                  # Create an object "b" and assign to it
>                         #    whatever is in object a.
> a <- c(1,2,3)           # Wipe out the "Andrew" and make it a vector
>                         #    with the values 1, 2, and 3.
>                         #    Never make c an object!
> b <- c(1:3)             # Wipe out the "Andrew" and make it a vector
>                         #    with the values 1, 2, and 3.
> b <- mean(a)            # Assign the mean of the object a to the object b.
> ls()                    # List all user-created objects

[1] "a" "b"
```

```
> rm(b)                    # Remove b
```

A couple of points are noteworthy: we didn't have to declare the variables as being any particular class. R coerced them into whatever was appropriate. Also we didn't have to declare the length of the vectors. That is convenient for the user.

6.2 Classes of Data

There are two fundamental kinds of data: numbers and characters (anything that is not a number is a character). There are several types of characters, each of which has unique properties. R distinguishes between these different types of object by their class.

R knows what these different classes are and what each is capable of. You can find out what the nature of any object is using the `class()` command. Alternatively, you can ask if it is a specific class using the `is.className()` command. You can often change the class too, using the `as.className()` command. This process can happen by default, and in that case is called *coercion*.

6.2.1 Numeric

A number. Could be a integer or a real number. R can generally tell the difference between them using context. We check by `is.numeric()` and change to by `as.numeric()`. R also handles complex numbers, but they're not important for this course. We can do all the usual things with numbers:

```
> a <- 2                   # create variable a, assign the number 2 to it.
> class(a)                 # what is it?
[1] "numeric"
> is.numeric(a)            # is it a number?
[1] TRUE
> b <- 4                   # create variable b, assign the number 4 to it.
> a + b                    # addition
[1] 6
> a - b                    # subtraction
[1] -2
> a * b                    # multiplication
[1] 8
> a / b                    # division
[1] 0.5
> a ^ b                    # exponentiation
[1] 16
> (a + b) ^ 3              # parentheses
[1] 216
> a == b                   # test of equality (returns a logical)
[1] FALSE
> a < b                    # comparison (returns a logical)
[1] TRUE
> max(a,b)                 # largest
[1] 4
> min(a,b)                 # smallest
[1] 2
```

6.2.2 Character

A collection of one or more alphanumerics, denoted by double quotes. We check whether or not our object is a character by `is.character()` and change to by `as.character()`. R provides numerous character manipulation functions, including search capabilities.

```
> a <- "string"         # create variable a, assign the value "string" to it.
> class(a)              # what is it?

[1] "character"

> is.numeric(a)         # is it a number?

[1] FALSE

> is.character(a)       # is it a character?

[1] TRUE

> b <- "spaghetti"      # create variable b, assign the value "spaghetti" to it.
> paste(a, b)           # join the characters

[1] "string spaghetti"

> paste(a, b, sep="")   # join the characters with no gap

[1] "stringspaghetti"

> d <- paste(a, b, sep="")
> substr(d, 1, 4)       # subset the character

[1] "stri"
```

6.2.3 Factor

Factors represent categorical variables.

In practice, factors are not terribly different than characters, except they can take only a limited number of values, which R keeps a record of, and R knows how to do very useful things with them. We check whether or not an object is a factor by `is.factor()` and change it to a factor, if possible, by `factor()`.

Even though R reports the results of operations upon factors by the levels that we assign to them, R represents factors internally as an integer. Therefore, factors can create considerable heartburn unless they're closely watched. This means: whenever you do an operation involving a factor you must make sure that it did what you wanted, by examining the output and intermediate steps. For example, factor levels are ordered alphabetically by default. This means that if your levels start with numbers, as many plot identifiers do, you might find that R thinks that plot 10 comes before plot 2. Not a problem, if you know about it!

```
> a <- c("A","B","A","B")   # create vector a
> class(a)                  # what is it?

[1] "character"

> is.character(a)           # is it a character?

[1] TRUE

> is.factor(a)              # is it a factor?

[1] FALSE

> a <- factor(a)            # make it so
> levels(a)                 # what are the levels?
```

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

> table(a)                  # what are the counts?

a
A B
2 2

> a <- factor(c("A","B","A","B"), levels=c("B","A"))
>                           # create a factor with different levels
```

Sometimes it will be necessary to work with a subset of the data, perhaps for convenience, or perhaps because some levels of a factor represent irrelevant data. If we subset the data then it will often be necessary to redefine any factors in the dataset, to let R know that it should drop the levels that are missing.

There will be much more on factors when we start manipulating vectors (Section 6.3.1).

6.2.4 Logical

A special kind of factor, that has only two levels: True and False. Logical variables are set apart from factors in that these levels are interchangeable with the numbers 1 and 0 (respectively) via coercion. The output of several useful functions are logical (also called boolean) variables. We can construct logical statements using the and (&), or (|), not (!) operators.

```
> a <- 2                # create variable a, assign the number 2 to it.
> b <- 4                # create variable b, assign the number 4 to it.
> d <- a < b            # comparison
> class(d)              # what is it?

[1] "logical"

> e <- TRUE             # create variable e, assign the value TRUE to it.
> d + e                 # what should this do?

[1] 2

> d & e                 # d AND e is True

[1] TRUE

> d | e                 # d OR e is also True

[1] TRUE

> d & !e                # d AND (NOT e) is not True

[1] FALSE
```

We can ask for the vector subscripts of all objects for which a condition is true via `which()`.

6.2.5 Missing Data

The last and oddest kind of data is called a missing value (NA). This is not a unique class, strictly speaking. They can be mixed in with all other kinds of data. It's easiest to think of them as place holders for data that should have been there, but for some reason, aren't. Unfortunately their treatment is not uniform in all the functions. Sometimes you have to tell the function to ignore them, and sometimes you don't. And, there are different ways of telling the function how to ignore them depending on who wrote the function and what its purpose is.

There are a few functions for the manipulation of missing values. We can detect missing values by `is.na()`.

```
> a <- NA               # assign NA to variable A
> is.na(a)              # is it missing?
```

```
[1] TRUE

> class(a)              # what is it?

[1] "logical"

> a <- c(11,NA,13)      # now try a vector
> mean(a)               # agh!

[1] NA

> mean(a, na.rm=TRUE)   # Phew! We've removed the missing value

[1] 12

> is.na(a)              # is it missing?

[1] FALSE  TRUE FALSE
```

We can identify the complete rows (*ie* rows that have no missing values) from a two-dimensional object via the `complete.cases()` command.

6.3 Structures for Data

Having looked at the most important data types, let's look at the mechanisms that we have for their collective storage and manipulation. There are more than we cover here - some of which (matrix, list) can be very useful.

6.3.1 Vector

A vector is a one-dimensional collection of atomic objects (atomic objects are objects which can't be broken down any further). Vectors can contain numbers, characters, factors, or logicals. All the objects that we created earlier were vectors, although some were of length 1. The key to vector construction is that all the objects must be of the same class. The key to vector manipulation is in using its subscripts. The subscripts are accessed by using the square brackets [ ].

```
> a <- c(11,12,13) # a is a vector
> a[1]             # the first object in a

[1] 11

> a[2]             # the second object in a

[1] 12

> a[-2]            # a, but without the second object

[1] 11 13

> a[c(2,3,1)]      # a, but in a different order

[1] 12 13 11

> a + 1            # Add 1 to all the elements of a

[1] 12 13 14

> length(a)        # the number of units in the vector a

[1] 3

> order(c(a,b))    # return the indices of a and b in increasing order
```

```
[1] 4 1 2 3

> c(a,b)[order(c(a,b))] # return a and b in increasing order

[1]  4 11 12 13

> a <- c(11,NA,13) # a is still a vector
> a[!is.na(a)]     # what are the elements of a that aren't missing?

[1] 11 13

> which(!is.na(a)) # what are the locations of the non-missing elements of a?

[1] 1 3
```

Notice that in `a[!is.na(a)]`, for example, we were able to nest a vector inside the subscript mechanism of another vector! This example also introduces a key facility in R for efficient processing: vectorization.

Vectorization

The concept underlying vectorization is simple: to make processing more efficient. Recall that in section 6.2.5, when we applied the `is.na()` function to the vector `a` it resulted in the function being applied to *each element* of the vector, and the output itself being a vector, without the user needing to intervene. This is vectorization.

Imagine that we have a set of 1,000,000 tree diameters and we need to convert them all to basal area. In C or Fortran we would write a loop. The R version of the loop would look like this (wrapped in a timer).

```
> diameters <- rgamma(n=1000000, shape=2, scale=20)
> basal.areas <- rep(NA, length(diameters))
> system.time(
+             for (i in 1:length(diameters)) {
+               basal.areas[i] <- diameters[i]^2 * pi / 40000
+             }
+ )

   user  system elapsed
  3.088   0.014   3.102
```

That took just over three seconds on my quite old computer. However, if we vectorize the operation, it becomes considerably faster.

```
> system.time(
+             basal.areas <- diameters^2 * pi / 40000
+             )

   user  system elapsed
  0.013   0.006   0.019
```

It's about 250 times faster. Of course, had we programmed this function in C or Fortran, the outcome would have been much faster still. The R programming mantra might be: compile only if you need to, loop only if you have to, and vectorize all the time.

Vectorization only works for some functions; *e.g.* it won't work for `mean()`, because that would make no sense; we want the mean of the numbers in the vector, not the mean of each individual unit. But, when vectorization works it makes life easier, code cleaner, and processing time faster.

Note that `pi` is a single value, and not of length 10^6, and R assumed that we would like it repeated 10^6 times. This is called recycling.

Recycling

You may have noticed above that R provided a convenience for the manipulation of vectors. When we typed

```
> a <- c(11,12,13)
> a + 1

[1] 12 13 14
```

R assumed that we wanted to add 1 to each element of `a`. This is called recycling, and is usually very useful and occasionally very dangerous. R recycled `1` until it had the same length as `a`. interpreted the function as:

```
> a + c(1, 1, 1)

[1] 12 13 14
```

For a further example, if we want to convert all the numbers in a vector from inches to centimetres, we simply do

```
> (a <- a * 2.54)

[1] 27.94 30.48 33.02
```

Recycling can be dangerous because sometimes we want the dimensions to match up exactly, and mismatches will lead to incorrect computations. If we fail to line up our results exactly - *e.g.* because of some missing values - R will go ahead and compute the result anyway. The only way to be sure is to watch for warnings, examine the output carefully, and keep in mind that most of the time this facility is really helpful.

```
> a + c(1, -1)

[1] 28.94 29.48 34.02
```

6.3.2 Dataframe

A dataframe is a powerful two-dimensional vector-holding structure. It is optimized for representing multidimensional datasets: each column corresponds to a variable and each row corresponds to an observation. A dataframe can hold vectors of any of the basic classes of objects at any given time. So, one column could be characters whilst another could be a factor, and a third be numeric.

We can still refer to the objects within the dataframe through their subscripts: using the square brackets. Now there are two dimensions: row, and column. If either is left blank, then the whole dimension is assumed. That is, `test[1:10,]` will grab the first ten rows of all the columns of dataframe test, using the above-noted expansion that the colon fills in the integers. `test[,c(2,5,4)]` will grab all the rows for only the second, fifth and fourth columns. These index selections can be nested or applied sequentially. Negative numbers in the index selections denote rows that will be omitted.

Each column, or variable, in a dataframe has a unique name. We can extract that variable by means of the dataframe name, the column name, and a dollar sign as: *dataframe$variable*.

A collection of columns can be selected by name by providing a vector of the column names in the index specification. This is useful in dataframes in which the locations of the columns is uncertain or dynamic, but the names are fixed.

If a comma-delimited file is read in using the commands in Section 5.1, R will assume that it is meant to be a dataframe. The command to check is `is.data.frame()`, and the command to change it is `as.data.frame()`. There are many functions to examine dataframes; we showcase some of them below.

```
> ufc <- read.csv("../data/ufc.csv")      # ufc is a dataframe
> is.data.frame(ufc)                      # we hope

[1] TRUE

> dim(ufc)                                # the size of the dimensions (r,c)
```

```
[1] 637   5

> names(ufc)                              # the labels of the columns

[1] "plot"    "tree"    "species" "dbh"     "height"

> ufc$height[1:5]                         # first 5 heights

[1]  NA 205 330  NA 300

> ufc$species[1:5]                        # first 5 species

[1]    DF WL WC GF
Levels:  DF ES F FG GF HW LP PP SF WC WL WP

> ufc[1:5, c(3,5)]                        # first 5 species and heights

  species height
1              NA
2      DF     205
3      WL     330
4      WC      NA
5      GF     300

> ufc[1:5, c("species","height")]         # first 5 species and heights again

  species height
1              NA
2      DF     205
3      WL     330
4      WC      NA
5      GF     300

> table(ufc$species)

     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

The last command suggests that there are ten trees with blank species. Here, it's vital to know about the data recording protocol. The ten trees without species are blank lines to represent empty plots. Let's remove them, for the moment. Note that we also redefine the species factor, so that it will drop the now empty level.

```
> ufc <- ufc[ufc$species != "",]
> ufc$species <- factor(ufc$species)
```

We can also create new variables within a dataframe, by naming them and assigning them a value. Thus,

```
> ufc$dbh.cm <- ufc$dbh/10            # Dbh now in cm
> ufc$height.m <- ufc$height/10       # Height now in metres
```

Finally, if we want to construct a dataframe from already existing variables, which is quite common, we use the `data.frame()` command, *viz*:

```
> temp <- data.frame(my.species=ufc$species,
+                    my.dbh=ufc$dbh.cm)
> temp[1:5,]

  my.species my.dbh
1         DF     39
2         WL     48
3         WC     15
4         GF     52
5         WC     31
```

Dataframes are the most useful data structures as far as we are concerned. We can use logical vectors that refer to one aspect of the dataframe to extract information from the rest of the dataframe, or even another dataframe. And, it's possible to extract pretty much any information using the tools that we've already seen.

```
> ufc$height.m[ufc$species=="LP"]          # Heights of lodgepole pine

[1] 24.5   NA   NA   NA 16.0 25.0   NA

> mean(ufc$height.m[ufc$species=="LP"], na.rm=TRUE)

[1] 21.83333

>                                          # Average height of lodgepole pine
```

Sapply

`sapply()` permits us to deploy a function upon each column in a dataframe. This is particularly useful if we want to find column sums or means, for example, but has other useful applications.

```
> sapply(ufc[,4:7], mean, na.rm=TRUE)

      dbh    height    dbh.cm  height.m
356.56619 240.66496  35.65662  24.06650
```

Below, for example, we use `sapply()` to tell us the class of each of the columns in our dataframe.

```
> sapply(ufc, class)

     plot      tree   species       dbh    height    dbh.cm  height.m
"integer" "integer"  "factor" "integer" "integer" "numeric" "numeric"
```

6.3.3 Matrix (Array)

A matrix is simply a vector that has extra attributes, called dimensions. R provides specific algorithms to enable us to treat the vector as though it were really two-dimensional. Many useful matrix operations are available.

```
> (mat.1 <- matrix(c(1,0,1,1), nrow=2))

     [,1] [,2]
[1,]    1    1
[2,]    0    1

> (mat.2 <- matrix(c(1,1,0,1), nrow=2))

     [,1] [,2]
[1,]    1    0
[2,]    1    1

> solve(mat.1)    # This inverts the matrix

     [,1] [,2]
[1,]    1   -1
[2,]    0    1

> mat.1 %*% mat.2 # Matrix multiplication

     [,1] [,2]
[1,]    2    1
[2,]    1    1

> mat.1 + mat.2   # Matrix addition

     [,1] [,2]
[1,]    2    1
[2,]    1    2

> t(mat.1)        # Matrix transposition

     [,1] [,2]
[1,]    1    0
[2,]    1    1

> det(mat.1)      # Matrix determinant

[1] 1
```

There are also various functions of matrices, for example, `qr()` produces the QR decomposition, `eigen()` produces the eigenvalues and eigenvectors of matrices, and `svd()` performs singular value decomposition.

Arrays are one, two, or three-dimensional matrices.

Apply

Apply permits us to deploy a function upon each row or column in a matrix or array. This is particularly useful if we want to find row or column sums or means, for example.

```
> apply(ufc[,4:7], 2, mean, na.rm=TRUE)

      dbh    height    dbh.cm  height.m
356.56619 240.66496  35.65662  24.06650
```

6.3.4 List

A list is a container for other objects. Lists are invaluable for, for example, collecting and storing complicated output of functions. Lists become invaluable devices as we become more comfortable with R, and start to think of different ways to solve our problems. We access the elements of a list using the double bracket, as below.

```
> (my.list <- list("one", TRUE, 3))

[[1]]
[1] "one"

[[2]]
[1] TRUE

[[3]]
[1] 3

> my.list[[2]]

[1] TRUE
```

If we use a single bracket then we extract the element, still wrapped in the list infrastructure.

```
> my.list[2]

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

We can also name the elements of the list, either during its construction or post-hoc.

```
> (my.list <- list(first = "one", second = TRUE, third = 3))

$first
[1] "one"

$second
[1] TRUE

$third
[1] 3

> names(my.list)

[1] "first"  "second" "third"

> my.list$second

[1] TRUE

> names(my.list) <- c("First element","Second element","Third element")
> my.list

$`First element`
[1] "one"

$`Second element`
[1] TRUE

$`Third element`
[1] 3

> my.list$`Second element`
```

```
[1] TRUE
```

Note the deployment of backticks to print the nominated element of the list, even though the name includes spaces.

The output of many functions is a list object. For example, when we fit a least squares regression, the regression object itself is a list, and can be manipulated using list operations.

Above, we saw the use of `tapply()`, which conveniently allowed us to apply an arbitrary function to all the elements of a grouped vector, group by group. We can use `lapply()` to apply an arbitrary function to all the elements of a list.

6.4 Merging Data

It is often necessary to merge datasets that contain data that occupy different hierarchical scales; for example, in forestry we might have some species-level parameters that we wish to merge with tree-level measures in order to make predictions from a known model. We tackled exactly this problem in writing the function that computes the bard-feet volume of a tree given its species, diameter and height. Let's see how this works on a smaller scale. First, we declare the dataframe that contains the species-level parameters.

Recall that by adding the parentheses we are asking R to print the result as well as saving it.

```
> (params <- data.frame(species = c("WP", "WL"),
+                       b0 = c(32.516, 85.150),
+                       b1 = c(0.01181, 0.00841)))

  species     b0      b1
1      WP 32.516 0.01181
2      WL 85.150 0.00841
```

Then let's grab the first three trees of either species from ufc that have non-missing heights. We'll only keep the relevant columns. Note that we can "stack" the index calls, and that they are evaluated sequentially from left to right.

```
> (trees <- ufc[ufc$species %in% params$species& !is.na(ufc$height.m),][1:3,])

   plot tree species dbh height dbh.cm height.m
3     2    2      WL 480    330   48.0       33
20    4    9      WP 299    240   29.9       24
26    5    6      WP 155    140   15.5       14
```

Now we merge the parameters with the species.

```
> (trees <- merge(trees, params))

  species plot tree dbh height dbh.cm height.m     b0      b1
1      WL    2    2 480    330   48.0       33 85.150 0.00841
2      WP    4    9 299    240   29.9       24 32.516 0.01181
3      WP    5    6 155    140   15.5       14 32.516 0.01181
```

There are many options for merging, for example, in how to deal with not-quite overlapping characteristics, or column names that do not match.

We can now compute the volume using a vectorized approach, which is substantially more efficient than using a loop. Also, note the use of `with()` to *temporarily* attach the dataframe to our search path. This usage simplifies the code.

```
> (trees$volume <-
+  with(trees, b0 + b1 * (dbh.cm/2.54)^2 * (height.m*3.281))*0.002359737)

[1] 0.9682839 0.3808219 0.1243991
```

6.5 Reshaping Data

Longitudinal datasets often come in the wrong shape for analysis. When longitudinal data are kept in a spreadsheet, it is convenient to devote a row to each object being measured, and include a column for each time point, so that adding new measurements merely requires adding a column. From the point of view of data analysis, however, it is easier to think of each observation on the object as being a row. R makes this switch convenient. Again, we'll see how this works on a small scale.

```
> (trees <- data.frame(tree = c(1, 2), species = c("WH", "WL"),
+                      dbh.1 = c(45, 52), dbh.2 = c(50, 55),
+                      ht.1 = c(30, 35), ht.2 = c(32, 36)))

  tree species dbh.1 dbh.2 ht.1 ht.2
1    1      WH    45    50   30   32
2    2      WL    52    55   35   36

> (trees.long <- reshape(trees,
+                        direction = "long",
+                        varying = list(c("dbh.1","dbh.2"),
+                                       c("ht.1","ht.2")),
+                        v.names = c("dbh","height"),
+                        timevar = "time",
+                        idvar = "tree"
+                        ))

    tree species time dbh height
1.1    1      WH    1  45     30
2.1    2      WL    1  52     35
1.2    1      WH    2  50     32
2.2    2      WL    2  55     36
```

The arguments are defined as follows:

direction tells R to go *wide* or go *long*,

varying is a list of vectors of column names that are to be stacked,

v.names is a vector of the new names for the stacked columns,

timevar is the name of the new column that differentiates between successive measurements on each object, and

idvar is the name of the existing column that differentiates between the objects.

6.6 Sorting Data

R provides the ability to order the elements of objects. Sorting is vital for resolving merging operations, for example, as we did in Chapter 3. Here are the top five trees by height.

```
> ufc[order(ufc$height.m, decreasing=TRUE),][1:5,]

    plot tree species  dbh height dbh.cm height.m
413   78    3      WP 1030    480  103.0     48.0
532  110    4      GF  812    470   81.2     47.0
457   88    3      WL  708    425   70.8     42.5
297   55    2      DF  998    420   99.8     42.0
378   68    1      GF  780    420   78.0     42.0
```

`order()` allows ordering by more than one vector, so, for example, to order by plot then species then height, we would use

```
> ufc[order(ufc$plot, ufc$species, ufc$height.m),][1:5,]
```

```
   plot tree species dbh height dbh.cm height.m
2     2    1      DF 390    205     39     20.5
3     2    2      WL 480    330     48     33.0
5     3    2      GF 520    300     52     30.0
8     3    5      WC 360    207     36     20.7
11    3    8      WC 380    225     38     22.5
```

Related functions include `sort()` and `rank()`, but they only permit the use of one index.

Chapter 7

Simple Descriptions

We have already seen a number of different data summary and analysis tools, but it is useful to consider them in a more coordinated and holistic fashion. We start by reading and processing the data according to our earlier strategy.

```
> ufc <- read.csv("../data/ufc.csv")
> ufc$height.m <- ufc$height/10
> ufc$dbh.cm <- ufc$dbh/10
> ufc$height.m[ufc$height.m < 0.1] <- NA
> ufc$species[ufc$species %in% c("F","FG")] <- "GF"
> ufc$species <- factor(ufc$species)
> ufc.baf <- 7
> cm.to.inches <- 1/2.54
> m.to.feet <- 3.281
> bd.ft.to.m3 <- 0.002359737
> ufc$g.ma2 <- ufc$dbh.cm^2*pi/40000
> ufc$tree.factor <- ufc.baf / ufc$g.ma2
> ufc$tree.factor[is.na(ufc$dbh.cm)] <- 0
```

Here, we will drop the plots that have no trees, although we will keep track of them in the plot count.

```
> number.of.plots <- length(unique(ufc$plot))
> ufc <- ufc[!is.na(ufc$dbh.cm), ]
> ufc$species <- factor(ufc$species)
```

7.1 Univariate

In what follows we distinguish between summaries of continuous data, also called numerical, and categorical data. Ordinal data can be considered a special case of the latter for our purposes. I don't see any particular reason to further subdivide continuous data into interval and ratio data.

7.1.1 Numerical

Getting standard estimates of the location and spread of variables is very easy; these are obtained by the following commands. Note the use of the argument `na.rm=TRUE`, which tells R to ignore the missing values.

Measures of location

```
> mean(ufc$dbh.cm, na.rm = TRUE)

[1] 35.65662

> median(ufc$dbh.cm, na.rm = TRUE)

[1] 32.9
```

Measures of spread

```
> sd(ufc$dbh.cm, na.rm = TRUE)

[1] 17.68945

> range(ufc$dbh.cm, na.rm = TRUE)

[1]  10 112

> IQR(ufc$dbh.cm, na.rm = TRUE)

[1] 23.45
```

Measures of skew

A `skewness()` function can be found in the `moments` package. Alternatively we can compare the data with a known non-skewed dataset; the normal distribution (see Figure 7.1). These data are, not surprisingly, positively skewed (i.e. they have a long right tail).

```
> qqnorm(ufc$dbh.cm, xlab = "Diameter (cm)")
> qqline(ufc$dbh.cm, col = "darkgrey")
```

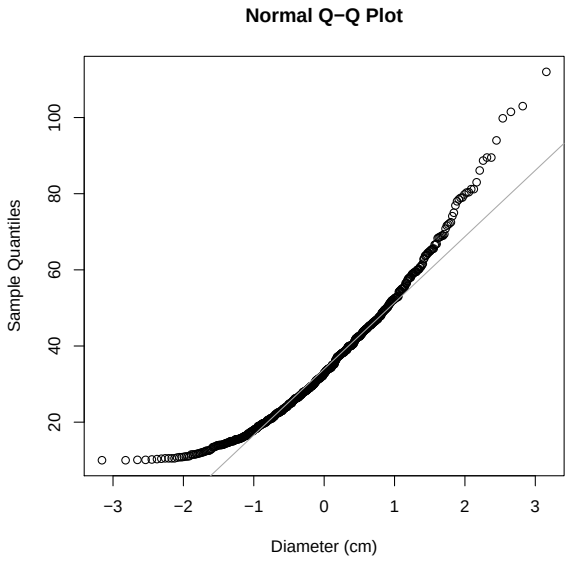

Figure 7.1: Demonstration of normal quantile plot to assess skew. These data are, not surprisingly, positively skewed (i.e. they have a long right tail).

7.1.2 Categorical

The most common data summary for categorical variables is tabulation. There are several ways that data can be tabulated in R. Their difference is in how flexible the commands are and the kind of output generated.

```
> table(ufc$species)

 DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 77   3 188   5   7   4  14 251  34  44

> tapply(ufc$species, ufc$species, length)
```

```
 DF  ES  GF  HW  LP  PP  SF  WC  WL  WP
 77   3 188   5   7   4  14 251  34  44
```

```
> aggregate(x=list(count=ufc$species),
+           by=list(species=ufc$species),
+           FUN = length)
```

```
   species count
1       DF    77
2       ES     3
3       GF   188
4       HW     5
5       LP     7
6       PP     4
7       SF    14
8       WC   251
9       WL    34
10      WP    44
```

These tables can be easily converted to figures (see Figure 7.2).

```
> plot(table(ufc$species), ylab = "Raw Stem Count")
> plot(as.table(tapply(ufc$tree.factor/number.of.plots, ufc$species,
+     sum)), ylab = "Stems per Hectare")
```

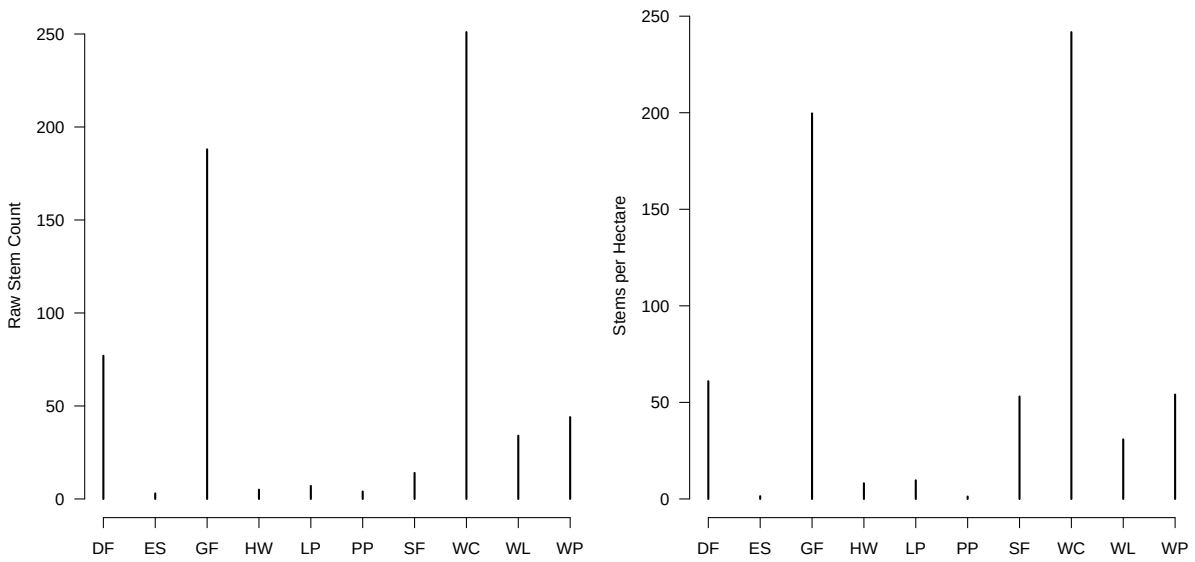

Figure 7.2: Raw count of trees by species (left panel) and weighted by tree factor (right panel).

Continuous variates can also be converted into ordinal variates for further analysis using the `cut()` function. For example, here we cut the tree diameters into 20 cm classes, making a new factor that has ordered levels.

```
> ufc$dbh.20 <- cut(ufc$dbh.cm, breaks = (0:6) * 20)
> table(ufc$dbh.20)
```

```
  (0,20]  (20,40]  (40,60]  (60,80] (80,100] (100,120]
     128       279       163        43        11         3
```

7.2 Multivariate

We will consider three cases that offer bivariate information; when there is conditioning on a categorical variable (e.g. species), and also when there is more than one variable of interest, be they numerical or categorical.

7.2.1 Numerical/Numerical

We have already seen a scatterplot of two continuous variates, and we are not interested in formally fitting models at this point. So, the correlation is a quick and useful summary of the level of agreement between two continuous variates. Note that the `na.rm` argument that we have become accustomed to is not used in this function, instead we use the `use` argument. See `?cor` for more details.

```
> cor(ufc$dbh.cm, ufc$height.m, use = "complete.obs")

[1] 0.7794116
```

7.2.2 Numerical/Categorical

We will most commonly be interested in obtaining summaries of the numerical variable conditioned upon the categorical variable. We have seen how to do this in several ways. The main difference between the `tapply()` and `aggregate()` functions is in the structure of the output. `tapply()` creates a named list, and `aggregate()` creates a dataframe with the labels as a separate column. Also, `aggregate()` will compute the nominated function for more than one variable of interest.

```
> tapply(ufc$dbh.cm, ufc$species, mean, na.rm=TRUE)

      DF       ES       GF       HW       LP       PP       SF       WC
38.37143 40.33333 35.20106 20.90000 23.28571 56.85000 13.64286 37.50757
      WL       WP
34.00588 31.97273

> aggregate(x = list(dbh.cm=ufc$dbh.cm, height.m=ufc$height.m),
+           by = list(species=ufc$species),
+           FUN = mean, na.rm = TRUE)

   species   dbh.cm height.m
1       DF 38.37143 25.30000
2       ES 40.33333 28.00000
3       GF 35.20106 24.34322
4       HW 20.90000 19.80000
5       LP 23.28571 21.83333
6       PP 56.85000 33.00000
7       SF 13.64286 15.41000
8       WC 37.50757 23.48777
9       WL 34.00588 25.47273
10      WP 31.97273 25.92500
```

7.2.3 Categorical/Categorical

Numerical and graphical cross-tabulations are useful for summarizing the relationship between more than one categorical variable.

```
> table(ufc$species, ufc$dbh.20)

    (0,20] (20,40] (40,60] (60,80] (80,100] (100,120]
  DF     10      33      27       5        2         0
  ES      0       2       1       0        0         0
  GF     37      88      47      12        4         0
  HW      3       2       0       0        0         0
  LP      2       5       0       0        0         0
```

PP	0	2	0	1	1	0
SF	14	0	0	0	0	0
WC	45	107	71	23	4	1
WL	4	19	10	1	0	0
WP	13	21	7	1	0	2

This table can be converted to a plot by calling the `plot()` function using the table function as the sole argument. Here we use the code to sum the tree factors to get unbiased estimates. Note that we need to replace the missing values with zeros.

```
> species.by.dbh <- as.table(tapply(ufc$tree.factor/number.of.plots,
+     list(ufc$dbh.20, ufc$species), sum))
> species.by.dbh[is.na(species.by.dbh)] <- 0
> plot(species.by.dbh, main = "Species by Diameter Class")
```

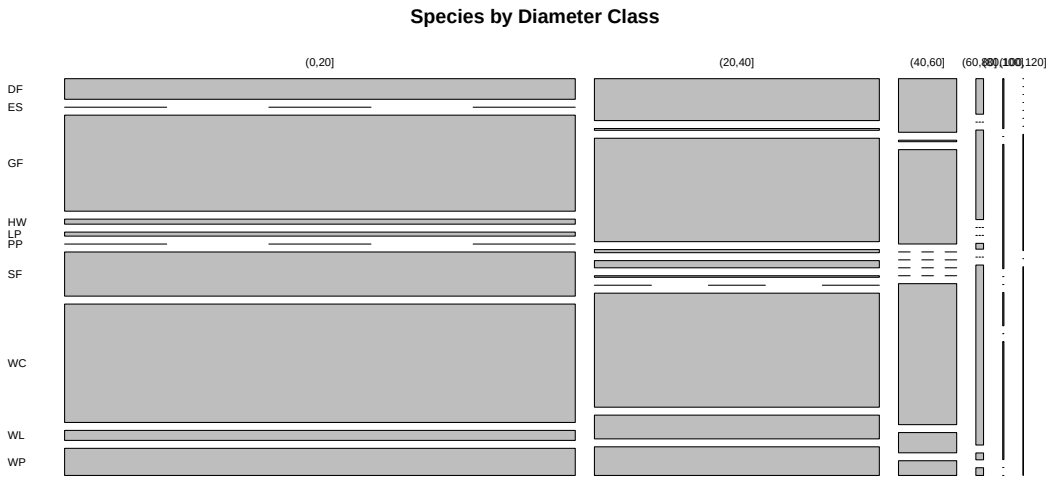

Figure 7.3: Species stems per hectare by 20 cm diameter class.

Chapter 8

Graphics

One major selling point for R is that it has better graphics-producing capabilities than many of the commercial alternatives. Whether you want quick graphs that help you understand the structure of your data, or publication-quality graphics that accurately communicate your message to your readers, R will suffice[1].

The graphics are controlled by scripts, and start at a very simple level. If you have `dbh.cm` and `height.m` in your ufc dataframe[2], for example

```
> plot(ufc$dbh.cm, ufc$height.m)
```

will open a graphical window and draw a scatterplot of dbh against height for the Upper Flat Creek data, labeling the axes appropriately. A small addition will provide more informative labels (see Figure 8.1). Also note that there is more than one way of calling a plot, sometimes the following construction may be more convenient.

```
> plot(height.m ~ dbh.cm, data=ufc, xlab = "Diameter (cm)", ylab = "Height (m)")
```

The `plot()` command offers a wide variety of options for customizing the graphic. Each of the following arguments can be used within the `plot()` statement, singly or together, separated by commas. That is, the plot statement will look like

```
> plot(x, y, xlab="An axis label", ... )
```

- `xlim=c(`a`,`b`)` will set the lower and upper limits of the x-axis to be a and b respectively. Note that we have to know a and b to make this work!
- `ylim=c(`a`,`b`)` will set the lower and upper limits of the y-axis to be a and b respectively. Note that we have to know a and b to make this work!
- `xlab="X axis label goes in here"`
- `ylab="Y axis label goes in here"`
- `main="Plot title goes in here"`
- `col="red"` makes all the points red. This is specially attractive for overlaid plots.

8.1 Organization Parameters

From here we have great flexibility in terms of symbol choice, color, size, axis labeling, over-laying, etc. We'll showcase a few of the graphical capabilities in due course. These options can be further studied through `?par`.

The cleanest implementation is to open a new set of parameters, create the graph, and restore the original state, by means of the following simple but rather odd commands:

[1] My brother is in marketing.

[2] You can check this by `str()`. If they are absent, the code to compute and add them is on page 14.

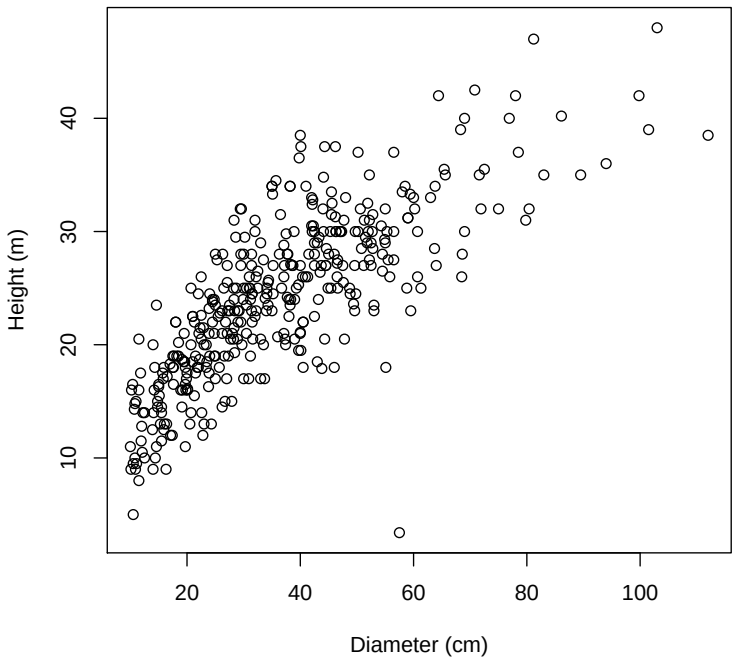

Figure 8.1: Diameter/Height plot for all species of Upper Flat Creek inventory data. Each point represents a tree.

```
> opar <- par( {parameter instructions go here, separated by commas}  )
> plot( {plot instructions go here} )
> par(opar)
```

The reason that these work is that when the `par()` function is used to set a new parameter, it invisibly returns the previous value of that parameter. So, the code

```
> opar <- par(las = 1)
```

changes the `las` parameter to 1 and store `las = 0` in `opar`. Subsequent invocation of

```
> par(opar)
```

changes the `las` parameter to 0 again.

There are several options for affecting the layout of the graphs on the page. These can all be used in conjunction with one another. There are many more than I note below, but these are the ones that I end up using the most often. My most common par call is something like:

```
> opar <- par(mfrow=c(3,2), mar=c(4,4,1,1), las=1)
> plot( ...
> par(opar)
```

- `par(mfrow=c(a,b))` where a and b are integers will create a matrix of plots on one page, with a rows and b columns.
- `par(mar=c(s,w,n,e))` will create a space of characters around the inner margin of the plot(s).
- `par(oma=c(s,w,n,e))` will create a space of characters around the outer margin of the plot(s).
- `par(las=1)` rotates the y-axis labels to be horizontal rather than vertical.
- `par(pty="s")` forces the plot shape to be square. The alternative is that the plot shape is mutable, which is the default, and is `pty="m"`.

- `par(new=TRUE)` when inserted between plots will plot the next one on the same place as the previous, effecting an overlay. It will not match the axes unless forced to do so. Increasing the `mar` parameters for the second plot will force it to be printed *inside* the first plot.
- Various character expansion factors may also be set.

You can also interrogate your last graph for details. For example,

```
> par("usr")

[1]   5.920 116.080   1.616  49.784
```

gives you the realized axis limits (compare with Figure 8.1).

8.2 Graphical Augmentation

A traditional plot, once created, can be augmented using any of a number of different tools.

The infrastructure of the plot can be altered. For example, axes may be omitted in the initial plot call (`axes = FALSE`) and added afterwards using the `axis()` function, which provides greater control and flexibility over the locations and format of the tickmarks, axis labels, and also the content. Note that `axis()` only provides axis information within the portion of the range of the data, to reduce unnecessary plot clutter. The usual plot frame can be added using the `box()` function. Text can be located in the margins of the plot, to label certain areas or to augment axis labels, using the `mtext()` function. A legend can be added using the `legend()` function, which includes a very useful legend location routine, as shown below. Additions can also be made to the content of the plot, using the `points()`, `lines()`, and `abline()` functions. A number of these different steps are shown in figure 8.2.

1. Start by creating the plot object, which sets up the dimensions of the space, but omit any plot objects for the moment.

```
> par(las = 1, mar=c(4,4,3,2))
> plot(ufc$dbh.cm, ufc$height.m, axes=FALSE, xlab="", ylab="", type="n")
```

2. Next, we add the points. Let's use different colours for different trees.

```
> points(ufc$dbh.cm, ufc$height.m, col="darkseagreen4")
> points(ufc$dbh.cm[ufc$height.m < 5.0],
+        ufc$height.m[ufc$height.m < 5.0], col="red")
```

3. Add axes. These are the simplest possible calls, we have much greater flexibility than shown here.

```
> axis(1)
> axis(2)
```

4. Add axis labels using margin text (switching back to vertical for the y-axis).

```
> par(las=0)
> mtext("Diameter (cm)", side=1, line=3, col="blue")
> mtext("Height (m)", side=2, line=3, col="blue")
```

5. Wrap the plot in the traditional frame.

```
> box()
```

6. Finally, add a legend.

```
> legend("bottomright",
+        c("A normal tree", "A weird tree"),
+        col=c("darkseagreen3","red"),
+        pch=c(1,1),
+        bty="n")
```

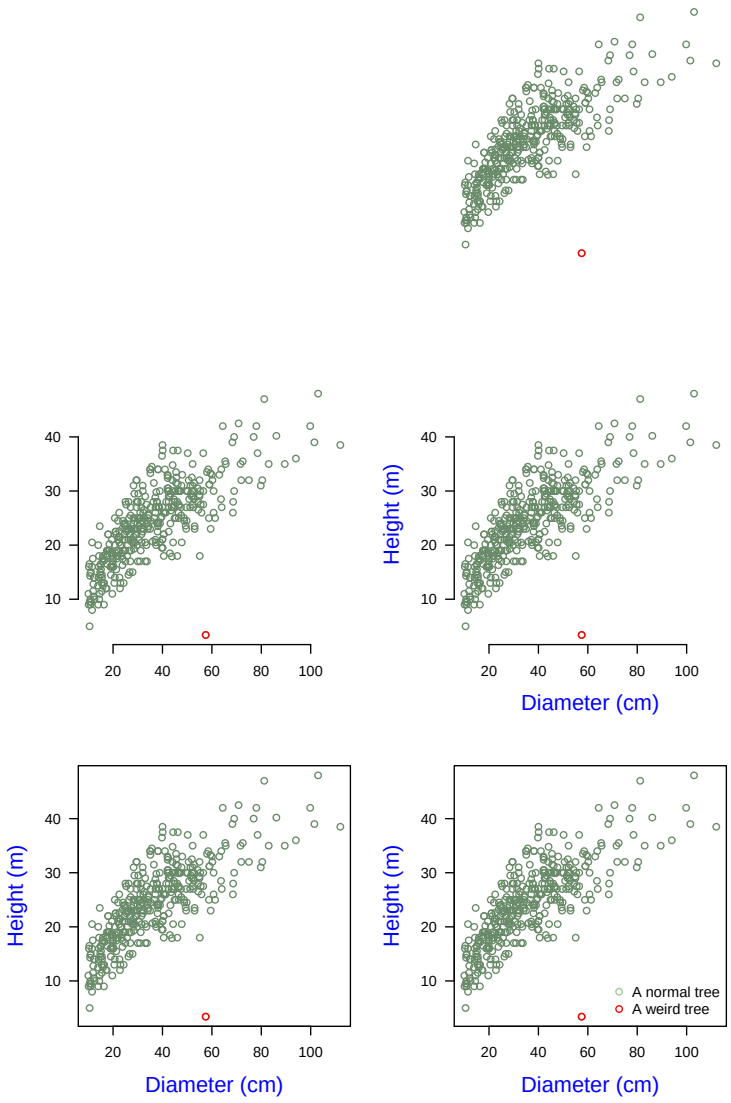

Figure 8.2: Building a plot up by components.

8.3 Permanence

Producing more permanent graphics is just as simple. For example, to create the graphic as a pdf file, which can be imported into various documents, we do the following:

```
> pdf(file="../graphics/graphic.pdf")
> plot(ufc$dbh.cm, ufc$height.m)
> abline(lm(height.m ~ dbh.cm, data=ufc), col="red")
> dev.off()
```

This will place the pdf in your graphics directory. This is an especially good option if the graphics that you want to produce would ordinarily cover more than one page, for example, if you are producing graphs in a loop. The pdf format is well accepted on the Internet as well, so the graphics are portable. Encapsulated postscript is also supported.

