ice breakeR

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

February 15 and 17, 2006

Contents

List of Figures **4**

1 Introduction **4**
1.1 R . . . 4
1.2 Why R? . . . 4
1.3 Why not R? . . . 5
1.4 The Open Source Ideal . . . 5

2 Infrastructure **6**
2.1 Using this Document . . . 6
2.2 Getting Help . . . 6
2.3 Working Directory . . . 7
2.4 Work Spaces . . . 7
2.5 History . . . 8
2.6 Writing scripts . . . 8

3 Interface **9**
3.1 Importing and Exporting Data . . . 9
3.1.1 Import . . . 10
3.1.2 Export . . . 11

4 Manipulating your Data **12**
4.1 Classes of Data . . . 12
4.1.1 Numeric . . . 12
4.1.2 String . . . 12
4.1.3 Factor . . . 13
4.1.4 Logical . . . 13
4.1.5 Missing Data . . . 13
4.2 Structures for Data . . . 14
4.2.1 Vector . . . 14
4.2.2 Dataframe . . . 15
4.3 Data References . . . 16

5 Graphics **18**
5.1 Organization Parameters . . . 19
5.2 Permanence . . . 19

6 Linear Regression **21**
6.1 Preparation . . . 21
6.2 Fitting . . . 23
6.3 Diagnostics . . . 23
6.4 Other Tools . . . 25
6.5 Examining the Model . . . 25

6.6 Other Angles . . . 26
6.7 Other Models . . . 26
6.8 Other Ways of Fitting . . . 26

7 More Graphics **27**
7.1 Trellis . . . 27

8 Hierarchical Models **30**
8.1 Introduction . . . 30
8.1.1 Methodological . . . 30
8.1.2 General . . . 31
8.2 Some Theory . . . 31
8.2.1 Effects . . . 31
8.2.2 Model Construction . . . 32
8.2.3 The Deep End . . . 36
8.2.4 Maximum Likelihood . . . 36
8.2.5 Restricted Maximum Likelihood . . . 37
8.3 A Simple Example . . . 37
8.4 Case Study . . . 39
8.4.1 Height/Diameter Data . . . 40
8.4.2 Extensions to the model . . . 60
8.5 The Model . . . 61
8.5.1 Z . . . 63
8.5.2 b . . . 63
8.5.3 D . . . 63

9 Extensibility - R Packages **64**

10 Programming **66**
10.1 Functions . . . 66
10.2 Scoping . . . 66
10.3 S3 Objects . . . 67
10.4 Control . . . 68
10.5 Other languages . . . 68
10.5.1 Write . . . 69
10.5.2 Compile . . . 70
10.5.3 Attach . . . 70
10.5.4 Call . . . 71
10.5.5 Benefit . . . 71

Bibliography **71**

List of Figures

5.1 Diameter/Height plot for all species of Upper Flat Creek inventory data. 18

6.1 Diagnostic plots for the regression of diameter against height. 24
6.2 Parameter estimate change as a result of dropping the outliers. 25

7.1 A random plot of coloured dots. 28
7.2 A lattice plot of height against predicted height by species for the four species that have the most trees. 29

8.1 Al Stage's Grand Fir stem analysis data: height (ft) against diameter (in). These were dominant and co-dominant trees. 34
8.2 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. 35
8.3 A simple dataset to show the use of mixed-effects models. 38
8.4 An augmented plot of the basic mixed-effects model with random intercepts fit to the sample dataset. 39
8.5 A sample plot showing the difference between basic.1 (single line), basic.2 (intercepts are fixed), and basic.4 (intercepts are random). 39
8.6 Regression diagnostics for the ordinary least squares fit of the Height/Diameter model with habitat type for Stage's data. 42
8.7 Selected diagnostics for the mixed-effects fit of the Height/Diameter ratio against habitat type and national forest for Stage's data. 46
8.8 The parameter estimates for the fixed effects and predictions for the random effects resulting from omitting one observation. 48
8.9 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. 49
8.10 Selected overall diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data. 52
8.11 Selected quantile-based diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data. 53
8.12 Selected random-effects based diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data. 54
8.13 Height against diameter by tree, augmented with predicted lines. 55
8.14 Selected diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data. 56
8.15 Selected diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data. 57
8.16 Innermost Pearson residuals against fitted values by habitat type. 58
8.17 Quantile plots of innermost Pearson residuals against the normal distribution, by habitat type. 58
8.18 Height against diameter by tree, augmented with predicted lines. 59
8.19 Added-variable plot for Age against the ratio of Height over Diameter. 61
8.20 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. . . . 62

Chapter 1

Introduction

1.1 R

R is a programming language that has been optimized 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 automatically.

We interact with R by typing commands into a command line in the console, or by creating them in a syntactically aware editor[1] and sending then to the console. The exclusive use of a command-line interface (CLI) makes the learning curve steeper. However, it also allows us to store collections of commands and run them without intervention, which simplifies the process of making templates for graphics or reports. Some interaction with R can be done through menus for some operating systems, but this is mainly administrative stuff. A principal advantage of the CLI is that it simplifies the development and use of scripts. These allow us to keep a permanent, documented record of the steps that we have done. This is also possible with a GUI, but it is much more cumbersome.

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

1.2 Why R?

1. R runs on Windows, Mac-OS, and Unix variants (FreeBSD, Linux)
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[2] 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] I use Xemacs.

[2] Screen is a very useful open-source terminal multiplexor.

1.3 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.
6. There is no guarantee that it is worth more than you paid for it.
7. R eschews the GUI.

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 with 180 different configurations, scoop up the results and assimilate them. 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, it goes to the next command. The failed command was a change of directory. My script wiped itself out, and all its companion scripts. A classic case of the power of ones tool exceeding ones competence. R is very powerful.

1.4 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: further distribution must be under the identical license and must include the source code. But, you can still be charged a reasonable cost for it.

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 essentially 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 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 can't be "corrupted".

Chapter 2

Infrastructure

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

2.1 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 by right-clicking in the console and selecting the "Paste commands only" option. This will save you a lot of tedious typing for the workshop.

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.

Finally, you will commonly see in later chapters the commands `print(plot( ... ))`. The `plot` command is nested inside the `print` command in order to make the lattice graphics available for this document. You do not need to include the `print` command if you only want to see the graphics on your screen. However, there is no problem if you include it anyway.

2.2 Getting Help

There are three main sources of assistance: the help files, the R-help archive, and R-help itself. 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 is because "regression" isn't an R command. If we want to know which commands refer to regression, we use

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

This will tell us to try `help(lm)`, which is one among a great deal of other commands that refer to regression. The level of help provided about the different commands is patchy.

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 often 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 ones own needs.

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 piped, and within which you can then point, click, and search for keywords.

There is a thriving community of programmers and users who will be happy to answer 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("{logistic regression}")          # matches exact phrase
> RSiteSearch("Baron Liaw", res = "Rhelp02")    # restricts search to year
```

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 people 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.

2.3 Working Directory

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

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.
```

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.

2.4 Work Spaces

R uses the concept of work spaces 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="Andrew.RData")   # Save all objects to a binary file
> save(test, file="Andrew.RData")   # Save named objects 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="Andrew.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 compared with, say, comma delimited files, but further space savings can be gained from further compression, such as provided by the zip family of functions.

Work spaces can be very useful, but 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.

2.5 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 2.6.

It's sometimes useful after a session to save the history to a file.

```
> 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 2.6). 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 develop scripts inside R. I prefer to use an external editor that is optimized for script development - examples are Emacs, XEmacs, and WinEdit.

2.6 Writing scripts

Using R effectively is as simple as writing scripts. We write the scripts in any text editor - MS Word, Open Office, 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 delicious, as well as running R as a sub-process, which means that the R output is also a manipulatable 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/filename/file.R", echo=T)
> source(file="../directory/file.R", echo=T)
> source(file="file.R", echo=T) # If file.R is in working directory
```

Note again 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.

Chapter 3

Interface

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.

We create objects and assign names to them using the left arrow: "`<-`".

```
> 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.
> 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.
                   #    This is a specific use for c!  Never make c an object!
> a <- 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.
```

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.

3.1 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 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.

3.1.1 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? (i.e. they should avoid underscores or spaces, percentage signs, etc).
3. Are any of the data missing? Are missing data explicitly represented in the data? If so, what symbol(s) represent(s) 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.

```
> ufc <- read.csv(file="C://path/to/filename/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=F`. See `?read.csv` for details.

Other commands are useful when the data have a more general structure: `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)

[1] 637   5

> names(ufc)

[1] "Plot"    "Tree"    "Species" "Dbh"     "Height"

> ufc[1:5, ]

  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 4.3.

3.1.2 Export

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

```
> write.csv(ufc, file="C://path/to/filename/file.csv")
> write.csv(ufc, file="path/to/filename/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. To save a graph as a pdf, for example, we will write

```
> pdf("fileName.pdf")  # Opens a pdf device and declares the file name
> plot(1:10,1:10)      # ... or 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 4

Manipulating your 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.

4.1 Classes of Data

There are two fundamental kinds of data: numbers and strings. There are several types of strings, each of which has unique properties. 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.

4.1.1 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?
> is.numeric(a)         # is it a number?
> b <- 4                # create variable b, assign the number 4 to it.
> a + b                 # addition
> a - b                 # subtraction
> a * b                 # multiplication
> a / b                 # division
> a ^ b                 # exponentiation
> (a + b) ^ a           # parentheses
> a == b                # logical test of equality
> a < b                 # comparison
> max(a,b)              # largest
> min(a,b)              # smallest
> order(c(a,b))         # return the indices of a and b in increasing order
> c(a,b)[order(c(a,b))] # return a and b in increasing order
```

4.1.2 String

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

```
> a <- "string"        # create variable a, assign the value "string" to it.
> class(a)             # what is it?
> is.numeric(a)        # is it a number?
> is.character(a)      # is it a string?
> b <- "spaghetti"     # create variable b, assign the value "spaghetti" to it.
> paste(a, b)          # join the strings
> paste(a, b, sep="")  # join the strings with no gap
> d <- paste(a, b, sep="")
> substr(d, 1, 4)      # subset the string
```

4.1.3 Factor

A categorical variable. These are not terribly different than strings, except they can take only a limited number of values, and R knows how to do very useful things with them. We check by `is.factor()` and change to by `factor()`. 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?
> is.character(a)           # is it a string?
> is.factor(a)              # is it a factor?
> a <- factor(a)            # make it so
> levels(a)                 # what are the options?
```

There will be much more on factors when we start manipulating vectors (Section 4.2.1).

4.1.4 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?
> e <- T               # create variable e, assign the value T to it.
> d + e                # what should this do?
> d & e                # d AND e is True
> d | e                # d OR e is also True
> d & !e               # d AND (NOT e) is not True
```

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

4.1.5 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?
> class(a)              # what is it?
> a <- c(11,NA,13)      # now try a vector
> mean(a)               # seems fine
> var(a)                # agh!
> var(a, na.rm=T)       # Phew!
> is.na(a)              # is it missing?
> a[!is.na(a)]          # what are the elements of a that aren't missing?
> which(!is.na(a))      # what are the locations of the non-missing elements of a?
```

4.2 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.

4.2.1 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
> a[2]             # the second object in a
> a[-2]            # a, but without the second object
> a[c(2,3,1)]      # a, but in a different order
> length(a)        # the number of units in the vector a
```

Notice that we were able to nest a vector inside the subscript mechanism of another vector! This is a clue to the power of object orientation. This also introduces a key facility in R for efficient processing: vectorization.