Under Windows, you can also copy directly from the plot window to the clipboard as either a metafile or a bmp (bitmap) image. Either can then be pasted directly into a Word document, for example. Alternatively using a similar approach to that noted above, one can create a JPEG image which can be imported into Word documents. My experience has been that the vividness of the colour suffers using JPEG, and some ameliorative action might be required.

8.4 Upgrades

The joy of advanced graphics in R is that all the good images can be made using the tools that we've already discussed. And, because the interface is scripted, it's very easy to take graphics that were created for one purpose and evolve them for another. Looping and judicious choice of control parameters can create informative and attractive output. For example, Figure 8.3 is constructed by the following code.

```
> opar <- par(oma=c(0,0,0,0), mar=c(0,0,0,0))
> x1 <- rep(1:10, 10)
> x2 <- rep(1:10, each=10)
> x3 <- 1:100
> interesting.colour.numbers <- c(1:152,253:259,362:657)
> plot.us <- sample(interesting.colour.numbers, size=max(x3))
> plot(x1, x2, col=colors()[plot.us], pch=20, cex=10, axes=F,
+      ylim=c(0,10), xlim=c(0, 10.5))
> text(x1, x2-0.5, colors()[plot.us], cex=0.3)
> text(x1+0.4, x2-0.4, plot.us, cex=0.5)
> par(opar)
```

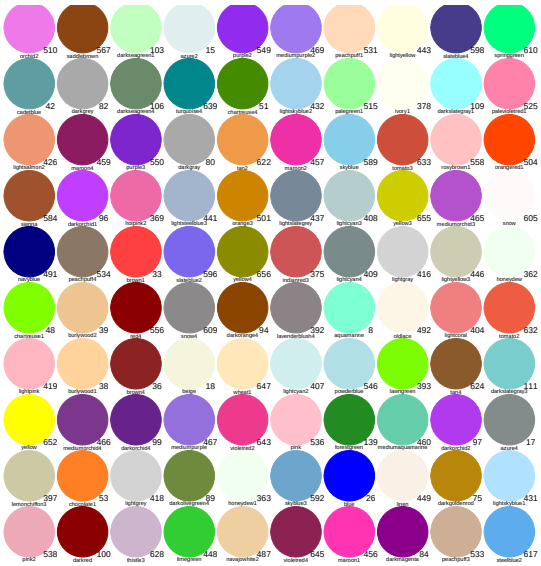

Figure 8.3: A random plot of coloured dots.

8.5 Common Challenges

This section is devoted to showing how to solve common graphical challenges using R.

8.5.1 Error Bars

Often we are asked to provide graphs that have error bars. We can use the `arrows()` function to provide the bars, and vectorization will keep the code simple for us. Figure 8.4 shows the mean and a 95% confidence interval (unweighted!) for the tree diameters by species.

```
> means <- tapply(ufc$dbh.cm, ufc$species, mean, na.rm=TRUE)
> ses <- tapply(ufc$dbh.cm, ufc$species, sd, na.rm=TRUE) /
+   sqrt(tapply(!is.na(ufc$dbh.cm), ufc$species, sum))
> ylim <- range(c(means + 1.96*ses, means - 1.96*ses))
> par(las=1, mar=c(4,4,2,2))
> plot(1:10, means, axes=FALSE, ylim=ylim, xlab="Species", ylab="Dbh (cm)")
```

```
> axis(1, at=1:10, labels=levels(ufc$species))
> axis(2)
> arrows(1:10, means-1.96*ses, 1:10, means+1.96*ses,
+        col="darkgrey", angle=90, code=3, length=0.1)
```

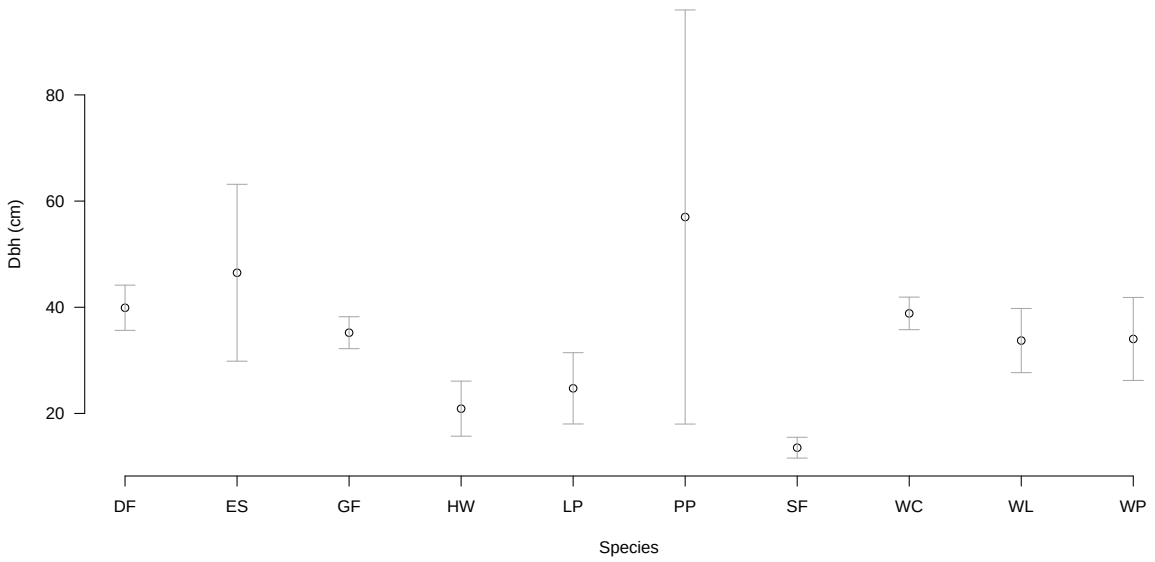

Figure 8.4: Means and 95% confidence intervals for the tree diameters, by species.

8.5.2 Colour graphics by groups

Integrating useful colour in another common challenge. There exist packages that contain functions to do this quickly and easily (`lattice`, see Section 8.6.1, and `car`, to name just two) but it is worth seeing how to do this with the tools available.

Using all the available species will make quite a mess, so let's constrain ourselves to trees from the four most common. The following code identifies the four most common species and constructs a smaller dataframe that contains trees only of those species.

```
> top.four <- levels(ufc$species)[order(table(ufc$species),
+                                       decreasing=TRUE)][1:4]
> ufc.small <- ufc[ufc$species %in% top.four,]
> ufc.small$species <- factor(ufc.small$species)
```

Now we can construct the plot using that smaller dataframe as follows (see Figure 8.5).

```
> my.colours <- c("red","blue","goldenrod","darkseagreen4")
> par(las=1, mar=c(4,4,3,2))
> plot(ufc.small$dbh.cm, ufc.small$height.m,
+      xlab = "Dbh (cm)", ylab = "Height (m)",
+      col = my.colours[ufc.small$species])
> legend("bottomright",
+        legend = levels(ufc.small$species),
+        col = my.colours,
+        lty = rep(NULL, 4),
+        pch = rep(19, 4),
+        bty = "n")
```

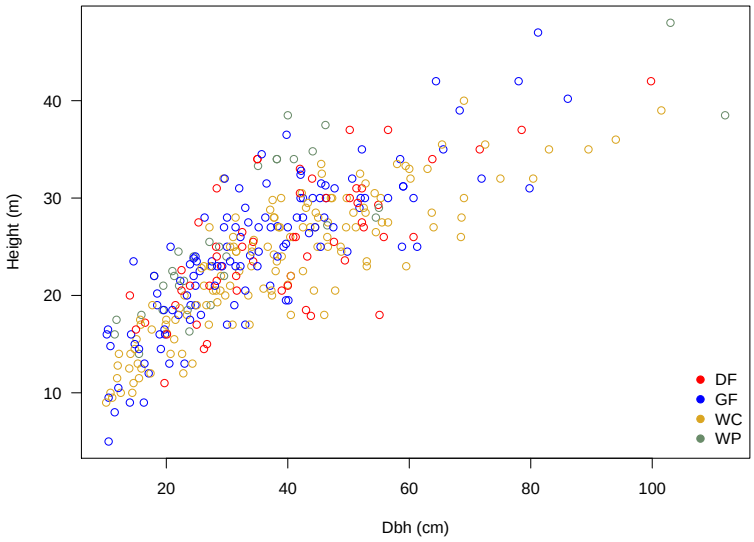

Figure 8.5: A plot of height against diameter for trees of the four most common species, coloured by species.

8.6 Contributions

Contributers to R have written packages that ease the construction of graphics. We will explore some of them here. Documentation of these facilities is challenging because the code is under continuous review and development.

8.6.1 Trellis

Trellis is a more formal tool for graphical virtuosity. Trellis allows great flexibility for producing conditioning plots. The R implementation of trellis is called *lattice*, and is written mainly by Deepayan Sankar.

We load the `lattice` package by means of the `library` function, which is explained in greater detail in Appendix A.

```
> library(lattice)
```

The goal of `lattice` is to simplify the production of high-quality and simple graphics that permit multi-dimensional data summaries.

We have already seen the use of these graphics in the Showcase chapter (Chapter 3), where we plotted height against diameter for each species, as well as diameter distributions for each species. We can get other useful graphics with a minimum of fuss, for example, focusing on the four most prolific species in the ufc data (Figure 8.6).

```
> ufc <- read.csv("../data/ufc.csv")      # ufc is a dataframe
> ufc$dbh.cm <- ufc$dbh/10                # Dbh now in cm
> ufc$height.m <- ufc$height/10           # Height now in metres
> ufc$height.m[ufc$height.m < 0.1] <- NA
> ufc <- ufc[complete.cases(ufc),]
> ufc$species[ufc$species %in% c("F","FG")] <- "GF"
> ufc$species <- factor(ufc$species)
> top.four <- levels(ufc$species)[order(table(ufc$species),
+                                       decreasing=TRUE)][1:4]
```

The panels in the following list refer directly to Figure 8.6 and do not necessarily correspond in position to what you will see on your screen.

- Top left panel:

```
> densityplot(~ dbh.cm | species, data=ufc,
+                   subset=species %in% top.four)
```

- Top right panel:

```
> bwplot(dbh.cm ~ species, data=ufc, subset=species %in% top.four)
```

- Bottom left panel:

```
> histogram( ~ dbh.cm | species, data=ufc,
+                 subset=species %in% top.four)
```

- Bottom right panel:

  (The object that we are loading was created in the Showcase chapter.)

```
> load("../images/ufc_plot.RData")

> contourplot(vol_m3_ha ~ east * north,
+             main = expression(paste("Volume (", m^3, "/ha)", sep="")),
+             xlab = "East (m)", ylab="North (m)",
+             region = TRUE,
+             col.regions = terrain.colors(11)[11:1],
+             data=ufc_plot)
```

Graphics produced by `lattice` are highly customizable. For example, if we were to wish to plot the height against the predicted height for the six species with the largest number of trees, using the mixed-effects imputation model from Chapter 3, and add some illustrative lines to the graphs, then we could do it with the following code. First we fit a suitable model and extract the predictions, to be used as imputations.

```
> library(nlme)
> hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) * species,
+               random = ~ I(log(dbh.cm)) | plot,
+               data = ufc)
> ufc$p.height.m <- exp(predict(hd.lme, level=1))
> top.six <- levels(ufc$species)[order(table(ufc$species),
+                                      decreasing=TRUE)][1:6]
> library(MASS)
```

Then we construct the plot, adding a new function that will be executed inside each panel. This version produces a scatterplot of the points, adds a blue 1:1 line, then a red linear regression line, fitted to the data inside the panel, a purple robust linear regression line, again fitted to the data inside the panel, and finally a black loess smooth.

```
> xyplot(height.m ~ p.height.m | species, data = ufc,
+        xlab="Predicted Height (m)",
+        ylab="Measured Height (m)",
+        panel = function(x, y) {
+          panel.xyplot(x, y)
+          panel.abline(0, 1, col="blue", lty=2)
+          panel.abline(lm(y ~ x), col="red")
+          panel.abline(rlm(y ~ x), col="purple")
+          panel.loess(x, y, col="black")
+        },
+        subset=species %in% top.six
+        )
```

If you are worried about the western larch panel (WL), try the same exercise using linear regression instead of the mixed-effects imputation model! That would require the following change to the predicted values:

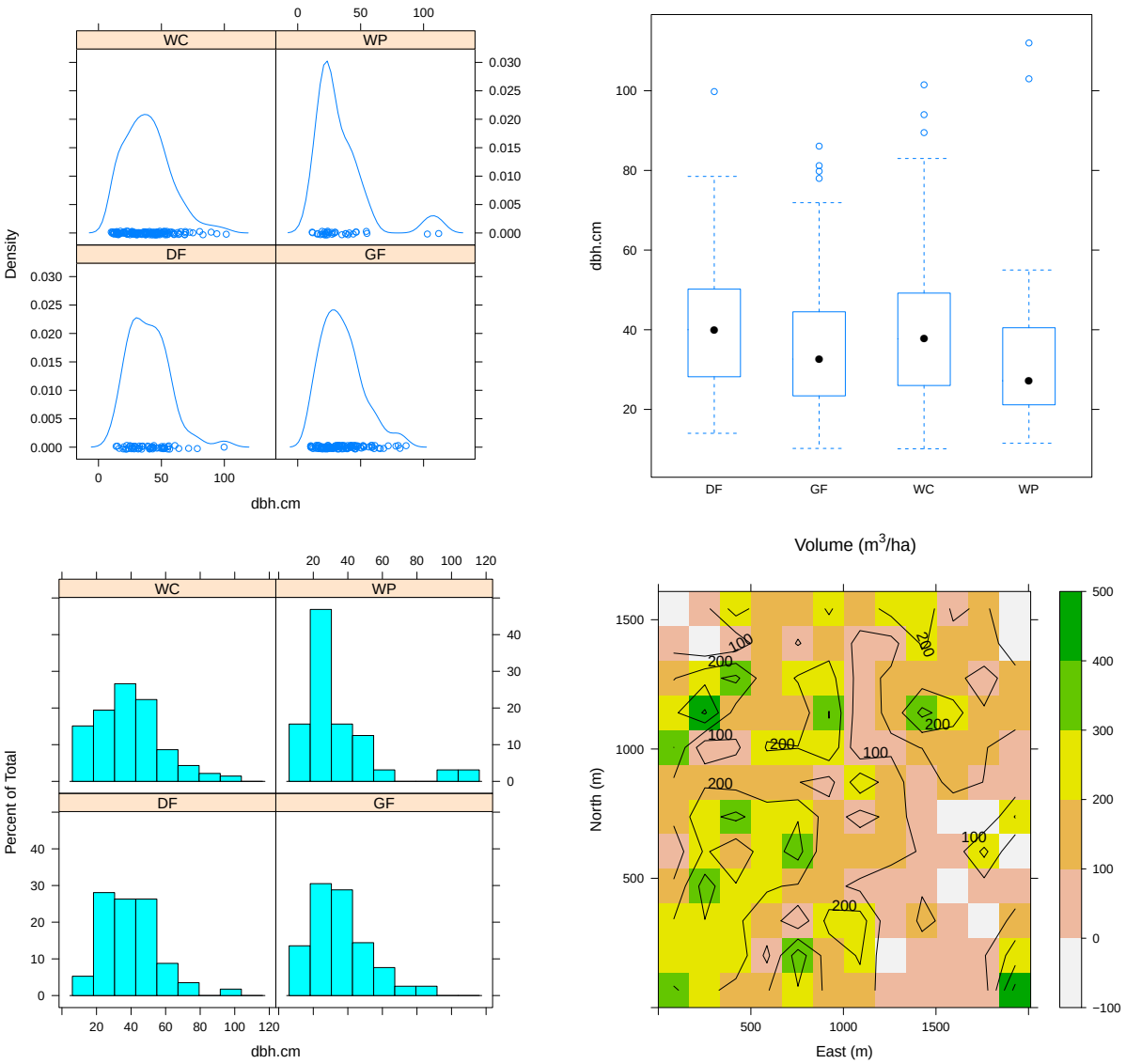

Figure 8.6: Example lattice plots, showing diameter information conditioned on species. The top left panel shows the empirical density curves of the diameters, the top right panel shows box plots, the bottom left shows histograms, and the bottom right shows.

```
> hd.lm <- lm(I(log(height.m)) ~ I(log(dbh.cm)) * species,
+             data = ufc)
> ufc$p.height.m <- exp(predict(hd.lm))
```

We can change the order of the panels using the `index.cond` option. We can also add other commands to each panel. `?xyplot` is very helpful to us here, providing very detailed information about the various options, and some attractive examples that we can adapt to our own uses.

Even though the code is quite mature, it is still under development to ease usage. For example, in up-to-date versions of lattice, to obtain a trellis plot that has points, regression lines of best fit, and smoothers superimposed, we need merely the following code:

```
> xyplot(height.m ~ p.height.m | species, data = ufc,
+        xlab = "Predicted Height (m)",
+        ylab = "Measured Height (m)",
+        type = c("p", "r", "smooth"),
+        subset = species %in% top.six
+ )
```

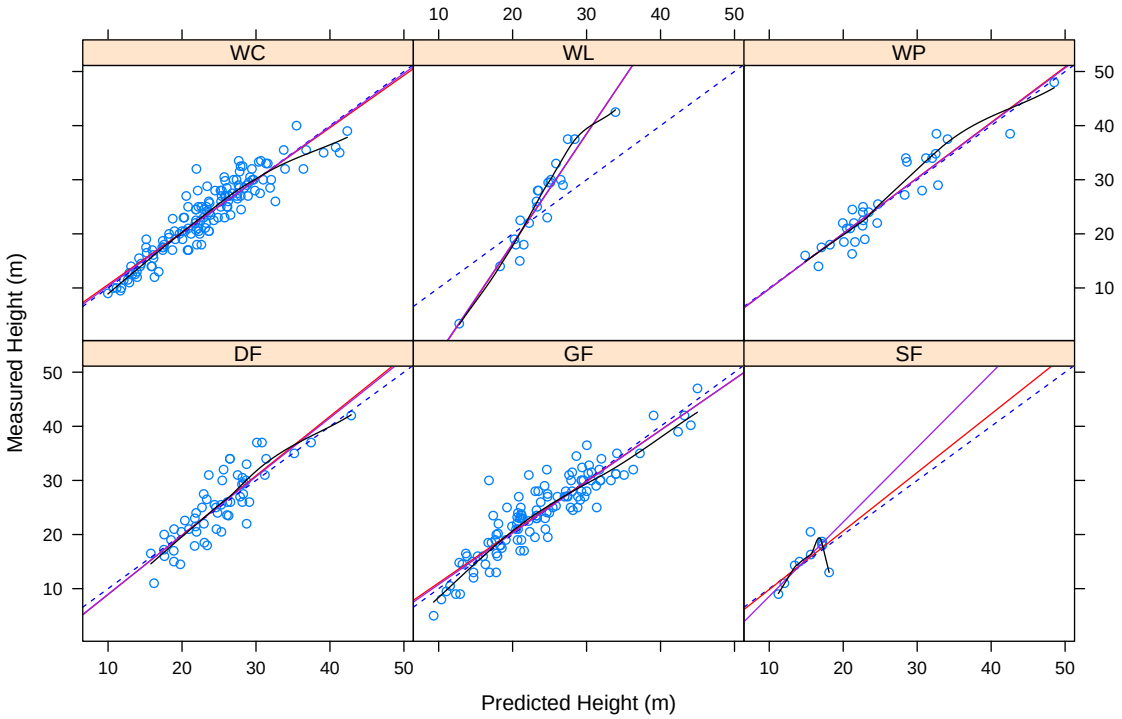

Figure 8.7: A lattice plot of height against predicted height by species for the six species that have the most trees. The blue dashed line is 1:1, the red solid line is the panel-level regression, the purple solid line is a robust regression fit and the black curved line is a loess smooth.

8.6.2 Grammar of Graphics

The key reference is now in its second edition (Wilkinson, 2005). Hadley Wickham has developed tools within R to realize some of the Grammar principles in an easily-usable format. The code is in version 2, and the original version is being phased out ("deprecated") hence we use the `ggplot2` package. This package is still under heavy development, but it also looks like a promising contribution to simplify statistical communication.

```
> library(ggplot2)
```

The differences between lattice and ggplot2 are mostly cosmetic. Publcation-quality graphics can be easily constructed using each of them, with patience. I find that ggplot2's ability to seamlessly add data from a range of sources greatly eases the challenge of juxtaposing model predictions and observations, but as we saw in the previous section, lattice has a somewhat more straightforward interface.

Getting Started

We trace some very simple developments in Figure 8.8.

1. Start by creating the plot space, and add points. Below, we identify the object to use, then pick out the so-called aesthetic elements of the object, and then tell R how to display them.

   ```
   > ggplot(ufc, aes(y = height.m, x = dbh.cm)) + geom_point()
   ```

2. Now we add a smooth line and (automatically) standard error spaces. The standard error polygon is attractive but its value is questionable, because it is point-wise, not curve-wise. So, it looks as though it tells us something really useful, but sadly, it does not.

   ```
   > ggplot(ufc, aes(y = height.m, x = dbh.cm)) + geom_point() +
   +       stat_smooth(fill = "blue", alpha = 0.2,
   +                   colour = "darkblue", size = 2)
   ```

Note that the `geom_point` and `stat_smooth` functions both know to take the x and y values from the `aes` portion of the `ggplot` function call.

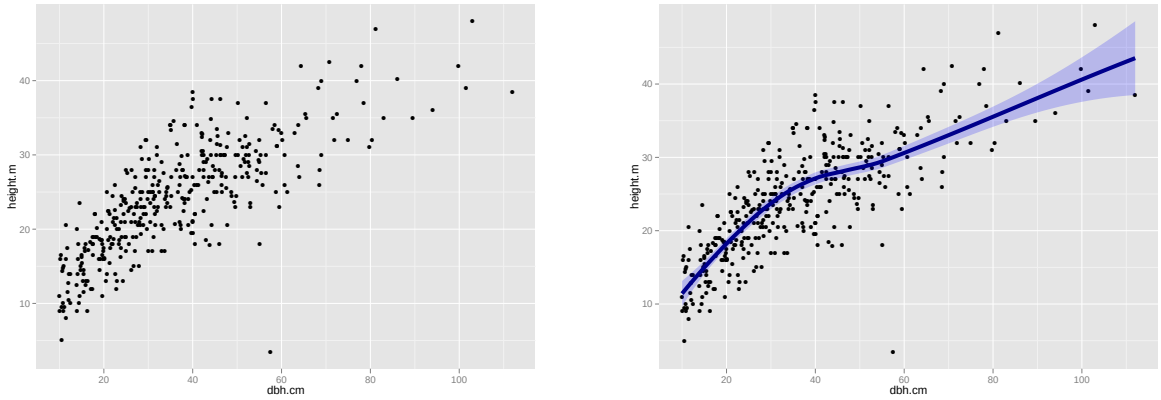

Figure 8.8: Building a plot up by ggplot2 components.

`qplot`: all-in-one

Building graphics interactively is useful but I tend not to do it, because my default approach to R coding is to write blocks of code and then execute them. Happily, ggplot2 provides the qplot function, which fits my needs nicely. `qplot` provides a convenient default system and neatens the interface (to my mind, in any case). It has the advantage of not bothering the user with the identification of aesthetics, at least in the first instance. See Figure 12.2 for an example of its usage.

Chapter 9

Linear Regression

This chapter focuses on the R tools that can be used for fitting ordinary linear regression, in various guises. I do not present the theory here as that has been covered much more effectively elsewhere.

9.1 Preparation

The data are forest inventory data from the Upper Flat Creek area of University of Idaho Experimental Forest. They were collected as a systematic random sample of variable radius plots. We have measures of diameter at breast height, 1.37 m, on all the sample trees and measures of height on a subset. Our immediate goal is to construct a relationship between height and diameter to allow us to predict the former from the latter. For the moment, we will ignore the mechanism by which trees were selected for each phase of measurement, and assume that they were selected equal probability. Some plots had no measured trees; to ensure that they are correctly accounted for an "empty" tree has been used to represent them.

Read in the data, having examined it within a spreadsheet.

```
> rm(list = ls())
> ufc <- read.csv("../data/ufc.csv")
```

How big is it?

```
> dim(ufc)
```

```
[1] 637   5
```

What are the variable names?

```
> names(ufc)
```

```
[1] "plot"    "tree"    "species" "dbh"     "height"
```

Let's take a snapshot - I usually eyeball the first 100-200 observations and choose a few sets of 100 at random. E.g. `ufc[1:100,]`, and `ufc[201:300,]`. These square brackets are fabulously useful, permitting great flexibility in data manipulation in a very simple format.

```
> ufc[1:10, ]
```

```
   plot tree species dbh height
1     1    1          NA     NA
2     2    1      DF 390    205
3     2    2      WL 480    330
4     3    1      WC 150     NA
5     3    2      GF 520    300
6     3    3      WC 310     NA
7     3    4      WC 280     NA
8     3    5      WC 360    207
9     3    6      WC 340     NA
10    3    7      WC 260     NA
```

Let's do some unit conversion: diameter at 1.37 m converted to cm and height to meters.

```
> ufc$dbh.cm <- ufc$dbh/10
> ufc$height.m <- ufc$height/10
```

Now we'll count the trees in each species, a few different ways

```
> table(ufc$species)
```

```
     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

```
> tapply(ufc$dbh.cm, ufc$species, length)
```

```
     DF  ES   F  FG  GF  HW  LP  PP  SF  WC  WL  WP
 10  77   3   1   2 185   5   7   4  14 251  34  44
```

```
> aggregate(x = list(num.trees = ufc$dbh.cm), by = list(species = ufc$species),
+     FUN = length)
```

```
   species num.trees
1                 10
2       DF        77
3       ES         3
4        F         1
5       FG         2
6       GF       185
7       HW         5
8       LP         7
9       PP         4
10      SF        14
11      WC       251
12      WL        34
13      WP        44
```

Note the 10 non-trees that mark empty plots - we don't need them for this purpose. Let's clean them up. Here, we do that by setting the species to be missing instead of its current value, which is a blank. (It's actually redundant with the following command, so you can ignore it if you wish).

```
> ufc$species[is.na(ufc$dbh)] <- NA
> ufc$species <- factor(ufc$species)
```

The F(ir) and the FG(?) are probably GF (Grand fir). Let's make that change.

```
> ufc$species[ufc$species %in% c("F", "FG")] <- "GF"
> ufc$species <- factor(ufc$species)
> table(ufc$species)
```

We redefined the species factor in order to drop the empty levels. Now, drop the trees with missing height measures.

```
> ufc <- ufc[!is.na(ufc$height.m), ]
```

Graph the tree data (see Figure 8.1). It's always informative. We see some evidence of curvature, and some peculiar points. What might have caused those? Nonetheless, an encouraging plot - we should be able to get a fix on the height within a fair range.

What kind of underlying variability do we have to work with?

```
> sd(ufc$height.m, na.rm = T)
```

```
[1] 7.491152
```

Here's the flexibility of `tapply`, showing us standard deviation *by species*:

```
> tapply(ufc$height.m, ufc$species, sd, na.rm = TRUE)
```

```
      DF       ES       GF       HW       LP       PP       SF       WC
6.633357 5.656854 7.610491 2.489980 5.057997 9.899495 3.656486 6.972851
      WL       WP
8.848308 9.177620
```

9.2 Fitting

Now let's see if we can produce some kind of regression line to predict height as a function of diameter. Note that R is object-oriented, so we can create objects that are themselves model fits.

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i; \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{9.1}$$

That is, $y_i \sim \mathcal{N}(\beta_0 + \beta_1 x_i, \sigma^2)$. We will denote the predictions from this model by $\hat{y}_i$, that is, $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$.

The parameters β_0 and β_1 are (usually) estimated by the method of least squares (equivalently, Maximum Likelihood), while σ^2 is estimated by

$$\hat{\sigma}^2 = \frac{\text{sum of squared residuals}}{n-2}$$

Using calculus, it is possible to find expressions for these estimates:

$$\begin{aligned}
\hat{\beta}_1 &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \\
\hat{\beta}_0 &= \bar{y} - \hat{\beta}_1 \bar{x} \\
\hat{\sigma} &= \sqrt{\frac{\sum_{i=1}^n (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2}{n-2}}
\end{aligned}$$

These estimates can readily be obtained using a scientific calculator (which will give, at least, $\hat{\beta}_0$ and $\hat{\beta}_1$), but it is even easier to use a statistical package which gives a lot more as well.

```
> hd.lm.1 <- lm(height.m ~ dbh.cm, data = ufc)
```

9.3 Diagnostics

First, let's examine the model diagnostics (Figure 9.1).

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1)
> plot(hd.lm.1)
> par(opar)
```

The default plot statement produces four graphs when applied to a linear model (`lm`).

- The top left panel shows a plot of the residuals against the fitted values, with a smooth red curve superimposed. Here we're looking for evidence of curvature, non-constant variance (heteroscedasticity), and outliers. Our plot shows the curvature that we're concerned about, no evidence of heteroscedasticity, and two points that are quite unalike the others.
- The top right panel shows a quantile plot of the standardized residuals against the normal distribution. Here the ideal plot is a straight line, although modest departures from straightness are usually acceptable (due to large-sample theory, see Section 15.2.1 for more on this topic), and outlying points are obvious. Departures from a straight line in this plot can indicate non-normality of the residuals *or* non-constant variance.
- The bottom left panel shows the square root of the absolute residuals against the fitted values, along with a smooth red line. Departures from a horizontal line signify heteroscedasticity.
- The bottom right panel shows a plot of the leverage of the observations against the standardized residuals. These are the two components of Cook's Distance, a statistic that reports the overall impact upon the parameter estimates of the observations (note that being a large residual or a high leverage point alone is no guarantee of having a substantial impact upon the parameter estimates). A reasonably well accepted rule of thumb is that Cook's Distances greater that 1 should attract our attention. Contours of these distances (isoCooks?) at 0.5 and 1.0 are added to the graph to assist interpretation.

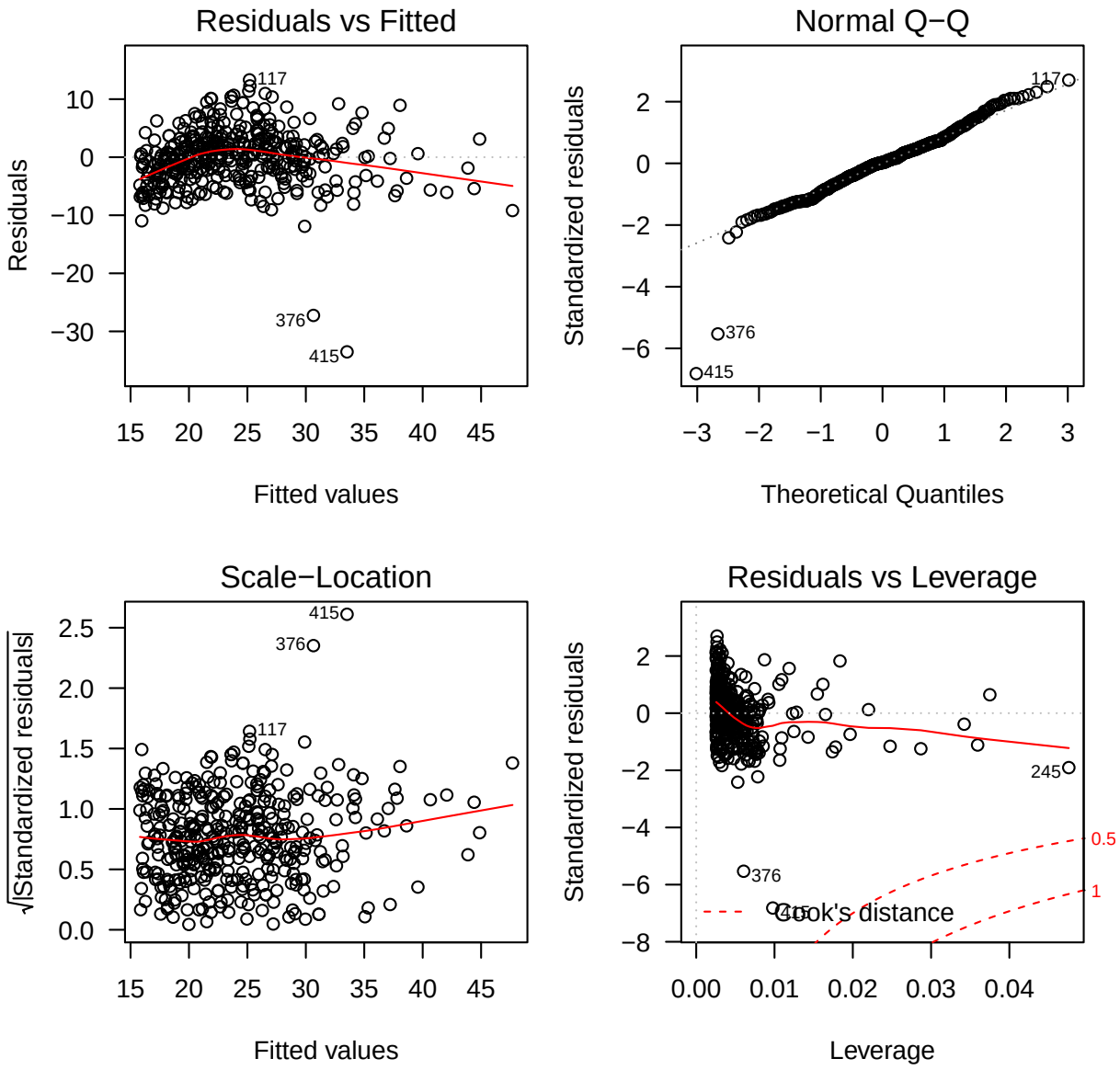

Figure 9.1: Diagnostic plots for the regression of diameter against height.

There are some worrying points, corresponding to the peculiar observations we noted earlier. However, the diagnostics imply that they shouldn't change things very much. None of the points are in the danger zone of the Cook's Distances, for example.

Let's see how we might examine them. The easiest route is probably to use the residuals to locate the offenders. It's a bit tricky - in order to be able to match the residuals with the observations, first we have to order them by the magnitude of the residual, then take the first two. The square brackets provide access to subscripting, which is one of the engines of S convenience.

```
> ufc[order(abs(residuals(hd.lm.1)), decreasing=TRUE), ][1:2, ]
```

```
    plot tree species dbh height dbh.cm height.m
415   78    5      WP 667      0   66.7      0.0
376   67    6      WL 575     34   57.5      3.4
```

It's clear that they're pretty odd looking trees! If we wish to exclude them from the model, we can do so via

```
> hd.res.1 <- abs(residuals(hd.lm.1))
> hd.lm.1a <- lm(height.m ~ dbh.cm, data = ufc, subset = (hd.res.1 <
```

```
+     hd.res.1[order(hd.res.1, decreasing = TRUE)][2]))
```

How did the estimates change? Here's a nice little plot that shows that the estimates changed very little. We can do this only because everything is about the same scale.

```
> opar <- par(las=1)
> plot(coef(hd.lm.1), coef(hd.lm.1a),
+      xlab="Parameters (all data)",
+      ylab="Parameters (without outliers)")
> text(coef(hd.lm.1)[1]-2, coef(hd.lm.1a)[1]-0.5,
+      expression(hat(beta)[0]), cex=2, col="blue")
> text(coef(hd.lm.1)[2]+2, coef(hd.lm.1a)[2]+0.5,
+      expression(hat(beta)[1]), cex=2, col="blue")
> points(summary(hd.lm.1)$sigma, summary(hd.lm.1a)$sigma)
> text(summary(hd.lm.1)$sigma+1, summary(hd.lm.1a)$sigma,
+      expression(hat(sigma)[epsilon]), cex=2, col="darkgreen")
> abline(0, 1, col="darkgrey")
> par(opar)
```

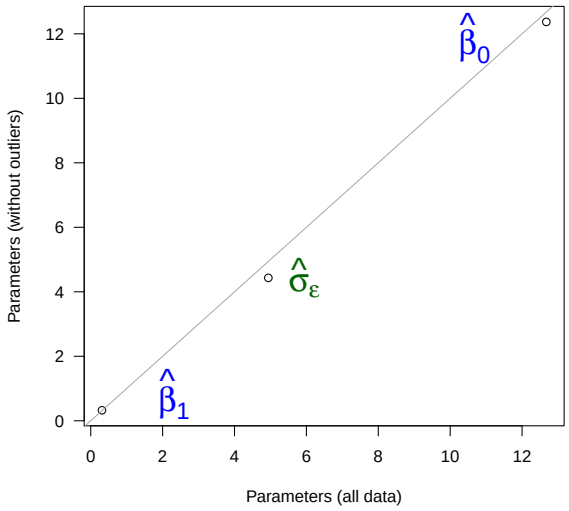

Figure 9.2: Parameter estimate change as a result of dropping the outliers.

9.4 Other Tools

Other tools for examining regression models should be mentioned:

```
> ?influence.measures
```

9.5 Examining the Model

Clearly, the peculiar observation doesn't affect the model very strongly; the intercept and slope are largely unchanged, and the variance of the residuals decreases slightly. Let's take a look at the model summary.

```
> summary(hd.lm.1)

Call:
lm(formula = height.m ~ dbh.cm, data = ufc)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-33.526  -2.862   0.132   2.851  13.321

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.67570    0.56406   22.47   <2e-16 ***
dbh.cm       0.31259    0.01388   22.52   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.941 on 389 degrees of freedom
Multiple R-squared: 0.566,        Adjusted R-squared: 0.5649
F-statistic: 507.4 on 1 and 389 DF,  p-value: < 2.2e-16
```

R first confirms the model that it has fitted, and then provides a 5-figure summary of the distribution of the residuals (we want them to be normal, remember).

The next section, `Coefficients`, summarizes the parameter estimates for the model. In addition to the parameter estimates, in the column labelled `Estimate`, there are other columns labelled `Std Error`, `t value` and `Pr(>|t|)`. The entries in the `Std Error` column are the *standard errors* of the estimates, those in the `t value` column are the values of the estimates divided by their standard error, and the values in the `Pr(>|t|)` column are the p-values for testing the hypothesis that the parameter is zero, against the 2-sided alternative.

The output also has values for `Residual Standard Error`, `Multiple R-squared`, and `Adjusted R-squared`.

Residual standard error, s The value of s is a measure of the variability of individual points as they vary from the fitted model.

$$s_{unexplained} = s_{residual} = \hat{\sigma} = \sqrt{\frac{\text{sum of } (residuals)^2}{\#\text{study units} - \#\text{things estimated}}}$$

$$\text{MS(residual)} = \frac{\sum_{i=1}^{n}[(\text{fuel consumption})_i - (\text{predicted consumption})_i]^2}{n-2} \tag{9.2}$$

$$= \frac{\sum_{i=1}^{n}\hat{e}_i^2}{n-2} = \hat{\sigma}^2. \tag{9.3}$$

R^2 The quantity R^2 (which R prints as `Multiple R-squared`) is calculated as $SS_{regression}/SS_{total}$ and is thus a measure of the variation explained by the model relative to the natural variation in the response values.

The SS (sums of squares) quantities are computed as follows. Recall that the predictions from the model are denoted $\hat{y}_i$.

$$SS_{regression} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 \tag{9.4}$$

$$SS_{total} = \sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{9.5}$$

We note in passing that there is a third such quantity.

$$SS_{residual} = \sum_{i=1}^{n}(y_i - \hat{y})^2 \tag{9.6}$$

and that for reasons that are not at all obvious, but are fundamentally Pythagorean,

$$SS_{total} = SS_{residual} + SS_{regression}. \tag{9.7}$$

We see that already, there is a big difference between the marginal variability, which is 7.49 m, and the conditional variability, which is 4.94 m. The model appears to be a useful one.

We can confirm this observation using the `anova` function, which provides a whole–model test.

```
> anova(hd.lm.1)

Analysis of Variance Table

Response: height.m
           Df  Sum Sq Mean Sq F value    Pr(>F)
dbh.cm      1 12388.1 12388.1  507.38 < 2.2e-16 ***
Residuals 389  9497.7    24.4
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

9.6 Other Angles

The process of model interrogation is simplified if we realize that the model fit is an object, and the summary of that model fit is a different object.

```
> names(hd.lm.1)

 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

> names(summary(hd.lm.1))

 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Some high-level functions, called `methods`, exist to enable the reliable extraction of model information, for example, `residuals()` and `fitted()`. We can learn what methods are available for our object as follows:

```
> class(hd.lm.1)

[1] "lm"

> methods(class = class(hd.lm.1))

 [1] add1.lm*           alias.lm*          anova.lm           case.names.lm*
 [5] confint.lm*        cooks.distance.lm* deviance.lm*       dfbeta.lm*
 [9] dfbetas.lm*        drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*        hatvalues.lm
[17] influence.lm*      kappa.lm           labels.lm*         logLik.lm*
[21] model.frame.lm     model.matrix.lm    nobs.lm*           plot.lm
[25] predict.lm         print.lm           proj.lm*           qr.lm*
[29] residuals.lm       rstandard.lm       rstudent.lm        simulate.lm*
[33] summary.lm         variable.names.lm* vcov.lm*

   Non-visible functions are asterisked
```

We can also extract or otherwise manipulate the attributes of the objects by means of the $ sign:

```
> hd.lm.1$call

lm(formula = height.m ~ dbh.cm, data = ufc)

> summary(hd.lm.1)$sigma

[1] 4.941222
```

9.7 Other Models

We can add further complexity to our model as follows. Try these and see what you learn.

```
> hd.lm.2 <- lm(height.m ~ dbh.cm + species, data = ufc)
> hd.lm.3 <- lm(height.m ~ dbh.cm * species, data = ufc)
```

9.8 Collinearity

A common problem for fitting and interpreting regression models in observational studies is in collinearity of predictor variables. When two predictor variables are correlated, it may be difficult to discern which of them explains the variation of the response variable, if either.

A consequence of collinearity is that interval estimates of parameters can be much wider than they would otherwise. Collinearity is a consequence of the structure and pattern of the design matrix, and therefore reflects the data as they have been collected. If you want to avoid collinearity, then perform a designed experiment.

We can detect and represent collinearity using the variance–inflation factor (VIF), for example as provided in the `car` package by Professor John Fox.

One strategy for handling collinearity that only sometimes makes sense is to replace the offending predictor variables, say x_1 and x_2, with statistically equivalent functions of them, for example, $x_3 = x_1 + x_2$ and $x_4 = x_1 - x_2$, or, equivalently, their average and their difference. The substitution of these constructed variables does not change the nature of the underlying space that the design matrix describes, but it does change the description, because x_3 and x_4 are independent.

9.9 Weights

Although the sample trees may be considered to be a random sample, they are not selected with equal probability, and they are clustered in location, because they are from a variable-radius plot sample. How much difference to the parameter estimates does this sample design make? Here, we will assess the effect of the sample weights. We will examine tools that illuminate the effect of clustering in the next chapter.

Recall that in a variable-radius plot, the probability of selection of a tree is proportional to its basal area. We can fit a model that accomodates this effect as follows.

```
> hd.lm.5 <- lm(height.m ~ dbh.cm * species,
+               weights = dbh.cm^-2,
+               data = ufc)
```

Now we would like to know what effect this change has on our parameter estimates. To pull out estimates of each slope and intercept we can use the `estimable()` function provided in the `gmodels` package, but for certain models there is a simpler way to obtain them.

```
> unweighted <- coef(lm(height.m ~ dbh.cm * species - 1 - dbh.cm,
+     data = ufc))
> weighted <- coef(lm(height.m ~ dbh.cm * species - 1 - dbh.cm,
+     weights = dbh.cm^-2, data = ufc))
```

We can now plot these estimates in various more or less informative ways. The following code constructs Figure 9.3.

```
> par(mfrow=c(1,2), las=1, mar=c(4,4,3,1))
> plot(unweighted[1:10], weighted[1:10], main="Intercepts", type="n",
+      xlab="Unweighted", ylab="Weighted")
> abline(0, 1, col="darkgrey")
> text(unweighted[1:10], weighted[1:10], levels(ufc$species))
> plot(unweighted[11:20], weighted[11:20], main="Slopes", type="n",
+      xlab="Unweighted", ylab="Weighted")
> abline(0, 1, col="darkgrey")
> text(unweighted[11:20], weighted[11:20], levels(ufc$species))
```

The effect of weighting appears to be pretty substantial. This effect can be interpreted as telling us that the smaller and the larger trees require different models.