Vectorization

The concept underlying vectorization is simple: processing can be made more efficient. Recall that in section 4.1.5 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. This is vectorization. It only works for some functions; *e.g.* it won't work for `mean()`, that makes no sense. But when it works it makes life easier, code cleaner, and processing time faster. Just think: otherwise we'd have had to write a loop that examined each element of the vector.

Anecdote: I have been working with a graduate student to simulate some of the processing of a large-scale forest growth model within R. Specifically, we wish to frequently apply the diameter growth model, which is an ordinary least squares regression, to 60,000 tree records. Using a loop takes a few hours. Using vectorization takes 15 minutes.

4.2.2 Dataframe

A dataframe is a powerful two-dimensional vector-holding structure. It is optimized for representing multi-dimensional 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 subscripts can be nested or applied sequentially.

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

If a comma-delimited file is read in using the commands in section 3.1.1, 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")
> is.data.frame(ufc)

[1] TRUE

> dim(ufc)

[1] 637   5

> names(ufc)

[1] "Plot"    "Tree"    "Species" "Dbh"     "Height"

> ufc$Height[1:5]

[1]  NA 205 330  NA 300

> ufc$Species[1:5]

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

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

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

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                 NA
2         DF     39
3         WL     48
4         WC     15
5         GF     52
```

4.3 Data References

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"]

[1] 24.5   NA   NA   NA 16.0 25.0   NA

> mean(ufc$height.m[ufc$Species == "LP"])

[1] NA
```

Now, let's try something a little more involved. How would we ask: what are the species of the three tallest trees? This command is best constructed piecemeal. Notice that we are able to nest the subscripts. R starts at the left and works its way right.

1. `order(ufc$height.m, decreasing = T)` provides the indices of the observations in order of increasing height.
2. `ufc$Species[order(ufc$height.m, decreasing = T)]` provides the species corresponding to those heights.
3. `ufc$Species[order(ufc$height.m, decreasing = T)][1:3]` provides only the first three.

```
> ufc$Species[order(ufc$height.m, decreasing = T)][1:3]

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

The next useful command is called `tapply()`. This lovely little function allows us to vectorize the application of certain functions to groups of data. In conjunction with factors, this makes for some exceptionally efficient code. `tapply()` requires three things: the target vector to which the function will be applied, the vector by which the target vector will be grouped, and the function.

```
> tapply(ufc$height.m, ufc$Species, mean)

      DF   ES    F   FG   GF   HW   LP   PP   SF   WC   WL   WP
  NA  NA   NA 27.0 27.5   NA 19.8   NA   NA   NA   NA   NA   NA
```

Ah. Many heights are missing. Let's fix that.

```
> tapply(ufc$height.m, ufc$Species, mean, na.rm = T)

               DF       ES        F       FG       GF       HW       LP
     NaN 25.43036 28.00000 27.00000 27.50000 24.26522 19.80000 21.83333
      PP       SF       WC       WL       WP
33.00000 15.41000 23.48777 25.25714 25.13939
```

And let's pretty it up a little.

```
> format(tapply(ufc$height.m, ufc$Species, mean, na.rm = T), dig = 3)

          DF     ES      F     FG     GF     HW     LP     PP     SF     WC
" NaN" "25.4" "28.0" "27.0" "27.5" "24.3" "19.8" "21.8" "33.0" "15.4" "23.5"
    WL     WP
"25.3" "25.1"
```

Now, let's try something a little more involved. How would we pull out the identity of the median height tree of the species that was second tallest on average? Ok that is ridiculous, but let's stretch the language. Again, piecemeal.

1. First get the mean height by species.

```
> ht.bar.by.species <- tapply(ufc$height.m, ufc$Species, mean,
+     na.rm = T)
> ht.bar.by.species
```

```
               DF       ES        F       FG       GF       HW       LP
     NaN 25.43036 28.00000 27.00000 27.50000 24.26522 19.80000 21.83333
      PP       SF       WC       WL       WP
33.00000 15.41000 23.48777 25.25714 25.13939
```

2. then get the index order of the species, sorted by average height.

```
> species.order.by.ht <- order(ht.bar.by.species, decreasing = T)
> species.order.by.ht
```

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

3. The species names in order of the average height by species are:

```
> species.by.ht <- levels(ufc$Species)[species.order.by.ht]
> species.by.ht
```

```
 [1] "PP" "ES" "FG" "F"  "DF" "WL" "WP" "GF" "WC" "LP" "HW" "SF" ""
```

4. The second tallest is

```
> species.by.ht[2]
```

```
[1] "ES"
```

5. The median of the heights of all the trees of that species is then

```
> median(ufc$height.m[ufc$Species == species.by.ht[2]], na.rm = T)
```

```
[1] 28
```

These things must be tackled strategically. If we pursue a rigorous naming policy then these intermediate objects become vital debugging tools as well.

Of course, this can all be expressed as a single operation:

```
> median(ufc$height.m[ufc$Species == levels(ufc$Species)[order(tapply(ufc$height.m,
+     ufc$Species, mean, na.rm = T), decreasing = T)][2]], na.rm = T)
```

```
[1] 28
```

But why would we want to do that?

Chapter 5

Graphics

One major selling point for R is that it has better graphics-producing capabilities than many of the commercial alternatives. The graphics are controlled by scripts, and start at a very simple level, 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 (Figure 5.1).

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

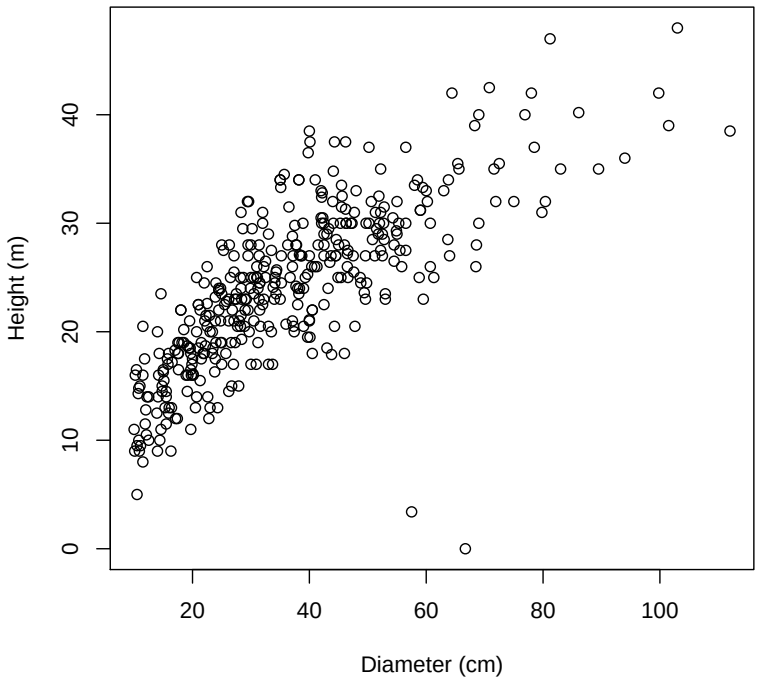

Figure 5.1: Diameter/Height plot for all species of Upper Flat Creek inventory data. Each point represents a tree.

The `plot()` command offers a wide variety of options for customizing the graphic. Each of the following can be used in the `plot()` statement, singly or together, separated by commas.

- `xlim=c(a,b)` will set the lower and upper limits of the x-axis to be a and b respectively.
- `ylim=c(a,b)` will set the lower and upper limits of the y-axis to be a and b respectively.
- `xlab="X axis label goes in here"`
- `ylab="Y axis label goes in here"`
- `main="Plot title goes in here"`
- `col="red"` makes it all red. Especially attractive for overlaid plots.
- `col=swatch[color.id]` makes the colour correspond to the colours listed in `swatch` corresponding to the rows in the variable `color.id`.

5.1 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:

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

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 more than I note below, but these are the ones that I end up using the most often.

- `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 axis labels to be horizontal rather than vertical
- `par(pty="s")` forces the plot shape to be square. The alternative, which is the default, is `"m"`.
- `par(new=T)` 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.
- Various character expansion factors may also be set.

5.2 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="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 working 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.

Chapter 6

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 will have been covered much more effectively elsewhere.

6.1 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)
```

We redefine the species factor in order to drop the empty levels. Drop the trees with missing height measures

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

Graph the tree data (see Figure 5.1). 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] 7.498069
```

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

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

       DF        ES         F        FG        GF        HW        LP        PP
6.6193084 5.6568542        NA 0.7071068 7.6939261 2.4899799 5.0579970 9.8994949
       SF        WC        WL        WP
3.6564859 6.9728507 9.0074176 9.1776202
```

6.2 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.

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

6.3 Diagnostics

First, let's examine the model diagnostics (Figure 6.1).

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

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

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 absolute 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 things change? Here's a nice little plot that shows that things changed very little. we can do this only because everything is about the same scale.