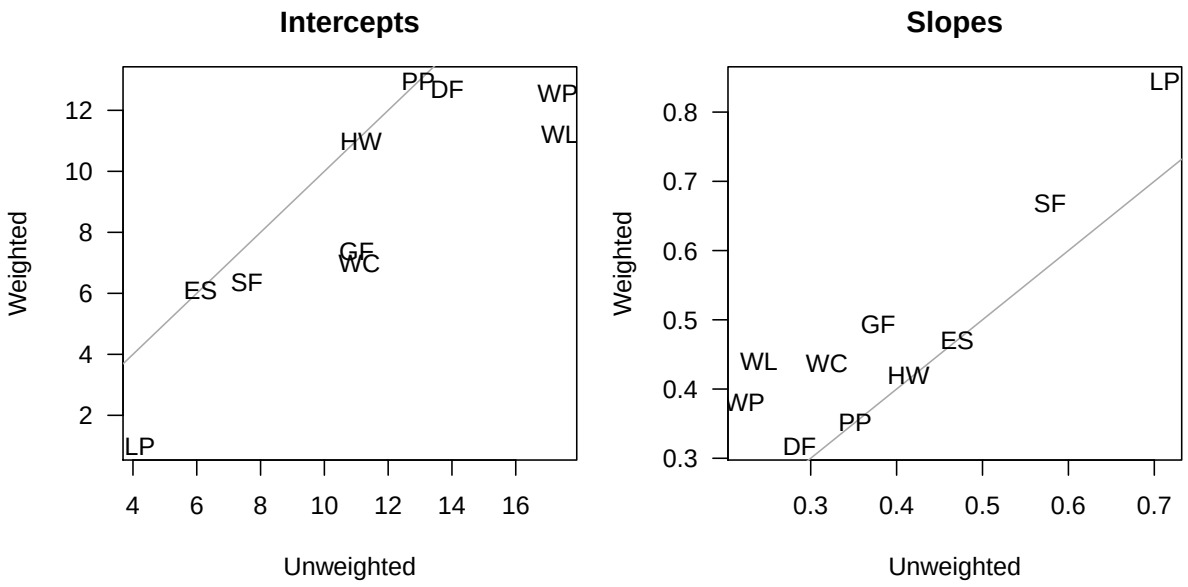

Figure 9.3: Summary of effect of sample weights upon species–specific height–diameter parameter estimates for UFC data.

9.10 Transformations

Sometimes the modeller has biological or statistical reasons to change the nature of the data that are being modelled. Such transformations can either be done by creating a new variable in the dataframe with the appropriate function, or by making the transformation inside the call to the model.

```
> ufc$log.height.m <- log(ufc$height.m)
> hd.lm.4a <- lm(log.height.m ~ dbh.cm * species, data = ufc,
+                subset=height.m > 0)
> hd.lm.4b <- lm(I(log(height.m)) ~ dbh.cm * species, data = ufc,
+                subset=height.m > 0)
```

Note the use of the `I()` function used in the latter approach. This tells R to interpret the function in the usual way, not in the context of the linear model. The difference becomes more obvious when we think about the double-usage that R is placing upon our familiar operators. For example, `*` usually means multiplication, but in the linear model it means include main effects and interaction. Likewise, `+` usually means addition, but in the linear model it is used to punctuate the model statement.

In my experience, transformation is an over-used tool. I believe that it is best if possible to work in the units that will make the most sense to the person who uses the model. Of course it is possible to back-transform arbitrarily, but why do so if it is unnecessary?

There are certainly circumstances where transformation is called for, but quite often a more appropriate and satisfying strategy is available, whether that be fitting a generalized linear model, an additive model, or performing a small Monte-Carlo experiment on the residuals and placing trust in the Central Limit Theorem.

9.11 Testing Specific Effects

R uses an algorithm to control the order in which terms enter a model. The algorithm respects hierarchy, in that all intereactions enter the model after the main effects, and so on. However, sometimes we need to change the order in which terms enter the model. This may be because we are interested in testing certain terms with other terms already in the model. For example, using the `npk` dataset provided by `MASS` (Venables and Ripley, 2002), imagine that N and P are design variables, and we wish to test the effect of K in a model that already includes the N:P interaction.

```
> require(MASS)
> data(npk)
```

```
> anova(lm(yield ~ block + N * P + K, npk))

Analysis of Variance Table

Response: yield
          Df Sum Sq Mean Sq F value   Pr(>F)
block      5 343.29  68.659  4.3911 0.012954 *
N          1 189.28 189.282 12.1055 0.003684 **
P          1   8.40   8.402  0.5373 0.475637
K          1  95.20  95.202  6.0886 0.027114 *
N:P        1  21.28  21.282  1.3611 0.262841
Residuals 14 218.90  15.636
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The preceding code fails, because of R's effect order algorithm. To force the order, we must use the terms function.

```
> anova(lm(terms(yield ~ block + N * P + K, keep.order = TRUE),
+     npk))

Analysis of Variance Table

Response: yield
          Df Sum Sq Mean Sq F value   Pr(>F)
block      5 343.29  68.659  4.3911 0.012954 *
N          1 189.28 189.282 12.1055 0.003684 **
P          1   8.40   8.402  0.5373 0.475637
N:P        1  21.28  21.282  1.3611 0.262841
K          1  95.20  95.202  6.0886 0.027114 *
Residuals 14 218.90  15.636
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Happily, we learn that the term order makes no difference in this case.

9.12 Other Ways of Fitting

We can also use other tools to fit the regression. For example, the following code fits the same regression model using maximum likelihood, and provides estimates of the parameters and their asymptotic standard errors.

```
> normal.ll <- function(parameters, x, y) {
+   sum( dnorm(y,
+              parameters[1] + parameters[2] * x,
+              parameters[3],
+              log = TRUE ))
+ }
> good.fit <- optim(c(1, 1, 1),
+                   normal.ll,
+                   hessian = TRUE,
+                   control = list(fnscale=-1),
+                   x = ufc$dbh.cm,
+                   y = ufc$height.m)
> good.fit$par

[1] 12.6781358  0.3127113  4.9327072

> sqrt(diag(solve(-good.fit$hessian)))
```

```
[1] 0.56309283 0.01385363 0.17661565
```

We can also fit a different error distribution by fitting a location-scale family. Recall that, for any pdf $f(x)$ and constants μ and $\sigma > 0$,

$$g(x|\mu,\sigma) = \frac{1}{\sigma} f\left(\frac{x-\mu}{\sigma}\right) \tag{9.8}$$

is also a pdf (see e.g. Casella and Berger, 1990, p. 116). Here we will use $\exp(\sigma)$ rather than σ because we know that $\exp(\sigma) > 0$

Here we're fitting a linear regression with errors described by a t-distribution with 3 degrees of freedom, which provides robustness to outliers.

```
> t3.ll <- function(parameters, x, y) {
+   sum( dt((y - parameters[1] - x * parameters[2]) / exp(parameters[3]),
+           df = 3,
+           log = TRUE ) - parameters[3])
+ }
> good.fit.t <- optim(c(1, 1, 1),
+                     t3.ll,
+                     hessian = TRUE,
+                     control = list(fnscale=-1),
+                     x = ufc$dbh.cm,
+                     y = ufc$height.m)
> good.fit.t$par

[1] 12.1153235  0.3310526  1.2474535

> sqrt(diag(solve(-good.fit.t$hessian)))

[1] 0.52052602 0.01352119 0.04918994
```

The interpretation of the parameters μ and σ is left to the analyst. Don't forget that the variance of the t_ν is $\frac{\nu}{\nu-2}$, so the reported scale parameter should not be interpreted as the conditional standard deviation of the data.

Finally, we can use a non-linear model, by changing the mean function.

```
> normal.ll.nl <- function(parameters, x, y) {
+   sum( dnorm(y,
+              parameters[1] * x ^ parameters[2],
+              parameters[3],
+              log = TRUE ))
+ }
> good.fit <- optim(c(1, 1, 1),
+                     normal.ll.nl,
+                     hessian = TRUE,
+                     control = list(fnscale=-1),
+                     x = ufc$dbh.cm,
+                     y = ufc$height.m)
> good.fit$par

[1] 4.4587482 0.4775113 4.7196199

> sqrt(diag(solve(-good.fit$hessian)))

[1] 0.33860126 0.02034234 0.16876952
```

All these parameter estimators are maximum likelihood estimators, conditional on the model, and therefore are asymptotically normal, efficient estimators.

I include more about writing your own functions in Chapter 15.

Chapter 10

Bootstrap

10.1 Introduction

We now take *another* big step in a different direction. Previously we have relied on theory such as the Central Limit Theorem to provide us with a rationale to construct interval estimates. What if the available sample is small and we are unwilling to take the plunge, and reluctant to transform the data to make it more symmetric? What if we don't know an appropriate estimator for the standard error of the population parameter? What if we suspect that our estimator might be biased, and we'd like to correct for it? When the data are sparse and our assumptions are uncertain, we can obtain a modest buffer by using computer–intensive tools.

Computer–intensive tools can be used to augment the theory available for data analysis and modelling. We will examine such a tool in this lab class: the bootstrap. It is not a panacea, but each can, when used carefully, provide useful insight into data.

10.2 Bootstrap

Recall that the sampling distribution of a statistic is the distribution of the statistic that results from the repeated process of taking a sample from a population using some fixed sampling prescription and computing the statistic upon each random sample.

The bootstrap idea proposes that to learn about the sampling distribution of a statistic, we could simulate the distribution from the available data.

This material will focus on the *non–parametric* bootstrap.

10.2.1 The Basic Idea

Imagine, for example, that we are interested in an interval estimate of the 50% trimmed mean of the population from which the following twelve numbers were randomly selected. Note that while we might reasonably expect the trimmed mean to follow the Central Limit Theorem, it is difficult to know what to use as an estimate of its variance.

```
> trim.me <- c(0, 1, 2, 3, 4, 6, 8, 10, 10, 12, 13, 15)
```

The point estimate is

```
> mean(trim.me, trim = 0.25)
```

```
[1] 6.833333
```

Now let's treat the sample as a population, and resample from it, with replacement, and take the mean again.

```
> trim.me.1 <- sample(trim.me, size = 12, replace = TRUE)
> mean(trim.me.1, trim = 0.25)
```

```
[1] 10
```

Naturally, it's very likely to be different. One iteration of this resample–and–estimate doesn't help us much. However, if we repeat the process, say, 1000 times, then we have a distribution that estimates the sampling distribution of the population parameter (Figure 10.1).

```
> samp.dist <- sapply(1:999,
+                     function(x) mean(sample(trim.me,
+                                             size = 12,
+                                             replace = TRUE),
+                                      trim = 0.25))
> par(las = 1, mar=c(4,4,1,1))
> plot(density(samp.dist), main="")
```

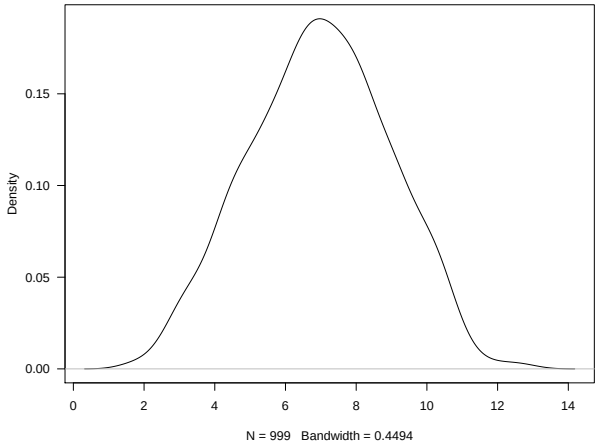

Figure 10.1: Sampling Distribution of Trimmed Mean

This distribution suggests to us a simple way to estimate and correct for any bias in our original estimator. We would estimate the bias in the original estimator by the difference between the original estimator and the resampled estimator:

```
> mean(samp.dist) - mean(trim.me, trim = 0.25)
```

```
[1] 0.1294628
```

So an *unbiased* estimate (the original estimate, with the bias subtracted) would look like this:

```
> 2 * mean(trim.me, trim = 0.25) - mean(samp.dist)
```

```
[1] 6.70387
```

To obtain an interval estimate using this rationale, we would apply the following approach:

```
> 2 * mean(trim.me, trim = 0.25) - quantile(samp.dist, p = c(0.975,
+     0.025))
```

```
97.5%  2.5%
  3.0  10.5
```

That is, to get the bias–corrected estimate for the lower interval, we add to the original estimate the difference between the original estimate and the *upper* quantile, and *vice versa*. This interval is called the *basic* bootstrap.

The distribution in Figure 10.1 also suggests to us a simple and direct way to obtain a non–parametric interval estimate of the trimmed mean. We could simply identify the 2.5% and 97.5% quantiles of the estimated density.

```
> quantile(samp.dist, p = c(0.025, 0.975))
```

```
    2.5%     97.5%
3.166667 10.666667
```

This approach to constructing an interval estimate is called the *percentile estimate*.

Although the percentile estimate is intuitively satisfying, it fails to correct for any bias.

We can take our analysis a step further. For example, we may believe that the underlying sampling distribution really should be normal. *In that case*, it would make more sense to estimate the standard error of the estimate and construct normal confidence intervals, which are substantially more stable to estimate than are the percentiles. We estimate the standard error using the standard deviation of the sampling distribution.

```
> mean(samp.dist) + 1.96 * c(-1, 1) * sd(samp.dist)

[1]  3.067104 10.858488
```

This approach to constructing an interval estimate is called the *normal estimate*. We can check to see whether normality seems justified using a familiar tool, as follows (Figure 10.2).

```
> par(las = 1)
> qqnorm(samp.dist, main = "")
> qqline(samp.dist)
```

That distribution looks defensibly normal to me.

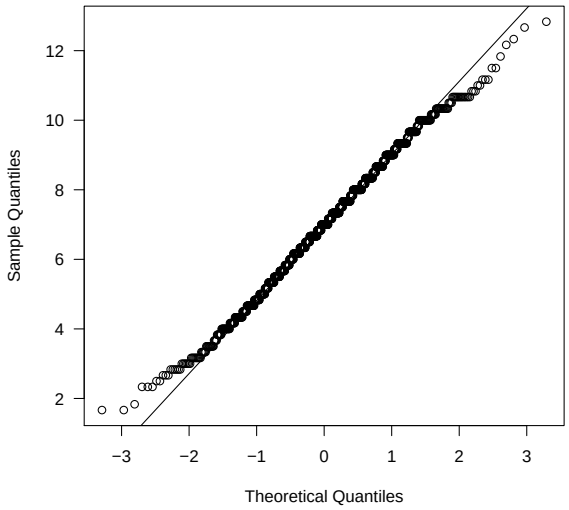

Figure 10.2: Normal quantile plot of estimated sampling distribution of 50% trimmed mean.

10.2.2 Formal Implementation

R provides access to this functionality via a package, `boot`. Using the functions in boot involves a bit of a learning curve, because they are designed to provide great flexibility rather than a simple approach. This learning curve is manifested as needing to know more about R functions and R objects.

In order to use the bootstrap within R, we need to write a function that will take at least two arguments: the sample to be input, and an index, which will permute the sample. In the case of the trimmed mean, we might write the function like this:

```
> trim.boot <- function(x, i) {
+   return(mean(x[i], trim = 0.25))
+ }
```

This function will permute the sample `x` using the index `i`, and return the 50% trimmed mean on the permutation.

We invoke the bootstrap upon it as follows. First, we load the `boot` library.

```
> library(boot)
```

Then we use the `boot` function, saving the outcome as an object. Here, we ask for 999 bootstrap replicates.

```
> trim.25 <- boot(trim.me, trim.boot, R = 999)
```

We can plot the bootstrap output to satisfy ourselves about the assumptions, as follows (Figure 10.3).

```
> par(las = 1)
> plot(trim.25)
```

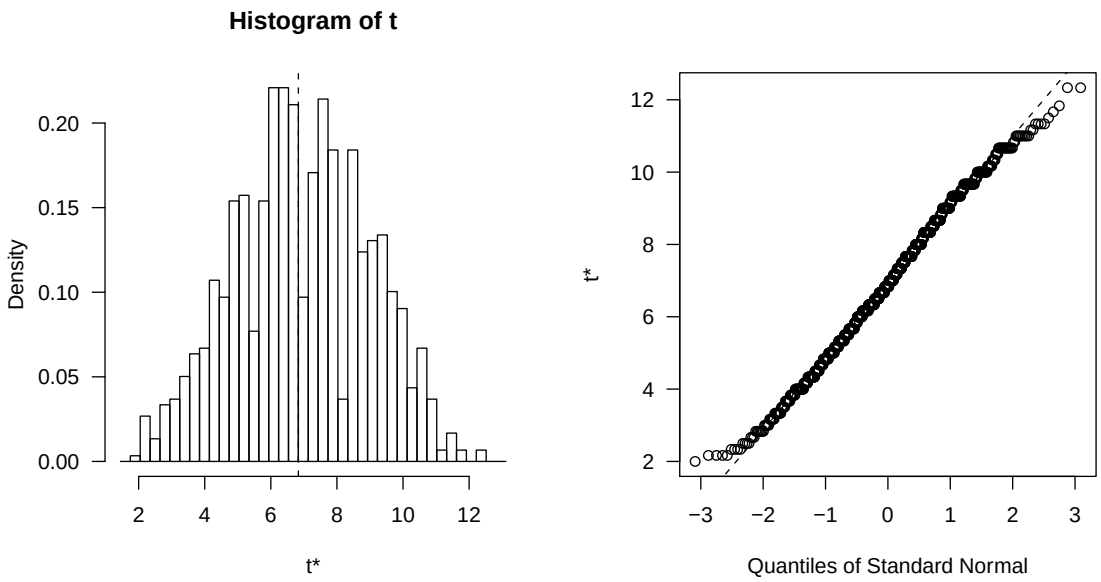

Figure 10.3: Diagnostic graphs for 50% trimmed mean bootstrap.

These diagnostics suggest that the assumption of normality seems reasonable. Finally, we examine the bootstrap object.

```
> trim.25

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = trim.me, statistic = trim.boot, R = 999)


Bootstrap Statistics :
    original    bias    std. error
t1* 6.833333 0.1006006    2.028981
```

This object provides us with information about the call, and with estimates of the bias and standard error of the estimator.

We now move to requesting interval estimates, as follows.

```
> boot.ci(trim.25, type = c("basic", "norm", "perc"))
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates

CALL :
boot.ci(boot.out = trim.25, type = c("basic", "norm", "perc"))

Intervals :
Level      Normal              Basic              Percentile
95%   ( 2.756, 10.709 )   ( 3.000, 10.667 )   ( 3.000, 10.667 )
Calculations and Intervals on Original Scale
```

The estimates reported above differ from those computed earlier because the process is random and our sample sizes are finite!

The multiplicity of estimates prompts us to ask: how can we choose amongst them? The recommendation is: if the estimates are close, then use the basic estimate. If they are not close, then do not use any of them — see below instead.

10.2.3 Improvements on the Theme

So far we have covered three types of bootstrap estimator: the basic estimate, the percentile estimate and the normal estimate. There are many more, for example, the *studentized* bootstrap, and the *accelerated, bias–corrected* bootstrap. It is also possible to improve upon some of these bootstraps by selecting a suitable transformation for the data. The possibilities are manifold.

Without going into unnecessarily great detail about these other approaches, we note the following points.

1. It is possible to use some characteristics of the estimated sampling distribution as a basis for improving on the quality of the percentile interval estimate. For example, the distribution may show that the estimator is biased, and the distribution itself might be skewed. This reasoning leads to the an interval estimate called the accelerated, bias–corrected bootstrap.

   ```
   > boot.ci(trim.25, type = c("bca"))

   BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
   Based on 999 bootstrap replicates

   CALL :
   boot.ci(boot.out = trim.25, type = c("bca"))

   Intervals :
   Level       BCa
   95%   ( 2.833, 10.667 )
   Calculations and Intervals on Original Scale
   ```

2. If you can estimate the variance of the estimator for each bootstrap sample, then it is possible to use a studentized bootstrap. The advantage of the studentized bootstrap is that it has better properties as an estimator. Broadly speaking, all bootstrap estimators rely upon asymptotic theory, and the studentized bootstrap gets there faster.

10.2.4 Diagnostics

In addition to the histogram and normal quantile plot noted above, we are often interested in knowing the contribution that any one observation makes to the overall picture. We can obtain this information from a bootstrap using the so-called *jackknife–after–bootstrap* tool (Figure 10.4).

```
> par(las = 1, mar = c(4, 4, 1, 1))
> jack.after.boot(trim.25)
```

The construction of the jackknife–after–bootstrap is as follows. For each observation, the bootstrap samples that *exclude* that observation are identified, and their estimates gathered. The quantiles of these

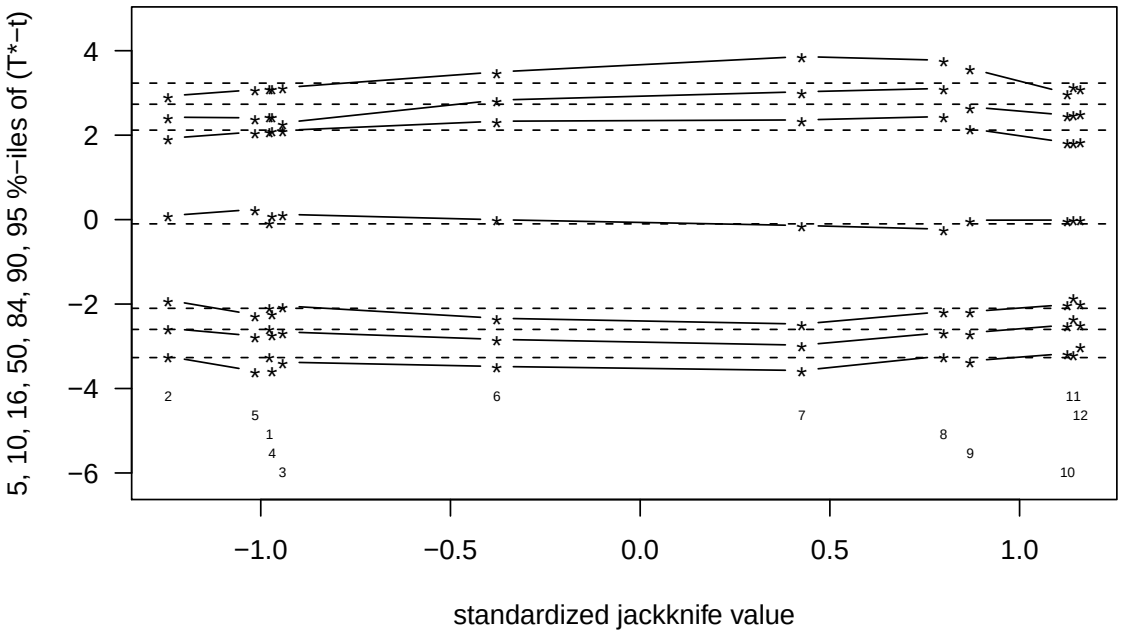

Figure 10.4: Jackknife–after–bootstrap graphical diagnostic tool.

estimates are then graphed as a sequence of points with the observation number underneath, at the x–location that corresponds to an estimate of their effect on the overall bootstrap estimate. The horizontal dotted lines can be thought of as representing the overall bootstrap distribution. Sharp deviation from these lines suggests a point that has substantial influence on the outcome.

For example, note that omission of points 10 and 11, on average, tend to shift the lower tail of $T^* - t$, which means that there are fewer large estimates from the bootstrap sample when those points are omitted.

For this dataset no points stand out in particular, but see Exercise 2.

10.2.5 Extensions

We can construct hypothesis tests of the parameters of interest by inverting the confidence intervals that we have computed. That is, if we start with a null hypothesis that a population parameter $\mu = k$, say, then in order to test that null hypothesis with size α, we construct a $1 - \alpha$ confidence interval for the population parameter. If the interval does not include k then we reject the null hypothesis.

10.2.6 Wrap–up

A common question is: how many bootstrap replicates should we use? Mostly, bootstrap replicates are really cheap. Then, we should think about doing thousands of them. If they are more expensive, then however many our time can afford.

Despite its attractions, the bootstrap does not always work. In particular, we should be wary of applying the bootstrap when there are data missing, when the data are dirty, and when the assumption of independence is questionable. The bootstrap will give us an answer in any of these cases, but the answer may well be misleading. *Caveat computator!*

Further reading: Efron and Tibshirani (1993) and Davison and Hinkley (1997).

10.3 A Case Study

We need a fairly challenging example. The following data are too good an example to pass up, and provide a nice break from forestry. These data are from OzDASL: `http://www.statsci.org/data/oz/rugby.html`. The following description is from Lee (1994). Rugby football is a popular quasi-amateur sport widely played in the United Kingdom, France, Australia, New Zealand, South Africa, and elsewhere. It is rapidly gaining popularity in the US, Canada, Japan and parts of Europe. Recently, some of the rules

of the game have been changed, with the aim of making play more exciting. In a study to examine the effects of the rule changes, Hollings and Triggs (1993) collected data on some recent games.

Typically, a game consists of bursts of activity which terminate when points are scored, if the ball is moved out of the field of play or if an infringement of the rules occurs. In 1992, the investigators gathered data on ten international matches which involved the New Zealand national team, the All Blacks. The first five games studied were the last international games played under the old rules, and the second set of five were the first internationals played under the new rules.

For each of the ten games, the data list the successive times (in seconds) of each passage of play in that game. One interest is to see whether the passages were on average longer or shorter under the new rules. (The intention when the rules were changed was almost certainly to make the play more continuous.)

```
> require(lattice)
> require(MASS)
> require(boot)
> require(nlme)
```

Read the data and make necessary adjustments.

```
> rugby <- read.csv("../data/rugby.csv")
> rugby$rules <- "New"
> rugby$rules[rugby$game < 5.5] <- "Old"
> rugby$game.no <- as.numeric(rugby$game)
> rugby$rules <- factor(rugby$rules)
> rugby$game <- factor(rugby$game)
> str(rugby)
```

```
'data.frame':        979 obs. of  4 variables:
 $ game   : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ time   : num  39.2 2.7 9.2 14.6 1.9 17.8 15.5 53.8 17.5 27.5 ...
 $ rules  : Factor w/ 2 levels "New","Old": 2 2 2 2 2 2 2 2 2 2 ...
 $ game.no: num  1 1 1 1 1 1 1 1 1 1 ...
```

The following analysis is from the OzDASL website:

> Can use one-way analysis of variance on this data to compare the ten games. A contrast would then be relevant to compare the first five games to the second five.
>
> Alternatively, one can pool the times for the first five games and the last five games together and conduct a two-sample test.
>
> The times are highly skew to the right. They need to be transformed to approximate symmetry; alternatively they could be treated as exponential or gamma distributed.

Figure 10.5 confirms that the data have substantial skew.

```
> densityplot(~time | game, data = rugby, layout = c(5, 2))
```

Assuming that the design choices made by the recommended analysis are reasonable, do we really need to transform the data for the analysis? Here is the standard approach. Figure 10.6 reports the important diagnostic; the qq probability plot of the standardized residuals.

```
> times.lm <- lm(time ~ rules, data = rugby)
```

```
> qqnorm(stdres(times.lm))
> qqline(stdres(times.lm))
```

```
> boxcox(times.lm)
```

If we were to follow the strategy advocated by the website, (and many popular regression books!) we'd now try something like a Box-Cox diagnostic, which gives us Figure 10.7. This seems to highly recommend a transformation of about 0.2.

Effecting the transformation and performing the analysis gives us the normal probability plot for residuals presented in Figure 10.8. Not perfect, but much better than in Figure 10.6.

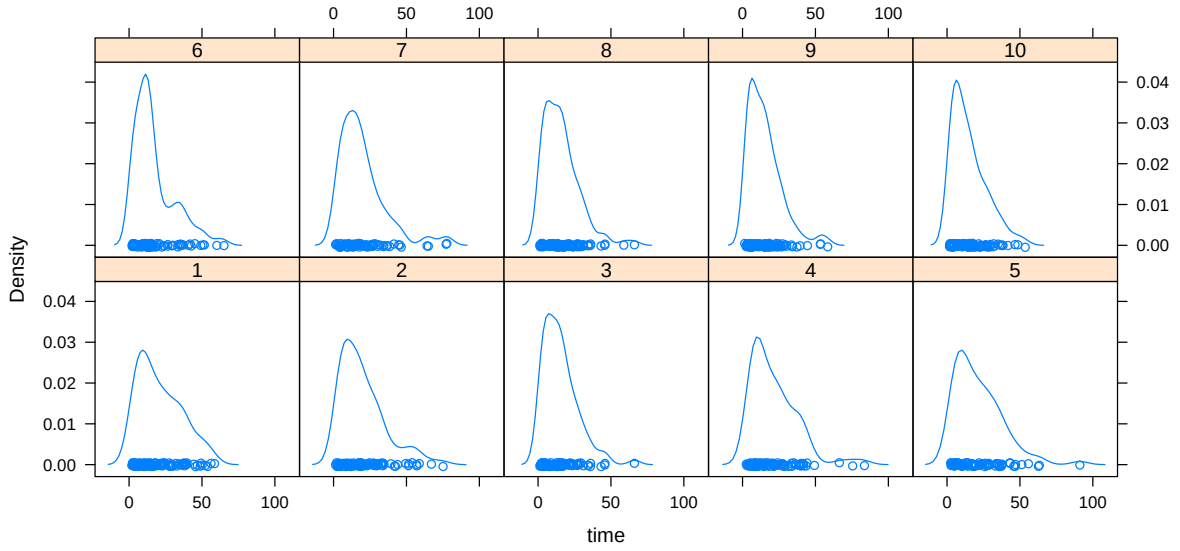

Figure 10.5: Lattice density plot of the rugby times data, by game.

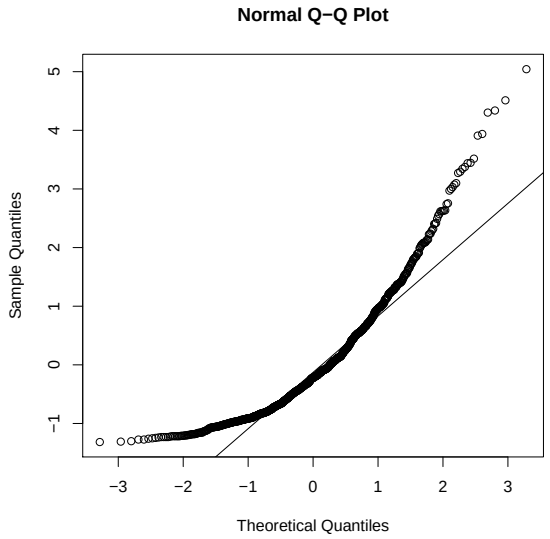

Figure 10.6: Key diagnostic graph for ordinary least squares regression upon rugby data: the qq probability plot of the standardized residuals.

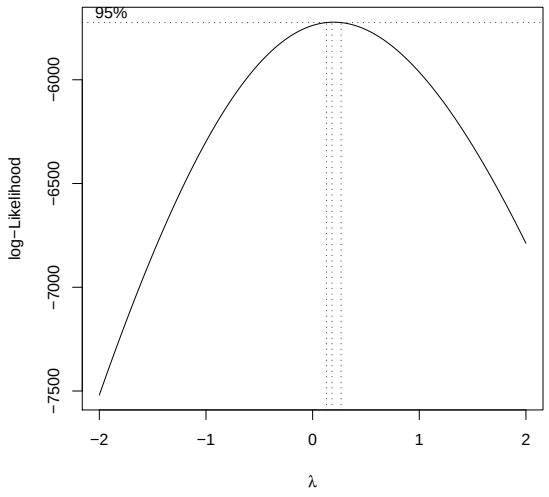

Figure 10.7: Box-Cox diagnostic test from MASS (Venables and Ripley, 2002). Note the tight confidence interval, which implies that a transformation is recommended.

```
> times.lm.2 <- lm(I(time^0.2) ~ rules, data = rugby)

> qqnorm(as.numeric(stdres(times.lm.2)))
> qqline(as.numeric(stdres(times.lm.2)))
```

The null hypothesis that there is no difference between the new rules and old can now be tested.

```
> anova(times.lm.2)

Analysis of Variance Table

Response: I(time^0.2)
          Df Sum Sq Mean Sq F value    Pr(>F)
```

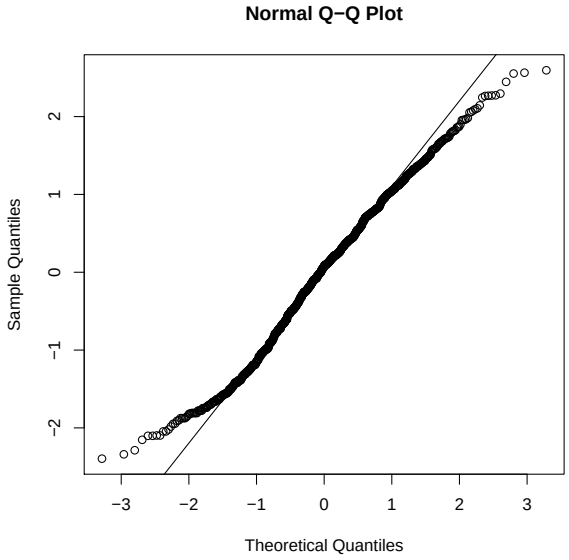

Figure 10.8: Key diagnostic graph for ordinary least squares regression upon transformed rugby data: the qq probability plot of the standardized residuals.

```
rules       1  1.147 1.14666  14.301 0.0001652 ***
Residuals 977 78.335 0.08018
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

A highly significant `Rules` effect suggests that there is indeed a difference - for the transformed data. Had we checked the model from the untransformed data, we would have concluded the same thing.

Regression books commonly advocate examination of the normal probability plot of the residuals, in order to assess the cleavage of the model/data combination to the assumption of normality. If the residuals are normal, then we can estimate confidence intervals, test parameters, conveniently, and do whole-model tests, etc.

In fairness I should add that not all such books do so, for example Weisberg (2005) and Gelman and Hill (2007) give this diagnostic a low priority.

Huber (1981) says (pp 159-160) that the sampling distribution of an arbitrary linear combination of the parameter estimates from least-squares regression is asymptotically normal if $h = \max_i(h_i) \to 0$ as n increases, where h_i are the diagonal elements of the Hat matrix (i.e. the leverages). Therefore the above condition of the normality of residuals is really too restrictive. That is, we ought to be able to perform those estimations and tests under the circumstances that either of two cases holds:

1. the residuals are normally distributed, or
2. the sample size is large enough that we can invoke large-sample theory.

The standard qq normal plot regression diagnostic addresses case 1 but not case 2. Therefore the diagnostic plot might imply that we have to transform, or use weights, or worry in some way, because the residuals are non-normal. However, in some situations, we don't have to worry because it is reasonable to interpret the Central Limit Theorem as asserting that for a sufficiently large sample size, the sampling distribution of a well-behaved estimate of a parameter can be treated as being normally distributed with negligible error. The problem is in identifying what a suffifficiently large sample size is, or perhaps more immediately, whether the sample we have is sufficiently large.

I'd like to avoid transforming because it involves messing with what is often a perfectly good abstract model. For example, I'd really rather not transform the rugby data, because the model loses the measurement units, and therefore a substantial slice of the real-world meaning and interpretation and utility. From this point of view, the Box-Cox diagnostic test, as in Figure 10.7, provides us with the exact asymptotic inverse of good advice. As the sample size increases, the 95% CI for the parameter that represents the optimal transformation narrows, making the transformation seem more urgent - when really it is less urgent because of the large-sample theory.

To avoid this vexing problem, we could bootstrap the data, fit the model, obtain non-parametric estimates of the sampling distributions of the parameter(s) of interest (be they estimates or test statistics), and then either assess their normality, or just use the estimated distributions directly. Such an approach will allow us to balance the effects of the underlying distribution of the parameters against large sample theory, and let us know whether we really need to transform, apply weights, or whatever.

So, we can write a function that will compute the parameter estimates from an arbitrary sample. For this model it will simply be the difference between the means of the timings for the old and the new rules, which doesn't require a formal linear model fit, but we'll keep the model anyway. We use a studentized bootstrap. We will see how to attack this problem with a loop in Section 15.2.

```
> boot.lm <- function(data, i) {
+     model.t <- coefficients(summary(lm(data[i, 2] ~ data[i, 3])))
+     c(model.t[2, 1], model.t[2.2]^2)
+ }
```

This function extracts the estimate of the time difference between the two groups and its variance.

The bootstrap command is:

```
> boot.rugby <- boot(rugby, boot.lm, R = 199)
```

We can extract and assess the distribution of the bootstrapped simulations via the following code, the output of which is presented in Figure 10.9:

```
> plot(boot.rugby)
```

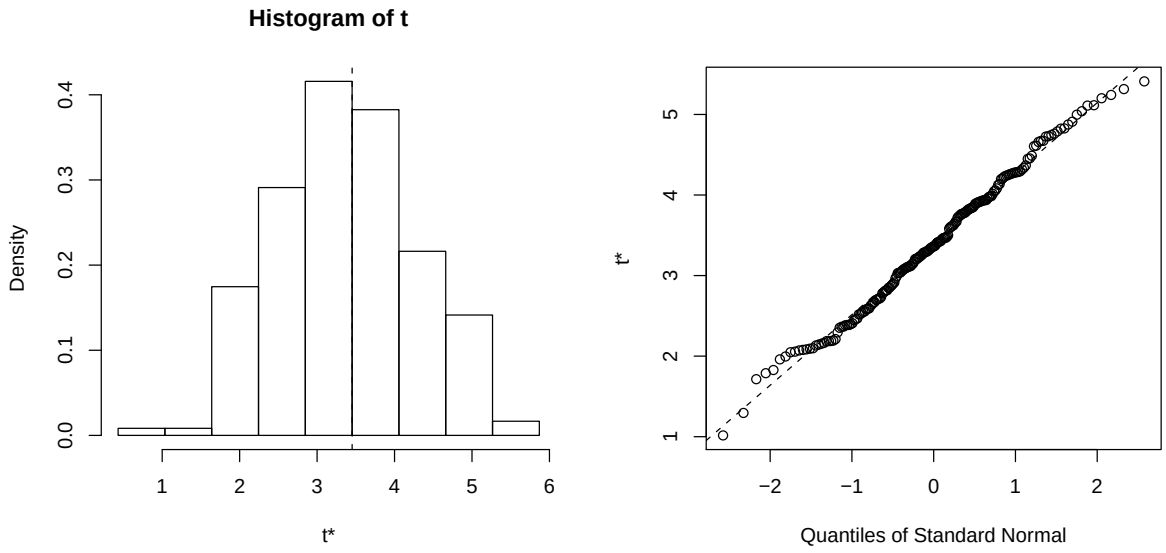

Figure 10.9: Key diagnostic graphs for bootstrap upon untransformed rugby data: the histogram and qq probability plot of the simulated values.

This distribution appears to be acceptably normal. This diagnostic suggests that we can proceed with the analysis of the bootstrap results (Davison and Hinkley, 1997).

To conclude, we can compare the bootstrap estimate of the standard error

```
> sd(boot.rugby$t[, 1])
```

```
[1] 0.8768196
```

with the model-based estimate from the original, untransformed data:

```
> coefficients(summary(times.lm))[2, 2]
```

```
[1] 0.9038698
```

These also appear acceptably close. This result suggests that, in this case, an invocation of large-sample theory for the initial model fit would have been reasonable. The transformation seems to have been unnecessary, and has cost us an easily-interpretable model. It would also be interesting to know whether the transformation costs power or size.

1. In addition to providing estimators in its own right, bootstrapping is a useful way to see whether or not the sample is suitable for large-sample theory. Consider the following datasets in the context of a confidence interval for the mean.

   - `> data.1 <- rexp(10)`
   - `> data.2 <- rexp(100)`
   - `> data.3 <- rcauchy(10)`
   - `> data.4 <- rcauchy(100)`

   For each,

   (a) Use a graphical tool to assess the suitability of the assumption that the population is normal.
   (b) Compute the 95% confidence interval for the population mean using this sample.
   (c) Create a bootstrap for the population mean, and examine the usual graphical diagnostics. What do they tell you?
   (d) Compute the four kinds of intervals for the mean that we have introduced in this chapter. What do they tell you?

2. Consider the following dataset in the context of a confidence interval for the mean.

   ```
   > data.5 <- c(rexp(9), 7)
   ```

   (a) Create a bootstrap for the population mean, and examine the usual graphical distributional diagnostics. What do they tell you?
   (b) Compute the four kinds of intervals for the mean that we have introduced in this chapter. What do they tell you?
   (c) Examine the jack–after–boot diagnostic. What does it tell you?

Chapter 11

Modelling, but not as We Know It

We now spend some time exploring tools that will be useful when, for whatever reason, the world no longer seems linear.

11.1 Non-linear models

This section focuses on the R tools that can be used for fitting non-linear regression, in various guises. As before, I do not present the theory here as that has been covered much more effectively elsewhere. Recommended reading includes: Ratkowsky (1983), Gallant (1987), Bates and Watts (1988), and Seber and Wild (2003). Our data are Norway spruce measurements from von Guttenberg (1915), kindly provided by Professor Boris Zeide.

First we read and examine the data.

```
> gutten <- read.csv("../data/gutten.csv")
> str(gutten)

'data.frame':        1287 obs. of  8 variables:
 $ Site    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Location: int  1 1 1 1 1 1 1 1 1 1 ...
 $ Tree    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Age.base: int  10 20 30 40 50 60 70 80 90 100 ...
 $ Height  : num  1.2 4.2 9.3 14.9 19.7 23 25.8 27.4 28.8 30 ...
 $ Diameter: num  NA 4.6 10.2 14.9 18.3 20.7 22.6 24.1 25.5 26.5 ...
 $ Volume  : num  0.3 5 38 123 263 400 555 688 820 928 ...
 $ Age.bh  : num  NA 9.67 19.67 29.67 39.67 ...
```

We check for missing values.

```
> colSums(is.na(gutten))

    Site Location     Tree Age.base   Height Diameter   Volume
       0        0        0        0        0       87        6
  Age.bh
      87
```

We will focus on fitting diameter growth models, and we should remove the unwanted columns and rows. They will cause problems later, otherwise.

```
> gutten <- gutten[!is.na(gutten$Diameter), c("Site",
+     "Location", "Tree", "Diameter", "Age.bh")]
> names(gutten) <- c("site", "location", "tree", "dbh.cm",
+     "age.bh")
> gutten$site <- factor(gutten$site)
> gutten$location <- factor(gutten$location)
```

We should also construct a unique site/location/tree identifier.

```
> gutten$tree.ID <- interaction(gutten$site, gutten$location,
+     gutten$tree)
```

The lattice package provides us with an easy way to plot the data. (Figure 11.1). Note that these are not particularly large trees!