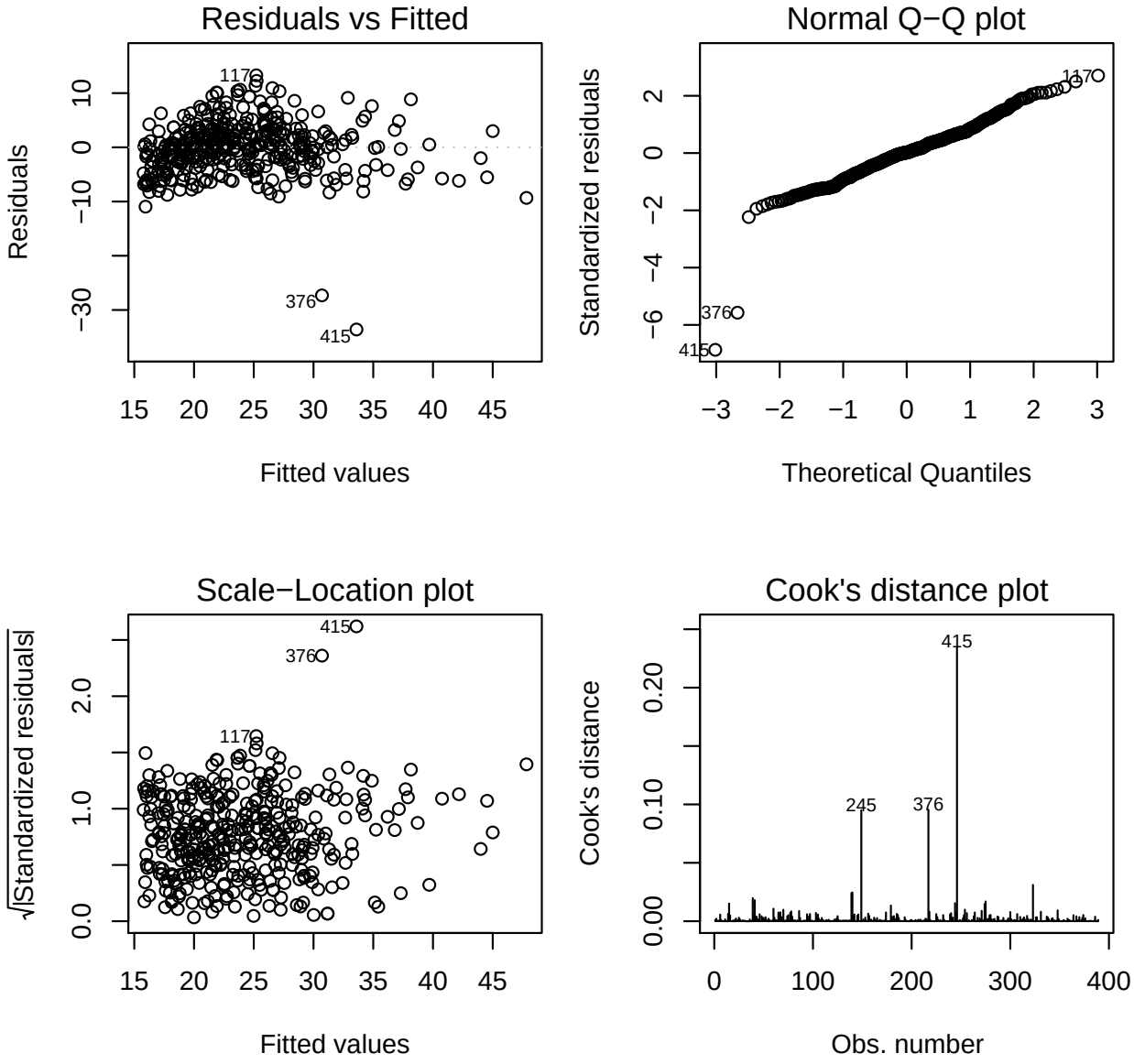

Figure 6.1: Diagnostic plots for the regression of diameter against height.

```
> 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)
```

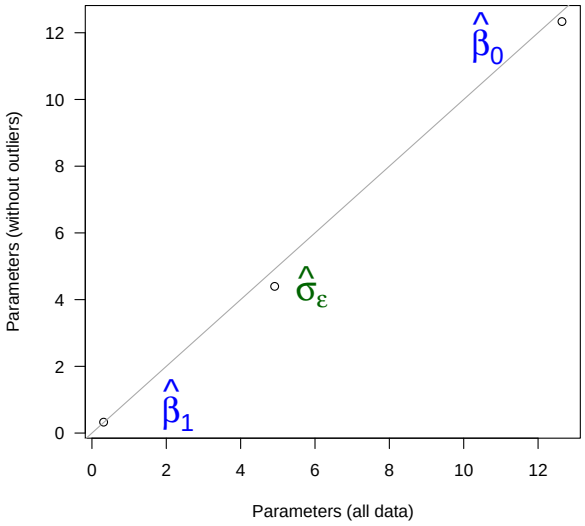

Figure 6.2: Parameter estimate change as a result of dropping the outliers.

6.4 Other Tools

Other tools for examining regressions should be mentioned:

```
> ?influence.measures
```

6.5 Examining the Model

In any case, they don't affect the model very strongly. 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
-33.59633  -2.86331   0.08956   2.81206  13.29113

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.64340    0.56124   22.53   <2e-16 ***
dbh.cm       0.31414    0.01383   22.72   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.915 on 387 degrees of freedom
Multiple R-Squared: 0.5714,        Adjusted R-squared: 0.5703
F-statistic:   516 on 1 and 387 DF,  p-value: < 2.2e-16
```

We see that already, there is a big difference between the marginal variability, which is 7.50, and the conditional variability, which is 4.91.

6.6 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 exist to enable the reliable extraction of moel information, for example, `residuals()` and `fitted()`, but these are not exhaustive. We extract 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] 4.914947
```

6.7 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)
```

6.8 Other Ways of Fitting

We can also use other tools to fit the regression. For example, what do you imagine the following values are? I include a little more about writing your own functions in Chapter 10.

```
> objective.function <- function(parameters, x, y) {
+     -sum(dnorm(y - parameters[1] - parameters[2] * x, 0, parameters[3],
+         log = T))
+ }
> good.fit <- optim(c(1, 1, 1), objective.function, hessian = TRUE,
+     x = ufc$dbh.cm, y = ufc$height.m)
> good.fit$par

[1] 12.6350150  0.3142022  4.9025639

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

[1] 0.55982729 0.01379412 0.17577988
```

Chapter 7

More Graphics

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 ...

```
> 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)
```

7.1 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. We load the `lattice` package by means of the `require` function, which is explained in greater detail in Chapter 9.

```
> require(lattice)

Loading required package: lattice
[1] TRUE
```

For example, if we were to wish to plot the height against the predicted height for the four species with the largest number of trees, and add some lines to the graphs, then we could do it with this code:

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 here, as are the following documents:

- http://zoonek2.free.fr/UNIX/48_R/04.html.
- http://addictedtor.free.fr/graphiques/
- http://www.stat.ucl.ac.be/ISpersonnel/lecoutre/stats/fichiers/_gallery.pdf

Figure 7.1: A random plot of coloured dots.

```
> ufc$height.hat <- fitted(hd.lm.1)
> top.nine <- levels(ufc$Species)[order(table(ufc$Species), decreasing = T)][1:9]
> print(xyplot(height.m ~ height.hat | Species, data = ufc, xlab = "Predicted Height (m)",
+     ylab = "Measured Height (m)", panel = function(x, y) {
+         panel.xyplot(x, y)
+         panel.abline(lm(y ~ x), col = "red")
+         panel.abline(0, 1, col = "blue", lty = 2)
+     }, subset = ufc$Species %in% top.nine))
```

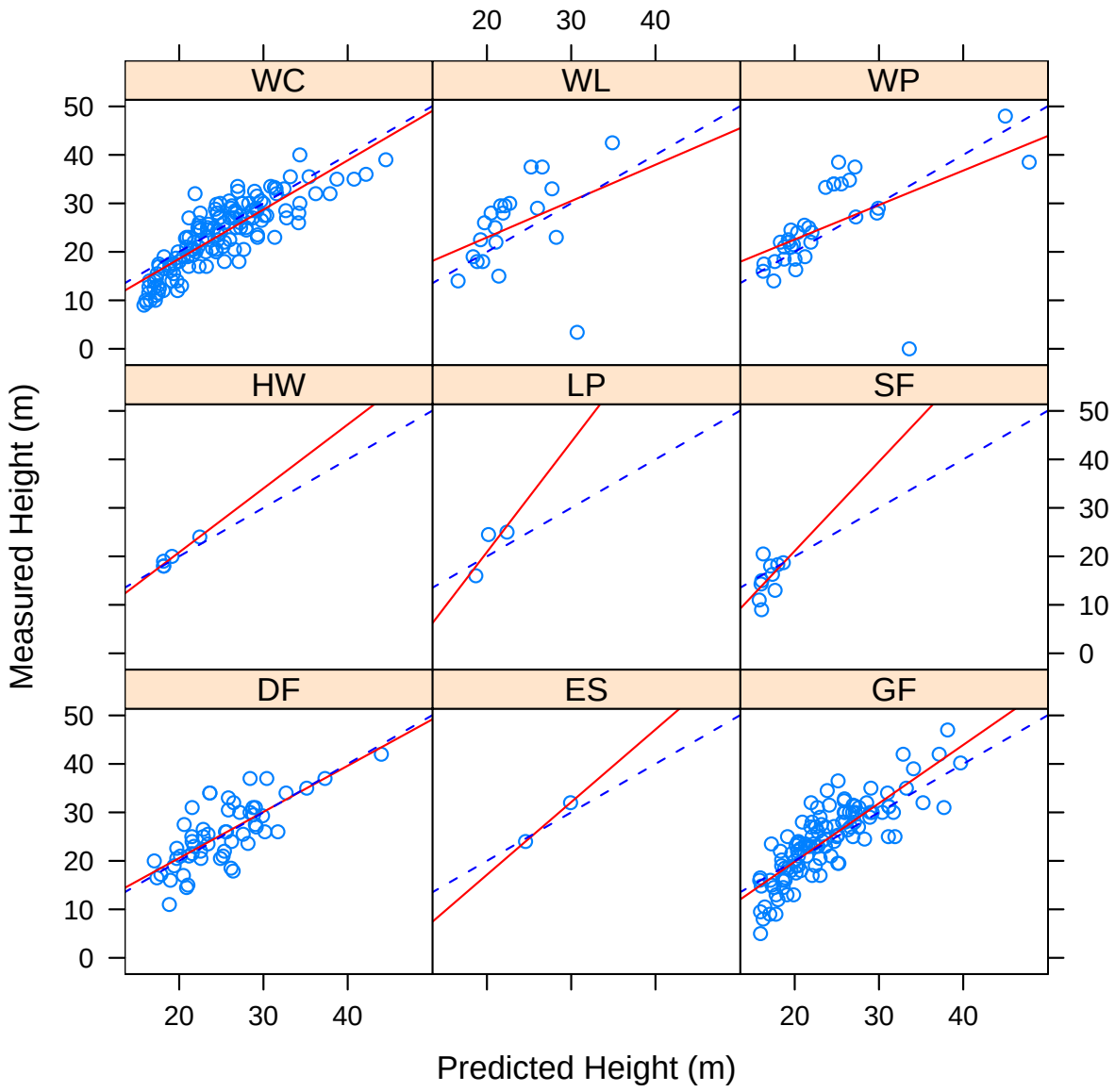

Figure 7.2: A lattice plot of height against predicted height by species for the four species that have the most trees. The blue dashed line is 1:1, and the red solid line is the panel-level regression.

Chapter 8

Hierarchical Models

We now shift to the analysis of hierarchical data using mixed-effects moels. These models are a natural match for many problems that occur commonly in natural resources.

8.1 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*.

8.1.1 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.

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.

[1] "The first law of ecology is that everything is related to everything else." – Barry Commoner, US biologist/environmentalist.

8.1.2 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: `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.

8.2 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.

8.2.1 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.

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 random 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

[2]The use of the label is a hang-over from experimental design, and no longer really suits the application, but that's how inertia goes.

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.

8.2.2 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 inthe 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 surficailly 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 8.1 and 8.2.