```
> require(lattice)
```

```
> xyplot(dbh.cm ~ age.bh | tree.ID, type = "l", data = gutten)
```

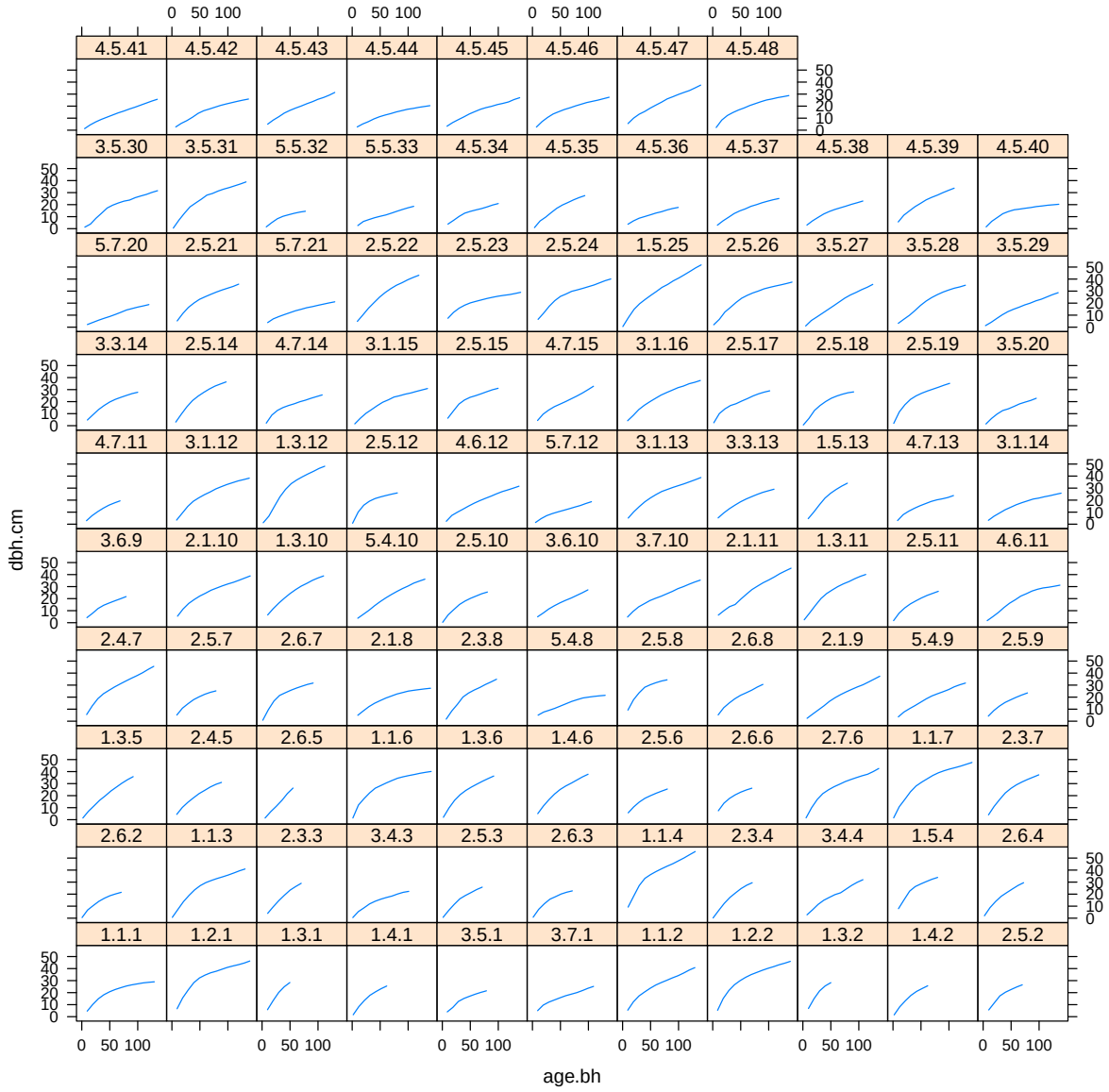

Figure 11.1: Guttenberg's diameter data.

After inspecting Figure 11.1, we shall use a simple non-linear model that passes through the origin and provides an asymptote. Passing through the origin is reasonable as a starting point because, by definition, dbh is zero until age at breast height is greater than 0. That may prove to be an ugly constraint however. The model is then

$$y_i = \phi_1 \times [1 - \exp(-\exp(\phi_2)\, x)] + \epsilon_{it} \tag{11.1}$$

where ϕ_1 is the fixed, unknown asymptote and ϕ_2 is the fixed, unknown scale (see Pinheiro and Bates, 2000), and the time until the tree reaches half its predicted maximum size $t_{0.5}$, is $t_{0.5} = \log 2 / \exp \phi_2$.

NB: it may seem peculiar that we adopt this model for which one of the parameters has a nice interpretation but the other is only a function of a quantity that has a nice interpretation. We choose this version because the parameterization of a non-linear model intimately affects its fittability[1]. Some parameterizations are very difficult to fit around; Schabenberger and Pierce (2002) have a nice discussion and demonstration of this point, and see also Ratkowsky (1983). The complicated part about it is that the quality of the parameterization also depends on the data, so a version of a model what works well for one dataset may work very poorly for another.

In order to fit a non-linear model we generally need to construct our own non-linear model function. We can do this simply by constructing a function that produces the appropriate predictions given the inputs. However, fitting the model turns out to be more efficient if we can also pass the first, and maybe second derivatives to the function as well. Writing a function to pass the predictions and the derivatives is simplified by using the `deriv()` function. NB: R provides some simple prepackaged models, about which we learn more later.

Our use of `deriv()` requires three things: a statement of the function, a vector of parameter names for which derivatives are required, and the template of the function call.

```
> diameter.growth <- deriv(~ asymptote * (1 - exp(-exp(scale) * x)),
+                          c("asymptote","scale"),
+                          function(x, asymptote, scale){})
```

11.1.1 One tree

Having written the non-linear model as a function, we should try it out. We select a handy tree:

```
> handy.tree <- gutten[gutten$tree.ID == "1.1.1", ]
```

In a departure from our earlier chapters, we also need to guess the starting estimates for the parameters. It's a very modest pain in this case. Here we will use the highest value as an estimate of the asymptote, and guess that the tree reaaches about half its maximum diameter by about 30 years.

```
> max(handy.tree$dbh.cm)

[1] 29

> log(log(2)/30)

[1] -3.76771

> handy.nls <- nls(dbh.cm ~ diameter.growth(age.bh, asymptote, scale),
+                  start = list(asymptote=29, scale=-3.76771),
+                  data = handy.tree)
```

We inspect the model object the usual way.

```
> summary(handy.nls)

Formula: dbh.cm ~ diameter.growth(age.bh, asymptote, scale)

Parameters:
          Estimate Std. Error t value Pr(>|t|)
asymptote  31.0175     0.4182   74.16 3.33e-16 ***
scale      -3.8366     0.0344 -111.52  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4507 on 11 degrees of freedom

Number of iterations to convergence: 4
Achieved convergence tolerance: 4.943e-07
```

[1] Pardon me!

Apart from the fact that we can't guarantee to have found a global minimum, these are the least-squares estimates. We should try a range of other starting points to be sure that we are happy with these.

However, it might also be of interest[2] to make inference on these estimates. What is their underlying population distribution? We can learn more about that with the handy `profile()` function, which provides insight as to the shape of the parameter space on to which our data have been projected. Ideally we want the distribution of the parameter estimates to be approximately normal; in order for this to be true the objective function should be approximately quadratic close to the nominated minimum.

```
> opar <- par(mfrow=c(2,2), mar=c(4,4,2,1), las=1)
> plot(fitted(handy.nls), handy.tree$dbh.cm,
+      xlab = "Fitted Values", ylab = "Observed Values")
> abline(0, 1, col="red")
> plot(fitted(handy.nls), residuals(handy.nls, type="pearson"),
+      xlab = "Fitted Values", ylab = "Standardized Residuals")
> abline(h=0, col="red")
> plot(profile(handy.nls), conf=0.95)
> par(opar)
```

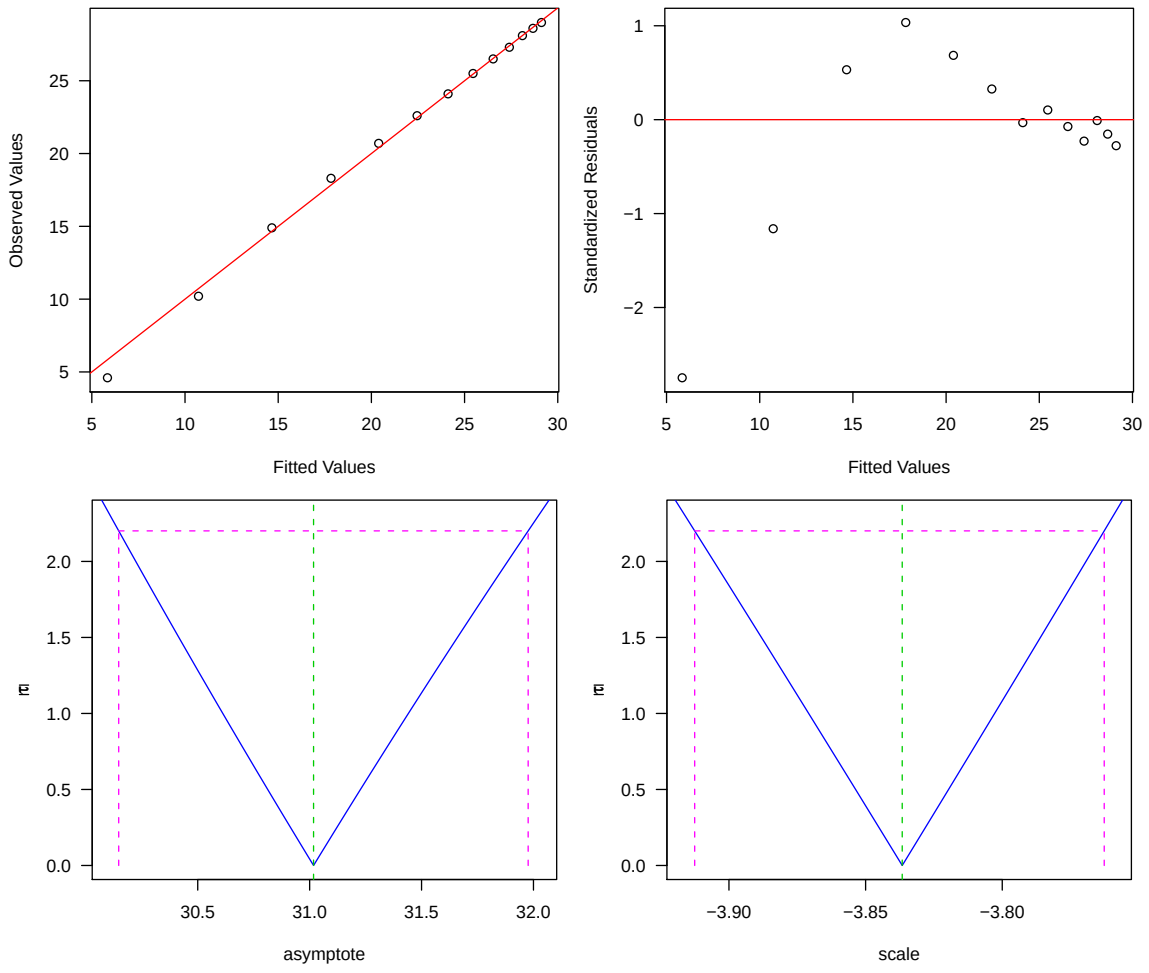

Figure 11.2: Diagnostic and profile plots for simple asymptotic model with handy tree.

Figure 11.2 provides a useful graphical summary of the model performance, combining some information about the residuals with some profile plots. The top-left frame is a plot of the fitted values against

[2] or it might not.

the observed values, with a $x = y$ line inposed. We'd like the points to be close to that line, and to not show fanning or curvature. The top right panel is a plot of the standardized residuals against the fitted values, with a $y = 0$ line imposed. As with OLS regression, we want to see no particular pattern, and no influential outliers. The bottom panels show the profile plots, one for each parameter estimate. These graphs provide information about the acceptability of the assumption of normality on the underlying distribution of the parameter estimates. Our ideal scenario is for the blue solid lines to be straight, and the vertical pink dashed lines to meet the x-axis at approximately the large-sample 95% confidence interval. We can quickly compute the large-sample estimates using:

```
> (my.t <- qt(0.975, summary(handy.nls)$df[2]))

[1] 2.200985

> coef(summary(handy.nls))[,1:2] %*% matrix(c(1,-my.t,1,my.t), nrow=2)

               [,1]      [,2]
asymptote 30.096977 31.938015
scale     -3.912319 -3.760880
```

and extract the profiled versions using:

```
> confint(handy.nls)

               2.5%     97.5%
asymptote 30.147020 31.976346
scale     -3.912546 -3.762685
```

Our diagnostics throw no substantial doubt on the validity of our model. But, the data appear to have a modest kink that our model failed to capture. This might be true just of this tree, or a subset of the trees, or of all the trees. We are naturally curious, but understandably reluctant to go through the same process for every tree. We will need some more general tools.

11.2 Splines

We previously adopted the traditional approach to choosing and fitting models for which a linear model seems unsuitable. Underlying our approach has been the unstated assumption that we are interested in knowing the values of certain parameters, in our case the asymptote and the scale. What if we were only interested in prediction? That is, what if our goal were to use the model for its outputs only, and the parameter estimates, whatever they be, are irrelevant? Under these circumstances it might make more sense to use a spline, or an additive model.

11.2.1 Cubic Spline

The `bs()` function produces the model matrix for a cubic spline, using B-spline basis. The B-spline basis is appealing because the functions are local, inasumuch as they are zero in areas where they are not wanted. See Wood (2006) for a very readable explanation[3].

Here we use the function to fit a linear model that represents a cubic spline with one knot (`df` − `degree` = 1). Note that we set the complexity of the model by this means.

```
> require(splines)
> handy.spline <- lm(dbh.cm ~ bs(age.bh, df = 4, degree = 3),
+     data = handy.tree)
> summary(handy.spline)

Call:
lm(formula = dbh.cm ~ bs(age.bh, df = 4, degree = 3), data = handy.tree)

Residuals:
```

[3] Actually this is a great book for many reasaons, not least amongst which is the authors clarity and willingness to drop into diagrams at any time.

```
      Min        1Q    Median        3Q       Max
-0.136378 -0.053526  0.009092  0.045355  0.096675

Coefficients:
                                Estimate Std. Error t value
(Intercept)                      4.55464    0.07893   57.71
bs(age.bh, df = 4, degree = 3)1 13.01407    0.17698   73.54
bs(age.bh, df = 4, degree = 3)2 20.74721    0.17750  116.88
bs(age.bh, df = 4, degree = 3)3 23.85037    0.16071  148.40
bs(age.bh, df = 4, degree = 3)4 24.43626    0.10916  223.85
                                Pr(>|t|)
(Intercept)                     9.03e-12 ***
bs(age.bh, df = 4, degree = 3)1 1.30e-12 ***
bs(age.bh, df = 4, degree = 3)2 3.21e-14 ***
bs(age.bh, df = 4, degree = 3)3 4.75e-15 ***
bs(age.bh, df = 4, degree = 3)4  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.08531 on 8 degrees of freedom
Multiple R-squared: 0.9999,        Adjusted R-squared: 0.9999
F-statistic: 2.406e+04 on 4 and 8 DF,  p-value: 2.386e-16
```

The model diagnostics are reported in Figure 11.3.

```
> opar <- par(las=1, mfrow=c(2,2), mar=c(4,4,2,2))
> plot(dbh.cm ~ age.bh, data=handy.tree)
> curve(predict(handy.spline, data.frame(age.bh=x)),
+       from=10, to=140, add=TRUE)
> plot(handy.spline, which=c(1,2,5))
> par(opar)
```

Overall the model looks good, as we might expect, with only one point of particularly high influence: 2.

```
> handy.tree["2", ]

  site location tree dbh.cm age.bh tree.ID
2    1        1    1    4.6   9.67   1.1.1
```

The first point measured. Otherwise the model seems fine, but the arbirtrary nature the selection of the knot count is a bit of a concern.

11.2.2 Additive Model

The cubic splines are sufficiently flexible that the cautious analyst will worry about the possibility of overfitting. It seems natural to try to rein in the number of knots, or degrees of freedom, that are available to the model. For example, we could fit a sequence of models, and compare them by some means, with a penalty for curviness. R has such a solution in the `mgcv` package (Wood, 2006), among others. This packages uses so-called generalized cross-validation (GCV) to select the complexity of the model, and conditional on the conplexity, reports suitable fit statistics.

GCV represents the expected prediction error variance, that is, the variance of the errors of predictions of unobserved points. *Loosely* speaking, the square root of the GCV statistic is in the units of the response variable, here, cm.

```
> library(mgcv)

This is mgcv  1.5-5 . For overview type 'help("mgcv-package")'.

> handy.gam <- gam(dbh.cm ~ s(age.bh), data = handy.tree)
> summary(handy.gam)
```

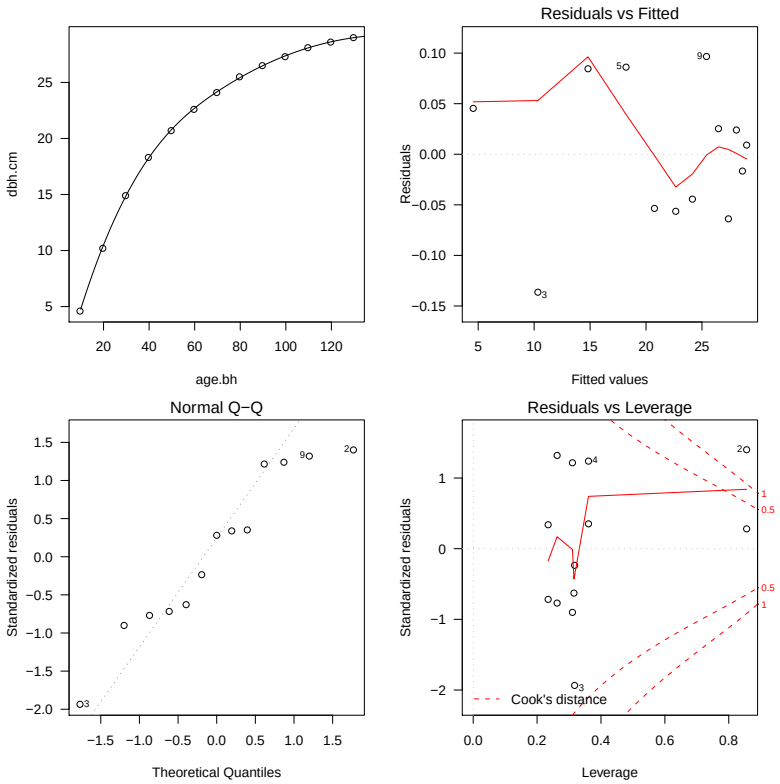

Figure 11.3: Graphical summary of the spline fit.

```
Family: gaussian
Link function: identity

Formula:
dbh.cm ~ s(age.bh)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 21.56923    0.01325    1628 2.47e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:
            edf Ref.df     F p-value
s(age.bh) 8.168  8.168 37580 3.9e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) =      1   Deviance explained =  100%
GCV score = 0.0077406  Scale est. = 0.0022818  n = 13
```

```
> sqrt(summary(handy.gam)$sp.criterion)

[1] 0.08798043
```

Figure 11.4 provides a summary of the model structure. The top left graph shows the predicted curve, *with a two standard-error shaded polygon superimposed*. The other three are standard regression diagnostic plots.

```
> par(las = 1, mfrow = c(2, 2), mar = c(4, 4, 2, 1))
> plot(handy.gam, rug = FALSE, shade = TRUE)
```

```
> plot(predict(handy.gam), residuals(handy.gam))
> abline(h = 0, col = "springgreen3")
> qqnorm(residuals(handy.gam))
> qqline(residuals(handy.gam), col = "springgreen3")
> scatter.smooth(predict(handy.gam), sqrt(abs(residuals(handy.gam))))
```

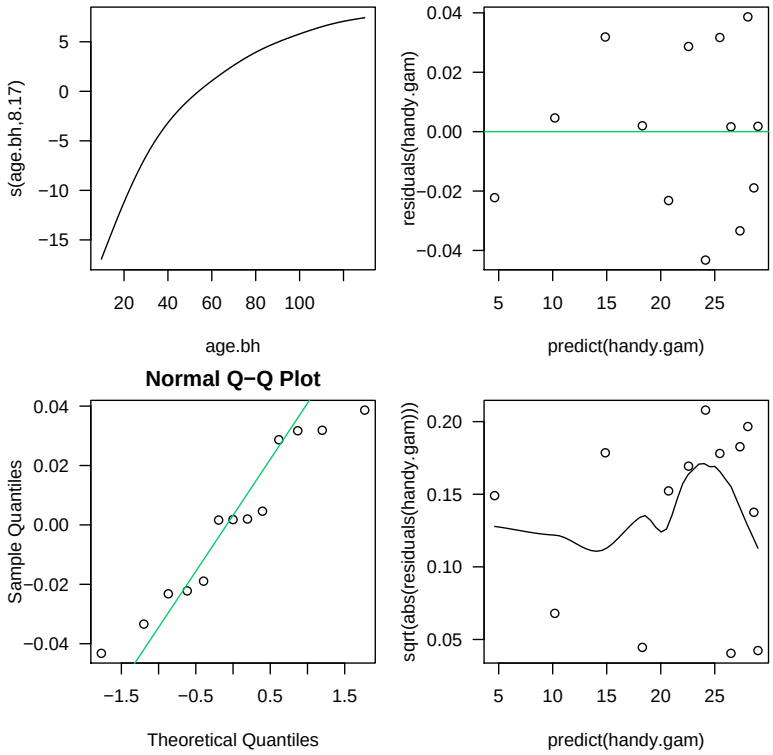

Figure 11.4: Diagnostic plots for generalized additive model for predicting diameter from age on a handy tree.

Chapter 12

Hierarchical Models

We now shift to the analysis of hierarchical data using mixed-effects models. These models are a natural match for many problems that occur commonly in natural resources.

12.1 Introduction

Recall that for fitting a linear regression using the ordinary techniques that you might be familiar with, you were required to make some assumptions about the nature of the residuals. Specifically, it was necessary to assume that the residuals were

1. independent
2. identically distributed, and, more often than not,
3. normally distributed.

The assumption of constant variance (homoscedasticity) lives in the identically distributed assumption (point 2, above). If these assumptions are true, or even defensible, then life is fine. However, more often than not, we know they're not true. This can happen in natural resources data collections because the data may have a temporal structure, a spatial structure, or a hierarchical structure, or all three[1]. That structure may or may not be relevant to your scientific question, but it's very relevant to the data analysis and modelling!

I mention several references. None are mandatory to purchase or read, but all will be useful at some point or other. They are mentioned in *approximate* order of decreasing utility *for this level*.

12.1.1 Methodological

Pinheiro and Bates (2000) details model fitting in R and Splus, which both provide first-rate graphical model discrimination and assessment tools through the libraries written by the authors. Good examples are provided. Schabenberger and Pierce (2002) is a treasure-trove of sensible advice on understanding and fitting the generalized and mixed-effects models that form the core of this class. There are copious worked examples and there is plenty of SAS code. You are welcome to interact with the material however you see fit. Finally, Fitzmaurice et al. (2004) and Gelman and Hill (2007) do a first-rate job of explaining the practice of fitting mixed-effects models.

In addition to these books, there are numerous articles that try to explain various elements of these topics in greater or lesser detail. In the past I have found Robinson (1991) (no relation!) and the discussions that follow it particularly useful.

12.1.2 General

Venables and Ripley (2002) is a must-have if you're interested in working with R or Splus. The three previous editions are now legendary in the R/S community for their thorough explication of modern statistical practice, with generous examples and code. The R community has also generated some excellent start-up documents. These are freely available for download at the R project website:

[1] "The first law of ecology is that everything is related to everything else." – Barry Commoner, US biologist/environmentalist. The Closing Circle: Nature, Man, and Technology. New York : Knopf, 1971.

`http://www.r-project.org`. Download and read any or all of them, writing code all the way. If you're interested in a deeper exploration of the programming possibilities in R or S then Venables and Ripley (2000) is very useful. Some larger-scale projects that I have been involved with have required calling C programs from R; this reference was very helpful then.

12.2 Some Theory

Mixed effects models contain both fixed and random effects. The model structure is usually suggested by the underlying design or structure of the data. I like to claim that random effects are suggested by the design of a study, and fixed effects are suggested by the hypotheses, but this is not always true.

12.2.1 Effects

"Effects" are predictor variables in a linear or non-linear model[2]. The discussion of fixed and random *effects* can get a little confusing. "Random" and "fixed" aren't normally held to be opposite to one another, or even mutually exclusive (except by sheer force of habit!). Why not "stochastic" and "deterministic"? Or, "sample" and "population"? Or, "local" and "global"? These labels might tell more of the story. Gelman and Hill (2007) decline to use random and fixed altogether.

There are different ways to look at these two properties. Unfortunately, it does affect the data analysis and the conclusions that can be drawn. Modellers may disagree on whether effects should be fixed or random, and the same effect can switch depending on circumstances. Certainly, statisticians haven't agreed on a strategy. Some will claim that it depends entirely on the inference, and some that it depends entirely on the design.

As the statistical tools that are used to analyze such data become more sophisticated, and models previously unthinkable become mainstream, the inadequacies of old vocabularies are increasingly obvious.

Random effects

Random effects are those whose levels are supposedly sampled randomly from a range of possible levels. Generally, although not always, when random effects are considered it is of interest to connect the results to the broader population. That is, the levels are assumed to be collectively representative of a broader class of potential levels, about which we wish to say something. Alternatively, one might say that a random effect is simply one for which the estimates of location are not of primary interest. Another alternative is that one might say that a random effect is one that you wish to marginalize, for whatever reason.

Fixed effects

Fixed effects are generally assumed to be purposively selected, and represent nothing other than themselves. If an experiment were to be repeated, and the exact same levels of an experimental effect were purposively produced, then the effect is fixed. However, some effects which might vary upon reapplication may also be fixed, so this is not definitive. Alternatively, one might say that a fixed effect is simply one for which the estimates of location are of first interest. Another alternative is that one might say that a fixed effect is one that you wish to condition on, for whatever reason.

Mixed-up effects

Some variables do not lend themselves to easy classification, and either knowledge of process and/or an epistemological slant is required. These are common in natural resources. For example, if an experiment that we feel is likely to be affected by climate is repeated over a number of years, would *year* be a fixed or a random effect? It is not a random sample of possible years, but the same years would not recur if the experiment were repeated. Likewise the replication of an experiment at known locations: some would claim that these should be a fixed effect, others that they represent environmental variation, and therefore they can be considered a random effect.

[2] The use of the label "effects" is a hang-over from experimental design, and no longer really suits the application, but that's how inertia goes.

12.2.2 Model Construction

The process of model construction becomes much more complex now. We have to balance different approaches and assumptions, each of which carries different implications for the model and its utility. If we think about the process of fitting an ordinary regression as being like a flow chart, then adding random effects adds a new dimension to the flow chart altogether. Therefore it's very important to plan the approach before beginning.

The number of potential strategies is as varied as the number of models we can fit. Here is one that we will rely on in our further examples.

1. Choose the minimal set of fixed and random effects for the model.
   (a) Choose the fixed effects that must be there. These effects should be such that, if they are not in the model, the model has no meaning.
   (b) Choose the random effects that must be there. These effects should be such that if they are not in the model, then the model will not adequately reflect the design.

   This is the baseline model, to which others will be compared.
2. Fit this model to the data using tools yet to be discussed, and check the assumption diagnostics. Iterate through the process of improving the random effects, including:
   (a) a heteroskedastic variance structure (several candidates)
   (b) a correlation structure (several candidates)
   (c) extra random effects (e.g. random slopes)
3. When the diagnostics suggest that the fit is reasonable, consider adding more fixed effects. At each step, re-examine the diagnostics to be sure that any estimates that you will use to assess the fixed effects are based on a good match between the data, model, and assumptions.

A further layer of complexity is that it may well be that the assumptions will not be met in the absence of certain fixed effects or random effects. In this case, a certain amount of iteration is inevitable.

It is important to keep in mind that the roles of the fixed and the random effects are distinct. Fixed effects *explain* variation. Random effects *organize* unexplained variation. At the end of the day you will have a model that surficially seems worse than an simple linear regression, by most metrics of model quality. Our goal is to find a model/assumption combination that matches the diagnostics that we examine. Adding random effects adds information, and improves diagnostic compatibility, but explains no more variation!

The bottom line is that the goal of the analyst is to find the simplest model that satisfies the necessary regression assumptions and answers the questions of interest. It is tempting to go hunting for more complex random effects structures, which may provide a higher maximum likelihood, but if the simple model satisfies the assumptions and answers the questions then maximizing the likelihood further is a mathematical exercise - not a statistical one.

Example

In order to illuminate some of these questions, consider the Grand fir stem analysis data. These data are plotted in Figures 12.1 and 12.2.

```
> rm(list = ls())
> stage <- read.csv("../data/stage.csv")
> stage$Tree.ID <- factor(stage$Tree.ID)
> stage$Forest.ID <- factor(stage$Forest, labels = c("Kaniksu",
+     "Coeur d'Alene", "St. Joe", "Clearwater", "Nez Perce", "Clark Fork",
+     "Umatilla", "Wallowa", "Payette"))
> stage$HabType.ID <- factor(stage$HabType, labels = c("Ts/Pach",
+     "Ts/Op", "Th/Pach", "AG/Pach", "PA/Pach"))
> stage$dbhib.cm <- stage$Dbhib * 2.54
> stage$height.m <- stage$Height / 3.2808399
```

These habitat codes refer to the climax tree species, which is the most shade-tolerant species that can grow on the site, and the dominant understorey plant, respectively. Ts refers to *Thuja plicata* and *Tsuga heterophylla*, Th refers to just *Thuja plicata*, AG is *Abies grandis*, PA is *Picea engelmanii* and *Abies lasiocarpa*, Pach is *Pachistima myrsinites*, and Op is the nasty *Oplopanaz horridurn*. Grand fir is considered a major climax species for AG/Pach, a major seral species for Th/Pach and PA/Pach, and a minor seral species for Ts/Pach and Ts/Op. Loosely speaking, a community is *seral* if there is evidence that at least some of the species are temporary, and *climax* if the community is self-regenerating Daubenmire (1952).

```
> opar <- par(las=1)
> plot(stage$dbhib.cm, stage$height.m, xlab="Dbhib (cm)", ylab="Height (m)")
> par(opar)
```

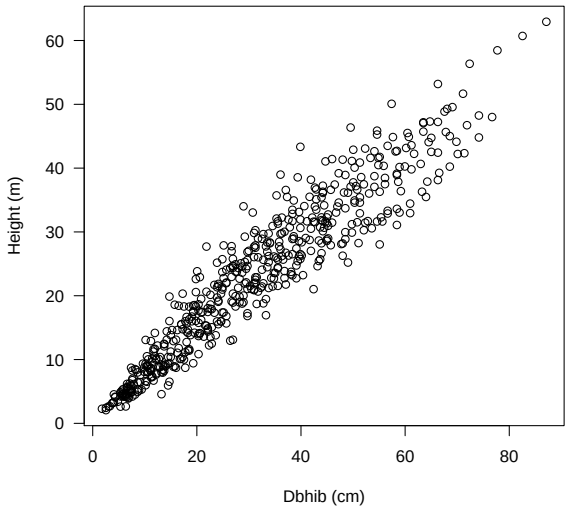

Figure 12.1: Al Stage's Grand Fir stem analysis data: height (ft) against diameter (in). These were dominant and co-dominant trees.

An earlier version of this document required a reasonably fussy nested loop to produce Figure 12.2, which I include below for reference, and portability.

```
> colours <- c("deepskyblue","goldenrod","purple",
+              "orangered2","seagreen")
> par(mfrow=c(3,3), pty="m", mar=c(2, 2, 3, 1) + 0.1, las=1)
> for (i in 1:length(levels(stage$Forest.ID))) {
+   thisForest <- levels(stage$Forest.ID)[i]
+   forestData <- stage[stage$Forest.ID==thisForest,]
+   plot(stage$dbhib.cm, stage$height.m, xlab = "", ylab = "",
+        main = thisForest, type="n")
+   theseTrees <- factor(forestData$Tree.ID)
+   legend("topleft",
+          unique(as.character(forestData$HabType.ID)),
+          xjust=0, yjust=1, bty="n",
+          col=colours[unique(forestData$HabType)],
+          lty=unique(forestData$HabType)+1)
+   for (j in 1:length(levels(theseTrees))) {
+     thisTree <- levels(theseTrees)[j]
+     lines(forestData$dbhib.cm[forestData$Tree.ID==thisTree],
+           forestData$height.m[forestData$Tree.ID==thisTree],
+           col=colours[forestData$HabType[forestData$Tree.ID==thisTree]],
```

```
+           lty=forestData$HabType[forestData$Tree.ID==thisTree]+1)
+   }
+ }
> mtext("Height (m)", outer=T, side=2, line=2)
> mtext("Diameter (cm)", outer=T, side=1, line=2)
```

However, application of Hadley Wickham's powerful ggplot2 package simplifies this challenge. Thanks are due to Charlotte Wickham for the code.

```
> library(ggplot2)

> qplot(dbhib.cm, height.m,
+       data = stage, group = Tree.ID,
+       geom = "line",
+       facets = ~ Forest.ID,
+       colour = HabType.ID,
+       linetype = HabType.ID,
+       xlab = "Diameter (cm)", ylab = "Height (m)"
+       ) +
+   scale_colour_manual(name = "Habitat Type",
+                       values = c("deepskyblue","goldenrod",
+                                  "purple","orangered2",
+                                  "seagreen")) +
+   scale_linetype_manual(name = "Habitat Type", values = c(1,2,4:6))
```

12.2.3 Dilemma

An easy way to approach the advantages to modelling that are offered by mixed-effects models is to think about a simple example. Imagine that we are interested in constructing a height-diameter relationship using two randomly selected plots in a forest, and that we have measured three trees on each. It turns out that the growing conditions are quite different on the plots, leading to a systematic difference between the height–diameter relationship on each (Figure 12.3).

The model is:

$$y_i = \beta_0 + \beta_1 \times x_i + \epsilon_i \tag{12.1}$$

where β_0 and β_1 are fixed but unknown population parameters, and ϵ_i are residuals. The following assumptions are required:

- The true relationship is linear.
- $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
- The ϵ_i are independent.

```
> trees <- data.frame(plot=factor(c(1, 1, 1, 2, 2, 2)),
+                     dbh.cm=c(30, 32, 35, 30, 33, 35),
+                     ht.m=c(25, 30, 40, 30, 40, 50))

> plot(trees$dbh.cm, trees$ht.m, pch=c(1, 19)[trees$plot], xlim=c(29, 36),
+      xlab="Diameter (cm)", ylab="Height (m)")
> abline(lm(ht.m ~ dbh.cm, data=trees), col="darkgrey")
```

If we fit a simple regression to the trees then we obtain a residual/fitted value plot as displayed in Figure 12.4).

```
> case.model.1 <- lm(ht.m ~ dbh.cm, data=trees)
> plot(fitted(case.model.1), residuals(case.model.1), ylab="Residuals",
+             xlab="Fitted Values", pch=c(1, 19)[trees$plot])
> abline(h=0, col="darkgrey")
```

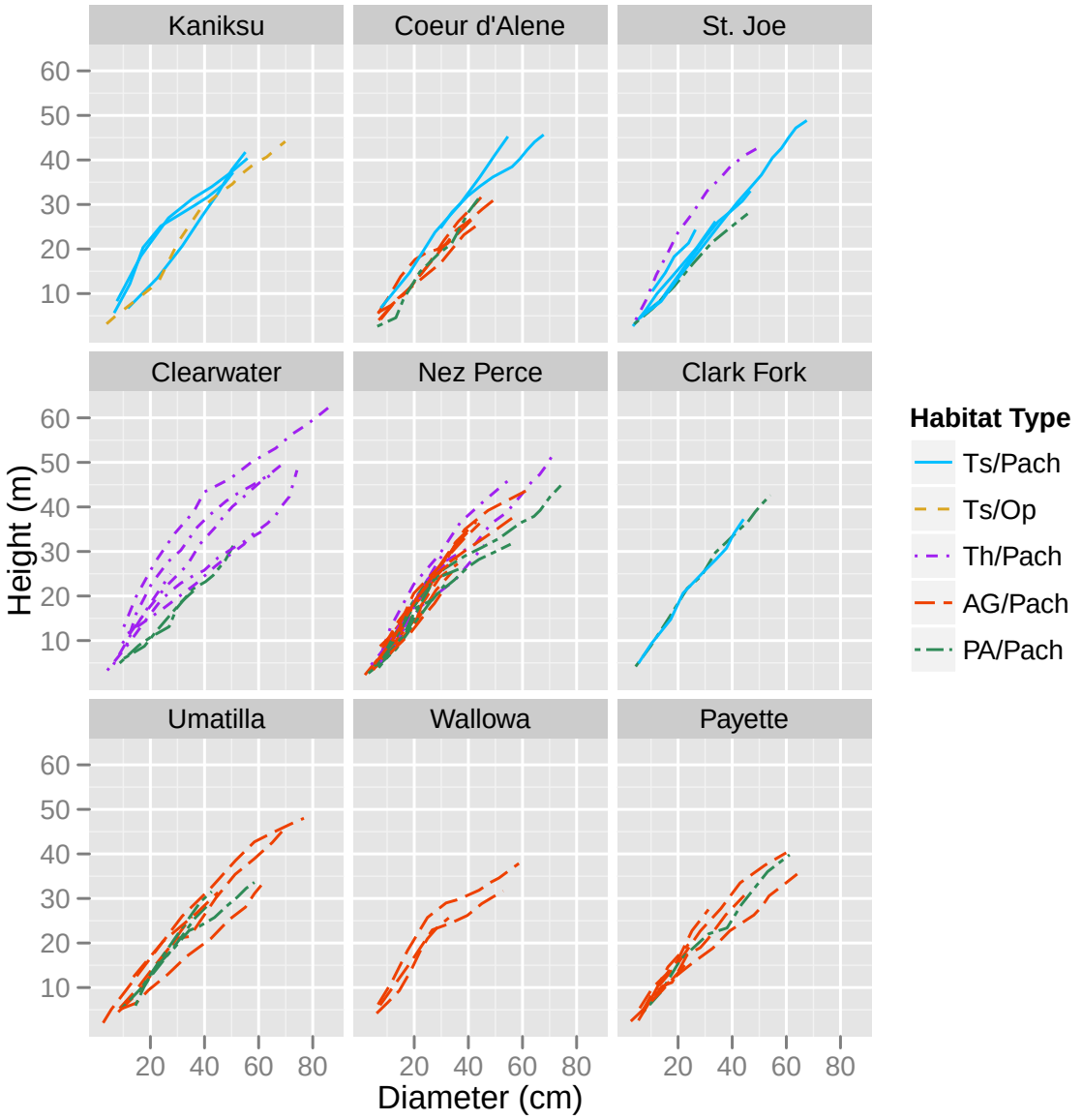

Figure 12.2: Al Stage's Grand Fir Stem Analysis Data: height (ft, vertical axes) against diameter (inches, horizontal axes) by national forest. These were dominant and co-dominant trees.

If we fit a simple regression to the trees with an intercept for each plot then we obtain a residual/fitted value plot as displayed in Figure 12.5.

```
> case.model.2 <- lm(ht.m ~ dbh.cm*plot, data=trees)
> plot(fitted(case.model.2), residuals(case.model.2),
+      xlab="Fitted Values", ylab="Residuals",
+      pch=c(1, 19)[trees$plot])
> abline(h=0, col="darkgrey")
```

These three figures (12.3—12.5) show the analyst's dilemma. The residuals in Figure 12.4 clearly show a correlation within plot; the plot conditions dictate that all the trees in the plot will have the same sign of residual. This phenomenon is not seen in Figure 12.5. However, the model described by Figure 12.5 has no utility, because in order to use it we have to choose whether the tree belongs to plot 1 or plot 2. If the tree belongs to neither, which is true of all of the unmeasured trees, then the model can make no prediction. So, the dilemma is: we can construct a useless model that satisfies the regression assumptions or a useful model that does not.

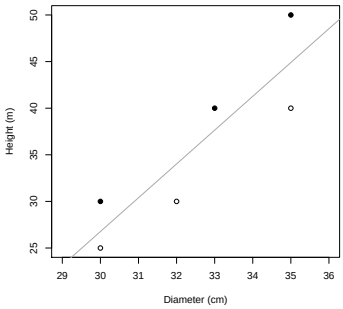

Figure 12.3: Height-diameter measures for three trees on two plots (full and outline symbols), with line of least squares regression.

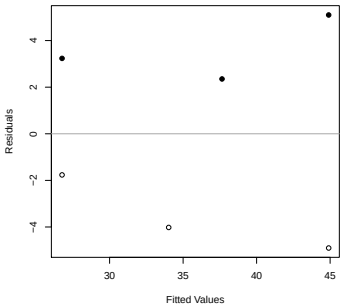

Figure 12.4: Residual plot for height-diameter model for three trees on two plots (full and outline symbols).

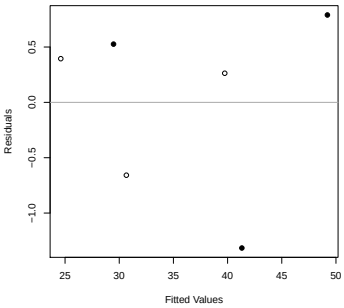

Figure 12.5: Residual plot for height-diameter model including plot for three trees on two plots (full and outline symbols).

12.2.4 Decomposition

The dilemma documented in section 12.2.3 has at least two solutions. One is the use of mixed-effects models, the other, which we do not cover here, is explicit modeling of the correlation structure using generalized least squares.

The mixed effects models approach is to decompose the unknown variation into smaller pieces, each of which themselves satisfies the necessary assumptions. Imagine that we could take the six residual values presented in Figure 12.4, which have the plot-level correlation structure, and decompose them into two plot-level errors and size within-plot errors. That is, instead of:

$$y_{ij} - \hat{y}_{ij} = \hat{\epsilon}_{ij} \tag{12.2}$$

we could try:

$$y_{ij} - \hat{y}_{ij} = \hat{b}_i + \hat{\epsilon}_{ij} \tag{12.3}$$

Then we merely need to assume that:

- The true relationship is linear.
- $b_i \sim \mathcal{N}(0, \sigma_b^2)$
- $\epsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$
- The ϵ_{ij} are independent.

However, when the time comes to use the model for prediction, we do not need to know the plot identity, as the fixed effects do not require it.

This example illustrates the use of random effects. Random effects do not explain variation, that is the role of the fixed effects. Random effects organize variation, or enforce a more complex structure upon it, in such a way that a match is possible between the model assumptions and the diagnostics. In fact, we would expect the overall uncertainty, measured as root mean squared error, to increase any time we fit in any way other than by least squares.

12.3 A Simple Example

We start with a very simple and abstract example. First we have to load the package that holds the mixed-effects code, `nlme`.

```
> library(nlme)
> library(lattice)
```

Now, we generate a simple dataset.

```
> straw <- data.frame(y = c(10.1, 14.9, 15.9, 13.1, 4.2, 4.8, 5.8, 1.2),
+                     x = c(1, 2, 3, 4, 1, 2, 3, 4),
+                     group = factor(c(1, 1, 1, 1, 2, 2, 2, 2)))
```

Let's plot the data (Figure 12.6).

```
> colours = c("red", "blue")
> plot(straw$x, straw$y, col = colours[straw$group])
```

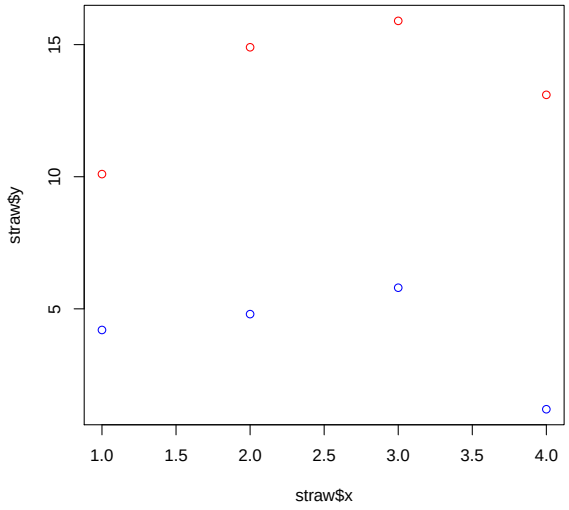

Figure 12.6: A simple dataset to show the use of mixed-effects models.

For each model below, examine the output using `summary()` commands of each model, and try to ascertain what the differences are between the models, and whether increasing the complexity seems to be worthwhile. Use `anova()` commands for the latter purpose.

Ordinary Least Squares

This model is just trying to predict y using x. Using algebra, we would write

$$y_i = \beta_0 + \beta_1 \times x_i + \epsilon_i \tag{12.4}$$

where β_0 and β_1 are fixed but unknown population parameters, and ϵ_i are residuals. The following assumptions are required:

- True relationship is linear.
- $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
- The ϵ_i are independent.

Note that the values of β_0 and β_1 that minimize the residual sum of squares are the least squares estimates in any case, and are unbiased if assumption 1 is true. If assumptions 2 and 3 are true as well then the estimates also have other desirable properties.

The model is fit using R with the following code:

```
> basic.1 <- lm(y ~ x, data=straw)
```

Let's let each `group` have its own intercept. In algebra,

$$y_i = \beta_{01} + \beta_{02} \times g_i + \beta_1 \times x_i + \epsilon_i \tag{12.5}$$

where β_{01}, β_{02}, and β_1 are fixed but unknown population parameters, g_i is an indicator variable with value 0 for group 1 and 1 for group 2, and ϵ_i are residuals. The same assumptions are required as for model 12.4.

The model is fit using R with the following code:

```
> basic.2 <- lm(y ~ x + group, data=straw)
```

Let's let each `group` have its own intercept *and* slope.

$$y_i = \beta_{01} + \beta_{02} \times g_i + (\beta_{11} + \beta_{12} \times g_i) \times x_i + \epsilon_i \tag{12.6}$$

where β_{01}, β_{02}, β_{12}, and β_{12} are fixed but unknown population parameters, g_i is an indicator variable with value 0 for group 1 and 1 for group 2, and ϵ_i are residuals. The same assumptions are required as for model 12.4.

The model is fit using R with the following code:

```
> basic.3 <- lm(y ~ x * group, data=straw)
```

Mixed Effects

Now we need to convert the data to a grouped object - a special kind of dataframe that allows special `nlme()` commands. The `group` will hereby be a random effect. One command that we can now use is `augPred`, as seen below. Try it on a few models.