```
> 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/Pac",
+     "Ts/Op", "Th/Pach", "AG/Pach", "PA/Pach"))
> stage$dbhib.cm <- stage$Dbhib * 2.54
> stage$height.m <- stage$Height/3.2808399
> stage[1:10, ]

   Tree.ID Forest HabType Decade Dbhib Height Age  Forest.ID HabType.ID
1        1      4       5      0  14.6   71.4  55 Clearwater    PA/Pach
2        1      4       5      1  12.4   61.4  45 Clearwater    PA/Pach
3        1      4       5      2   8.8   40.1  35 Clearwater    PA/Pach
4        1      4       5      3   7.0   28.6  25 Clearwater    PA/Pach
5        1      4       5      4   4.0   19.6  15 Clearwater    PA/Pach
6        2      4       5      0  20.0  103.4 107 Clearwater    PA/Pach
7        2      4       5      1  18.8   92.2  97 Clearwater    PA/Pach
8        2      4       5      2  17.0   80.8  87 Clearwater    PA/Pach
9        2      4       5      3  15.9   76.2  77 Clearwater    PA/Pach
10       2      4       5      4  14.0   70.7  67 Clearwater    PA/Pach
   dbhib.cm height.m
1    37.084 21.76272
2    31.496 18.71472
3    22.352 12.22248
4    17.780  8.71728
5    10.160  5.97408
6    50.800 31.51632
7    47.752 28.10256
8    43.180 24.62784
9    40.386 23.22576
10   35.560 21.54936

> opar <- par(las = 1)
> plot(stage$dbhib.cm, stage$height.m, xlab = "Dbhib (cm)", ylab = "Height (m)")
> par(opar)


> colours <- c("deepskyblue", "goldenrod", "purple", "orangered2",
+     "seagreen")
> par(mfrow = c(3, 3), pty = "m", mar = c(3, 2, 3, 1) + 0.1)
> 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(0, max(stage$height.m), unique(as.character(forestData$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)
```

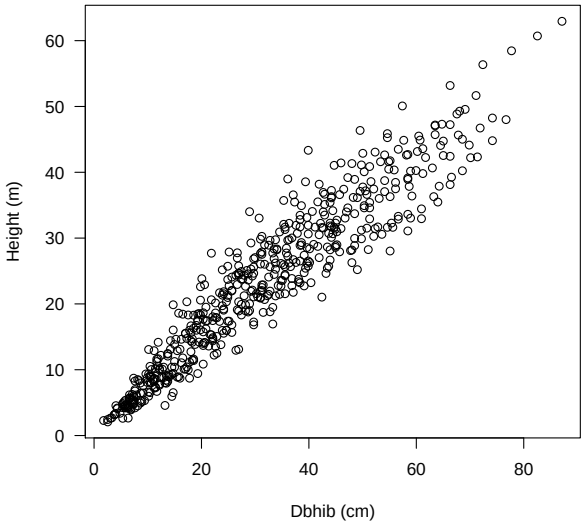

Figure 8.1: Al Stage's Grand Fir stem analysis data: height (ft) against diameter (in). These were dominant and co-dominant trees.

```
+     }
+ }
> mtext("Height (m)", outer = T, side = 2, line = 2)
> mtext("Diameter (cm)", outer = T, side = 1, line = 2)
```

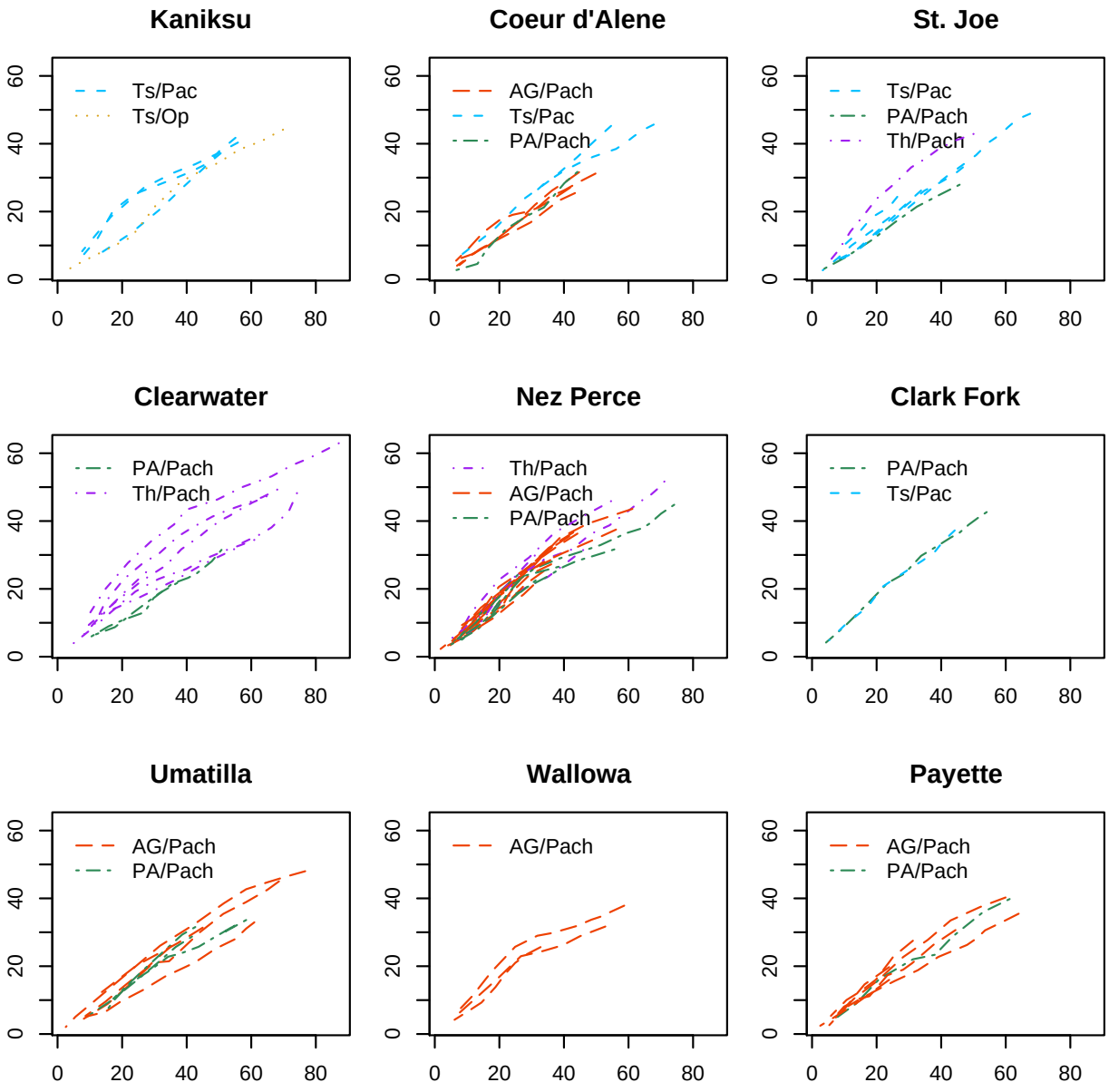

Figure 8.2: Al Stage's Grand Fir Stem Analysis Data: height (ft, vertical axes) against diameter (inches, horizontal axes) by National Forest. These were dominant and co-dominant trees.

8.2.3 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.

8.2.4 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:

$$f(\mathbf{Y} \mid \mu, \mathbf{V}) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}-\mu)'\mathbf{V}^{-1}(\mathbf{Y}-\mu)}$$

So in terms of the linear model $\mathbf{Y} = \mathbf{X}\beta$, the conditional joint density is

$$f(\mathbf{Y} \mid \mathbf{X}, \beta, \mathbf{V}) = \frac{|\mathbf{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} e^{-\frac{1}{2}(\mathbf{Y}-\mathbf{X}\beta)'\mathbf{V}^{-1}(\mathbf{Y}-\mathbf{X}\beta)}$$

Reversing the conditioning and taking logs yields:

$$\mathcal{L}(\beta, \mathbf{V} \mid \mathbf{Y}, \mathbf{X}) = -\frac{1}{2}\ln(|\mathbf{V}|) - \frac{n}{2}\ln(2\pi) - \frac{1}{2}(\mathbf{Y}-\mathbf{X}\beta)'\mathbf{V}^{-1}(\mathbf{Y}-\mathbf{X}\beta) \tag{8.1}$$

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:

$$\hat{\beta}_{MLE} = \left(\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{X}\right)^{-1}\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{Y} \tag{8.2}$$

After some tedious algebra, which is well documented in Schabenberger and Pierce (2002), we also get the $BLUP$s.

$$\hat{b}_{MLE} = \mathbf{D}\mathbf{Z}'\hat{\mathbf{V}}\left(\mathbf{Y} - \mathbf{X}\hat{\beta}\right) \tag{8.3}$$

where $\mathbf{D}$ is the covariance matrix of the random effects.

8.2.5 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 $\mathbf{K}'\mathbf{V}\mathbf{K}$.

8.3 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`.