```
> straw.mixed <- groupedData(y ~ x | group, data=straw)
```

Now let's fit the basic mixed-effects model that allows the intercepts to vary randomly between the groups. We'll add a subscript for clarity.

$$y_{ij} = \beta_0 + b_{0i} + \beta_1 \times x_{ij} + \epsilon_{ij} \tag{12.7}$$

where β_0 and β_1 are fixed but unknown population parameters, the b_{0i} are two group-specific random intercepts, and ϵ_{ij} are residuals. The following assumptions are required:

- True relationship is linear.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_0})$
- $\epsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$
- The ϵ_{ij} are independent.

The model is fit using R with the following code:

```
> basic.4 <- lme(y ~ x, random = ~1 | group, data = straw.mixed)
```

The `random` syntax can be a little confusing. Here, we're instructing R to let each group have its own random intercept. If we wanted to let each group have its own slope and intercept, we would write `random = x`. If we wanted to let each group have its own slope but not intercept, we would write `random = x - 1`.

We can examine the model in a useful graphic called an *augmented prediction plot*. This plot provides a scatterplot of the data, split up by group, and a fitted line which represents the model predictions (Figure 12.7). We should also check the regression diagnostics that are relevant to our assumptions, but we have so few data here that examples aren't useful. We will develop these ideas further during the case study, to follow.

```
> plot(augPred(basic.4))
```

If we are satisfied with the model diagnostics then we can examine the structure of the model, including the estimates, using the `summary()` function. The summary function presents a collection of useful information about the model. Here we report the default structure for a `summary.lme` object. The structure may change in the future, but the essential information will likely remain the same.

```
> summary(basic.4)
```

Firstly, the `data.frame` object is identified, and fit statistics reported, including Akaike's "An Information Criterion", Schwartz's "Bayesian Information Criterion", and the log likelihood.

```
Linear mixed-effects model fit by REML
 Data: straw.mixed
       AIC      BIC    logLik
  43.74387 42.91091 -17.87193
```

The random effects structure is then described, and estimates are provided for the parameters. Here we have an intercept for each group, and the standard deviation is reported, as well as the standard deviation of the residuals within each group.

```
Random effects:
 Formula: ~1 | group
        (Intercept) Residual
StdDev:    6.600769 2.493992
```

The fixed effects structure is next described, in a standard t-table arrangement. Estimated correlations between the fixed effects follow.

```
Fixed effects: y ~ x
            Value Std.Error DF   t-value p-value
(Intercept)   8.5  5.142963  5 1.6527437  0.1593
x             0.1  0.788670  5 0.1267958  0.9040
 Correlation:
  (Intr)
x -0.383
```

The distribution of the within-group residuals, also called the *innermost residuals* in the context of strictly hierarchical models by Pinheiro and Bates (2000), is then described.

```
Standardized Within-Group Residuals:
       Min         Q1        Med         Q3        Max
-1.2484738 -0.4255493  0.1749470  0.6387985  1.0078956
```

Finally the hierarchical structure of the model and data is presented.

```
Number of Observations: 8
Number of Groups: 2
```

Next let's fit a unique variance to each group. The model form will be the same as in equation 12.7, but the assumptions will be different. Now, we will need to assume that

- True relationship is linear.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_1})$
- $\epsilon_{1j} \sim \mathcal{N}(0, \sigma^2_{b01})$
- $\epsilon_{2j} \sim \mathcal{N}(0, \sigma^2_{b02})$
- $Cov(\epsilon_{ab}, \epsilon_{cd}) = 0$ for $a \neq c$ or $d \neq d$

The model is fit using R with the following code:

```
> basic.5 <- lme(y ~ x, random = ~1 | group,
+                weights = varIdent(form = ~1 | group),
+                data = straw.mixed)
```

The summary output is essentially identical to the previous in structure, with the addition of a new section that summarizes the newly-added variance model. Here we show only the new portion.

```
> summary(basic.5)
```

```
Variance function:
 Structure: Different standard deviations per stratum
 Formula: ~1 | group
 Parameter estimates:
       2        1
1.000000 1.327843
```

Finally let's allow for temporal autocorrelation within each group. Again, the model form will be the same as in equation 12.7, but the assumptions will be different. Now, we will need to assume that

- True relationship is linear.
- $b_{0i} \sim \mathcal{N}(0, \sigma^2_{b_1})$
- $\epsilon_{1j} \sim \mathcal{N}(0, \sigma^2_{b01})$
- $\epsilon_{2j} \sim \mathcal{N}(0, \sigma^2_{b02})$
- $Cov(\epsilon_{ab}, \epsilon_{ac}) = \rho$ for $b \neq c$
- The ϵ_{ij} are independent otherwise.

The model is fit using R with the following code:

```
> basic.6 <- lme(y ~ x, random = ~1 | group,
+                weights = varIdent(form = ~1 | group),
+                correlation = corAR1(),
+                data = straw.mixed)
```

The summary output is again essentially identical to the previous in structure, with the addition of a new section that summarizes the newly-added correlation model. Here we show only the new portion.

```
> summary(basic.6)

Correlation Structure: AR(1)
 Formula: ~1 | group
 Parameter estimate(s):
      Phi
0.8107325
```

We can summarize some of these differences in a graph (Figure 12.8).

```
> opar <- par(las=1)
> colours <- c("blue", "darkgreen", "plum")
> plot(straw$x, straw$y)
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+                      fitted(basic.1)[straw$group ==
+                                      levels(straw$group)[g]],
+                      col = colours[1])
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+                      fitted(basic.2)[straw$group ==
+                                      levels(straw$group)[g]],
+                      col = colours[2])
> for (g in 1:2) lines(straw$x[straw$group == levels(straw$group)[g]],
+                      fitted(basic.4)[straw$group ==
+                                      levels(straw$group)[g]],
+                      col = colours[3])
> legend(2.5, 13, lty = rep(1, 3), col = colours,
+        legend = c("Mean Only", "Intercept Fixed", "Intercept Random"))
> par(opar)
```

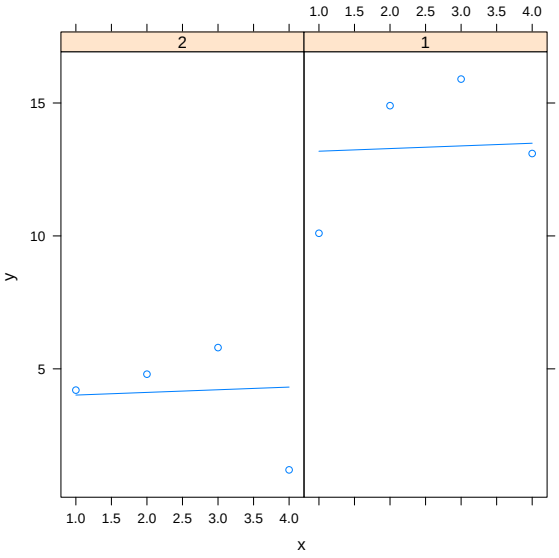

Figure 12.7: An augmented plot of the basic mixed-effects model with random intercepts fit to the sample dataset.

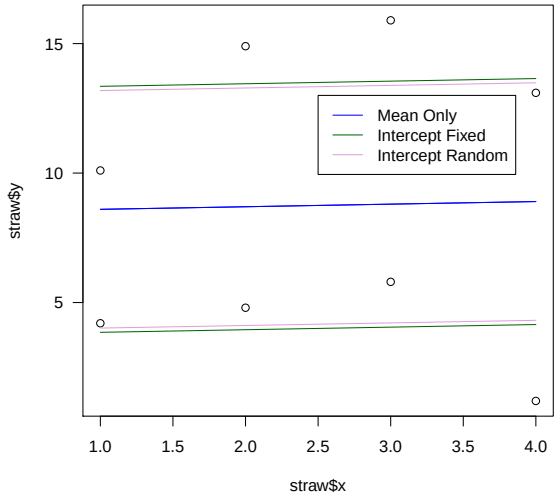

Figure 12.8: A sample plot showing the difference between basic.1 (single line), basic.2 (intercepts are fixed), and basic.4 (intercepts are random).

12.3.1 The Deep End

There are numerous different representations of the linear mixed-effects model. We'll adopt that suggested by Laird and Ware (1982):

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

Here, $\mathbf{D}$ and $\mathbf{R}$ are preferably constructed using a small number of parameters, which will be estimated from the data. We'll think first about estimation using maximum likelihood.

12.3.2 Maximum Likelihood

Recall that the principle behind maximum likelihood was to find the suite of parameter estimates that were best supported by the data. This began by writing down the conditional distribution of the observations. For example the *pdf* for a single observation from the normal distribution is:

$$f\left(y_i \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(y_i-\mu)^2}{2\sigma^2}}$$

So if $Y \overset{\mathrm{d}}{=} \mathrm{N}(\mu, \mathbf{V})$ then by definition:

$$f\left(\mathbf{Y} \mid \mu, \mathbf{V}\right) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}-\mu)'\mathbf{V}^{-1}(\mathbf{Y}-\mu)}$$

So in terms of the linear model $\mathbf{Y} = \mathbf{X}\beta$, the conditional joint density is

$$f\left(\mathbf{Y} \mid \mathbf{X}, \beta, \mathbf{V}\right) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}-\mathbf{X}\beta)'\mathbf{V}^{-1}(\mathbf{Y}-\mathbf{X}\beta)}$$

Reversing the conditioning and taking logs yields:

$$\mathcal{L}\left(\beta, \mathbf{V} \mid \mathbf{Y}, \mathbf{X}\right) = -\frac{1}{2}\ln\left(|\mathbf{V}|\right) - \frac{n}{2}\ln\left(2\pi\right) - \frac{1}{2}\left(\mathbf{Y}-\mathbf{X}\beta\right)'\mathbf{V}^{-1}\left(\mathbf{Y}-\mathbf{X}\beta\right) \tag{12.8}$$

Notice that the parameters we're interested in are now embedded in the likelihood. Solving for those parameters should be no more difficult than maximizing the likelihood. In theory. Now, to find $\hat{\beta}$ we take the derivative of $\mathcal{L}(\beta, \mathbf{V} \mid \mathbf{y}, \mathbf{X})$ with regards to β:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\beta} = \frac{\mathrm{d}}{\mathrm{d}\beta}\left[-\frac{1}{2}(\mathbf{y}-\mathbf{X}\beta)'\mathbf{V}^{-1}(\mathbf{y}-\mathbf{X}\beta)\right]$$

this leads to, as we've seen earlier

$$\hat{\beta}_{MLE} = \left(\mathbf{X}'\mathbf{V}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}'\mathbf{V}^{-1}\mathbf{Y}$$

but this only works *if we know* $\mathbf{V}$!

Otherwise, we have to maximize the likelihood as follows. First, substitute

$$(\mathbf{X}'\mathbf{V}^{-1}\mathbf{X})^{-1}\mathbf{X}'\mathbf{V}^{-1}\mathbf{Y}$$

for β in the likelihood. That is, remove all the instances of β, and replace them with this statement. By this means, β is *profiled out* of the likelihood. The likelihood is now only a function of the data and the covariance matrix V. This covariance matrix is itself a function of the covariance matrices of the random effects, which are structures that involve hopefully only a few unknown parameters, and that are organized by the model assumptions.

Maximize the resulting likelihood in order to estimate $\hat{V}$, and then calculate the estimate of the fixed effects via:

$$\hat{\beta}_{MLE} = \left(\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{Y} \tag{12.9}$$

After some tedious algebra, which is well documented in Schabenberger and Pierce (2002), we also get the $BLUP$s.

$$\hat{b}_{MLE} = \mathbf{D}\mathbf{Z}'\hat{\mathbf{V}}\left(\mathbf{Y}-\mathbf{X}\hat{\beta}\right) \tag{12.10}$$

where $\mathbf{D}$ is the covariance matrix of the random effects.

12.3.3 Restricted Maximum Likelihood

It was noted earlier that maximum likelihood estimators of covariance parameters are usually negatively biased. This is because in profiling out the fixed effects, we're effectively pretending that we know them, and therefore we are not reducing the degrees of freedom appropriately. *Restricted* or *Residual* Maximum Likelihood will penalize the estimation based on the model size, and is therefore a preferred strategy. $ReML$ is not unbiased, except under certain circumstances, but it is less biased than maximum likelihood.

Instead of maximizing the conditional joint likelihood of $\mathbf{Y}$ we do so for an almost arbitrary linear transformation of $\mathbf{Y}$, which we shall denote $\mathbf{K}$. It is almost arbitrary inasmuch as there are only two constraints: $\mathbf{K}$ must have full column rank, or else we would be creating observations out of thin air, and $\mathbf{K}$ must be chosen so that $\mathrm{E}[\mathbf{K}'\mathbf{Y}] = 0$.

The easiest way to guarantee that these hold is to ensure that $[\mathbf{K}'\mathbf{X}] = 0$, and that $\mathbf{K}$ has no more than $n-p$ independent columns, where p is the number of independent parameters in the model. Notice that we would like $\mathbf{K}$ to have as many columns as it can because this will translate to more realizations for fitting the model. This removes the fixed effects from consideration and in so doing also penalizes the estimation for model size. So, the likelihood is restricted by the fixed effects being set to 0, thus, restricted maximum likelihood. Finally, notice that having a $\mathbf{0}$ column in $\mathbf{K}$ doesn't actually add any information to the problem.

So, briefly, $ReML$ involves applying ML, but replacing $\mathbf{Y}$ with $\mathbf{KY}$, $\mathbf{X}$ with $\mathbf{0}$, $\mathbf{Z}$ with $\mathbf{K}'\mathbf{Z}$, and $\mathbf{V}$ with $\mathbf{K}'\mathbf{V}\mathbf{K}$.

12.4 Case Study

Recall our brutal exposition of the mixed-effects model:

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

$\mathbf{D}$ and $\mathbf{R}$ are covariance matrices, constructed using a small number of parameters, and the structure of which is *suggested* by what is known about the data and can be *tested* by comparing nested models.

12.4.1 Stage Data

A brief synopsis: a sample of 66 trees was selected in national forests around northern and central Idaho. According to Stage (*pers. comm.* 2003), the trees were selected purposively rather than randomly. Stage (1963) notes that the selected trees "... appeared to have been dominant throughout their lives" and "... showed no visible evidence of crown damage, forks, broken tops, etc." The habitat type and diameter at 4'6" were also recorded for each tree, as was the national forest from which it came. Each tree was then split, and decadal measures were made of height and diameter inside bark at 4'6".

First, eyeball the data in your spreadsheet of choice. Then import the data as follows:

```
> rm(list = ls())
> stage <- read.csv("../data/stage.csv")
> dim(stage)

[1] 542   7

> names(stage)

[1] "Tree.ID" "Forest"  "HabType" "Decade"  "Dbhib"   "Height"
[7] "Age"

> sapply(stage, class)

  Tree.ID    Forest   HabType    Decade     Dbhib    Height
"integer" "integer" "integer" "integer" "numeric" "numeric"
      Age
"integer"
```

Some cleaning will be necessary. Let's start with the factors.

```
> stage$Tree.ID <- factor(stage$Tree.ID)
> stage$Forest.ID <- factor(stage$Forest, labels = c("Kan",
+     "Cd'A", "StJoe", "Cw", "NP", "CF",
+     "Uma", "Wall", "Ptte"))
> stage$HabType.ID <- factor(stage$HabType, labels = c("Ts/Pach",
+     "Ts/Op", "Th/Pach", "AG/Pach", "PA/Pach"))
```

The measurements are all imperial (this was about 1960, after all).

```
> stage$dbhib.cm <- stage$Dbhib * 2.54
> stage$height.m <- stage$Height / 3.2808399
> str(stage)

'data.frame':        542 obs. of  11 variables:
 $ Tree.ID   : Factor w/ 66 levels "1","2","3","4",..: 1 1 1 1 1 2 2 2 2 2 ...
 $ Forest    : int  4 4 4 4 4 4 4 4 4 4 ...
 $ HabType   : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Decade    : int  0 1 2 3 4 0 1 2 3 4 ...
 $ Dbhib     : num  14.6 12.4 8.8 7 4 20 18.8 17 15.9 14 ...
```

```
 $ Height    : num  71.4 61.4 40.1 28.6 19.6 ...
 $ Age       : int  55 45 35 25 15 107 97 87 77 67 ...
 $ Forest.ID : Factor w/ 9 levels "Kan","Cd'A","StJoe",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ HabType.ID: Factor w/ 5 levels "Ts/Pach","Ts/Op",..: 5 5 5 5 5 5 5 5 5 5 ...
 $ dbhib.cm  : num  37.1 31.5 22.4 17.8 10.2 ...
 $ height.m  : num  21.76 18.71 12.22 8.72 5.97 ...
```

Height from Diameter

The prediction of height from diameter provides useful and inexpensive information. It may be that the height/diameter relationship differs among habitat types, or climate zones, or tree age. Let's examine the height/diameter model of the trees using a mixed-effects model. We'll start with a simple case, using only the oldest measurement from each tree that provides one.

```
> stage.old <- stage[stage$Decade == 0, ]
```

Note that this code actually drops a tree, but we can afford to let it go for this demonstration.

To establish a baseline of normalcy, let's first fit the model using ordinary least squares. We drop the sole observation from the Ts/Op habitat type. It will cause trouble otherwise (the leverage will prove to be very high).

```
> hd.lm.1 <- lm(height.m ~ dbhib.cm * HabType.ID,
+         data = stage.old,
+         subset = HabType.ID !=  "Ts/Op")
```

Formally, I think it is good practice to examine the diagnostics upon which the model is predicated *before* examining the model itself, tempting though it may be, so see Figure 12.9. The graph of the residuals vs. fitted values plot (top left) seems good. There is no suggestion of heteroskedasticity. The Normal Q-Q plot suggests a little wiggle but seems reasonably straight. There seem to be no points of egregious influence (bottom left; all Cook's Distances < 1).

```
> opar <- par(mfrow=c(2,2), mar=c(4, 4, 4, 1))
> plot(hd.lm.1)
> par(opar)
```

So, having come this far, we should examine the model summary.

```
> summary(hd.lm.1)

Call:
lm(formula = height.m ~ dbhib.cm * HabType.ID, data = stage.old,
    subset = HabType.ID != "Ts/Op")

Residuals:
     Min       1Q   Median       3Q      Max
-10.3210  -2.1942   0.2218   1.7992   7.9437

Coefficients:
                            Estimate Std. Error t value Pr(>|t|)
(Intercept)                  8.33840    4.64118   1.797   0.0778
dbhib.cm                     0.58995    0.08959   6.585 1.67e-08
HabType.IDTh/Pach            2.42652    5.78392   0.420   0.6764
HabType.IDAG/Pach            0.29582    5.13564   0.058   0.9543
HabType.IDPA/Pach            0.02604    5.96275   0.004   0.9965
dbhib.cm:HabType.IDTh/Pach -0.03224    0.10670  -0.302   0.7637
dbhib.cm:HabType.IDAG/Pach -0.08594    0.10116  -0.850   0.3992
dbhib.cm:HabType.IDPA/Pach -0.10322    0.11794  -0.875   0.3852

(Intercept)                 .
dbhib.cm                    ***
```

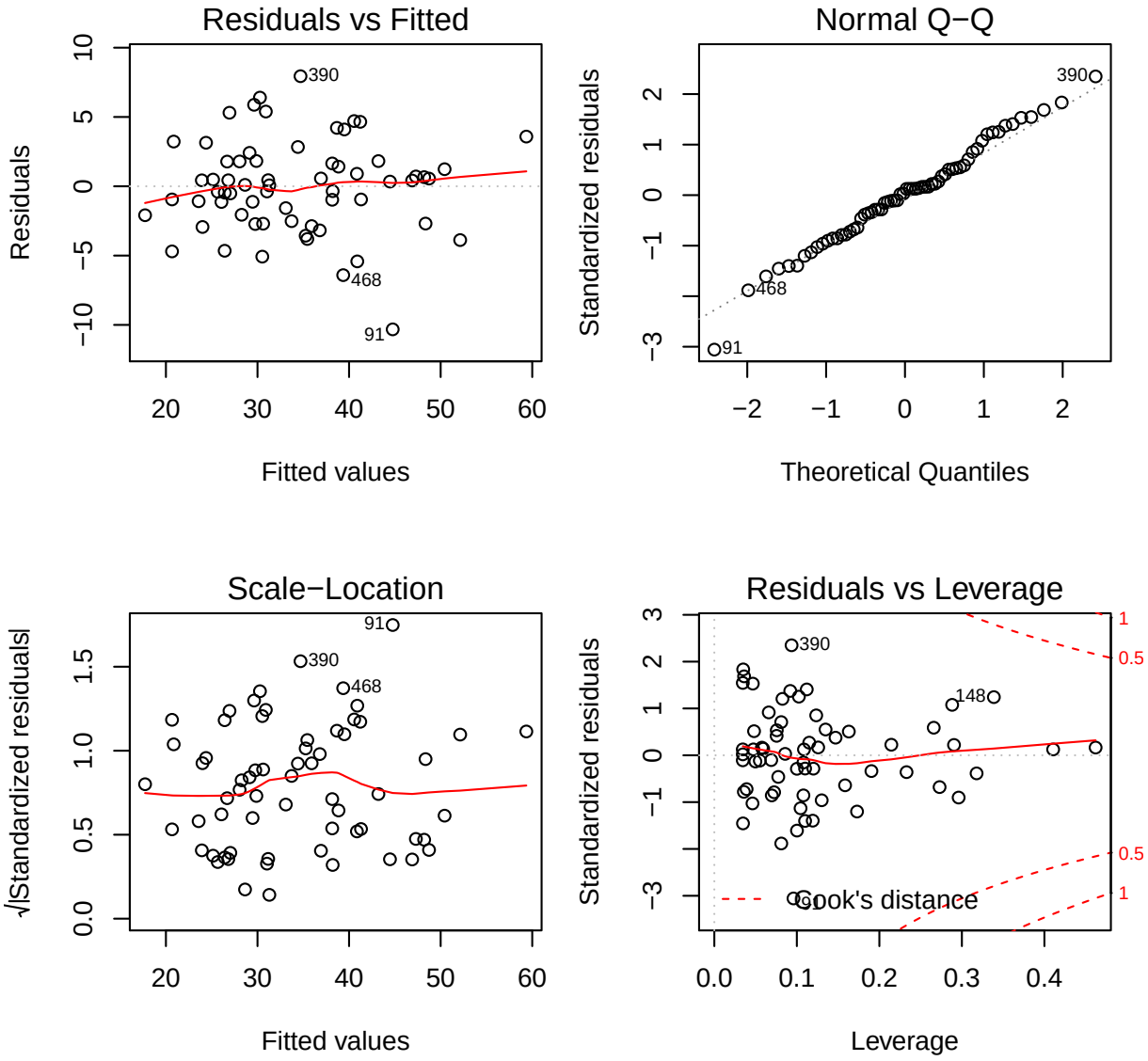

Figure 12.9: Regression diagnostics for the ordinary least squares fit of the Height/Diameter model with habitat type for Stage's data.

```
HabType.IDTh/Pach
HabType.IDAG/Pach
HabType.IDPA/Pach
dbhib.cm:HabType.IDTh/Pach
dbhib.cm:HabType.IDAG/Pach
dbhib.cm:HabType.IDPA/Pach
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.551 on 56 degrees of freedom
Multiple R-squared: 0.8748,        Adjusted R-squared: 0.8591
F-statistic: 55.89 on 7 and 56 DF,  p-value: < 2.2e-16
```

For comparison: the following quantities are in metres. The first is the standard deviation of the height measures. The second is the standard deviation of the height measures conditional on the diameter measures *and* the model.

```
> sd(stage.old$height.m)

[1] 9.468042

> summary(hd.lm.1)$sigma

[1] 3.551062
```

It's also interesting to know how much variation is explained by the habitat type information. We can assess this similarly. Here we will not worry about diagnostics, although it should be done.

```
> summary(lm(height.m ~ dbhib.cm, data = stage.old,
+         subset = HabType.ID !=  "Ts/Op"))$sigma

[1] 4.10135
```

Not much!

Mixed effects

Based on our knowledge of the locations of national forests, it seems reasonable to believe that there will be similarities between trees that grow in the same forest *relative to the population of trees*.

However, we'd like to create a model that doesn't rely on knowing the national forest, that is, a model that can plausibly be used for trees in other forests. This is acceptable as long as we are willing to believe that the sample of trees that we are using is representative of the conditions for which we wish to apply the model. In the absence of other information, this is a judgement call. Let's assume it for the moment.

Then, based on the above information, national forest will be a random effect, and habitat type a fixed effect. That is, we wish to construct a model that can be used for any forest, that might be more accurate if used correctly within a named national forest, and provides unique estimates for habitat type. We can later ask how useful the knowledge of habitat type is, and whether we want to include that in the model.

So, we'll have two random effects: national forest and tree within national forest. We have one baseline fixed effect: diameter at breast height inside bark, with to potential additions: age and habitat type. The lowest-level sampling unit will be the tree, nested within national forest.

It is convenient to provide a basic structure to R. The structure will help R create useful graphical diagnostics later in the analysis. Note that you only need to call a package once per session, but it doesn't hurt to do it more often. Here we will scatter them liberally to remind you what packages you should be using.

```
> library(nlme)
> stage.old <- groupedData(height.m ~ dbhib.cm | Forest.ID,
+                          data = stage.old)
```

Now, let's look to our model.

$$y_{ij} = \beta_0 + b_{0i} + \beta_1 \times x_{ij} + \epsilon_{ij} \tag{12.11}$$

y_{ij} is the height of tree j in forest i, x_{ij} is the diameter of the same tree. β_0 and β_1 are fixed but unknown parameters and b_{0i} are the forest-specific random and unknown intercepts. Later we might see if the slope also varies with forest. So, in matrix form,

$$Y = \beta\mathbf{X} + b\mathbf{Z} + \epsilon \tag{12.12}$$

Y is the column of tree heights, X will be the column of diameters, with a matrix of 0s and 1s to allocate the observations to different habitat types, along with a column for the ages, if necessary. β will be a vector of parameter estimates. Z will be a matrix of 0s and 1s to allocate the observations to different forests. b will be a vector of means for the forests and trees within forests. Finally, we'll let $\mathbf{D}$ be a 9×9 identity matrix multiplied by a constant σ_h^2, and $\mathbf{R}$ be a 66×66 identity matrix multiplied by a constant σ^2.

```
> hd.lme.1 <- lme(height.m ~ dbhib.cm, random = ~1 | Forest.ID,
+       data = stage.old)
```

Automatic functions are available to extract and plot the different pieces of the model. I prefer to extract them and choose my own plotting methods. I recommend that you do the same. For the pre-programmed versions see Pinheiro and Bates (2000).

A quick burst of jargon: for hierarchical models there is more than one level of fitted values and residuals. Pinheiro and Bates (2000) adopt the following approach: the outermost residuals and fitted values are conditional only on the fixed effects, the innermost residuals and fitted values are conditional on the fixed and all the random effects, and there are as many levels between these extremes as are necessary. So, in a two-level model like this,

- the outermost residuals are the residuals computed from the outermost fitted values, which are computed from only the fixed effects. Let's refer to them as r_0.

$$r_0 = y_{ij} - \hat{\beta}_0 - \hat{\beta}_1 \times xij \tag{12.13}$$

- the innermost residuals are the residuals computed from the innermost fitted values, which are computed from the fixed effects and the random effects. Let's refer to them as r_1.

$$r_1 = y_{ij} - \hat{\beta}_0 - \hat{b}_{0i} - \hat{\beta}_1 \times xij \tag{12.14}$$

Furthermore, the mixed-effects apparatus provides us with three kinds of innermost and outermost residuals:

1. *response* residuals, simply the difference between the observation and the prediction;
2. *Pearson* residuals, which are the response residuals scaled by dividing by their standard deviation; and
3. *normalized* residuals, which are the Pearson residuals pre-multiplied by the inverse square-root of the estimated correlation matrix from the model.

The key assumptions that we're making for our model are that:

1. the model structure is correctly specified;
2. the random effects are normally distributed;
3. the innermost residuals are normally distributed;
4. the innermost residuals are homoscedastic within and across the groups; and
5. the innermost residuals are independent within the groups.

Notice that we're not making any assumptions about the outermost residuals. However, they are useful for summarizing the elements of model performance.

We should construct diagnostic graphs to check these assumptions. Note that in some cases, the assumptions are stated in an untenably broad fashion. Therefore the sensible strategy is to check for the conditions that can be interpreted in the context of the design, the data, and the incumbent model. For example, there are infinite ways that the innermost residuals could fail to have constant variance. What are the important ways? The situation most likely to lead to problems is if the variance of the residuals is a function of something, whether that be a fixed effect or a random effect.

Rather than trust my ability to anticipate what the programmers meant by the labels etc., I want to know what goes into each of my plots. The best way to do that is to put it there myself. To examine each of the assumptions in turn, I have constructed the following suite of graphics. These are presented in Figure 12.10.

1. A plot of the outermost fitted values against the observed values of the response variable. This graph allows an overall summary of the explanatory power of the model.
   (a) How much of the variation is explained?
   (b) How much remains?
   (c) Is there evidence of lack of fit anywhere in particular?

2. A plot of the innermost fitted values against the innermost Pearson residuals. This graph allows a check of the assumption of correct model structure.
   (a) Is there curvature?
   (b) Do the residuals fan out?
3. a qq-plot of the estimated random effects, to check whether they are normally distributed with constant variance.
   (a) Do the points follow a straight line, or do they exhibit skew or kurtosis?
   (b) Are any outliers evident?
4. a qq-plot of the Pearson residuals, to check whether they are normally distributed with constant variance.
   (a) Do the points follow a straight line, or do they exhibit skew or kurtosis?
   (b) Are any outliers evident?
5. a notched boxplot of the innermost Pearson residuals by the grouping variable, to see what the within-group distribution looks like.
   (a) Do the notches intersect 0?
   (b) Is there a trend between the medians of the within-group residuals and the estimated random effect?
6. a scatterplot of the variance of the Pearson residuals within the forest against the forest random effect.
   (a) Is there a distinct positive or negative trend?

NB: I do not tend to use any of the widely-available statistical tests for homoskedasticity, normality, etc, for diagnostics. I like (Box, 1953): "... to make preliminary tests on variances is rather like putting to sea in a rowing boat to find out whether conditions are sufficiently calm for an ocean liner to leave port".

We use the following code to produce Figure 12.10. Of course there is no need to pack all the graphical diagnostics into one figure.

```
> opar <- par(mfrow = c(3, 2), mar = c(4, 4, 3, 1), las = 1, cex.axis = 0.9)
> #### Plot 1
> plot(fitted(hd.lme.1, level=0), stage.old$height.m,
+      xlab = "Fitted Values (height, m.)",
+      ylab = "Observed Values (height, m.)",
+      main = "Model Structure (I)")
> abline(0, 1, col = "blue")
> #### Plot 2
> scatter.smooth(fitted(hd.lme.1), residuals(hd.lme.1, type="pearson"),
+                xlab = "Fitted Values",
+                ylab = "Innermost Residuals",
+                main = "Model Structure (II)")
> abline(h = 0, col = "red")
> #### Plot 3
> ref.forest <- ranef(hd.lme.1)[[1]]
> ref.var.forest <- tapply(residuals(hd.lme.1, type="pearson", level=1),
+                      stage.old$Forest.ID,  var)
> qqnorm(ref.forest, main="Q-Q Normal - Forest Random Effects")
> qqline(ref.forest, col="red")
> #### Plot 4
> qqnorm(residuals(hd.lme.1, type="pearson"), main="Q-Q Normal - Residuals")
> qqline(residuals(hd.lme.1, type="pearson"), col="red")
> #### Plot 5
> boxplot(residuals(hd.lme.1, type="pearson", level=1) ~ stage.old$Forest.ID,
```

```
+         ylab = "Innermost Residuals", xlab = "National Forest",
+         notch=T, varwidth = T, at=rank(ref.forest))
> axis(3, labels=format(ref.forest, dig=2), cex.axis=0.8,
+      at=rank(ref.forest))
> abline(h=0, col="darkgreen")
> #### Plot 6
> plot(ref.forest, ref.var.forest, xlab="Forest Random Effect",
+      ylab="Variance of within-Forest Residuals")
> abline(lm(ref.var.forest ~ ref.forest), col="purple")
> par(opar)
```

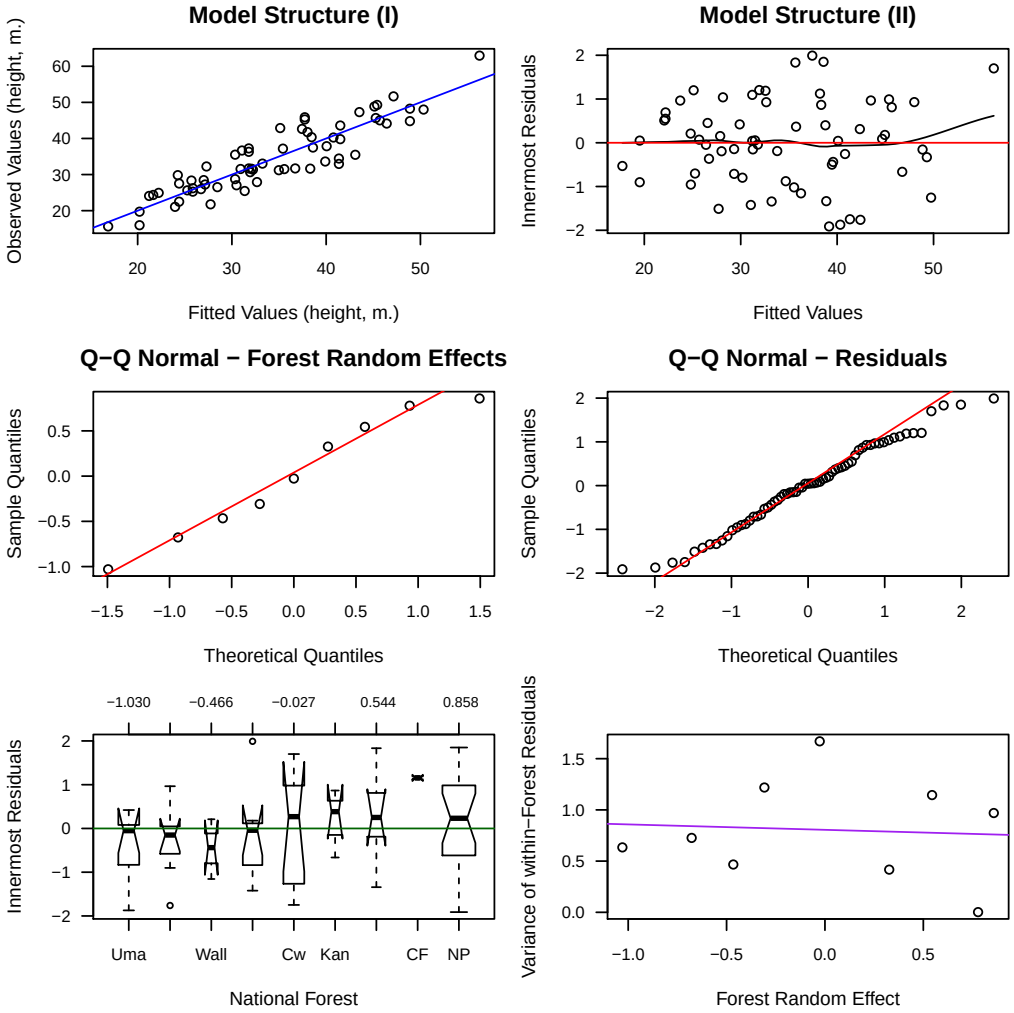

Figure 12.10: Selected diagnostics for the mixed-effects fit of the Height/Diameter ratio against habitat type and national forest for Stage's data.

Cross-reference these against Figure 12.10. In fact, all of these residual diagnostics look good.

But, let's accept the model as it stands for the moment, and go on to examine the summary. Refer to Section 12.3 for a detailed description of the output that follows.

```
> summary(hd.lme.1)

Linear mixed-effects model fit by REML
 Data: stage.old
       AIC     BIC    logLik
  376.6805 385.253 -184.3403
```

```
Random effects:
 Formula: ~1 | Forest.ID
        (Intercept) Residual
StdDev:    1.151405 3.937486

Fixed effects: height.m ~ dbhib.cm
              Value Std.Error DF   t-value p-value
(Intercept) 6.58239 1.7763571 55  3.705556   5e-04
dbhib.cm    0.57036 0.0335347 55 17.008062   0e+00
 Correlation:
         (Intr)
dbhib.cm -0.931

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-1.91215622 -0.70233393  0.04308139  0.81189065  1.99133843

Number of Observations: 65
Number of Groups: 9
```

1. Here we have the overall metrics of model fit, including the log likelihood (recall that this is the quantity we're maximizing to make the fit), and the AIC and BIC statistics. The fixed effects are profiled out of the log-likelihood, so that the log-likelihood is a function only of the data and two parameters: σ_h^2 and σ^2.

2. The formula reminds us of what we asked for: that the forest be a random effect, and that a unique intercept be fit for each level of Forest. The square roots of the estimates of the two parameters are also here.

3. Another metric of model quality is RMSE, which is the estimate of the standard deviation of the response residuals conditional on only the fixed effects. Note that 3.94 is *not* theRMSE, it is instead an estimate of the standard deviation of the response residuals conditional on the fixed and the random effects. Obtaining the RMSE is relatively easy because the random effects and the residuals are assumed to be independent.

$$\text{RMSE} = \sqrt{\sigma_h^2 + \sigma^2} = 4.1$$

   The last metric of model quality we can get here is the intra-class correlation. This is the variance of the random effect divided by the sum of the variances of the random effects and the residuals.

$$\rho = \frac{\sigma_h^2}{\sigma_h^2 + \sigma^2} = 0.0788$$

   so about 7.9 % of the variation in height (that isn't explained by diameter) is explained by forest. Not very much.

4. Now we have a reminder of the fixed effects model and the estimates of the fixed effects. We have several columns:

   (a) the value of the estimate,

   (b) its standard error (not identical here because of the lack of balance),

   (c) the degrees of freedom (simply mysterious for various reasons),

   (d) the t-value associated with the significance test of the null hypothesis that the estimate is 0 against the two-tailed alternative that it is not 0, which is really rather meaningless for this model, and

   (e) the p-value associated with that rather meaningless test.

5. This is the correlation matrix for the estimates of the fixed effects. It is estimated from the design matrix. This comes from the covariance matrix of the fixed effects, which can be estimated by

$$(\mathbf{X}'\mathbf{V}^{-1}\mathbf{X})^{-1}$$

6. Information about the within-group residuals. Are they symmetric? Are there egregious outliers? Compare these values to what we know of the standard normal distribution, for which the median should be about 0, the first quartile at -0.674, and the third at 0.674.

7. And finally, confirmation that we have the correct number of observations and groups. This is a useful conclusion to draw; it comforts us that we fit the model that we thought we had!

A compact summary of the explanatory power of the model can be had from:

```
> anova(hd.lme.1)
```

```
            numDF denDF   F-value p-value
(Intercept)     1    55 2848.4436  <.0001
dbhib.cm        1    55  289.2742  <.0001
```

Deeper design

Let's now treat the Grand fir height/diameter data from Stage (1963) in a different way. We actually have numerous measurements of height and diameter for each tree. It seems wasteful to only use the largest.

Let's still assume that the national forests represent different, purposively selected sources of climatic variation, and that habitat type represents a randomly selected treatment of environment (no, it's probably not true, but let's assume that it is). This is a randomized block design, where the blocks and the treatment effects are crossed. This time we're interested in using all the data. Previously we took only the first measurement. How will the model change? As always, we begin by setting up the data.

```
> library(nlme)
> stage <- groupedData(height.m ~ dbhib.cm | Forest.ID/Tree.ID, data = stage)
```

Let's say that, based on the above information, national forest will be a random effect, and habitat type a candidate fixed effect. So, we'll have anywhere from one to three fixed effects (dbhib, age, and habitat) and two random effects (forest and tree within forest). The response variable will now be the height measurement, nested within the tree, possibly nested within habitat type. Let's assume, for the moment, that the measurements are independent within the tree (definitely not true). Now, let's look to our model. A simple reasonable candidate model is:

$$y_{ijk} = \beta_0 + b_{0i} + b_{0ij} + \beta_1 \times x_{ijk} + \epsilon_{ijk} \tag{12.15}$$

y_{ijk} is the height of tree j in forest i at measurement k, x_{ijk} is the diameter of the same tree. β_0 and β_1 are fixed but unknown parameters, b_{0i} are the forest-specific random and unknown intercepts, and b_{0ij} are the tree-specific random and unknown intercepts. Later we might see if the slope also varies with forest. So, in matrix form, we have:

$$Y = \beta\mathbf{X} + b\mathbf{Z} + \epsilon \tag{12.16}$$

- Y is the vector of height measurements. The basic unit of Y will be a measurement within a tree within a forest. It has 542 observations.
- X will be a matrix of 0s, 1s, and diameters, to allocate the observations to different national forests and different tree diameters at the time of measurement.
- β will be a vector of parameter estimates.
- Z will be a matrix of 0s and 1s to allocate the observations to different forests, and trees within forests.
- b will be a vector of means for the forests and the trees.

- **D** will be a block diagonal matrix comprising a 9×9 identity matrix multiplied by a constant σ_f^2, and then a square matrix for each forest, which will be a diagonal matrix with variances on the diagonals.
- **R** will now be a 542×542 identity matrix multiplied by a constant σ^2.

```
> hd.lme.3 <- lme(height.m ~ dbhib.cm,
+                 random = ~1 | Forest.ID/Tree.ID,
+                 data = stage)
```

Now, the key assumptions that we're making are that:

1. the model structure is correctly specified
2. the tree and forest random effects are normally distributed,
3. the tree random effects are homoscedastic within the forest random effects.
4. the inner-most residuals are normally distributed,
5. the inner-most residuals are homoscedastic within and across the tree random effects.
6. the innermost residuals are independent within the groups.

We again construct diagnostic graphs to check these assumptions. To examine each of the assumptions in turn, I have constructed the earlier suite of graphics, along with some supplementary graphs.

1. an extra qq-plot of the tree-level random effects, to check whether they are normally distributed with constant variance.
   (a) Do the points follow a straight line, or do they exhibit skew or kurtosis?
   (b) Are any outliers evident?
2. a notched boxplot of the tree-level random effects by the grouping variable, to see what the within-group distribution looks like.
   (a) Do the notches intersect 0?
   (b) Is there a trend between the medians of the within-group residuals and the estimated random effect?
3. a scatterplot of the variance of the tree-level random effects within the forest against the forest random effect.
   (a) Is there a distinct positive or negative trend?
4. an autocorrelation plot of the within-tree errors.

As a rule of thumb, we need four plots plus three for each random effect. Cross-reference these against Figures 12.11, 12.12, and 12.13. Each graphic should ideally be examined separately in its own frame. Here's the code:

```
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1,
+         cex.axis = 0.9)
>   plot(fitted(hd.lme.3, level=0), stage$height.m,
+       xlab = "Fitted Values", ylab = "Observed Values",
+       main = "Model Structure (I)")
>   abline(0, 1, col = "gray")
>   scatter.smooth(fitted(hd.lme.3), residuals(hd.lme.3, type="pearson"),
+       main = "Model Structure (II)",
+       xlab = "Fitted Values", ylab = "Innermost Residuals")
>   abline(h = 0, col = "gray")
> acf.resid <- ACF(hd.lme.3, resType = "normal")
> plot(acf.resid$lag[acf.resid$lag < 10.5],
+      acf.resid$ACF[acf.resid$lag < 10.5],
```

```
+      type="b", main="Autocorrelation",
+      xlab="Lag", ylab="Correlation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       -stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> abline(0,0,col="gray")
> par(opar)
```

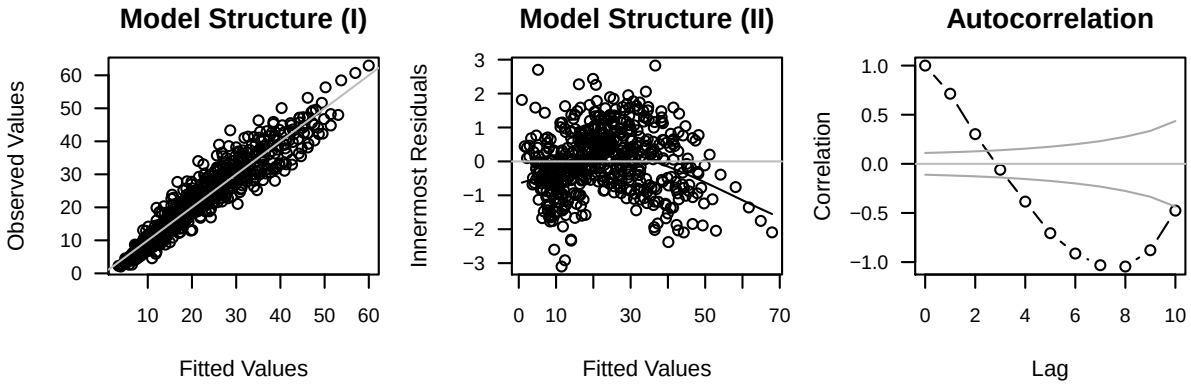

Figure 12.11: Selected overall diagnostics for the mixed-effects fit of the Height and Diameter model for Stage's data.

```
> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1,
+         cex.axis = 0.9)
>   ref.forest <- ranef(hd.lme.3, level=1, standard=T)[[1]]
>   ref.tree <- ranef(hd.lme.3, level=2, standard=T)[[1]]
>   ref.tree.frame <- ranef(hd.lme.3, level=2, augFrame=T, standard=T)
>   ref.var.tree <- tapply(residuals(hd.lme.3, type="pearson", level=1),
+                        stage$Tree.ID,  var)
>   ref.var.forest <- tapply(ref.tree, ref.tree.frame$Forest, var)
>   qqnorm(ref.forest, main = "QQ plot: Forest")
>   qqline(ref.forest)
>   qqnorm(ref.tree, main = "QQ plot: Tree")
>   qqline(ref.tree)
>   qqnorm(residuals(hd.lme.3, type="pearson"), main="QQ plot: Residuals")
>   qqline(residuals(hd.lme.3, type="pearson"), col="red")
> par(opar)
```

```
> opar <- par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1,
+         cex.axis = 0.9)
> boxplot(ref.tree ~ ref.tree.frame$Forest,
+         ylab = "Tree Effects", xlab = "National Forest",
+         notch=T, varwidth = T, at=rank(ref.forest))
> axis(3, labels=format(ref.forest, dig=2), cex.axis=0.8,
+      at=rank(ref.forest))
> abline(h=0, col="darkgreen")
> boxplot(residuals(hd.lme.3, type="pearson", level=1) ~ stage$Tree.ID,
+         ylab = "Innermost Residuals", xlab = "Tree",
+         notch=T, varwidth = T, at=rank(ref.tree))
> axis(3, labels=format(ref.tree, dig=2), cex.axis=0.8,
```

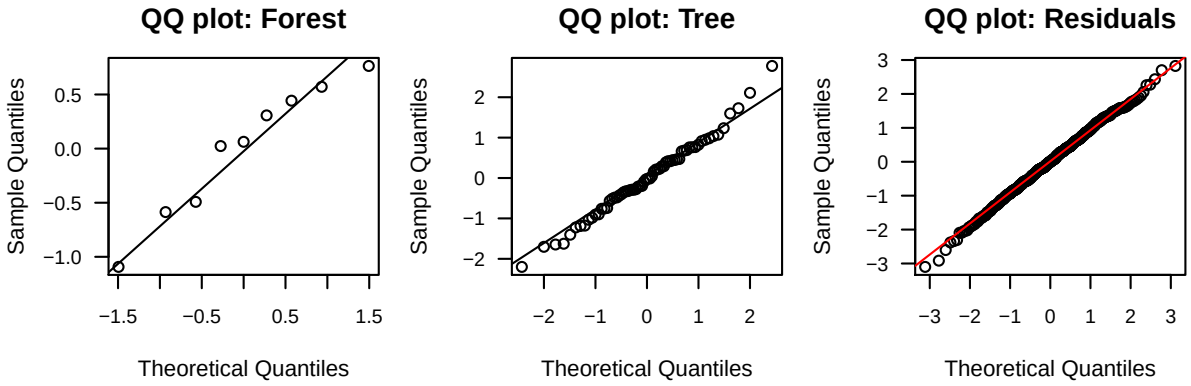

Figure 12.12: Selected quantile-based diagnostics for the mixed-effects fit of the Height and Diameter model for Stage's data.

```
+      at=rank(ref.tree))
> abline(h=0, col="darkgreen")
> plot(ref.forest, ref.var.forest, xlab="Forest Random Effect",
+      ylab="Variance of within-Forest Residuals")
> abline(lm(ref.var.forest ~ ref.forest), col="purple")
> plot(ref.tree, ref.var.tree, xlab="Tree Random Effect",
+      ylab="Variance of within-Tree Residuals")
> abline(lm(ref.var.forest ~ ref.forest), col="purple")
> par(opar)
```

Everything in these figures look good except for the residual plots and the correlation of the within-tree residuals, which show an unacceptably strong signal. At this point one might think that the next step is to try to fit an autocorrelation function to the within-tree residuals, but the kink in the residual plot suggests that it seems more valuable to take a look at a different diagnostic first.

The augmented prediction plot overlays the fitted model with the observed data, at an optional level within the model. It is constructed using `xyplot()` from lattice graphics, and accepts arguments that are relevant to that function, for further customization. This allows us to sort the trees by national forest, to help us pick up any cluster effects.

```
> trees.in.forests <- aggregate(x=list(measures=stage$height.m),
+     by=list(tree=stage$Tree.ID, forest=stage$Forest.ID), FUN=length)
> panel.order <- rank(as.numeric(as.character(trees.in.forests$tree)))

> plot(augPred(hd.lme.3), index.cond=list(panel.order))
```

The augmented prediction plot (Figure 12.14) shows that a number of the trees have curvature in the relationship between height and diameter that the model fails to pick up, whilst others seem pretty linear. It also shows that the omission of a random slope appears to be problematic.

At this point we have several options, each of which potentially leads to different resolutions for our problem, or, more likely, to several further approaches, and so on. How we proceed depends on our goal. We can:

1. add a quadratic fixed effect;
2. add a quadratic random effect;
3. add quadratic fixed and random effects;
4. correct the model by including a within-tree correlation; and
5. switch to non-linear mixed-effects models and use a more appropriate functional form.

Since we do not believe that the true relationship between height and diameter could reasonably be a straight line, let's add a fixed and a random quadratic diameter effect, by tree, and see how things go. For a start this will increase the number of diagnostic graphs that we want to look at to about 22! We'll show only a sample here.

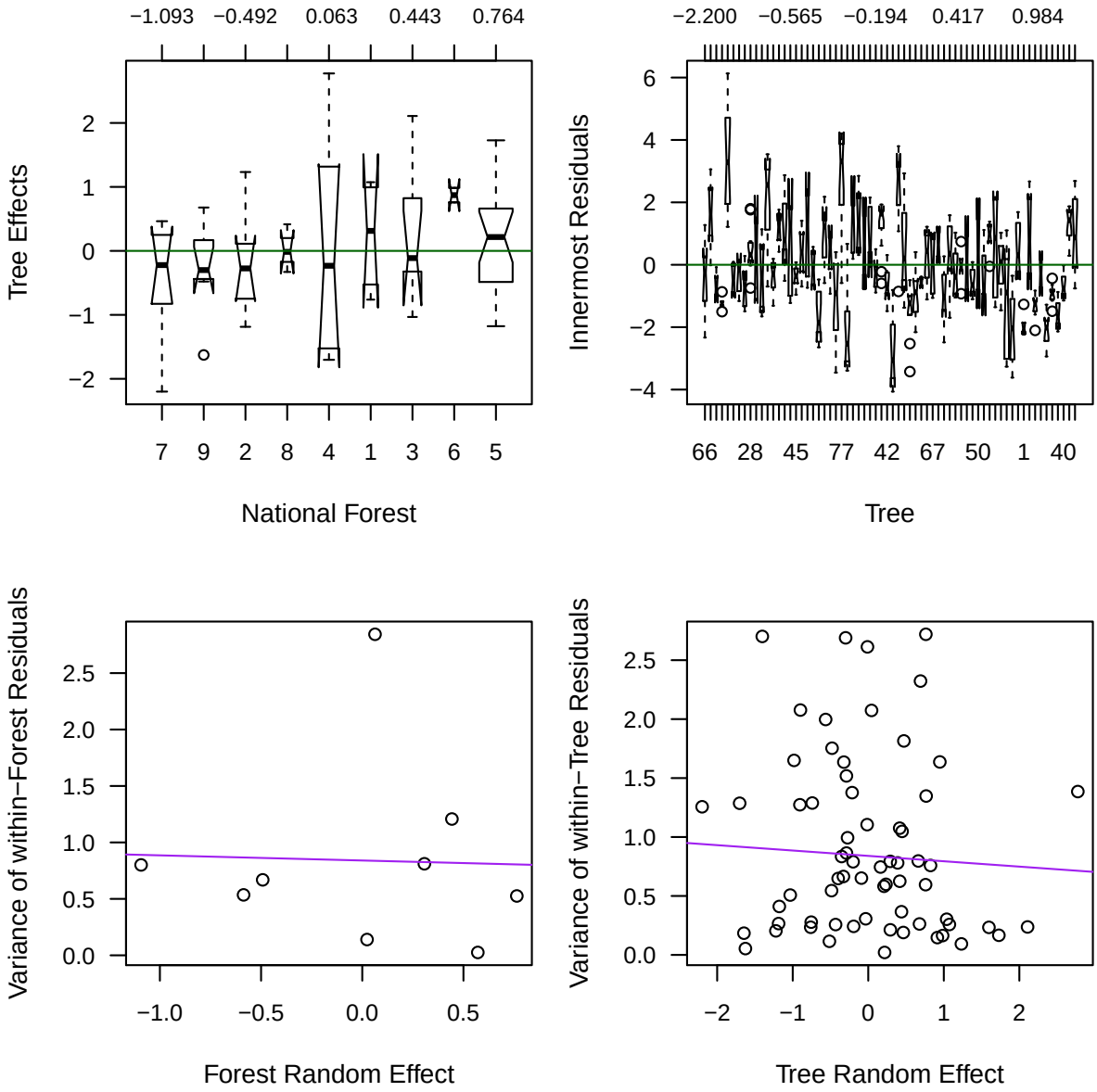

Figure 12.13: Selected random-effects based diagnostics for the mixed-effects fit of the Height and Diameter model for Stage's data.

```
> hd.lme.4 <- lme(height.m ~ dbhib.cm + I(dbhib.cm^2),
+    random = ~ dbhib.cm + I(dbhib.cm^2) | Tree.ID,
+      data = stage)

> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1,
+         cex.axis = 0.9)
>   plot(fitted(hd.lme.4, level=0), stage$height.m,
+       xlab = "Fitted Values", ylab = "Observed Values",
+       main = "Model Structure (I)")
>   abline(0, 1, col = "gray")
>   scatter.smooth(fitted(hd.lme.4), residuals(hd.lme.4, type="pearson"),
+       main = "Model Structure (II)",
+       xlab = "Fitted Values", ylab = "Innermost Residuals")
>   abline(0, 0, col = "gray")
> acf.resid <- ACF(hd.lme.4, resType = "n")
> plot(acf.resid$lag[acf.resid$lag < 10.5],
```

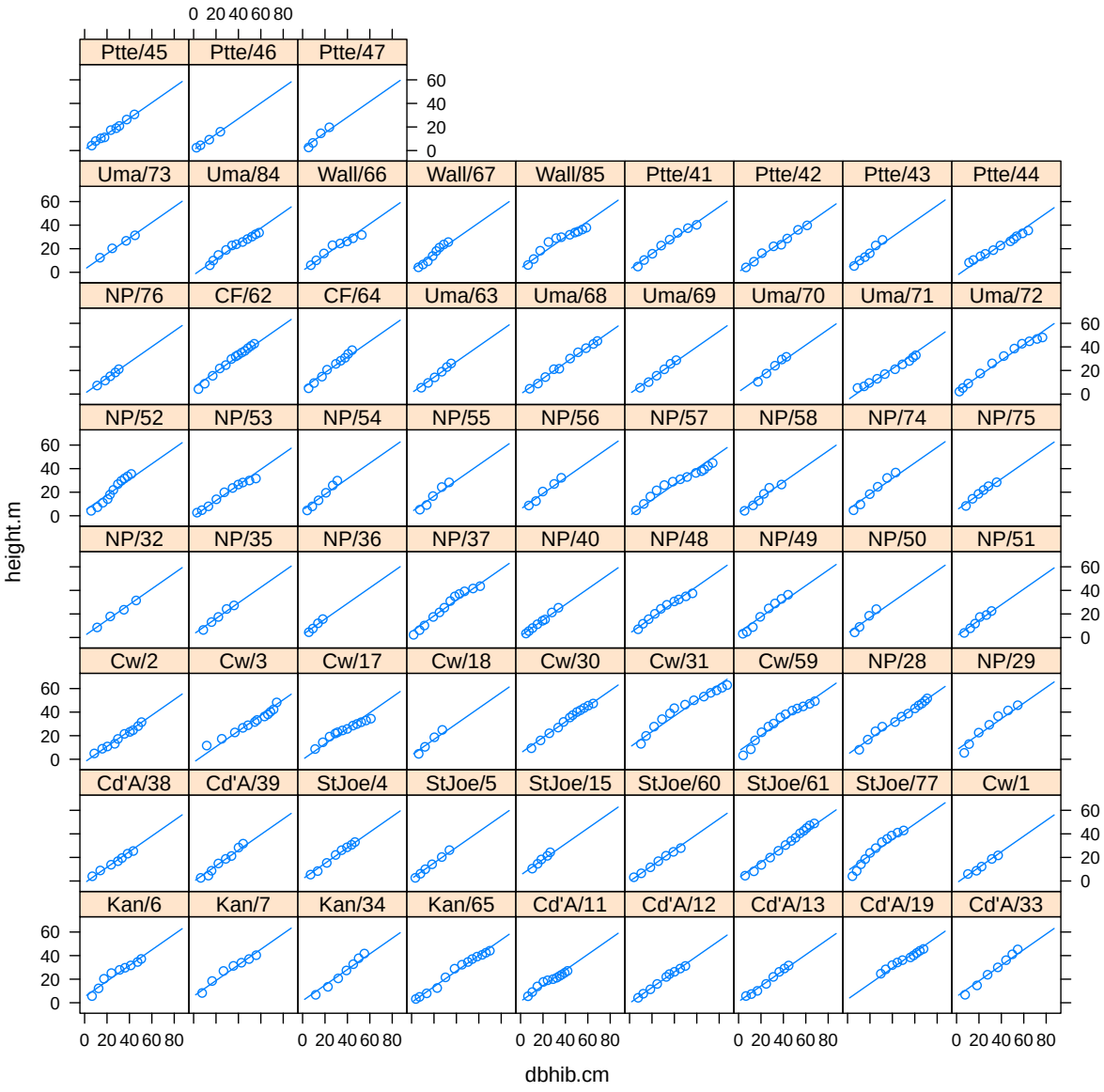

Figure 12.14: Height against diameter by tree, augmented with predicted lines.

```
+       acf.resid$ACF[acf.resid$lag < 10.5],
+       type="b", main="Autocorrelation",
+       xlab="Lag", ylab="Correlation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       -stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> abline(0,0,col="gray")
> par(opar)
```

This has improved the model somewhat, but it looks like we do need to include some accounting for the within-tree correlation. Pinheiro and Bates (2000) detail the options that are available. Also, we'll use `update()` because that starts the model fitting at the most recently converged estimates, which speeds up fitting considerably. Finally, we need to use a different fitting engine, for greater stability.

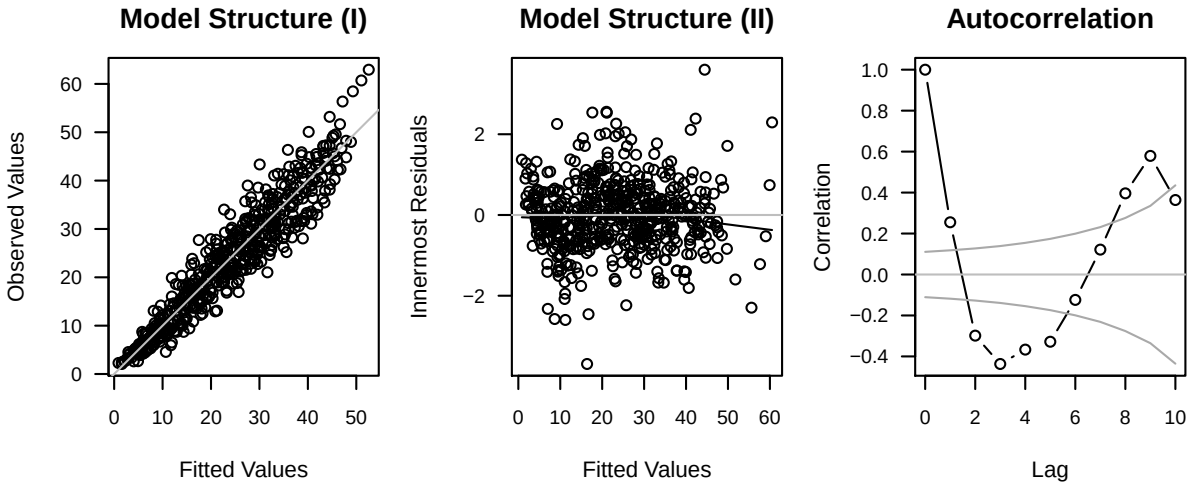

Figure 12.15: Selected diagnostics for the mixed-effects fit of the Height and Diameter model for Stage's data.

```
> hd.lme.5 <- update(hd.lme.4, correlation = corCAR1(),
+                    control = lmeControl(opt="optim"))

> opar <- par(mfrow = c(1, 3), mar = c(4, 4, 3, 1), las = 1,
+         cex.axis = 0.9)
>   plot(fitted(hd.lme.5, level=0), stage$height.m,
+       xlab = "Fitted Values", ylab = "Observed Values",
+       main = "Model Structure (I)")
>   abline(0, 1, col = "gray")
>   scatter.smooth(fitted(hd.lme.5), residuals(hd.lme.5, type="pearson"),
+       main = "Model Structure (II)",
+       xlab = "Fitted Values", ylab = "Innermost Residuals")
>   abline(0, 0, col = "gray")
> acf.resid <- ACF(hd.lme.5, resType = "n")
> plot(acf.resid$lag[acf.resid$lag < 10.5],
+      acf.resid$ACF[acf.resid$lag < 10.5],
+      type="b", main="Autocorrelation",
+      xlab="Lag", ylab="Correlation")
> stdv <- qnorm(1 - 0.01/2)/sqrt(attr(acf.resid, "n.used"))
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> lines(acf.resid$lag[acf.resid$lag < 10.5],
+       -stdv[acf.resid$lag < 10.5],
+       col="darkgray")
> abline(0,0,col="gray")
> par(opar)
```

The correlation is small now.

Another element of the model that we have control over is the variance of the random effects. We haven't seen any red flags for heteroskedasticity in the model diagnostics, so we haven't worried about it. However, such situations are common enough to make an example worthwhile.

Two kinds of heteroskedasticity are common and worthy of concern: firstly, that the variance of the response variable is related to the response variable, and secondly, that the conditional variance of the observations varied within one or more stratum. Some combination of the two conditions is also possible.

We can detect these conditions by conditional residual scatterplots of the following kind. The first is a scatterplot of the innermost Pearson residuals against the fitted values stratified by habitat type. The code to create this graphic is part of the `nlme` package.

```
> plot(hd.lme.5, resid(.) ~ fitted(.) | HabType.ID, layout=c(1, 5))
```

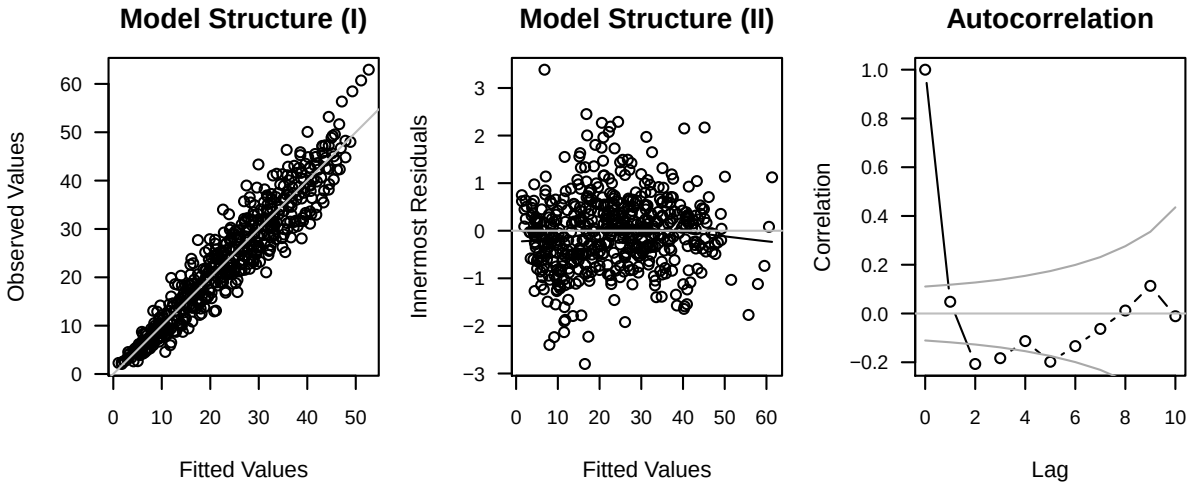

Figure 12.16: Selected diagnostics for the mixed-effects fit of the Height and Diameter model for Stage's data.

The second is a quantile plot of the innermost Pearson residuals against the normal distribution, stratified by habitat type. This code is provided by the `lattice` package, and we found a template under `?qqmath`.

```
> qqmath(~ resid(hd.lme.5) | stage$HabType.ID,
+        prepanel = prepanel.qqmathline,
+        panel = function(x, ...) {
+          panel.qqmathline(x, distribution = qnorm)
+          panel.qqmath(x, ...)
+        })
```

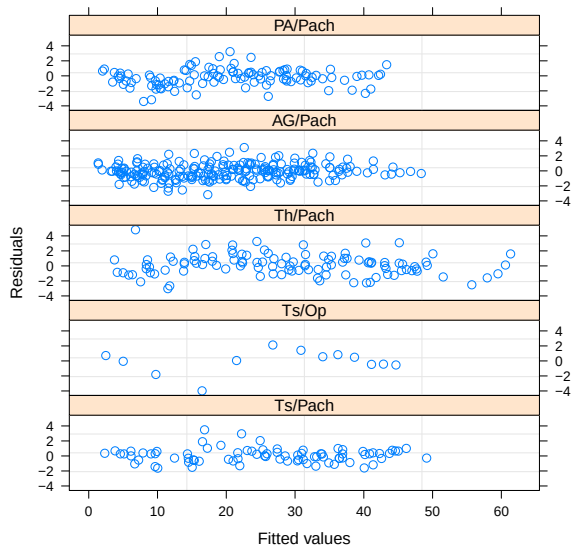

Figure 12.17: Innermost Pearson residuals against fitted values by habitat type.

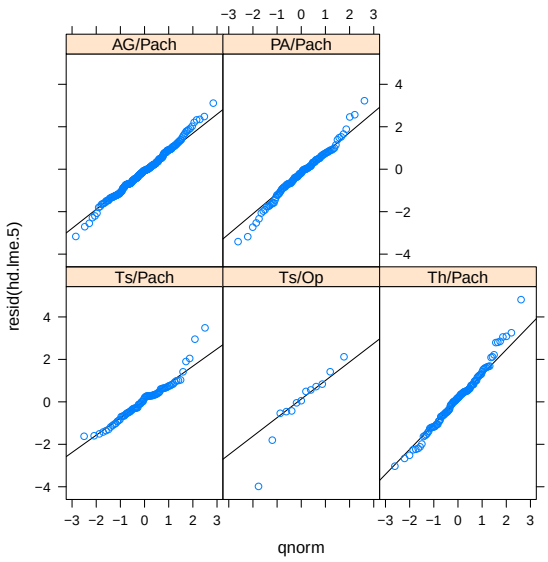

Figure 12.18: Quantile plots of innermost Pearson residuals against the normal distribution, by habitat type.

There seems little evidence in either of figures 12.17 and 12.18 to suggest that the variance model is inadequate.

Had the variance model seemed inadequate, we could have used the `weights` argument in a call to `update` with one of the following approaches:

- `weights = varIdent(form = 1 | HabType.ID)` This option would allow the observations within each habitat type to have their own variance.
- `weights = varPower()` This option would fit a power function for the relationship between the variance and the predicted mean, and estimate the exponent.
- `weights = varPower(form = dbhib.cm | HabType.ID)` This option would fit a power function for the relationship between the variance and the diameter uniquely within each habitat type, and estimate the exponent.
- `weights = varConstPower()` This option would fit a power function with a constant for the relationship between the variance and the predicted mean, and estimate the exponent and constant.

Other options are available; the function is fully documented in Pinheiro and Bates (2000).

Then, let's accept the model as it stands for the moment. This is the baseline model, as it provides predictions of height from diameter, and satisfies the regression assumptions. Other options may later prove to be better fitting, for example it may be that including habitat type or age in the model obviates our use of the quadratic diameter term. Whether or not this makes for a better model in terms of actual applications will vary!

```
> plot(augPred(hd.lme.5), index.cond=list(panel.order))
```

```
> summary(hd.lme.5)

Linear mixed-effects model fit by REML
 Data: stage
       AIC      BIC    logLik
  1945.521 1992.708 -961.7604

Random effects:
 Formula: ~dbhib.cm + I(dbhib.cm^2) | Tree.ID
 Structure: General positive-definite, Log-Cholesky parametrization
              StdDev       Corr
(Intercept)   0.0007992448 (Intr) dbhb.c
dbhib.cm      0.1844016124 -0.243
I(dbhib.cm^2) 0.0030927949 -0.175 -0.817
Residual      1.4223449956

Correlation Structure: Continuous AR(1)
 Formula: ~1 | Tree.ID
 Parameter estimate(s):
      Phi
0.6660391
Fixed effects: height.m ~ dbhib.cm + I(dbhib.cm^2)
                   Value  Std.Error  DF   t-value p-value
(Intercept)   -0.4959656 0.25444240 474 -1.949226  0.0519
dbhib.cm       0.8918030 0.02985028 474 29.875871  0.0000
I(dbhib.cm^2) -0.0032310 0.00052633 474 -6.138857  0.0000
 Correlation:
              (Intr) dbhb.c
dbhib.cm      -0.514
I(dbhib.cm^2)  0.417 -0.867

Standardized Within-Group Residuals:
         Min           Q1          Med           Q3          Max
-2.799939334 -0.482647581 -0.008763721  0.414566863  3.384428017

Number of Observations: 542
Number of Groups: 66
```

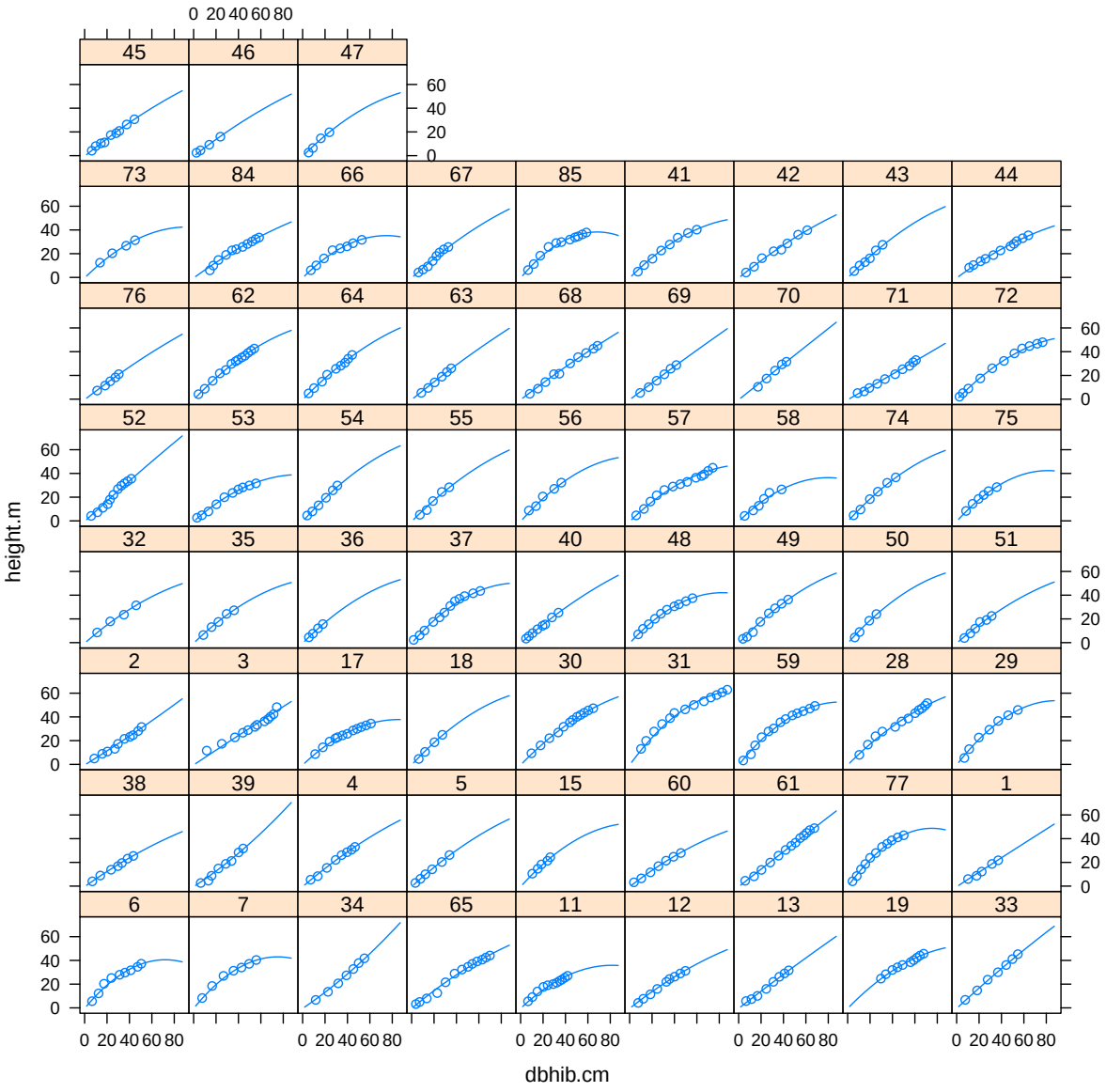

Figure 12.19: Height against diameter by tree, augmented with predicted lines.

12.4.2 Extensions to the model

We can try to extend the baseline model to improve its performance, based on our knowledge of the system. For example, it might be true that the tree age mediates its diameter - height relationship in a way that has not been captured in the model. We can formally test this assertion, using the `anova` function, or we can examine it graphically, using an added-variable plot, or we can try to fit the model with the term included and see what effect that has on the residual variation.

An added-variable plot is a graphical summary of the amount of variation that is uniquely explained by a predictor variable. It can be constructed in R as follows. Here, we need to decide what level of residuals to choose, as there are several. We adopt the outermost residuals.

```
> age.lme.1 <- lme(Age ~ dbhib.cm, random = ~1 | Forest.ID/Tree.ID,
+       data = stage)
> res.Age <- residuals(age.lme.1, level = 0)
> res.HD <- residuals(hd.lme.5, level = 0)
> scatter.smooth(res.Age, res.HD, xlab = "Variation unique to Age",
+      ylab = "Variation in Height after all but Age")
```

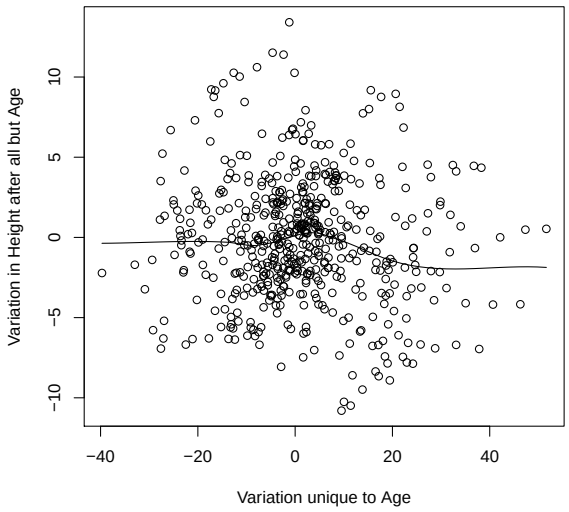

Figure 12.20: Added-variable plot for Age against the ratio of Height over Diameter.

In order to assess whether we would be better served by adding habitat type to the model, we can construct a graphical summary, thus:

```
> xyplot(stage$height.m ~ fitted(hd.lme.5, level=0) | HabType.ID,
+         xlab="Predicted height (m)",
+         ylab="Observed height (m)",
+         data=stage,
+         panel = function(x, y, subscripts) {
+                 panel.xyplot(x, y)
+                 panel.abline(0, 1)
+                 panel.abline(lm(y ~ x), lty=3)
+         }
+ )
```

Neither of figures 12.20 or 12.21 suggest that significant or important improvements would accrue from adding these terms to the model.

The incumbent model represents the best compromise so far. It seems to have addressed most of our major concerns in terms of model assumptions. It may be possible to find a better model with further searching. However, there comes a point of diminishing returns. Note finally that although the presentation of this sequence of steps seems fairly linear, in fact there were numerous blind-alleys followed, much looping, and retracing of steps. This is not a quick process! Introducing random effects to a fixed effects model increases the number of diagnostics to check and possibilities to follow.

12.5 The Model

Let's examine our final model.

$$\begin{aligned} y_{ijk} &= \beta_0 + b_{0i} + b_{0ij} && (12.17)\\ &\quad + (\beta_1 + b_{1i} + b_{1ij}) \times x_{ijk} && (12.18)\\ &\quad + (\beta_2 + b_{2i} + b_{2ij}) \times x_{ijk}^2 && (12.19)\\ &\quad + \epsilon_{ijk} && (12.20) \end{aligned}$$

In matrix form, it is still:

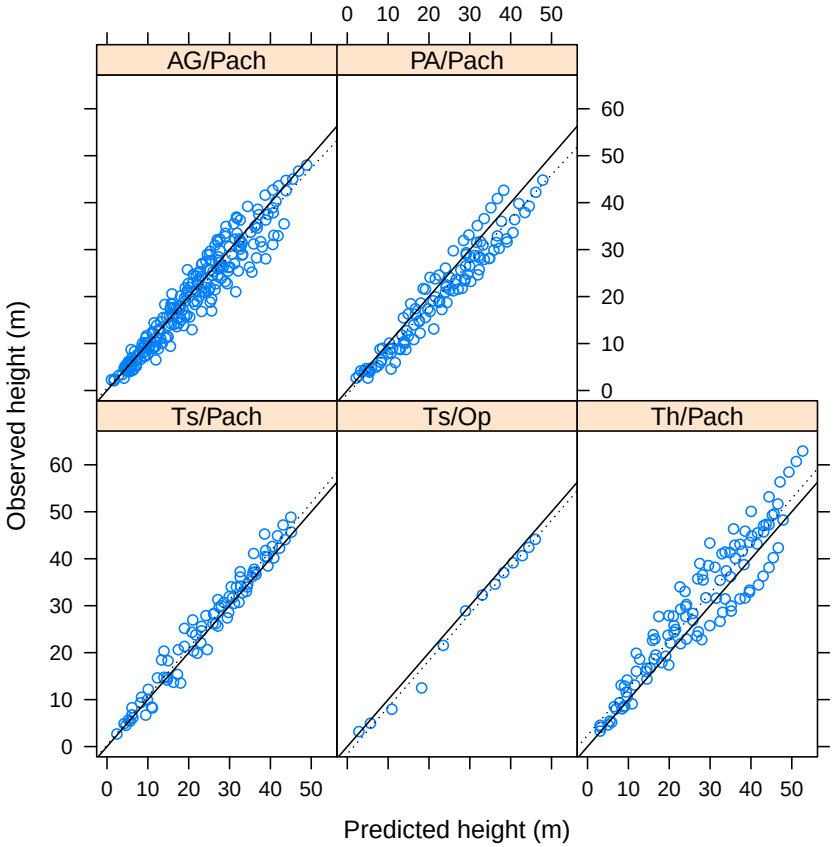

Figure 12.21: Plot of predicted height against observed height, by habitat type. The solid line is 1:1, as predicted by the model. The dotted line is the OLS line of best fit within habitat type.

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

Here's the structure.

- Y is the vector of height measurements. It has 542 observations.
- X is a 3×542 matrix of 1s, diameters and squared diameters.
- β is a vector of length three: it has an intercept, a slope for the linear diameter term, and a slope for the quadratic diameter term.
- Z is a 225×542 unit brute. See below.
- b is a vector of intercepts, and slopes for diameter and diameter squared for each forest, then for each tree. It will be $27 + 198 = 225$ elements long. See below. The predictions can be obtained by `ranef(hd.lme.5)`.
- $\mathbf{D}$ will be a block diagonal matrix comprising 9 3×3 identical matrices, followed by 66 3×3 identical matrices. Each matrix will express the covariance between the 3 random effects within forest or within tree. See below.
- $\mathbf{R}$ will now be a 542×542 symmetric matrix for which the off diagonals are 0 between trees, and a geometric function of the inter-measurement time within trees.

12.5.1 Z

The only role of $\mathbf{Z}$ is to allocate the random effects to the appropriate element. This can be somewhat complicated. Our $\mathbf{Z}$ can be divided into two independent sections; a 27×542 matrix $\mathbf{Z_f}$ associated with the forest level effects, and a 198×542 matrix $\mathbf{Z_t}$ associated with the tree-level effects. In matrix nomenclature:

$$\mathbf{Z} = [\mathbf{Z_f} \mid \mathbf{Z_t}] \tag{12.21}$$

Now, $\mathbf{Z_f}$ allocates the random intercept and two slopes to each observation from each forest. There are 9 forests, so any given row of $\mathbf{Z_f}$ will contain 24 zeros, a 1, and the corresponding $dbhib$ and $dbhib^2$. For example, for the row corresponding to measurement 4 on tree 2 in forest 5, we'll have

$$Z_f = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, d_{524}, d_{524}^2, 0, 0, 0, \ldots) \tag{12.22}$$

Similarly, $\mathbf{Z_t}$ allocates the random intercept and two slopes to each observation from each tree. There are 66 trees, so any given row of $\mathbf{Z_t}$ will contain 195 zeros, a 1, and the corresponding $dbhib$ and $dbhib^2$. It will have the same fundamental pattern as above.

12.5.2 b

The purpose of $\mathbf{b}$ is to contain all the predicted random effects. Thus it will be 225 units long, which corresponds to 3 for each level of forest (intercept, slope for diameter, and slope for diameter squared) and 3 for each level of tree (intercept, slope for diameter, and slope for diameter squared).

$$\mathbf{b} = (b_{f10}, b_{f1d}, b_{f1d2}, b_{f20}, b_{f2d}, b_{f2d2}, \ldots, b_{t10}, b_{t1d}, b_{t1d2}, b_{t20}, b_{t2d}, b_{t2d2}, \ldots)' \tag{12.23}$$

The combination of $\mathbf{b}$ and $\mathbf{Z}$ serves to allocate each random effect to the appropriate unit and measurement.

12.5.3 D

Finally, $\mathbf{D}$ dictates the relationships between the different random effects within the levels of forest and tree. We've assumed that the random effects will be independent between habitat types and trees. So, there are only two sub-matrices to this matrix, called $\mathbf{D_f}$ and $\mathbf{D_t}$.

$$\mathbf{D_f} = \begin{bmatrix} \sigma_{bf0}^2 & \sigma_{bf0d} & \sigma_{bf0d2} \\ \sigma_{bf0d} & \sigma_{bfd}^2 & \sigma_{bfdd2} \\ \sigma_{bf0d2} & \sigma_{bfdd2} & \sigma_{bfd2}^2 \end{bmatrix} \tag{12.24}$$

$$\mathbf{D_t} = \begin{bmatrix} \sigma_{bt0}^2 & \sigma_{bt0a} & \sigma_{bt0d2} \\ \sigma_{bt0d} & \sigma_{btd}^2 & \sigma_{btdd2} \\ \sigma_{bt0d2} & \sigma_{btdd2} & \sigma_{btd2}^2 \end{bmatrix} \tag{12.25}$$

Then the structure of $\mathbf{D}$ is simply 9 repetitions of $\mathbf{D_f}$, laid on a diagonal line, followed by 66 repetitions of $\mathbf{D_t}$ laid on the same diagonal, and zeros everywhere else.

12.6 Wrangling

We observed earlier that the use of the `control` argument was a key tool for the modeller. This element can introduce a little culture shock. Having ourselves come from traditions of model fitting for which exact solutions were easily obtained,and convergence was unequivocal, it was surprising, not to say disheartening, to find that algorithms sometimes quit before convergence. Probably we display our naivete.

The statistical tools that we have been discussing in this chapter are too complicated to admit exact solutions. Accordingly, we have to try to maximize the likelihood, for example, by iterative means. It is necessary and correct that the authors of the code we use will have put in checks, to halt the code in situations where they deem continuing unprofitable.

In any case, bitter experience, and ruthless experimentation have taught us that the code authors do not necessarily have exactly our problem in mind when they are choosing the default parameters for their software. In such cases, it is necessary to roll up our sleeves and plunge our arms into the organs of our analysis. Most of the fitting tools that we use have control arguments that will report or modify the process of model fitting. Experimenting with these will often lead to model configurations that fit reliably.

In short, don't be reluctant to experiment. Any or all of the following strategies might be necessary to achieve a satisfactory fit of your model to your data.

12.6.1 Monitor

In order to be better informed about the progress of model fitting, we use the `msVerbose` argument. It provides a brief, updating description of the progress of the model fit. It will also point out problems along the way, which help the user decide what is the best thing to do next.

12.6.2 Meddle

This strategy involves adjusting the fitting tool.

If the model is failing to converge, then often all that is required is an increase in the number of allowable iterations. The mixed-effects model fitting algorithms in `lme` use a hybrid optimization scheme that starts with the EM algorithm and then changes to Newton–Raphson (Pinheiro and Bates, 2000, p. 80). The latter algorithm is implemented with two loops, so we have three iteration caps. We have found that increasing both `maxIter` and `msMaxIter` is a useful strategy. If we are feeling patient, we will increase them to about 10000, and monitor the process to see if the algorithm still wishes to search. We have occasionally seen iteration counts in excess of 8000 for models that subsequently converged.

We have also had success with changing the optimization algorithm. That is, models that have failed to converge with `nlminb`, by getting caught in a singular convergence, have converged successfully using `Nelder-Mead` in `optim`. The default is to use `nlminb`, but it may be worth switching to `optim`, and within `optim`, choosing between `Nelder-Mead`, `BFGS`, and `L-BFGS-B`. Each of these algorithms has different properties, and different strengths and weaknesses. Any might lead more reliably to a satisfactory solution.

12.6.3 Modify

This strategy involves changing the relationship between the model and the data.

The `update` function fits a new model using the output of an old model as a starting point. This is an easy way to set the starting points for parameter estimates, and should be in common use for iterative model building in any case, due to its efficiency. Try dropping out components of the model that complicate it, fitting a smaller, simpler model, and then using `update` to fit the full model.

Alternatively, a number of the model components permit the specification of a starting point. For example, if we provide the `corAR1` function with a suitable number then the algorithm will use that number as a starting point. Specifying this value can help the algorithm converge speedily, and sometimes, at all. Experimenting with subsets of the full model to try to find suitable starting points can be profitable, for example if one has a correlation model and a variance model.

We can also think about how the elements in the data might be interacting with the model. Is the dataset unbalanced, or are there outliers, or is it too small? Any of these conditions can cause problems for fitting algorithms. Examining the data before fitting any model is standard practice. Be prepared to

temporarily delete data points, or augment under-represented portions, in order to provide the `update` function with a reasonable set of starting values.

12.6.4 Compromise

Sometimes a model involves a complicated hierarchy of random effects. It is worth asking whether or not such depth is warranted, and whether a superficially more complex, but simpler model, might suffice. The case study in this chapter serves as a good example: although model fit benefited by allowing each individual tree to have a random slope, there was no need to allow each national forest to have a random slope. Including a slope for each forest made the model unnecessarily complicated, and also made fitting the model much harder. Specifying the smaller model was a little less elegant, however.

Finally, sometimes no matter what exigencies we try, a model will not converge. There is a point in every analysis where we must decide to cut our losses and go with the model we have. If we know that the model has shortcomings, then it is our responsibility to draw attention to those shortcomings. For example, if we are convinced that there is serial autocorrelation in our residuals, but cannot achieve a reasonable fit using the available resources, then providing a diagnostic plot of that autocorrelation is essential. Furthermore, it is important to comment on the likely effect of the model shortcoming upon inference and prediction. If we are fortunate enough to be able to fit a simpler model that does include autocorrelation, for example, we might demonstrate what effect the inclusion of that portion of the model has upon our conclusions. We would do this by fitting three models: the complex model, the simple model with the autocorrelation, and the simple model without the autocorrelation. If the difference between the latter two models is modest, then we have some modest degree of indirect evidence that perhaps our conclusions will be robust to misspecification of the complex model. It is not ideal, but we must be pragmatic.

12.7 Appendix - Leave-One-Out Diagnostics

Another important question is whether there are any outliers or high-influence points. In a case like this it is relatively easy to see from the diagnostics that no point is likely to dominate the fit in this way. However, a more formal examination of the question is valuable. To date, there is little peer-reviewed development of the problem of outlier and influence detection. Schabenberger (2005) provides an overview of the extensive offerings available in SAS, none of which are presently available in R. Demidenko and Stukel (2005) also provide some alternatives.

The simplest thing, in the case where a model fit is relatively quick, is to refit the model dropping each observation one by one, and collecting the results in a vector for further analysis. This is best handled by using the `update()` function.