```
> require(nlme)
```

```
Loading required package: nlme
[1] TRUE
```

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 8.3).

```
> colours = c("red", "blue")
> plot(straw$x, straw$y, col = colours[straw$group])
```

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.

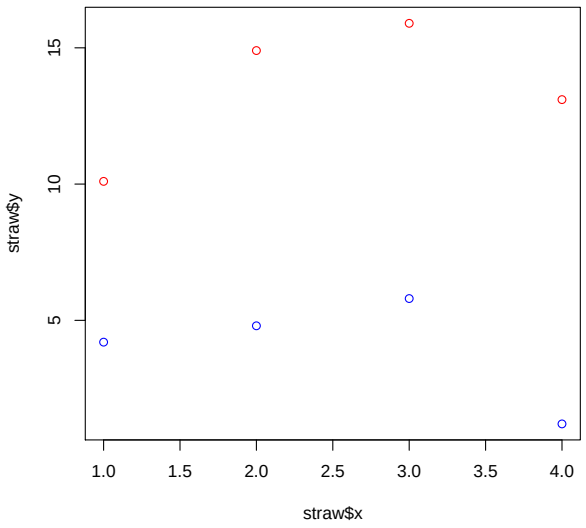

Figure 8.3: A simple dataset to show the use of mixed-effects models.

Ordinary Least Squares

This model is just trying to predict y using x.

```
> basic.1 <- lm(y ~ x, data = straw)
```

Let's let each `group` have its own intercept.

```
> basic.2 <- lm(y ~ x + group, data = straw)
```

Let's let each `group` have its own intercept *and* slope.

```
> 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)
```

Let's fit the basic mixed-effects model that allows the intercepts to vary randomly between the groups.

```
> basic.4 <- lme(y ~ x, random = ~1 | group, data = straw.mixed)
```

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 8.4).

```
> print(plot(augPred(basic.4)))
```

Next let's fit a unique variance to each group.

```
> basic.5 <- lme(y ~ x, random = ~1 | group, weights = varIdent(form = ~1 |
+     group), data = straw.mixed)
```

Finally let's allow for temporal autocorrelation within each group.

```
> basic.6 <- lme(y ~ x, random = ~1 | group, weights = varIdent(form = ~1 |
+     group), correlation = corAR1(), data = straw.mixed)
```

We can summarize some of these differences in a graph (Figure 8.5).

```
> 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)
```

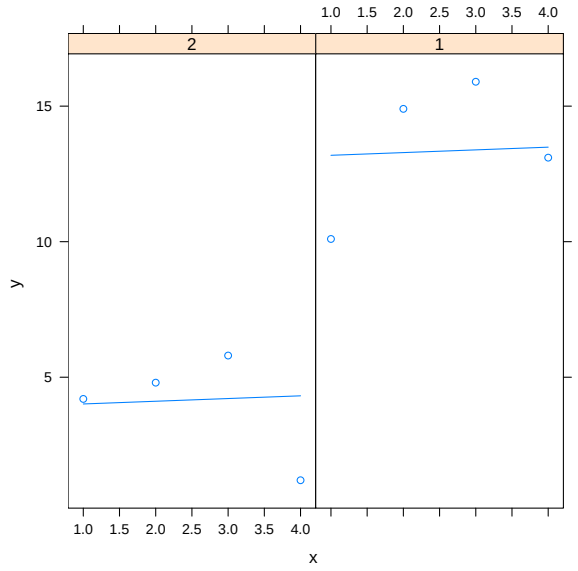

Figure 8.4: An augmented plot of the basic mixed-effects model with random intercepts fit to the sample dataset.

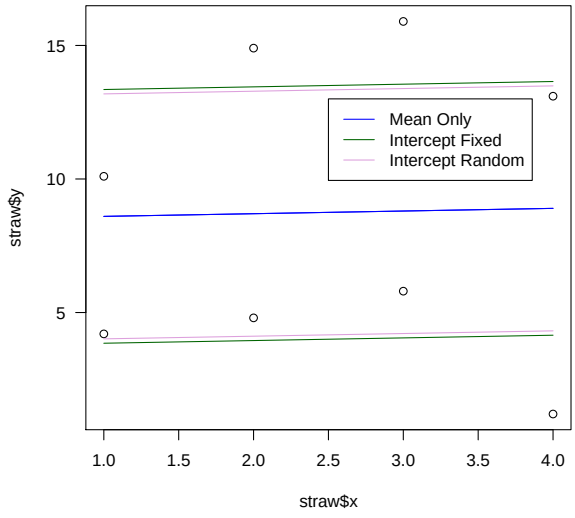

Figure 8.5: A sample plot showing the difference between basic.1 (single line), basic.2 (intercepts are fixed), and basic.4 (intercepts are random).

8.4 Case Study

Recall our brutal exposition of the mixed-effects model:

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon} \\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D}) \\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

D and **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.

8.4.1 Height/Diameter 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"  "Age"

> sapply(stage, class)

  Tree.ID    Forest   HabType    Decade     Dbhib    Height       Age
"integer" "integer" "integer" "integer" "numeric" "numeric" "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("Kaniksu",
+     "Coeur d'Alene", "St. Joe", "Clearwater", "Nez Perce", "Clark Fork",
+     "Umatilla", "Wallowa", "Payette"))
> stage$HabType.ID <- factor(stage$HabType, labels = c("Ts/Pac",
+     "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
> stage[1:10, ]

   Tree.ID Forest HabType Decade Dbhib Height Age  Forest.ID HabType.ID
1        1      4       5      0  14.6   71.4  55 Clearwater    PA/Pach
2        1      4       5      1  12.4   61.4  45 Clearwater    PA/Pach
3        1      4       5      2   8.8   40.1  35 Clearwater    PA/Pach
4        1      4       5      3   7.0   28.6  25 Clearwater    PA/Pach
5        1      4       5      4   4.0   19.6  15 Clearwater    PA/Pach
6        2      4       5      0  20.0  103.4 107 Clearwater    PA/Pach
7        2      4       5      1  18.8   92.2  97 Clearwater    PA/Pach
```

```
8        2      4       5      2  17.0   80.8  87 Clearwater    PA/Pach
9        2      4       5      3  15.9   76.2  77 Clearwater    PA/Pach
10       2      4       5      4  14.0   70.7  67 Clearwater    PA/Pach
   dbhib.cm height.m
1    37.084 21.76272
2    31.496 18.71472
3    22.352 12.22248
4    17.780  8.71728
5    10.160  5.97408
6    50.800 31.51632
7    47.752 28.10256
8    43.180 24.62784
9    40.386 23.22576
10   35.560 21.54936
```

Height/Diameter

The prediction of height from diameter provides useful and inexpensive information. It may be that the height/diameter relationship differs among hibitat 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, tempting though it may be, so see Figure 8.6. 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|)
```

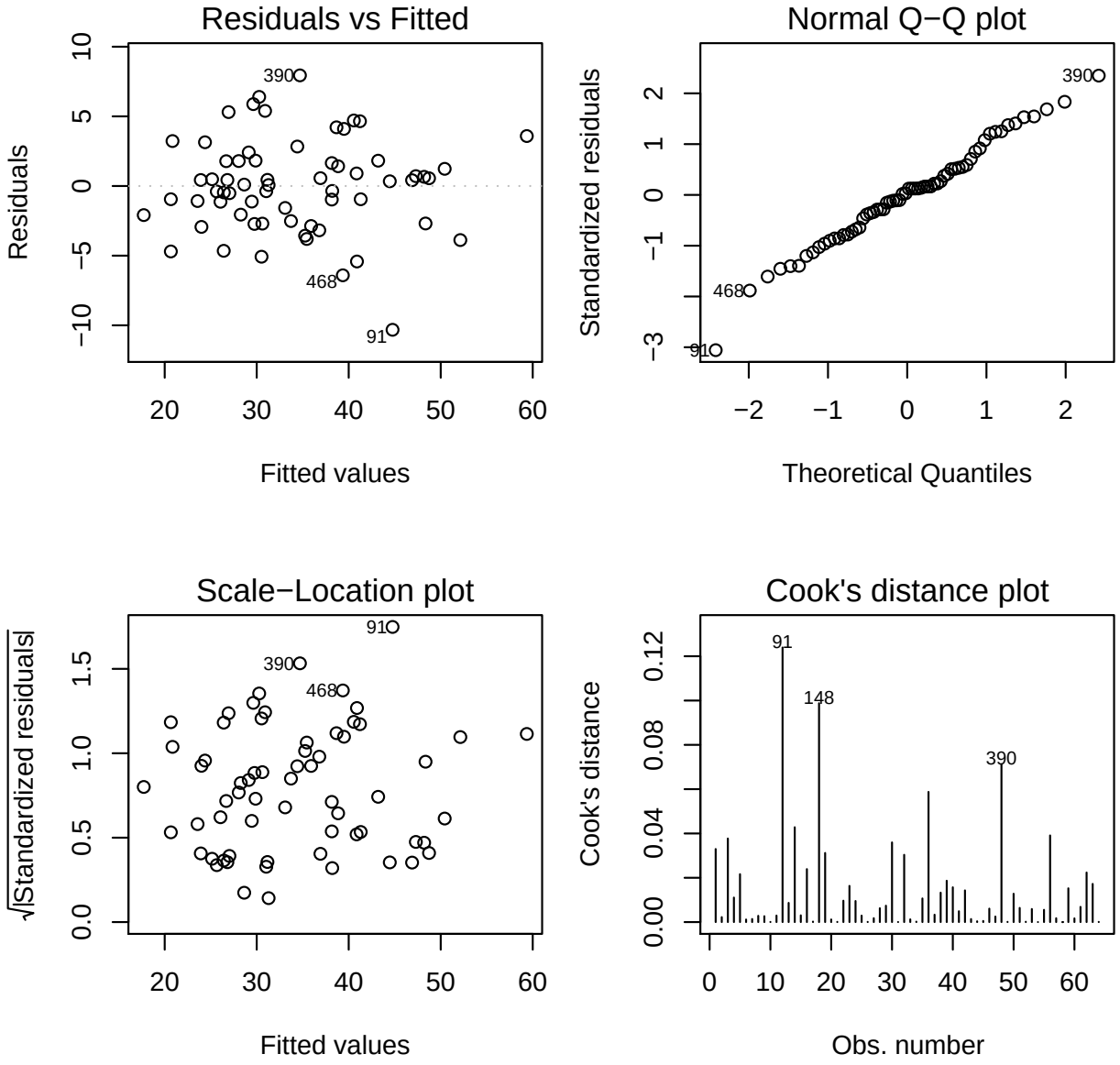

Figure 8.6: Regression diagnostics for the ordinary least squares fit of the Height/Diameter model with habitat type for Stage's data.