```
> all.betas <- data.frame(labels=names(unlist(hd.lme.1$coefficients)))
> cook.0 <- cook.1 <- rep(NA, dim(stage.old)[1])
> p.sigma.0 <- length(hd.lme.1$coefficients$fixed) *
+          var(residuals(hd.lme.1, level=0))
> p.sigma.1 <- length(hd.lme.1$coefficients$fixed) *
+          var(residuals(hd.lme.1, level=1))
> for (i in 1:dim(stage.old)[1]) {
+   try({ hd.lme.n <- update(hd.lme.1, data = stage.old[-i,])
+         new.betas <- data.frame(labels=names(unlist(hd.lme.n$coefficients)),
+                                 coef=unlist(hd.lme.n$coefficients))
+         names(new.betas)[2] <- paste("obs", i, sep=".")
+         all.betas <- merge(all.betas, new.betas, all.x = TRUE)
+         cook.0[i] <- sum((predict(hd.lme.1, level=0, newdata=stage.old) -
+                           predict(hd.lme.n, level=0, newdata=stage.old))^2) /
+                             p.sigma.0
+         cook.1[i] <- sum((predict(hd.lme.1, level=1, newdata=stage.old) -
+                           predict(hd.lme.n, level=1, newdata=stage.old))^2) /
+                             p.sigma.1
+       })
+ }
```

We can then examine these results with graphical diagnostics (Figures 12.22 and 12.23). The Cook's Distances presented here are only approximate.

```
> all.betas <- t(all.betas[,-1])
> len.all <- length(unlist(hd.lme.1$coefficients))
> len.fixed <- length(hd.lme.1$coefficients$fixed) # 2
> len.ran <- length(hd.lme.1$coefficients$random$Forest.ID) # 9

> opar <- par(mfrow=c(len.all, 1), oma=c(2,0,1,0), mar=c(0,4,0,0), las=1)
> for (i in 1:len.fixed) {
+   plot(all.betas[,i], type="l", axes=F, xlab="", ylab="")
+   text(length(all.betas[,i])-1, max(all.betas[,i], na.rm=T),
+        names(unlist(hd.lme.1$coefficients))[i],
+        adj=c(1,1), col="red")
+   axis(2)
+   box()
+ }
> for (i in (len.fixed+1):(len.all)) {
+   plot(all.betas[,i], type="l", axes=F, xlab="", ylab="")
+   text(length(all.betas[,i])-1, max(all.betas[,i], na.rm=T),
+        names(unlist(hd.lme.1$coefficients))[i],
+        adj=c(1,1), col="red")
+   axis(2)
+   box()
+ }
> axis(1)
> par(opar)


> cook <- data.frame(id=stage.old$Tree.ID, fixed=cook.0, forest=cook.1)
> influential <- apply(cook[,2:3], 1, max) > 1
> plot(cook$fixed, cook$forest, type="n",
+     xlab="Outermost (Fixed effects only)",
+     ylab="Innermost (Fixed effects and random effects)")
> points(cook$fixed[!influential], cook$forest[!influential])
> if(sum(influential) > 0)
+    text(cook$fixed[influential], cook$forest[influential],
+       cook$id[influential], col="red", cex=0.85)
```

And, what about the removal of entire forests? We can compute the effects similarly.

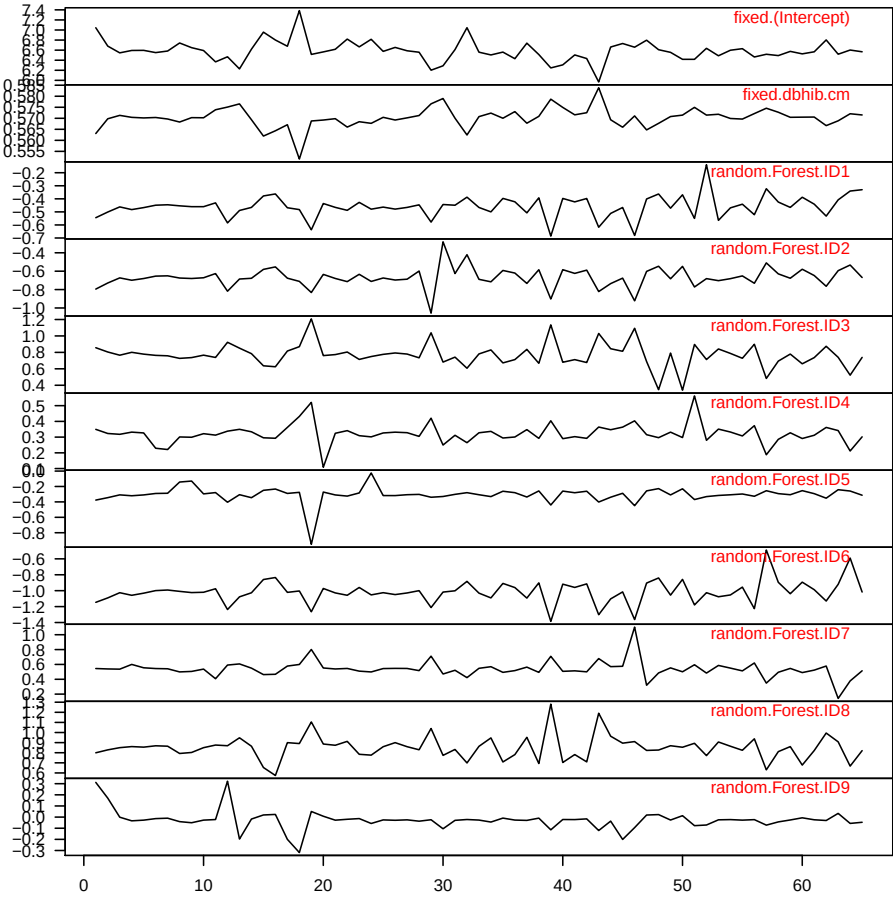

Figure 12.22: The parameter estimates for the fixed effects and predictions for the random effects resulting from omitting one observation.

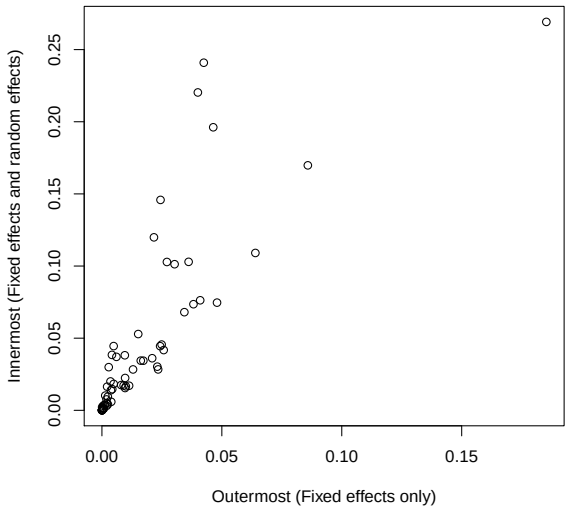

Figure 12.23: Cook's Distances for outermost and innermost residuals. Values greater than 1 appear in red and are identified by the tree number. The corresponding observations bear further examination.

Chapter 13

Nonlinear Mixed Effects

13.1 Many Units

In Section 11.1, we fit a nice non–linear model to a single tree. It would be good to be able to fit the same model to a collection of trees with minimal fuss. The `nlme` package again provides us with a way forward. Firstly, we can use the `groupedData` structure to simplify the problem of fitting a model to many trees, and secondly, we can use a so–called self-starting function, which provides its own starting values for any data that are presented to it. This self-starting function is one of the prepackaged functions mentioned earlier, and its adoption simplifies our approach considerably.

```
> gutten <- read.csv("../data/gutten.csv")
> gutten <- gutten[!is.na(gutten$Diameter), c("Site",
+     "Location", "Tree", "Diameter", "Age.bh")]
> names(gutten) <- c("site", "location", "tree", "dbh.cm",
+     "age.bh")
> gutten$site <- factor(gutten$site)
> gutten$location <- factor(gutten$location)
> gutten$tree.ID <- interaction(gutten$site, gutten$location,
+     gutten$tree)
> diameter.growth <- deriv(~asymptote * (1 - exp(-exp(scale) *
+     x)), c("asymptote", "scale"), function(x, asymptote,
+     scale) {
+ })
```

```
> require(nlme)
> gutten.d <- groupedData(dbh.cm ~ age.bh | tree.ID, data=gutten)
```

The relevant self-starting function is called `SSasympOrig`.

```
> gutten.nlsList <-
+   nlsList(dbh.cm ~ SSasympOrig(age.bh, asymptote, scale), data = gutten.d)
```

We are now in a position to speculate as to whether or not the kink observed in the residuals for tree 1.1.1 is repeated in the other trees (Figure 13.1). These results suggest that there certainly is a systematic lack of fit across the board. We may need to adopt a more flexible function.

```
> plot(gutten.nlsList, residuals(., type = "pearson") ~
+     fitted(.) | tree.ID)
```

We could print out all the parameter estimates but it is more useful to construct a graphical summary (Figure 13.2). Note that the intervals are based on large-sample theory.

```
> plot(intervals(gutten.nlsList), layout = c(2, 1))
```

We can extract and manipulate the coefficient estimates with a little bit of digging. The digging follows, and may be skipped. First we try to find what functions are available to manipulate objects of the class of the object that we have just created.

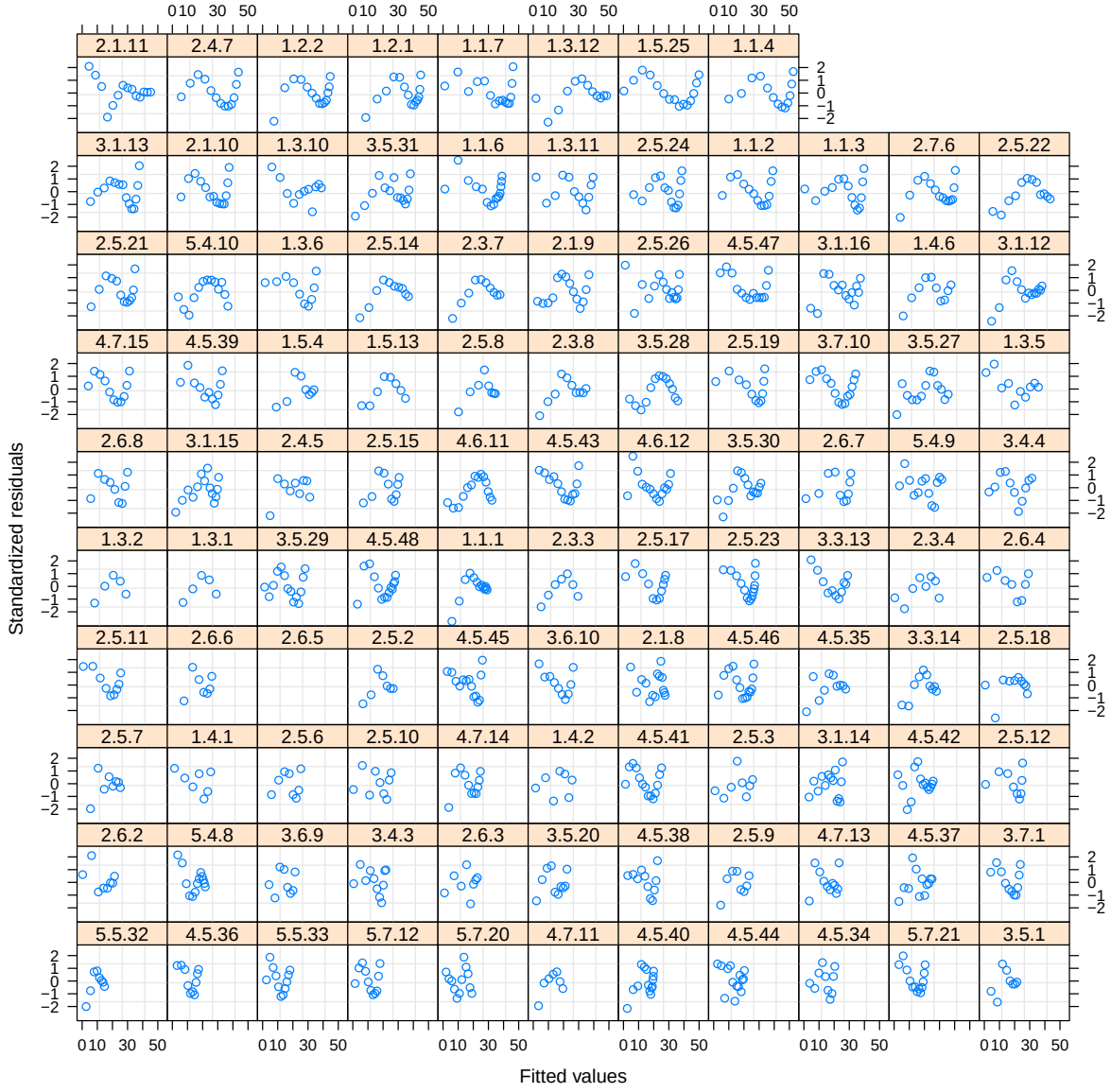

Figure 13.1: Plot of residuals against fitted values from non-linear models as fit to each tree.

```
> methods(class = class(gutten.nlsList))

[1] formula.nlsList* nlme.nlsList     summary.nlsList*
[4] update.nlsList*

   Non-visible functions are asterisked
```

A `summary` method is available. Let's find out what functions are available to manipulate summary objects.

```
> methods(class = class(summary(gutten.nlsList)))

[1] coef.summary.nlsList*

   Non-visible functions are asterisked
```

A `coef` method is available. What does its output look like?

```
> str(coef(summary(gutten.nlsList)))
```

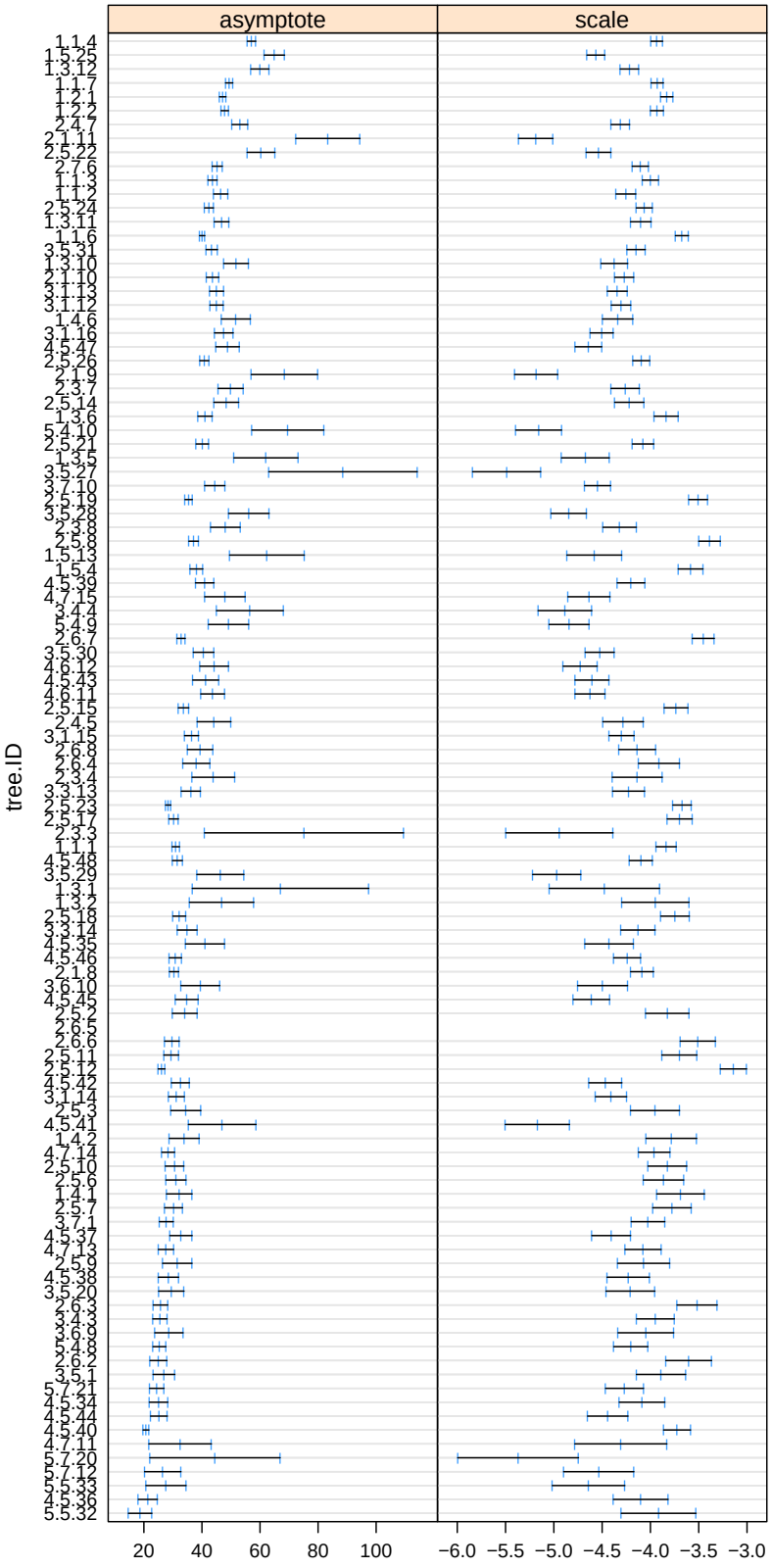

Figure 13.2: Interval plot for diameter prediction model fitted to Guttenberg data.

```
num [1:107, 1:4, 1:2] 18.7 21.4 27.6 26.5 44.5 ...
- attr(*, "dimnames")=List of 3
```

```
  ..$ : chr [1:107] "5.5.32" "4.5.36" "5.5.33" "5.7.12" ...
  ..$ : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)"
  ..$ : chr [1:2] "asymptote" "scale"
```

It is an array. We can extract its elements in the following way.

```
> asymptote <- coef(summary(gutten.nlsList))[,"Estimate","asymptote"]
> half.age <- log(2) /
+   exp(coef(summary(gutten.nlsList))[,"Estimate","scale"])
```

So, back to the exercise. Figure 13.3 shows a scatterplot of the estimated parameters for each tree: the estimated asymptote on the y-axis and the estimated age at which the tree reaches half its maximum diameter on the x-axis, with a *loess* smooth added to describe the mean of the pattern. There is clearly a relationship between the asymptote and the estimated age at which half the asymptote is reached.

```
> opar <- par(las=1, mar=c(4,4,1,1))
> scatter.smooth(half.age, asymptote, xlab="Age", ylab="Asymptote")
> par(opar)
```

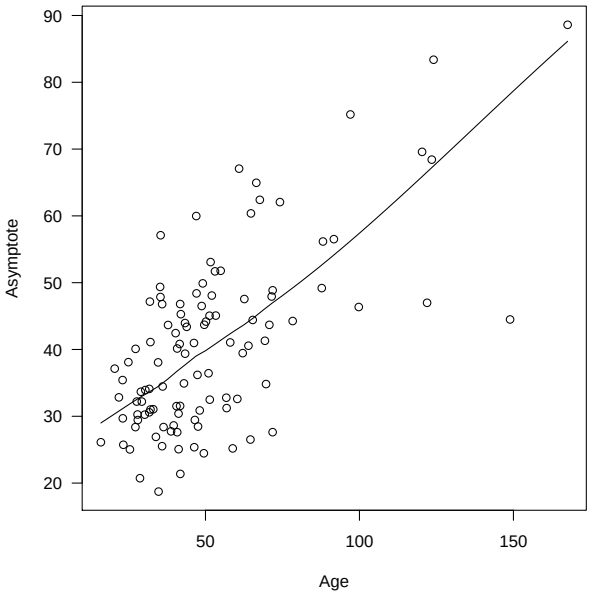

Figure 13.3: Plot of the estimated age at which the tree reaches half its maximum diameter against the estimated tree-level maximum diameter.

This approach provides us with a convenient way to think about fitting data to many different objects. But, what practical use does the model have? Not much. This model is analogous to the linear model that includes one intercept for each plot: the model assumptions are (probably) satisfied, but the model isn't really useful.

We could also try to fit the model to all the trees at once, that is, ignoring the differences between trees.

```
> (gutten.nls <- nls(dbh.cm ~ diameter.growth(age.bh, asymptote, scale),
+                    start = list(asymptote=50, scale=-5),
+                    data = gutten))

Nonlinear regression model
  model:  dbh.cm ~ diameter.growth(age.bh, asymptote, scale)
   data:  gutten
asymptote     scale
    38.56     -4.20
 residual sum-of-squares: 39130
```

```
Number of iterations to convergence: 5
Achieved convergence tolerance: 3.027e-06
```

This model has diagnostics reported in Figure 13.4, and constructed as follows.

```
> opar <- par(mfrow=c(2,2), mar=c(4,4,2,1), las=1)
> plot(fitted(gutten.nls), gutten$dbh.cm,
+      xlab="Fitted Values", ylab="Observed Values")
> abline(0, 1, col="red")
> plot(fitted(gutten.nls), residuals(gutten.nls, type="pearson"),
+      xlab="Fitted Values", ylab="Standardized Residuals")
> abline(h=0, col="red")
> plot(profile(gutten.nls), conf=0.95)
> par(opar)
```

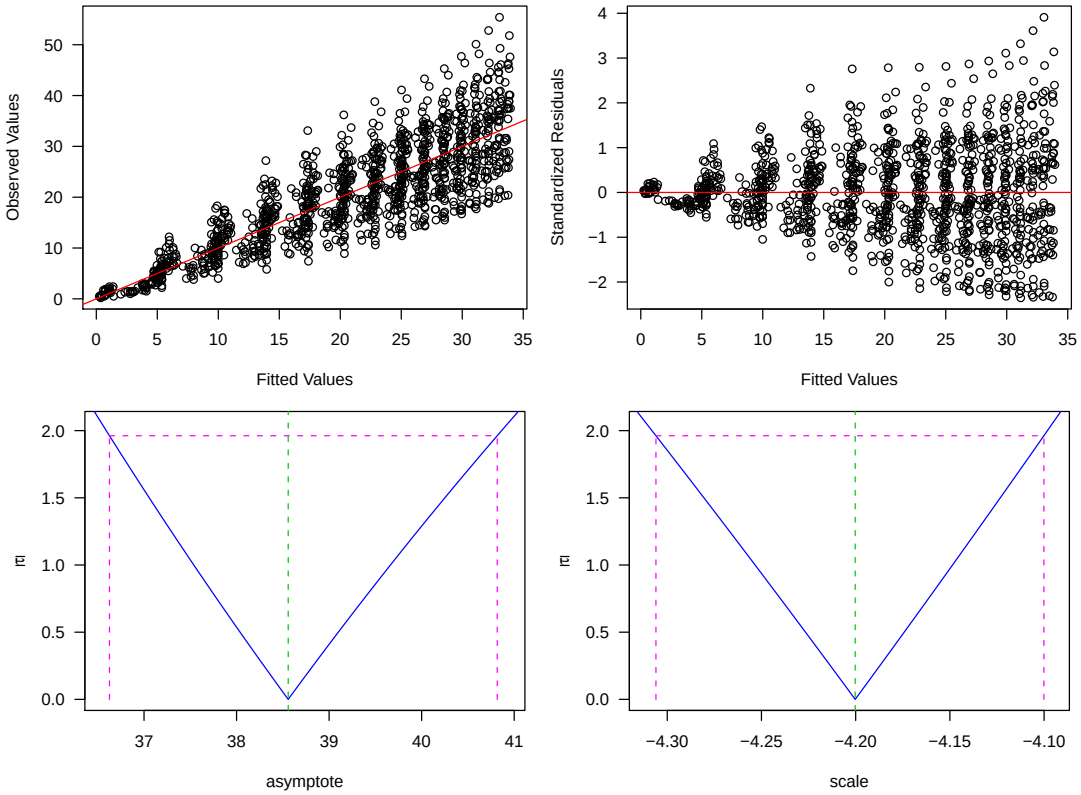

Figure 13.4: Profile plots for simple asymptotic model with all data.

How does the model seem? Here are the approximate large-sample 95% confidence interval estimates.

```
> (my.t <- qt(0.975, summary(gutten.nls)$df[2]))

[1] 1.961946

> coef(summary(gutten.nls))[,1:2] %*% matrix(c(1,-my.t,1,my.t), nrow=2)

               [,1]      [,2]
asymptote 36.522095 40.593577
scale     -4.300579 -4.099923
```

Compare them with these:

```
> confint(gutten.nls)
```

```
               2.5%     97.5%
asymptote 36.626648 40.816835
scale     -4.306002 -4.100122
```

They seem just fine. It's a shame about the diagnostics.

13.2 Hierarchical Resolution

We now focus on fitting the same kinds of models to hierarchical data. This direction produces simplifications and complications, but sadly more of the latter than the former.

We start with the non-linear mixed effects model that generalizes our earlier non-linear model (Equation 11.1. We will start with allowing each tree to have a random asymptote and scale. That is, for diameter measure t in tree i

$$y_{it} = (\phi_1 + \phi_{i1}) \times [1 - \exp(-\exp(\phi_2 + \phi_{i2})\, x)] + \epsilon_{it} \tag{13.1}$$

where ϕ_1 is the fixed, unknown asymptote and ϕ_2 is the fixed, unknown scale, and

$$\begin{bmatrix} \phi_{i1} \\ \phi_{i2} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix} \right) \tag{13.2}$$

The process of fitting and critiquing these models is well documented in Pinheiro and Bates (2000). Given that we have already used `nlsList()`, the easiest approach both from the point of view of typing, and of having sensible starting points, is:

```
> gutten.nlme.0 <- nlme(gutten.nlsList)
> summary(gutten.nlme.0)
```

```
Nonlinear mixed-effects model fit by maximum likelihood
  Model: dbh.cm ~ SSasympOrig(age.bh, asymptote, scale)
 Data: gutten.d
       AIC      BIC    logLik
  3627.763 3658.304 -1807.882

Random effects:
 Formula: list(asymptote ~ 1, scale ~ 1)
 Level: tree.ID
 Structure: General positive-definite, Log-Cholesky parametrization
          StdDev     Corr
asymptote 12.2386790 asympt
scale      0.4191592 -0.596
Residual   0.7092730

Fixed effects: list(asymptote ~ 1, scale ~ 1)
             Value Std.Error   DF    t-value p-value
asymptote 40.19368 1.2164375 1092   33.04212       0
scale     -4.20333 0.0417687 1092 -100.63355       0
 Correlation:
      asympt
scale -0.613

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-4.50181460 -0.51490231 -0.03653829  0.44912909  3.96736354

Number of Observations: 1200
Number of Groups: 107
```

If we wanted to construct this model from scratch, then we would need to do this:

```
> gutten.nlme.0 <- nlme(dbh.cm ~ SSasympOrig(age.bh, asymptote, scale),
+                       fixed = asymptote + scale ~ 1,
+                       random = asymptote + scale ~ 1,
+                       start = c(asymptote=50, scale=-5),
+                       data = gutten.d)
```

As with the linear mixed-effects models, we have a wide arrray of different diagnostic plots that we can deploy, and the functions to obtain those diagnostics are pretty much identical. Here we will focus on examining the within-tree autocorrelation (Figure 13.5).

```
> plot(ACF(gutten.nlme.0, form = ~1 | tree.ID), alpha = 0.01)
```

The figure shows that there is substantial within-tree autocorrelation, which suggests systematic lack of fit. At this point we have two options: we can try a different mean function, or we can try to model the autocorrelation. In general we should try them in that order. For the moment, though, we will focus on modeling the autocorrelation. After some experimentation, I arrived at:

```
> gutten.nlme.1 <- update(gutten.nlme.0,
+                         correlation = corARMA(p=1, q=2))
```

This produces residuals with autocorrelation pattern presented in Figure 13.6.

```
> plot(ACF(gutten.nlme.1, resType = "n", form = ~1 |
+     tree.ID), alpha = 0.01)
```

This is clearly superior to the preceding model, but still needs some tinkering. Perhaps too much lack of fit?

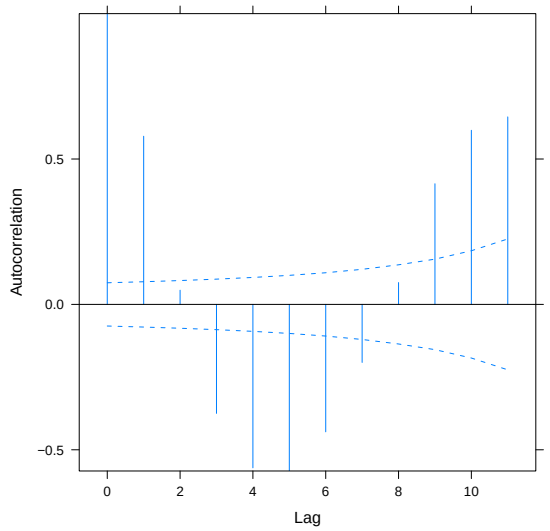

Figure 13.5: Autocorrelation of within-tree residuals from non-linear mixed-effects model.

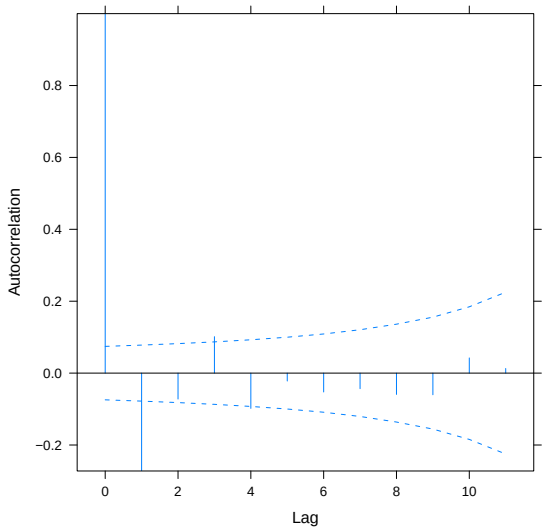

Figure 13.6: Autocorrelation of within-tree residuals from non-linear mixed-effects model with explicit autocorrelation model.

Because we constructed the model from a `groupedData` object, we can use the augmented prediction plots as a very useful summary of the model behaviour. This graphic is presented in Figure 13.7.

```
> plot(augPred(gutten.nlme.1))
```

We can get the approximate 95% confidence intervals of the estimates from the following call:

```
> intervals(gutten.nlme.1)
```

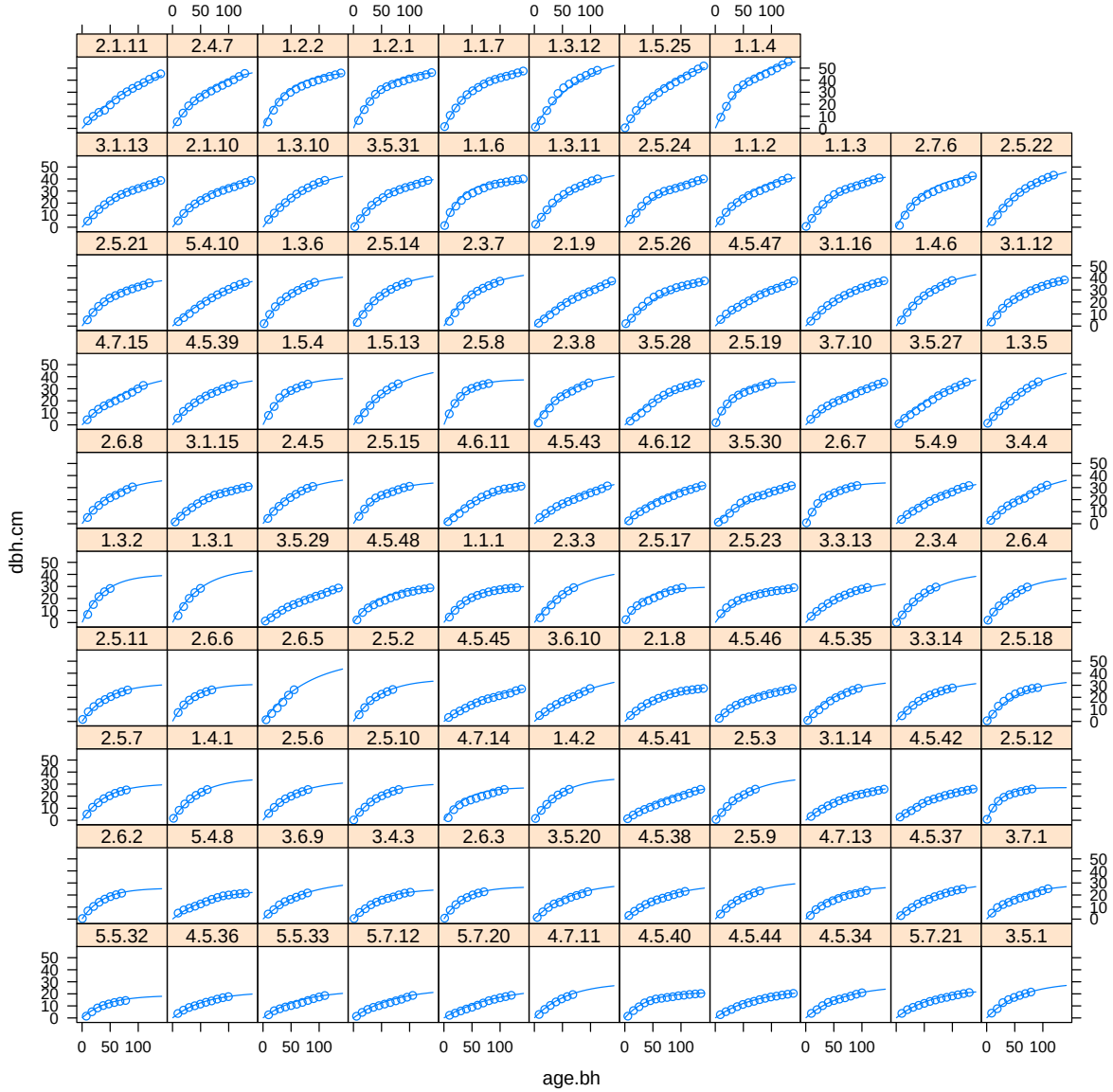

Figure 13.7: von Guttenberg's diameter data with asymptotic model passing through origin fitted, and within-tree autocorrelation accommodated. Here the tree-level models (Equation 13.1) are plotted.

```
Approximate 95% confidence intervals

 Fixed effects:
              lower      est.     upper
asymptote 36.477392 38.649203 40.821014
scale     -4.203253 -4.119008 -4.034763
attr(,"label")
[1] "Fixed effects:"

 Random Effects:
  Level: tree.ID
                          lower       est.      upper
sd(asymptote)         8.9042315 10.5142706 12.4154325
sd(scale)             0.3232857  0.3855331  0.4597661
cor(asymptote,scale) -0.6726260 -0.5225912 -0.3312863
```

```
 Correlation structure:
           lower      est.     upper
Phi1   0.7549473 0.8217214 0.8716312
Theta1 0.2940804 0.4007537 0.5090448
Theta2 0.5607426 0.6827465 0.7757331
attr(,"label")
[1] "Correlation structure:"

 Within-group standard error:
   lower     est.    upper
1.291170 1.524274 1.799463
```

Chapter 14

Showcase: equivalence tests

14.1 Introduction

Model validation and environmental monitoring are essential components of the management and protection of biodiversity and natural resources. Yet, validation and monitoring are poorly served by traditional, established statistical tools.

Equivalence tests, described in detail below, are a relatively new branch of statistics. Equivalence tests are ideal for demonstrating that predictions from a model conform to field observations, or that field observations conform to legislated norms. Equivalence tests are more appropriate for these tasks than the traditional hypothesis tests.

A characteristic of the traditional hypothesis tests is that they start from the position that two things are the same, and try to prove otherwise (that is, they are designed to *split*). Equivalence tests start from the position that two things are different, and try to prove that they are the same. That is, equivalence tests are designed to *lump*.

The goal of both validation and monitoring is to compare measurements with some expectation, whether those be from model predictions, as in the case of a model validation exercise, or from a nominal standard, as in the case of environmental or natural resource monitoring. In each case, it is desirable to be able to conclude that the predictions and the observations are the same, or that the field observations meet the required standard.

Of course, we can never conclude that these quantities are identical, instead we wish to conclude that they are sufficiently similar that differences are irrelevant.

Various authors have recommended the use of equivalence tests instead of the traditional hypothesis tests for specific situations, for example in environmental management (McBride, 1999), biology (Parkhurst, 2001), ecology (Dixon and Pechmann, 2005), medical studies (see Berger and Hsu, 1996, among many others), psychology (Rogers et al., 1993), and model validation (Robinson and Froese, 2004).

Some equivalence tests are available in R via the equivalence package.

```
> library(equivalence)
```

14.2 TOST 1

A popular equivalence test is an intersection–union test called the TOST, for Two-One-Sided Tests, which can be used to try to assess the equivalence between the means of two populations based only on a sample from each (Schuirmann, 1981; Westlake, 1981). The TOST is applied as follows.

1. Choose
   (a) a test statistic, for example, the mean of the differences between the samples,
   (b) a *size*, which is essentially the probability of a false positive for the test, e.g. $\alpha = 0.05$, and
   (c) a *region of similarity*, which is a span of numbers within which we think that if the two population means are this far apart then we can treat them as though they were the same.
2. Compute the statistic and two one-sided confidence intervals, one lower and one upper, each with nominal coverage equal to $1 - \alpha$.

3. If the overlap of the two intervals computed in step 2 is *inside* the region determined in step 1c then conclude that the means of the populations are statistically similar.

This test is easy to apply and interpret, although it is not the most powerful available (Berger and Hsu, 1996; Wellek, 2003). Here, we run the TOST in R using the equivalence package.

We have a dataset of height and diameter measurements taken from the University of Idaho Experimental Forest (UIEF). In the summer of 1991, a stand examination was made of the Upper Flat Creek unit of the UIEF. 144 plots were located on a square grid, with north-south inter-plot distance of 134.11 m and east-west inter-plot distance of 167.64 m. A 7.0 m^2/ha BAF variable-radius plot was installed at each plot location. Every tree in the plot was measured for species and diameter at 1.37 m (dbh), recorded in mm, and a subsample of trees was measured for height, recorded in dm. The inventory data are stored in the `ufc` object, which is provided by the equivalence package.

Our goal is to assess whether the Prognosis height–diameter model is suitable for this stand. We will assume that the sampling units are the trees, and we will ignore the clustering in the sample design for the purposes of this demonstration, this point is discussed further below.

```
> data(ufc)
> str(ufc, strict.width = "cut")

'data.frame':        633 obs. of  10 variables:
 $ Plot       : int  1 2 2 3 3 3 3 3 3 ...
 $ Tree       : int  1 1 2 1 2 3 4 5 6 7 ...
 $ Species    : Factor w/ 13 levels "","DF","ES","F",..: 1 2 12 11 6..
 $ Dbh        : int  NA 390 480 150 520 310 280 360 340 260 ...
 $ Height     : int  NA 205 330 NA 300 NA NA 207 NA NA ...
 $ Height.ft  : num  NA 67.3 108.3 NA 98.4 ...
 $ Dbh.in     : num  NA 15.4 18.9 5.9 20.5 ...
 $ Height.ft.p: num  NA 83.5 107.3 44.2 106.1 ...
 $ Height.m   : num  NA 20.5 33 NA 30 NA NA 20.7 NA NA ...
 $ Height.m.p : num  NA 25.4 32.7 13.5 32.3 ...
```

The height predictions have already been computed for the trees. Hence our goal is to perform an equivalence test on the two variables: measured (`Height.m`) and predicted (`Height.m.p`) heights, both expressed in metres. Let's check these data first.

```
> lapply(ufc[, c("Height.m", "Height.m.p")], summary)

$Height.m
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
    0.0    18.7    24.0    24.0    29.0    48.0   248.0

$Height.m.p
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  8.921  20.100  25.240  24.260  28.980  46.350  18.000
```

Our test involves computing a confidence interval for the differences of two means. We'll need to invoke the central limit theorem (CLT) in order to compute the confidence interval. Sometimes we have to do this on blind faith, but here we have a sample of differences that we can assess for proximity to normality. We provide the qq-norm plot in Figure 14.1, using the following code.

```
> error.hats.m <- ufc$Height.m - ufc$Height.m.p
> qqnorm(error.hats.m)
> qqline(error.hats.m)
```

The plot shows good agreement with our needs in order to invoke the CLT, but also shows that two of the observations differ quite substantially from the predictions. Let's drill down and inspect the trees.

```
> subset(ufc, error.hats.m < -15,
+        select = c(Plot, Tree, Species, Dbh, Height.m, Height.m.p))
```

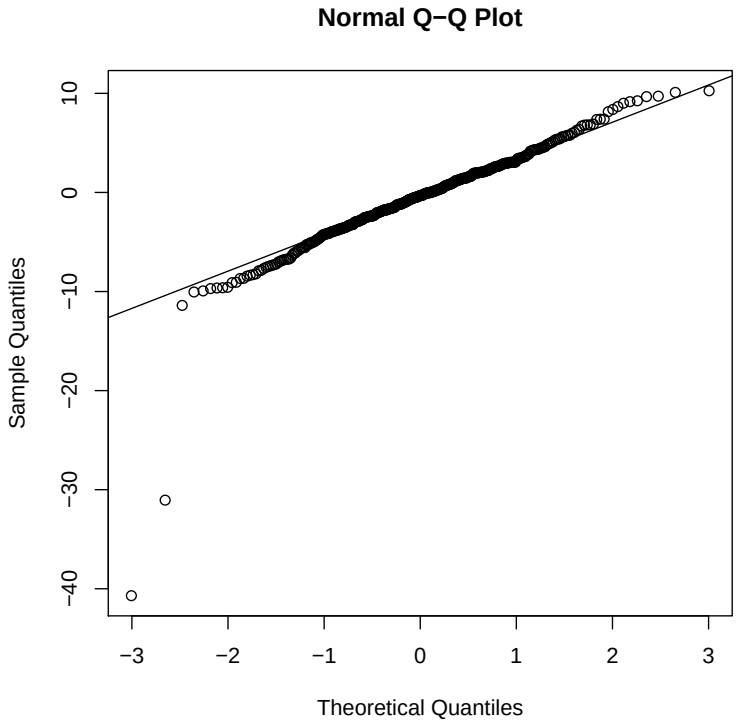

Figure 14.1: QQ norm plot of the treee height prediction errors, in metres.

```
    Plot Tree Species Dbh Height.m Height.m.p
372   67    6      WL 575      3.4   34.45379
411   78    5      WP 667      0.0   40.69514
```

The output shows that the two trees in question have very small heights for their diameters (recall that the diameters are measured in mm). We could quite reasonably exclude them from further consideration, and then feel comfortable with our use of the CLT.

```
> ufc.nice <- subset(ufc, error.hats.m > -15)
```

We use 2 metres as our region of indifference; that is, we'll say that if the average height of the predictions is demonstrably within 2 metres of the average height of the observations, then the null hypothesis should be rejected. Here, we save the test object that the function creates for subsequent examination.

```
> fvs.test <- tost(x = ufc.nice$Height.m.p,
+                  y = ufc.nice$Height.m,
+                  epsilon = 2)
```

The returned object is a list of objects, so we'll dig around it here.