```
(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
---
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] 4.101350
```

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.

```
> require(nlme)

[1] TRUE

> stage.old <- groupedData(height.m ~ dbhib.cm | Forest.ID, data = stage.old)
```

Now, let's look to our model.

$$y_{ij} = \beta_0 + b_{0i} + \beta_1 \times x_{ij} + \epsilon_{ij} \tag{8.4}$$

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,

$$Y = \beta\mathbf{X} + b\mathbf{Z} + \epsilon \tag{8.5}$$

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.

$$r_0 = y_{ij} - \hat{\beta}_0 - \hat{\beta}_1 \times xij \tag{8.6}$$

- 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.

$$r_1 = y_{ij} - \hat{\beta}_0 - \hat{b}_{0i} - \hat{\beta}_1 \times xij \tag{8.7}$$

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 8.7.

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?

We use the following code to produce Figure 8.7. 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(fitted(hd.lme.1, level = 0), stage.old$height.m, xlab = "Fitted Values (height, m.)",
+     ylab = "Observed Values (height, m.)", main = "Model Structure (I)")
> abline(0, 1, col = "blue")
> scatter.smooth(fitted(hd.lme.1), residuals(hd.lme.1, type = "pearson"),
+     main = "Model Structure (II)", xlab = "Fitted Values", ylab = "Innermost Residuals")
> abline(h = 0, col = "red")
> ref.forest <- ranef(hd.lme.1)[[1]]
> ref.var.forest <- tapply(residuals(hd.lme.1, type = "pearson",
+     level = 1), stage.old$Forest.ID, var)
> qqnorm(ref.forest, main = "Q-Q Normal - Forest Random Effects")
```

```
> qqline(ref.forest, col = "red")
> qqnorm(residuals(hd.lme.1, type = "pearson"), main = "Q-Q Normal - Residuals")
> qqline(residuals(hd.lme.1, type = "pearson"), col = "red")
> 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(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)
```

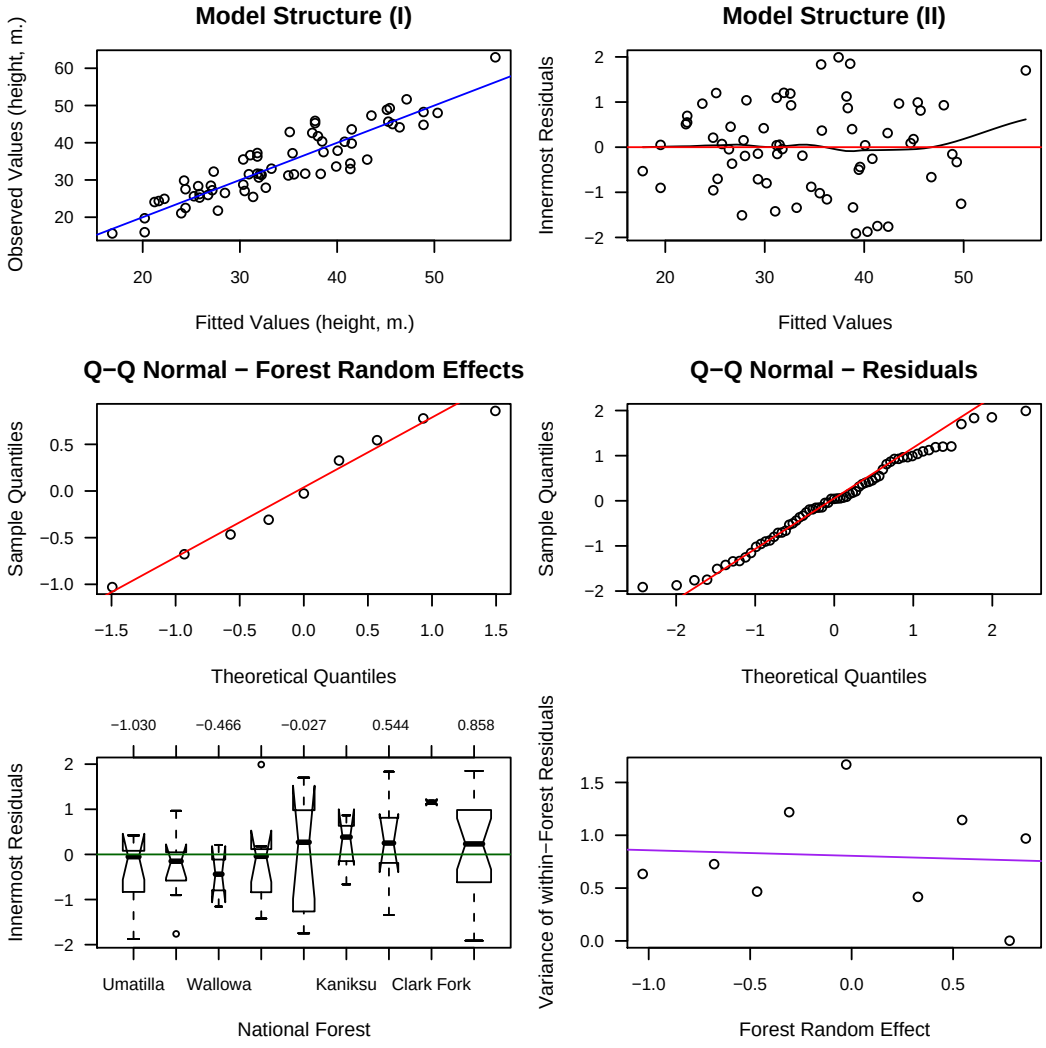

Figure 8.7: Selected diagnostics for the mixed-effects fit of the Height/Diameter ratio against habitat type and national forest for Stage's data.

Cross-reference these against Figure 8.7. In fact, all of these residual diagnostics look good.

The next important question is whether there are any outliers or high-influence points. In a case like this it is realtively 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. **?** 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 8.8 and 8.9). 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$fixed)
> len.ran <- length(hd.lme.1$coefficients$random$Forest.ID)

> 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)
```

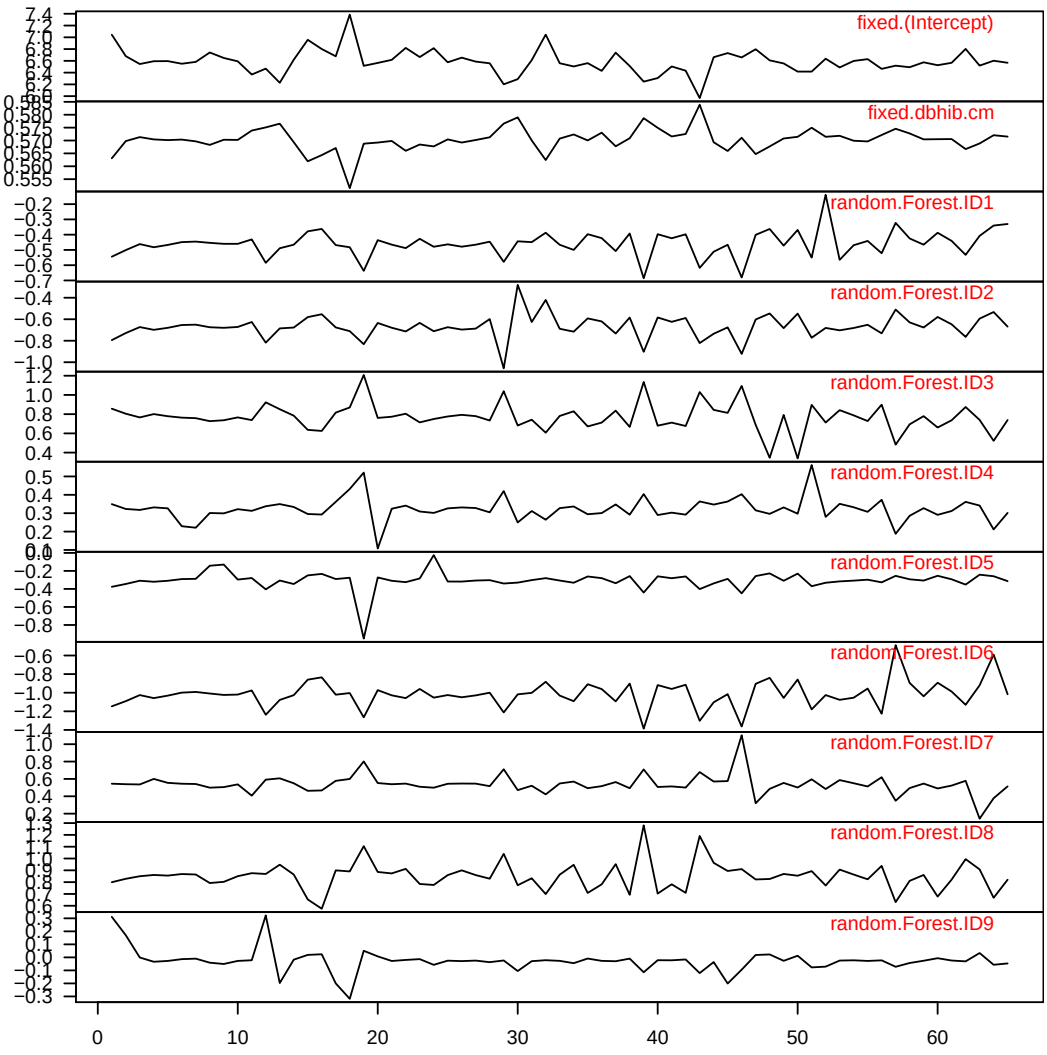

Figure 8.8: The parameter estimates for the fixed effects and predictions for the random effects resulting from omitting one observation.

```
> 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. But, let's accept the model as it stands for the moment, and go on to examine the summary.

```
Linear mixed-effects model fit by REML
 Data: stage.old
       AIC      BIC    logLik
  376.6805 385.2530 -184.3403

Random effects:
```

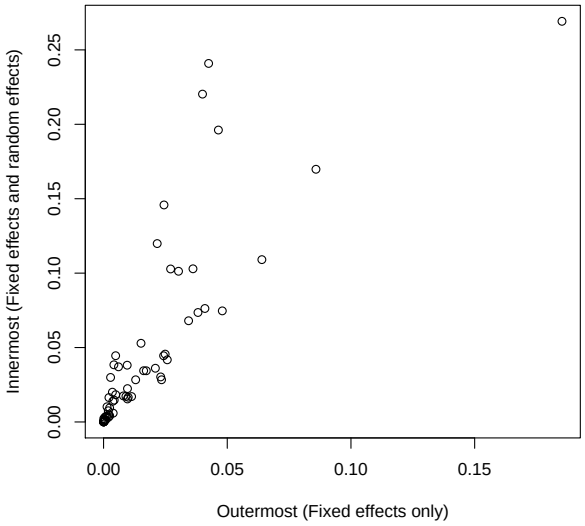

Figure 8.9: Cook's Distances for outermost and innermost residuals. Values greater than 1 appear in red and are identified by the tree number. The corresponding observations bear further examination.

```
 Formula: ~1 | Forest.ID
        (Intercept) Residual
StdDev:    1.151436 3.937481

Fixed effects: height.m ~ dbhib.cm
               Value Std.Error DF   t-value p-value
(Intercept) 6.58238 1.7763600 55  3.705544   5e-04
dbhib.cm    0.57036 0.0335347 55 17.008069   0e+00
 Correlation:
         (Intr)
dbhib.cm -0.931

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-1.91216370 -0.70233864  0.04308706  0.81188545  1.99134351

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* the RMSE, 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 (53 = 66 trees - 1 - (9-1) national forests - (5-1) habitat types)
   (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 2848.3870  <.0001
dbhib.cm        1    55  289.2744  <.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.

```
> 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 potential fixed effect. So, we'll have 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, 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.

$$y_{ijk} = \beta_0 + b_{0i} + b_{0ij} + \beta_1 \times x_{ijk} + \epsilon_{ijk} \tag{8.8}$$

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:

$$Y = \beta\mathbf{X} + b\mathbf{Z} + \epsilon \tag{8.9}$$

- Y is the vector of height/diameter 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.
- $\mathbf{D}$ will be a block diagonal matrix comprising a 9×9 identity matrix multiplied by a constant σ^2_f, and then a square matrix for each forest, which will be a diagonal matrix with variances on the diagonals.
- $\mathbf{R}$ 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 Figure **??**. 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)
```

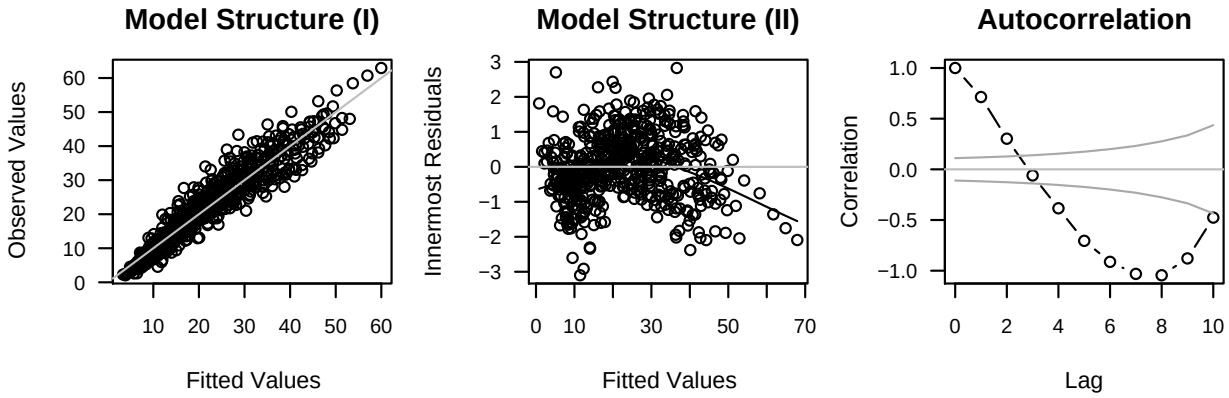

Figure 8.10: Selected overall diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data.

```
> 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)
```

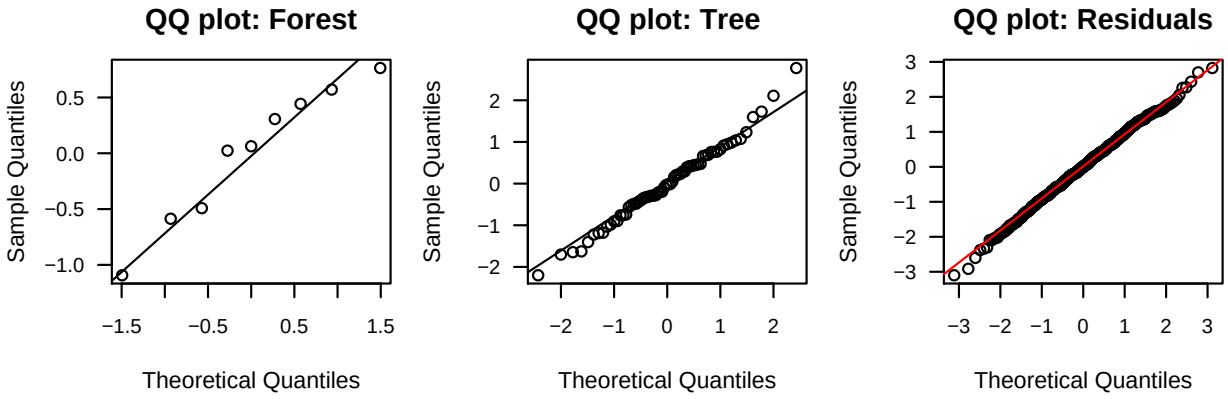

Figure 8.11: Selected quantile-based diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data.

```
> 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, 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

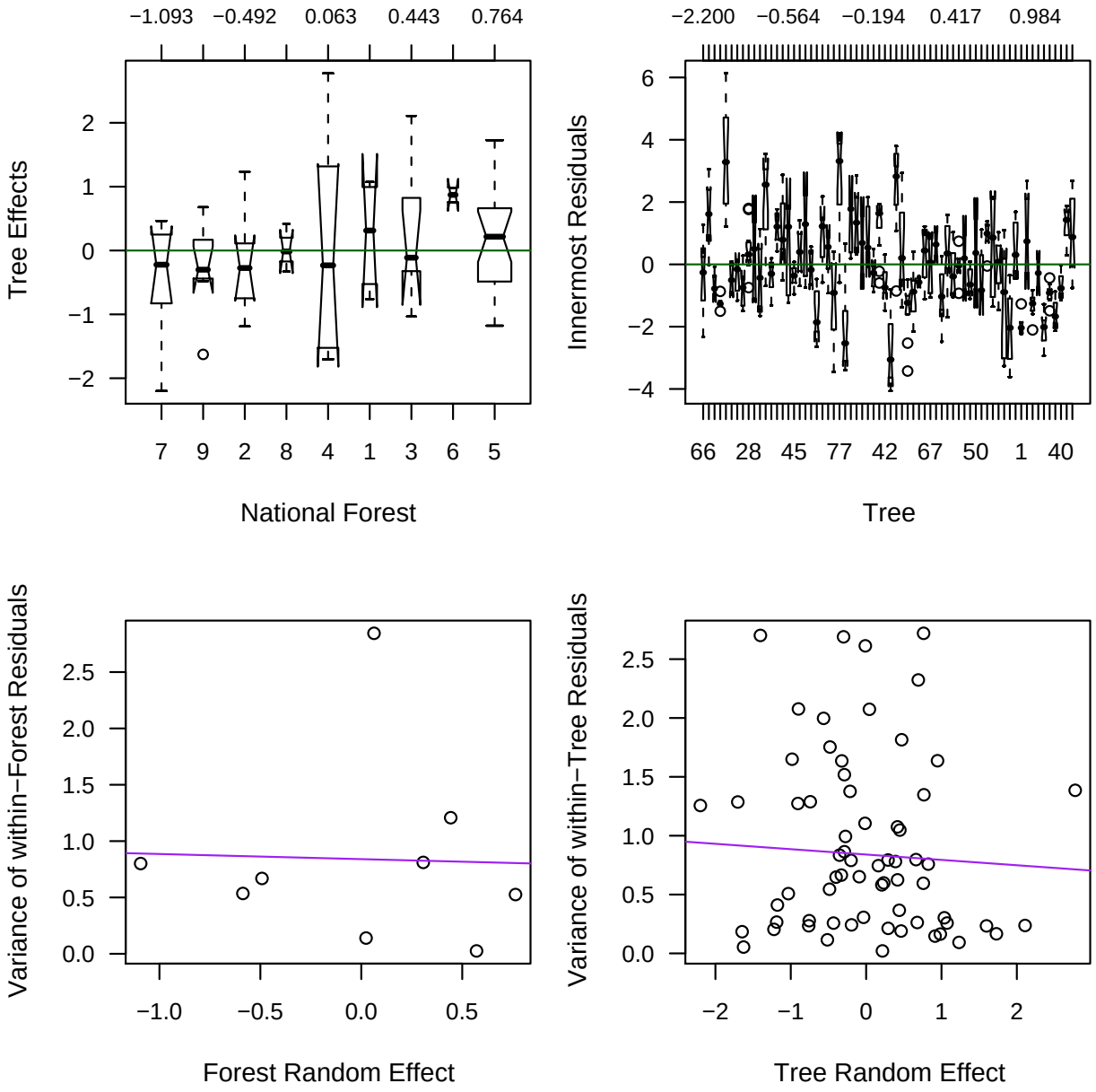

Figure 8.12: Selected random-effects based diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data.