```
> str(fvs.test)
```

```
List of 9
 $ mean.diff: num 0.412
 $ se.diff  : num 0.511
 $ alpha    : num 0.05
 $ ci.diff  : atomic [1:2] -0.429 1.254
  ..- attr(*, "conf.level")= num 0.9
 $ df       : Named num 738
  ..- attr(*, "names")= chr "df"
 $ epsilon  : num 2
```

```
 $ result   : chr "rejected"
 $ p.value  : num 0.000982
 $ check.me : atomic [1:2] -1.18 2
  ..- attr(*, "conf.level")= num 0.998
```

The object comprises the following pieces:

1. `mean.diff` is the difference between the sample means,
2. `se.diff` is the estimated standard error of the difference between the sample means,
3. `alpha` is the nominated size of the test,
4. `ci.diff` is the estimated $(1-\alpha)$ confidence interval of the difference between the population means,
5. `df` is the number of degrees of freedom used for the calculation,
6. `epsilon` is the nominated region of similarity,
7. `result` is the outcome of the hypothesis test — the null hypothesis is either rejected or not rejected,
8. `p.value` reports 1 minus the coverage of the largest possibly interval that will result in rejection of the null hypothesis, and
9. `check.me` reports the largest possibly interval that will result in rejection of the null hypothesis. As much as anything, it is produced to cross-check the reported p-value.

Here, the outcome of the hypothesis test is:

```
> fvs.test$result

[1] "rejected"
```

with p-value

```
> fvs.test$p.value

[1] 0.0009824095
```

We conclude that there is strong evidence to support the use of the FVS height–diameter model in this stand.

The rationale behind the interpretation of the p-value is the same as that that underpins the connection between confidence intervals and traditional (splitting) hypothesis tests: the p-value is 1 minus the coverage of the largest possible interval that includes the null hypothesis value.

14.3 Equivalence plot

Robinson et al. (2005) suggested an extension of the TOST that could be used to identify components of lack-of-fit, and provide an integrated summary of the relationship between two variables along with the necessary information to make inference. For want of a better name it is presently termed an "equivalence plot".

The equivalence plot involves making a scatterplot of the raw data, and imposing graphical images on the plot that summarize two TOST-based tests: a test on the intercept (equivalent to a test on the sample mean) and a test on the slope. The plot is called using the following code:

```
> ufc.ht <- ufc.nice[!is.na(ufc.nice$Height), ]
> equivalence.xyplot(ufc.ht$Height.m ~ ufc.ht$Height.m.p,
+     alpha = 0.05, b0.ii = 1, b1.ii = 0.1, b0.absolute = TRUE,
+     xlab = "Predicted height (m)", ylab = "Measured height (m)")
```

and is shown in Figure 14.2. We are comparing field measurements of tree height against predictions from models published by Wykoff et al. (1982). Here the test *rejects* the null hypothesis of difference from the intercept to 0 and the slope to 1. The mean of the model predictions is close to the mean of the height measurements, and the tall trees are predicted to be taller than the short trees. We can use this model for the population from which the sample came with some confidence.

It is worth noting that the quality of the match between the data and the model here is quite extraordinary, and we would normally not expect the test to clear quite so high a bar.

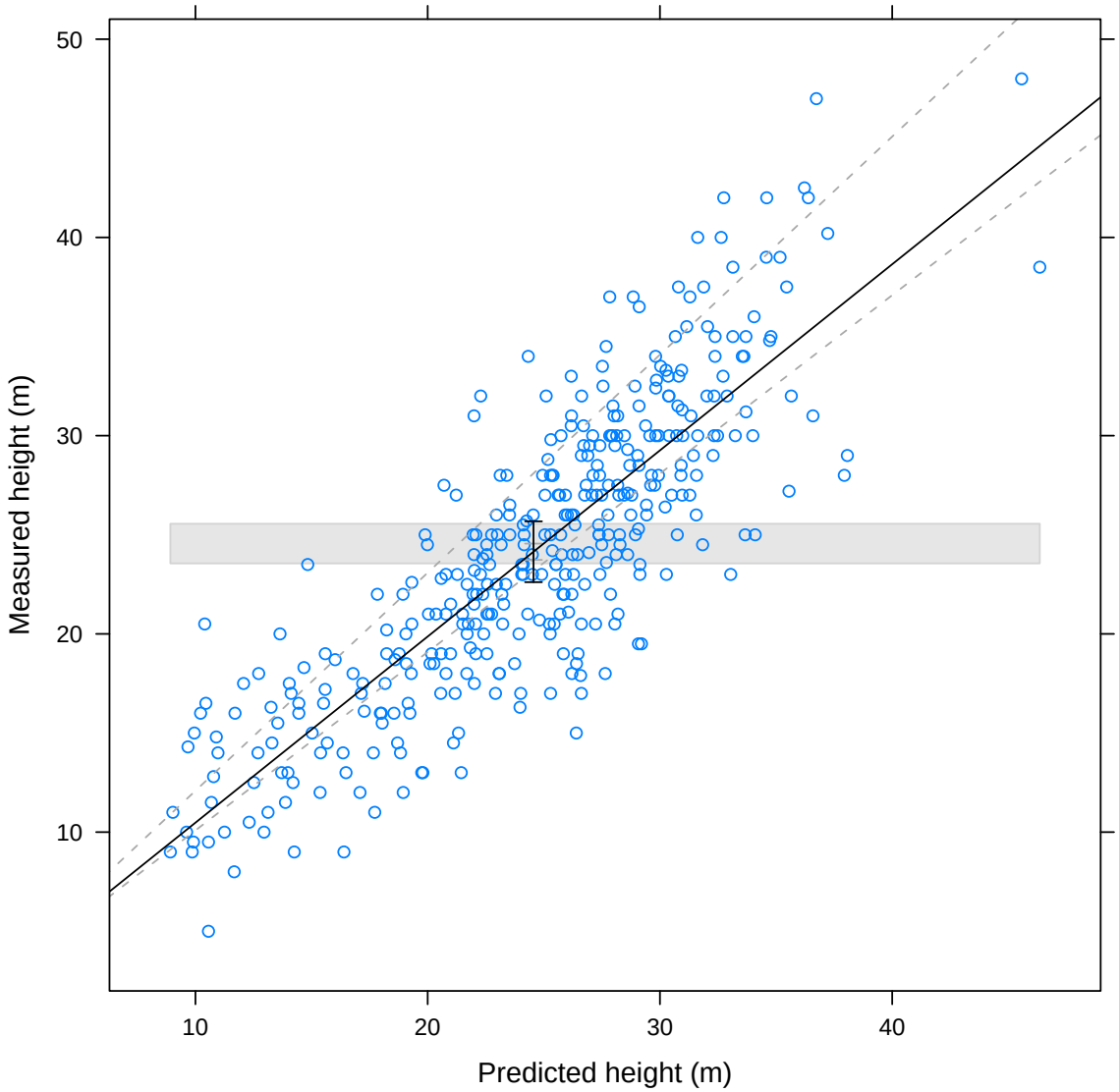

Figure 14.2: TOST embedded in a regression framework. The TOST for the intercept is carried out by assessing whether the (grey) intercept error bar is inside the grey rectangle, which represents a region of similarity of $\pm 1m$ of the mean. The TOST for the slope is carried out by assessing whether the (black) slope error bar is inside the dotted rays, which represent a region of similarity of $\pm 10\%$ of the slope. Here, the slope and intercept error bars are coincident, owing to the large sample size. The test rejects the null hypothesis of difference from the intercept to 0 and the slope to 1.

Chapter 15

Programming

One of R's greatest strengths is its extensibility. In fact, many of the tools that we've been enjoying were created by other people. R can be extended by writing functions using the R language (Section 15.1) or other languages such as C and Fortran (Section 15.6). Being comfortable writing functions is very useful, as it adds an entirely new dimension to efficient and enjoyable programming.

15.1 Functions

The topic of writing functions in R deserves its own workshop, so we'll merely skim the surface. We declare a function in the following way.

```
> my.function <- function(arguments) {
+ }
```

We then put R commands between the braces {}. Unless we use an explicit return() command, R will return the outcome of the last statement in the braces. If the function is only one line long, then the braces may be omitted. Here's a function that might be useful for NASA:

```
> cm.to.inches <- function(data) {
+     data/2.54
+ }
```

We can now call that function just like any other.

```
> cm.to.inches(25)

[1] 9.84252

> cm.to.inches(c(25, 35, 45))

[1]  9.84252 13.77953 17.71654
```

It's even vectorized (cf Section 6.3.1)! Of course this is trivial. But as far as R is concerned now, this function is just like any other.

Functions can be used in many different ways. One useful example is to gather together function calls that we often use. This function opens a file a field data, and optionally solves some common problems and imputes height measurements.

```
> get.field.data <- function(file = "../data/ufc.csv",
+                            clean = TRUE,
+                            impute = FALSE) {
+   field <- read.csv(file)
+   if (clean) {
+     field$dbh.cm <- field$dbh/10
+     field$height.m <- field$height/10
+     field$height.m[field$height.m < 0.1] <- NA
```

```
+     field$species[field$species %in% c("F","FG")] <- "GF"
+     field$species <- factor(field$species)
+     if (impute) {
+       require(nlme)
+       hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) * species,
+                     random = ~ I(log(dbh.cm)) | plot,
+                     data = field)
+       field$p.height.m <- exp(predict(hd.lme, level=1))
+       }
+     }
+   return(field)
+ }
```

More sophisticated versions would include error checking and reporting, more cleaning, and so on. Most likely a function like this will grow organically.

15.2 Flow Control

R provides access to a number of control structures for programming - `if()`, `for()`, and `while()`, for example. Take some time to look these up in the help system. In general it is best to avoid these functions if at all possible, as many problems that can be solved using these functions can be solved more quickly and with tighter code using vectorization (see section 6.3.1).

15.2.1 A Case Study

For the sake of example, though, imagine that we are interested in estimating how close to normally-distributed are the parameter estimates for a linear regression, and what the empirical joint distribution might look like. Let's do it now with an explicit loop. First we get the data.

```
> rugby <- read.csv("../data/rugby.csv")
> rugby$rules <- "New"
> rugby$rules[rugby$game < 5.5] <- "Old"
> rugby$game.no <- as.numeric(rugby$game)
> rugby$rules <- factor(rugby$rules)
> rugby$game <- factor(rugby$game)
> str(rugby)
```

```
'data.frame':        979 obs. of  4 variables:
 $ game   : Factor w/ 10 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ time   : num  39.2 2.7 9.2 14.6 1.9 17.8 15.5 53.8 17.5 27.5 ...
 $ rules  : Factor w/ 2 levels "New","Old": 2 2 2 2 2 2 2 2 2 2 ...
 $ game.no: num  1 1 1 1 1 1 1 1 1 1 ...
```

Looping

Then we construct the loop.

```
> reps <- 200                    # Choose a number of replicates
> differences <- rep(NA, reps)   # Prepare a receptacle
> (n <- nrow(rugby))             # Find the sample size
```

```
[1] 979
```

```
> system.time({                  # Just to keep time
+   for (i in 1:reps) {          # Execute once for each value of i
+     new.index <- sample(1:n, replace=TRUE)  # Sample the index
+     differences[i] <-          # Fit the model, extract the estimate.
+       coef(lm(time[new.index] ~ rules[new.index], data=rugby))[2]
+   }
+ })
```

```
   user  system elapsed
  1.199   0.082   1.331
```

Finally, we can examine the mean, standard error, and distribution of the parameter estimates (Figure 15.1).

```
> mean(differences)

[1] 3.407116

> sd(differences)

[1] 0.9208683

> qqnorm(differences, main = "QQ plot of Estimates")
> qqline(differences)
```

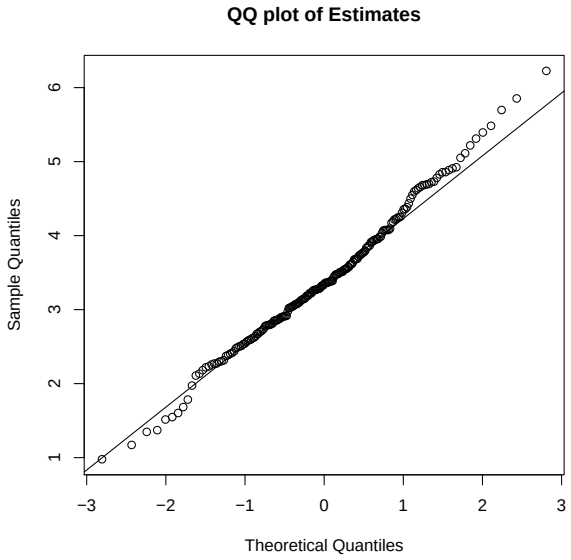

Figure 15.1: Normal quantile plot of random parameter estimates from loop.

15.2.2 Looping on other things

The `for()` is more generic than one might suspect, given its simplicity. The range of values to take the index from doesn't have to be sequential, or in any order, or even numbers. This can be very useful when you wish to, for example, read in a large number of files for later operations.

```
> (file.names <- list.files("../data"))

 [1] "DBT_Data_For_A_Robinson_20090702.csv"
 [2] "DBT_Metadata_ec28_20090702.doc"
 [3] "Sawlog_Data_For_A_Robinson_20090702.csv"
 [4] "blowdown.txt"
 [5] "ehc.csv"
 [6] "gutten.csv"
 [7] "hanford.csv"
 [8] "ndvi.csv"
 [9] "rugby.csv"
[10] "stage.csv"
[11] "ufc.csv"
[12] "wind-river.csv"
```

```
> (f.c <- nchar(file.names))
```

```
 [1] 36 30 39 12  7 10 11  8  9  9  7 14
```

```
> (file.names <- file.names[substr(file.names, f.c - 3, f.c) ==
+     ".csv"])
```

```
 [1] "DBT_Data_For_A_Robinson_20090702.csv"
 [2] "Sawlog_Data_For_A_Robinson_20090702.csv"
 [3] "ehc.csv"
 [4] "gutten.csv"
 [5] "hanford.csv"
 [6] "ndvi.csv"
 [7] "rugby.csv"
 [8] "stage.csv"
 [9] "ufc.csv"
[10] "wind-river.csv"
```

```
> for (my.file in file.names) {
+     in.file <- read.csv(paste("../data", my.file, sep = "/"))
+     cat(paste("File", my.file, "has", nrow(in.file), "rows.\n",
+         sep = " "))
+ }
```

```
File DBT_Data_For_A_Robinson_20090702.csv has 2319 rows.
File Sawlog_Data_For_A_Robinson_20090702.csv has 4821 rows.
File ehc.csv has 488 rows.
File gutten.csv has 1287 rows.
File hanford.csv has 9 rows.
File ndvi.csv has 62 rows.
File rugby.csv has 979 rows.
File stage.csv has 542 rows.
File ufc.csv has 637 rows.
File wind-river.csv has 5933 rows.
```

15.3 Scoping

When writing functions in some languages, we have to pass or declare all the variables that will be used in the function. This is called a scoping rule, and it means that the language always knows where to look to find its variables. R has a more relaxed approach: it will first look in the local function, and if it doesn't find what it's looking for, it will go up a level - in to the calling environment - and look there instead, and so on. This has two important consequences. Firstly, we don't have to pass all the variables that we might want to use. This makes code neater and easier to read in many cases. However, secondly, you may find upstream variables affecting downstream results if you have a mistake in your code, and you'll never know about it. So, whenever I develop code I make a habit of saving the script, deleting all the objects, and running it all again, perhaps several times. Scoping is important - it can save you a lot of hassle, but it can cost a lot of time if you're not careful.

```
> x <- pi
> radius.to.area <- function(radius) {
+     x * radius^2
+ }
> radius.to.area(4)
```

```
[1] 50.26548
```

15.4 Debugging

Advice that pertains to debugging in general is useful for R also. That is, profile your code using simple inputs where you are confident that you know what the answer should be, provide generous commenting and ensure that you examine the intermediate steps.

Also, use `browser()` inside your functions. `browser()` will stop the processing, and allow you to inspect all the objects in, and in some cases out, of your function. See `?browser` for more information. You can use

```
> eval(expression(function()),
+     envir = sys.frame(n),
+     enclos = NULL)
```

to execute `function()` in relative environment `n`, which might, for example, be the environment from which your function has been called. For example, to list the objects that are one level up, you might do the following.

```
> rm(list = ls())
> ls()

character(0)

> x <- 2
> ls.up <- function() {
+     eval(expression(ls()), envir = sys.frame(-1), enclos = NULL)
+ }
> ls.up()

[1] "ls.up" "x"
```

This example is not particularly useful in and of itself, but shows how to construct commands that operate in different environments, and such commands are very useful when debugging inside a browser.

15.5 S3 Classes and Objects

It is not mandatory to apply the principles of object-oriented programming when writing R code, but doing so can have advantages. Here I briefly demonstrate the use of S3-style classes. Briefly, we will create a class, which is a specific type of object, and we will write functions that apply specifically to that class. The functions will replace so-called *generic* functions, such as `print()`, `plot()`, and `mean()`.

Note that there is no such class as `inventory`.

```
> methods(class = "inventory")

no methods were found
```

Let us invent one. Let's say that the `inventory` class is a `list`, with certain extra attributes. To start we write a function that creates the class from a file and extra information.

```
> inventory <- function(file = "../data/ufc.csv",
+                       clean = TRUE,
+                       impute = FALSE,
+                       design = c("VRP","FAP"),
+                       weight = NULL,
+                       area = NULL
+                       ) {
+   data <- read.csv(file)
+   if (clean) {
+     data$dbh.cm <- data$dbh/10
+     data$height.m <- data$height/10
+     data$height.m[data$height.m < 0.1] <- NA
+     data$species[data$species %in% c("F","FG")] <- "GF"
+     data$species <- factor(data$species)
+     if (impute) {
+       require(nlme)
+       hd.lme <- lme(I(log(height.m)) ~ I(log(dbh.cm)) * species,
```

```
+                     random = ~ I(log(dbh.cm)) | plot,
+                     na.action=na.exclude,
+                     data = data)
+       predicted.log.heights <-
+         predict(hd.lme, na.action=na.omit, newdata=data)
+       data$p.height.m[!is.na(data$dbh.cm)] <- exp(predicted.log.heights)
+       data$p.height.m[!is.na(data$height.m)] <-
+         data$height.m[!is.na(data$height.m)]
+     } else {
+       data$p.height.m <- data$height.m
+     }
+   }
+   results <- list(data = data,
+                   design = design,
+                   weight = weight,
+                   area = area)
+   class(results) <- "inventory"
+   return(results)
+ }
```

The class will assume that there are only two kinds of inventories; variable-radius plots (VRP), in which case the weight argument will be interpreted as the BAF, and fixed-area plots (FAP), in which case the weight argument will be interpreted as the plot size in hectares. Notice the assumptions that the function is making about the format of the data, and what data are being included.

Now we write a print function that will be used for all objects of class `inventory`. We will assume that the goal of printing the inventory is a description of the data. Printing can then be invoked in one of two ways.

```
> print.inventory <- function(x) str(x)
```

Now we can try it out, e.g.

```
> ufc <- inventory(file = "../data/ufc.csv",
+                  clean = TRUE,
+                  impute = TRUE,
+                  design = "VRP",
+                  weight = 7,
+                  area = 300)
> ufc

List of 4
 $ data  :'data.frame':        637 obs. of  8 variables:
  ..$ plot      : int [1:637] 1 2 2 3 3 3 3 3 3 3 ...
  ..$ tree      : int [1:637] 1 1 2 1 2 3 4 5 6 7 ...
  ..$ species   : Factor w/ 11 levels "","DF","ES","GF",..: 1 2 10 9 4 9 9 9 9 9 ...
  ..$ dbh       : int [1:637] NA 390 480 150 520 310 280 360 340 260 ...
  ..$ height    : int [1:637] NA 205 330 NA 300 NA NA 207 NA NA ...
  ..$ dbh.cm    : num [1:637] NA 39 48 15 52 31 28 36 34 26 ...
  ..$ height.m  : num [1:637] NA 20.5 33 NA 30 NA NA 20.7 NA NA ...
  ..$ p.height.m: num [1:637] NA 20.5 33 13.8 30 ...
 $ design: chr "VRP"
 $ weight: num 7
 $ area  : num 300
 - attr(*, "class")= chr "inventory"
```

Now we can start to think about how we interact with inventory data. For example, we might decide that a plot of inventory data should always produce a set of four panels, thus:

```
> plot.inventory <- function(inventory, ...) {
+   plot(inventory$data$dbh.cm, inventory$data$height.m,
```

```
+        main="Height v. Diameter",
+        xlab="Dbh (cm)", ylab="Height (m)", ...)
+   require(MASS)
+   truehist(inventory$data$dbh.cm, h=5, x0=0, xlab="Dbh (cm)",
+            main="Diameter Distribution (unweighted)", ...)
+   boxplot(dbh.cm ~ species, data=inventory$data, ylab="Dbh (cm)",
+           varwidth=TRUE)
+   boxplot(height.m ~ species, data=inventory$data, xlab="Height (m)",
+           varwidth=TRUE)
+   }
```

Note firstly that we had to extract the data from two layers of infrastructure, using a sequence of `$` signs, and secondly, that it was very easy to do so. The role of the three dots in the argument list is to alert the function that it may receive other arguments, and if it does it should expect to pass those arguments on to the `plot()` and `boxplot()` functions unaltered.

We then use this function as follows, to create Figure 15.2:

```
> par(mfrow = c(2, 2), mar = c(4, 4, 3, 1), las = 1)
> plot(ufc, col = "seagreen3")
```

In brief, then, to write a function, say `fu()`, that will be *automatically* invoked to replace an existing generic function `fu()` when applied to an object of class `bar`, we need merely call the function `fu.bar`.

Non-generic functions can also be written, but don't require the trailing object name. For example, we wish to compute a volume for all the trees, but to have options amongst our many volume functions.

```
> compute.tree.volume <- function(inventory, vol.function) {
+   tree.volume.m3 <- do.call(vol.function, list(inventory$data$species,
+                                                inventory$data$dbh.cm,
+                                                inventory$data$p.height.m))
+   return(tree.volume.m3)
+ }
```

Again, note that we are making important assumptions about what information will be available to the function, and how it will interact with it. For example, we are assuming that there will be a variable called `p.height.m` available for each tree. We will use the northerm Idaho FVS volume function from Wykoff et al. (1982), but with the unit conversions inside the function rather than outside.

```
> vol.fvs.ni.m3 <- function(spp, dbh.cm, ht.m){
+   dbh.in <- dbh.cm / 2.54
+   ht.ft <- ht.m * 3.281
+   bf.params <-
+     data.frame(
+                 species = c("WP", "WL", "DF", "GF", "WH", "WC", "LP", "ES",
+                   "SF", "PP", "HW"),
+                 b0.small = c(26.729, 29.790, 25.332, 34.127, 37.314, 10.472,
+                   8.059, 11.851, 11.403, 50.340, 37.314),
+                 b1.small = c(0.01189, 0.00997, 0.01003, 0.01293, 0.01203,
+                   0.00878, 0.01208, 0.01149, 0.01011, 0.01201, 0.01203),
+                 b0.large = c(32.516, 85.150, 9.522, 10.603, 50.680, 4.064,
+                   14.111, 1.620, 124.425, 298.784, 50.680),
+                 b1.large = c(0.01181, 0.00841, 0.01011, 0.01218, 0.01306,
+                   0.00799, 0.01103, 0.01158, 0.00694, 0.01595, 0.01306)
+                 )
+   dimensions <- data.frame(dbh.in = dbh.in,
+                            ht.ft = ht.ft,
+                            species = as.character(spp),
+                            this.order = 1:length(spp))
+   dimensions <- merge(y=dimensions, x=bf.params, all.y=TRUE, all.x=FALSE)
+   dimensions <- dimensions[order(dimensions$this.order, decreasing=FALSE),]
+   b0 <- with(dimensions, ifelse(dbh.in <= 20.5, b0.small, b0.large))
```

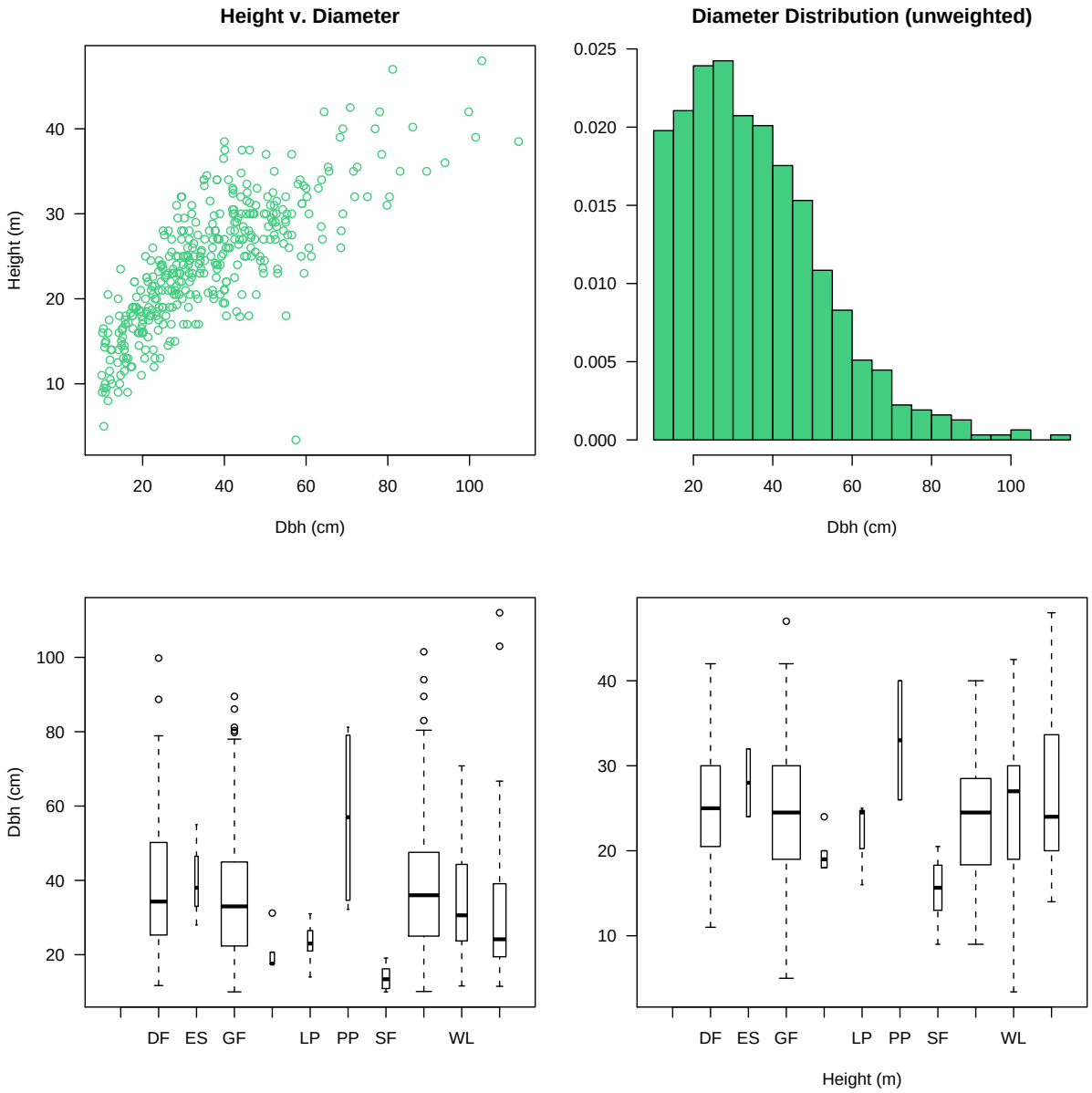

Figure 15.2: Summary graphic for an *inventory* object.

```
+   b1 <- with(dimensions, ifelse(dbh.in <= 20.5, b1.small, b1.large))
+   volumes.bf <- b0 + b1 * dimensions$dbh.in^2 * dimensions$ht.ft
+   volumes.m3 <- volumes.bf * 0.002359737
+   return(volumes.m3)
+ }
```

Now we can compute and save the tree-level volumes in the ufc inventory object as follows:

```
> ufc$data$vol.m3 <- compute.tree.volume(ufc, vol.fvs.ni.m3)
```

Note that adding a column to the data attribute of the object hasn't changed the object's class.

```
> class(ufc)

[1] "inventory"
```

Finally, we write a mean function that produces a mean per hectare of a single nominated attribute, rather than the mean of the attribute.

```
> mean.inventory <- function(inventory, attribute, ...) {
+   data <- inventory$data[, names(inventory$data) == attribute]
+   if (inventory$design == "VRP") {
+     tree.factor <- inventory$weight * 40000 /
+       (inventory$data$dbh.cm ^ 2 * pi)
+   } else {
+     tree.factor <- 1 / inventory$weight
+   }
+   total <- sum(data * tree.factor, na.rm = TRUE)
+   mean <- total / length(unique(inventory$data$plot))
+   return(mean)
+ }
> mean(ufc, "vol.m3")

[1] 149.5207
```

We are starting to build a nice little collection of class-specific functions for our new class.

```
> methods(class = "inventory")

[1] mean.inventory  plot.inventory  print.inventory
```

There is a final caveat. What if more than one author writes a set of functions that require an inventory class, and the class definitions are incompatible? This is one of the problems that S4 classes are designed to sidestep. Coverage of S4 classes is beyond this workshop, but see Chambers (1998) for much background information.

15.6 Other languages

Just an anecdote here, with some instructions on how to compile and run your own programs in a Unix environment. I was working with Robert Froese, my first doctoral student, on developing tests of equivalence for model validation. We were using Forest Inventory and Analysis (FIA) data to validate the diameter engine of the Forest Vegetation Simulator (FVS) by species. That amounts to 41000 tree records for 10 species. The best-represented is Douglas-fir, with 12601 observations, the least is white pine, with 289, and the average is 3725.

```
> (n <- tapply(species, species, length))

 ABGR  ABLA  LAOC  OTHE  PICO  PIEN  PIMO  PIPO  PSME  THPL  TSHE
 3393  4532  3794  1629  6013  3251   289  2531 12601  2037   909
```

This will all matter shortly.

We were applying the sign rank test. We'd love to have used the paired t-test: it's simple and straightforward, and doesn't require any devious looping. But unfortunately, the prediction errors were far from normal; they were quite badly skewed. So we decided to use this non-parametric test. The problem, or should I say challenge, or even `opportunity`, was that calculating the metric requires a double loop across the data sets.

$$\hat{U}_+ = \binom{n}{2}^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I_+(D_i + D_j) \tag{15.1}$$

where $I_+(x)$ is an indicator function that takes value 1 if $x > 0$ and 0 otherwise. That's

```
> sum(choose(n,2))
[1] 133017037
```

more than 100 million operations. But that's not all! In order to estimate the variance of the metric we had to implement a *triple* loop across the datasets. And at each of those combinations we have to do about seven things.

$$\hat{\sigma}^2_{\hat{U}_+} = \binom{n}{2}^{-1} \{2(n-2)[\hat{q}^+_{1(2,3)} - \hat{U}^2_+) + \hat{U}_+ - \hat{U}^2_+\} \tag{15.2}$$

where

$$\hat{q}^+_{1(2,3)} = \binom{n}{3}^{-1} \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{3} \left[I_{ij} I_{ik} + I_{ij} I_{jk} + I_{ik} I_{jk} \right] \tag{15.3}$$

for which, say, $I_{ij} = I_+(D_i + D_j)$. That's now

```
> sum(choose(n,3))*7
[1] 2.879732e+12
```

about 2.9 trillion operations! Well, I expect that this will take some time. So I wrote up a script,

```
n <- length(diffs)
cat("This is the sign rank test for equivalence.\n")
zeta <- 0
for (i in 1:(n-1)) {
    for (j in (i+1):n) {
      zeta <- zeta + ((diffs[i] + diffs[j]) > 0)
    }
  }
Uplus <- zeta / choose(n, 2)
cat("UPlus ", Uplus, ". \n")
zeta <- 0
for (i in 1:(n-2)) {
  for (j in (i+1):(n-1)) {
```

```
      for (k in (j+1):n) {
        IJpos <- ((diffs[i] + diffs[j]) > 0)
        IKpos <- ((diffs[i] + diffs[k]) > 0)
        JKpos <- ((diffs[j] + diffs[k]) > 0)
        zeta <- zeta + IJpos*IKpos + IJpos*JKpos + IKpos*JKpos
      }
    }
    cat("Triple loop. We have ", n-2-i, "iterations left from ", n,". \n")
  }
  q123 <- zeta / 3 / choose(n, 3)
```

fired up my remote server and waited. All night. By which point it hadn't even got through maybe 10% of the first species (a smallish one!). I didn't want to wait that long. It seemed like this was absolutely the case that the dynamical loading of compiled code was designed for.

15.6.1 Write

The code has to be modular, and written so that all communication is through the passed arguments. So, in C it must always be type void, and in Fortran it must be a subroutine. Here's the example:

```
void signRankSum(double *a, int *na, double *zeta)
{
  int i, j, k, m;
  int IJpos, IKpos, JKpos;
  zeta[0] = ((a[i] + a[j]) > 0);
  zeta[1] = 0;
  for (i = 0; i < *na-2; i++) {
    for (j = i+1; j < *na-1; j++) {
      zeta[0] += ((a[i] + a[j]) > 0);
      for (k = j+1; k < *na; k++) {
        IJpos = ((a[i] + a[j]) > 0);
        IKpos = ((a[i] + a[k]) > 0);
        JKpos = ((a[j] + a[k]) > 0);
        zeta[1] += (IJpos*IKpos + IJpos*JKpos + IKpos*JKpos);
      }
    }
  }
  for (m = 0; m < *na-1; m++)
    zeta[0] += ((a[m] + a[*na-1]) > 0);
}
```

Basically all I did was take the triple loop and convert it to C code. Note that I passed the length of the object as well, rather than trusting C (well, trusting me) to figure it out. Do in R what you can do well in R.

- Also notice that C indexes differently than R: the first element of an array is element 0, *not* element 1! That can bite your bum.
- Also note the use of pointers. These are documented elsewhere. For simple problems, they're easy to figure out.
- I also figured out how to nest the double loop inside the triple loop, which accounts for some performance enhancement.
- Finally, notice that the arguments are the inputs and the outputs, and this subroutine doesn't care how long they are.

15.6.2 Compile

Compilation for Linux or BSD is easy. Simply be on a Linux or BSD box, create the above file (called e.g. signRankSum.c) and in the same directory as the file, type:

```
$ R CMD SHLIB signRankSum.c
```

It will compile and link the code, and create an object that you can link to in the R session (see below). If it can't, it will present reasonably helpful error messages.

On Windows, the tools that we need to develop software are a little more scattered. Refer to the available resources for advice on this point. Briefly, one needs to:

- Install Unix-like command-line tools for compilation and linking, e.g. MinGW and some others.
- Edit your PATH variable so that these are visible to the OS.
- Write appropriate code in appropriately named directories.
- In the DOS shell, execute: `R CMD SHLIB signRankSum.c`

15.6.3 Attach

In the body of the function that I'm writing to apply the test, I added the following call:

```
dyn.load("~/Rosetta/Programming/r/signRankSum/signRankSum.so")
signRankSum <- function(a)
  .C("signRankSum",
     as.double(a),
     as.integer(length(a)),
     zeta = double(2))$zeta
```

This tells R where to find the library. Note that I included the full path to the *.so object, and in the declaration of the function I named the C subroutine and the arguments. I tell it that when I signRankSum(a), it should know that a points to a double-precision array, that it will need the length of `a`, which is an integer, and zeta, which is a size two, double-precision array for the output. Then the `$zeta` at the end tells it to return that array.

15.6.4 Call

Later in the function it's time to call the routine. That's easy:

```
zeta <- signRankSum(diffs)
Uplus <- zeta[1] / choose(n, 2)
cat("UPlus ", Uplus, ". \n")
q123 <- zeta[2] / 3 / choose(n, 3)
```

Note that we're back in R, now, so the first element is referred to as 1.

15.6.5 Profit!

The following output reports a comparison of the code using C first and the code in pure R, second.

```
> test <- rnorm(1000)
> system.time(sgnrk(test, limit=1000))
This is the sign rank test for equivalence.
Starting. Actual:  1000 .  Used:  1000 .
UPlus  0.484042 .
VarUplus is calculated:  0.0003321017
Levels are calculated:  0.2397501  and  0.76025 , ncp is  203.9437
Cutoff:  12.63603
Value computed:  -0.8756727 .
Finished. Outcome: dissimilarity is  rejected.
[1] 3.882812 0.000000 3.906250 0.000000 0.000000
> system.time(sgnrk.old(test, limit=1000))
This is the sign rank test for equivalence.
Starting. Actual:  1000 .  Used:  1000 .
```

```
UPlus  0.484042 .
VarUplus is calculated:  0.0003321017
Levels are calculated:  0.2397501  and  0.76025 , ncp is  203.9437
Cutoff:  12.63603
Value computed:  -0.8756727 .
Finished. Outcome: dissimilarity is  rejected.
[1] 5092.87500   14.27344 5120.23438    0.00000    0.00000
>
```

Yes, that's four seconds compared to nearly an hour and a half. And the time to complete the test for all species and all data was:

```
[1] 9605.687500    5.367188 9621.734375    0.000000    0.000000
```

less than 3 *hours*, instead of 169 *days*.

This example was quite extreme, requiiring a triple-loop across a large dataset. However, moving to C can make substantial gains even under more modest circumstances.

Bibliography

Bates, D. M., Watts, D. G., 1988. Nonlinear regression analysis and its applications. John Wiley & Sons, Inc., 365 p. 83

Berger, R. L., Hsu, J. C., 1996. Bioequivalence trials, intersection-union tests and equivalence confidence sets. Statistical Science 11 (4), 283–319. 138, 139

Box, G. E. P., 1953. Non-normality and tests on variances. Biometrika 40 (3/4), 318–335. 109

Casella, G., Berger, R. L., 1990. Statistical Inference. Duxbury Press, 650 p. 71

Chambers, J. M., 1998. Programming with data: A guide to the S language. Springer, 469 p. 151

Daubenmire, R., 1952. Forest vegetation of Northern Idaho and adjacent Washington, and its bearing on concepts of vegetation classification. Ecological Monographs 22, 301–330. 94

Davison, A. C., Hinkley, D. V., 1997. Bootstrap methods and their application. Cambridge University Press, 582 p. 77, 81

Demidenko, E., Stukel, T. A., 2005. Influence analysis for linear mixed-effects models. Statistics in Medicine 24, 893–909. 126

Dixon, P. M., Pechmann, J. H. K., 2005. A statistical test to show negligible trend. Ecology 86 (7), 1751–1756. 138

Efron, B., Tibshirani, R. J., 1993. An Introduction to the Bootstrap. No. 57 in Monographs on Statistics and Applied Probability. Chapman and Hall, 115 Fifth Hall, New York, NY 10003. 77

Fitzmaurice, G., Laird, N., Ware, J., 2004. Applied Longitudinal Analysis. John Wiley & Sons, 506 p. 91

Gallant, A. R., 1987. Nonlinear statistical models. John Wiley & Sons, 624 p. 83

Gelman, A., Hill, J., 2007. Data Analysis Using Regression and Multilevel/Hierarchical Models. Cambridge University Press, 625 p. 80, 91, 92

Hollings, Triggs, 1993. Influence of the new rules in international rugby football: Implications for conditioning, Unpublished. 78

Huber, P. J., 1981. Robust Statistics. John Wiley & Sons, Inc., 308 p. 80

Laird, N. M., Ware, J. H., 1982. Random-effects models for longitudinal data. Biometrics 38, 963–974. 102

Lee, A., 1994. Data Analysis: An introduction based on R. Department of Statistics, University of Auckland. 77

McBride, G. B., 1999. Equivalence tests can enhance environmental science and management. Australian and New Zealand Journal of Statistics 41 (1), 19–29. 138

Parkhurst, D. F., 2001. Statistical significance tests: equivalence and reverse tests should reduce misinterpretation. Bioscience 51 (12), 1051–1057. 138

Pinheiro, J. C., Bates, D. M., 2000. Mixed-effects models in S and Splus. Springer-Verlag, 528 p. 84, 91, 100, 108, 117, 120, 125, 134

Ratkowsky, D. A., 1983. Nonlinear regression modeling. Marcel Dekker, New York, 276 p. 83, 85

Robinson, A. P., Duursma, R. A., Marshall, J. D., 2005. A regression-based equivalence test for model validation: shifting the burden of proof. Tree Physiology 25, 903–913. 141

Robinson, A. P., Froese, R. E., 2004. Model validation using equivalence tests. Ecological Modelling 176, 349–358. 138

Robinson, A. P., Wykoff, W. R., 2004. Imputing missing height measures using a mixed-effects modeling strategy. Canadian Journal of Forest Research 34, 2492–2500. 19

Robinson, G. K., 1991. That BLUP is a good thing: The estimation of random effects. Statistical Science 6 (1), 15–32. 91

Rogers, J. L., Howard, K. I., Vessey, J. T., 1993. Using significance tests to evaluate equivalence between two experimental groups. Psychological Bulletin 113 (3), 553–565. 138

Schabenberger, O., 2005. Mixed model influence diagnostics. In: SUGI 29. SAS Institute, pp. Paper 189–29. 126

Schabenberger, O., Pierce, F. J., 2002. Contemporary statistical models for the plant and soil sciences. CRC Press, 738 p. 85, 91, 103

Schuirmann, D. L., 1981. On hypothesis testing to determine if the mean of a normal distribution is contained in a known interval. Biometrics 37, 617. 138

Seber, G. A. F., Wild, C. J., 2003. Nonlinear regression. John Wiley & Sons, 792 p. 83

Stage, A. R., 1963. A mathematical approach to polymorphic site index curves for grand fir. Forest Science 9 (2), 167–180. 104, 112

Venables, W. N., Ripley, B. D., 2000. S programming. Springer-Verlag, 264 p. 92

Venables, W. N., Ripley, B. D., 2002. Modern applied statistics with S and Splus, 4th Ed., 4th Edition. Springer-Verlag, 495 p. 5, 69, 79, 91

von Guttenberg, A. R., 1915. Growth and yield of spruce in Hochgebirge. Franz Deuticke, Vienna, 153 p. (In German). 83

Weisberg, S., 2005. Applied Linear Regression, 3rd Edition. John Wiley & Sons, Inc., 310 p. 80

Wellek, S., 2003. Testing statistical hypotheses of equivalence. Chapman and Hall/CRC. 139

Westlake, W. J., 1981. Response to T.B.L. Kirkwood: bioequivalence testing–a need to rethink. Biometrics 37, 589–594. 138

Wilkinson, L., 2005. The Grammar of Graphics, 2nd Edition. Springer, 694 p. 59

Wood, S. N., 2006. Generalized additive models: an introduction with R. Chapman & Hall/CRC, 391 p. 87, 88

Wykoff, W. R., Crookston, N. L., Stage, A. R., 1982. User's guide to the Stand Prognosis model. GTR-INT 133, USDA Forest Service. 19, 141, 149

Appendix A

Extensibility

One notable thing about R is how quickly it loads. This is because much of its functionality is kept in the background, ignored, until it is explicitly asked for. There are three layers of functions in R:

1. those that are loaded by default at startup (`base`),
2. those that are loaded on to your hard drive upon installation but not explicitly loaded into RAM when R is run, and
3. those that are available on the Internet, which have to be installed before they can be loaded.

A function in the `base` package, such as `mean()`, is always available. A function in one of the loaded packages, such as the linear mixed-effects model fitting tool `lme()`, can only be used by first loading the package. We load the package using the `require()` command. `help.search()` will tell you which package to load to use a command, or to get more help on it.

```
> require(nlme)
```

If we need to find out what kinds of packages are installed, R will tell us, but, characteristically, in the most general fashion. We need to work with the output to get the information that we want.

```
> installed.packages()           # Blurts output all over the screen

> sessionInfo()                  # Learn about the current state of your session.

R version 2.9.0 (2009-04-17)
i386-apple-darwin8.11.1

locale:
C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] nlme_3.1-94

loaded via a namespace (and not attached):
[1] grid_2.9.0      lattice_0.17-25

> ip <- installed.packages()     # Save the output as an object!
> class(ip)                      # It is a matrix

[1] "matrix"

> ip <- as.data.frame(ip)        # Now it is a dataframe
> names(ip)                      # These are the variable names
```

```
 [1] "Package"  "LibPath"  "Version"  "Priority" "Bundle"   "Contains"
 [7] "Depends"  "Imports"  "Suggests" "OS_type"  "Built"

> length(ip$Package)             # And the packages are numerous

[1] 150

> head(ip$Package)

    AER    DAAG     DBI  Design   Ecdat Formula
    AER    DAAG     DBI  Design   Ecdat Formula
150 Levels: AER DAAG DBI Design Ecdat Formula HSAUR Hmisc KernSmooth ... zoo
```

Your package names will differ from mine. To find out what kinds of packages are available, use

```
> available.packages()           # Blurts output all over the screen
```

A function in an as-yet unloaded package is inaccessible until the package is downloaded and installed. This is very easy to do if one is connected to the Internet, and has administrator (root) privileges. Let's say, for example, that through searching on the R website we determine that we need the wonderful package called "equivalence". Then we simply install it using the Packages menu or by typing:

```
> install.packages("equivalence")
```

After it has been installed then we can load it during any particular session using the `require(equivalence)` command, as above.

If we do not have sufficient privileges to install packages then we are required to provide more information to R. Specifically, we need to download and install the package to a location in which we do have write privileges, and then use the `library` command to attach it to our session.

Let's install `equivalence`, using a folder called "library". The `download.packages()` command below will grab the appropriate version for your operating system and report the version number, which we save in the object called `get.equivalence`. We can use its output in the subsequent steps.

```
> (get.equivalence <- download.packages("equivalence",
+                                       destdir="../library",
+                                       repos="http://cran.stat.sfu.ca/"))
> install.packages(get.equivalence[1,2], repos=NULL, lib="../library")
> library(equivalence, lib.loc="../library")
```

Ah, equivalence needs `boot`. Ok,

```
> (get.boot <- download.packages("boot",
+                                destdir="../library",
+                                repos="http://cran.stat.sfu.ca/"))
> install.packages(get.boot[1,2], repos=NULL, lib="../library")
> library(boot, lib.loc="../library")
> library(equivalence, lib.loc="../library")
```

The equivalence package is now available for your use. Please note again the use of forward slashes as directory delimiters is platform independent.