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)))
```

```
> print(plot(augPred(hd.lme.3), index.cond = list(panel.order),
+     strip = strip.custom(par.strip.text = list(cex = 0.5))))
```

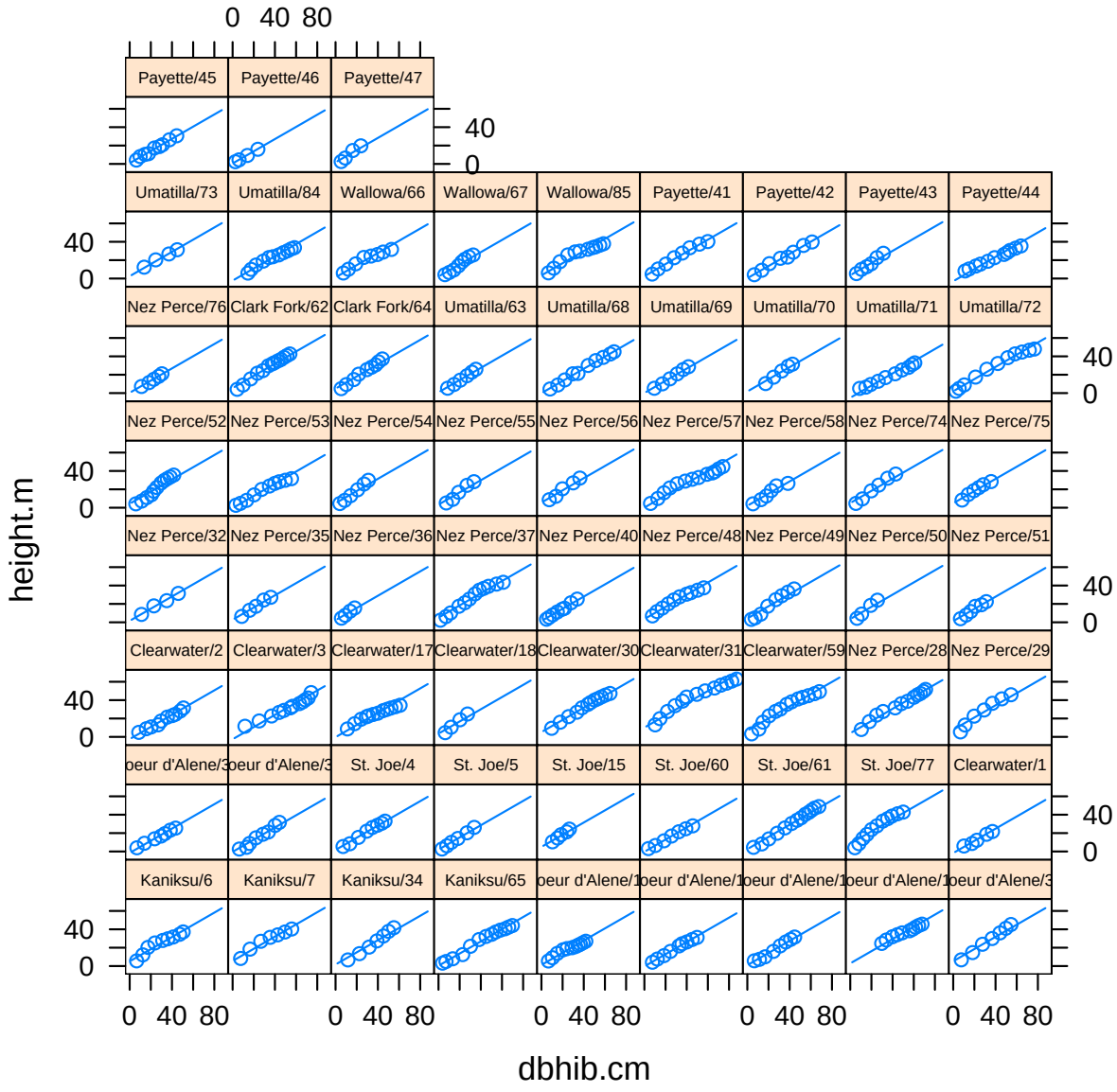

Figure 8.13: Height against diameter by tree, augmented with predicted lines.

The augmented prediction plot (Figure 8.13) 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 reasaonbly 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.

```
> hd.lme.4 <- lme(height.m ~ dbhib.cm + I(dbhib.cm^2), random = ~dbhib.cm +
+     I(dbhib.cm^2) | Forest.ID/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$lag < 10.5], acf.resid$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)
```

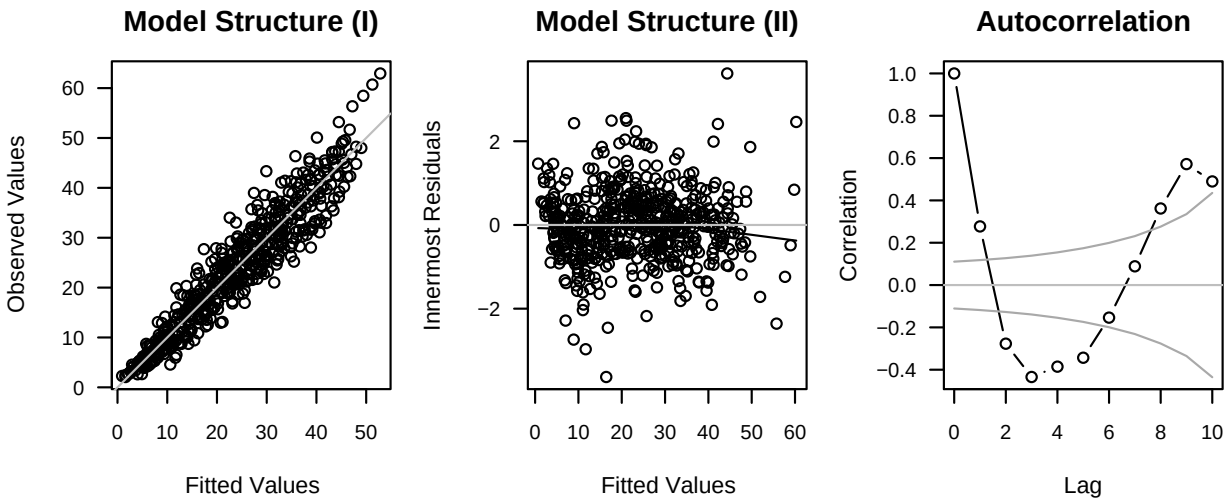

Figure 8.14: Selected diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data.

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.

```
> hd.lme.5 <- update(hd.lme.4, correlation = corCAR1())

> 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)
```

Figure 8.15: Selected diagnostics for the mixed-effects fit of the Height/Diameter model for Stage's data.

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.

```
> print(plot(hd.lme.5, resid(.) ~ fitted(.) | HabType.ID, layout = c(1,
+     5)))
```

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`.

```
> print(qqmath(~resid(hd.lme.5) | stage$HabType.ID, prepanel = prepanel.qqmathline,
+     panel = function(x, y) {
+         panel.qqmathline(y, distribution = qnorm)
+         panel.qqmath(x, y)
+     }))
```

There seems little evidence in either of figures 8.16 and 8.17 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:

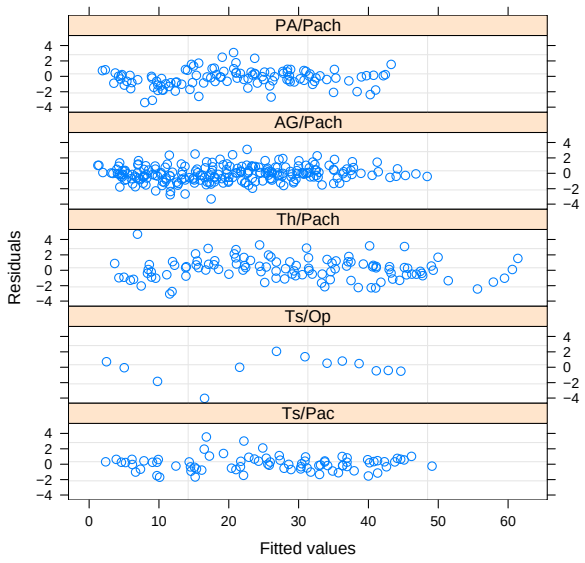

Figure 8.16: Innermost Pearson residuals against fitted values by habitat type.

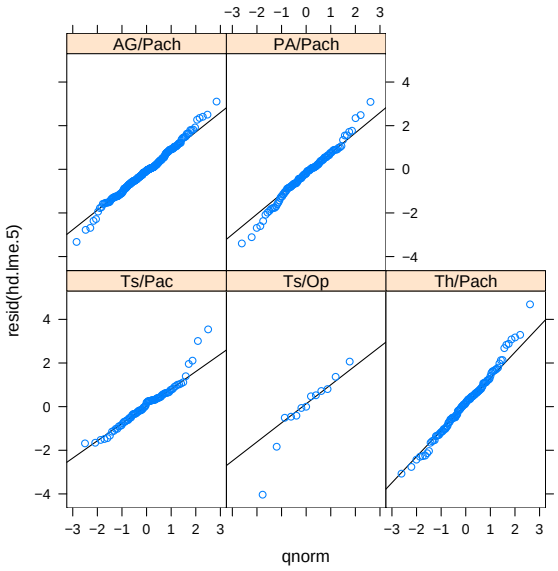

Figure 8.17: Quantile plots of innermost Pearson residuals against the normal distribution, by habitat type.

- `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!

```
> print(plot(augPred(hd.lme.5), index.cond = list(panel.order),
+     strip = strip.custom(par.strip.text = list(cex = 0.5))))
```

```
> summary(hd.lme.5)

Linear mixed-effects model fit by REML
 Data: stage
       AIC      BIC    logLik
  1954.898 2027.823 -960.449
```

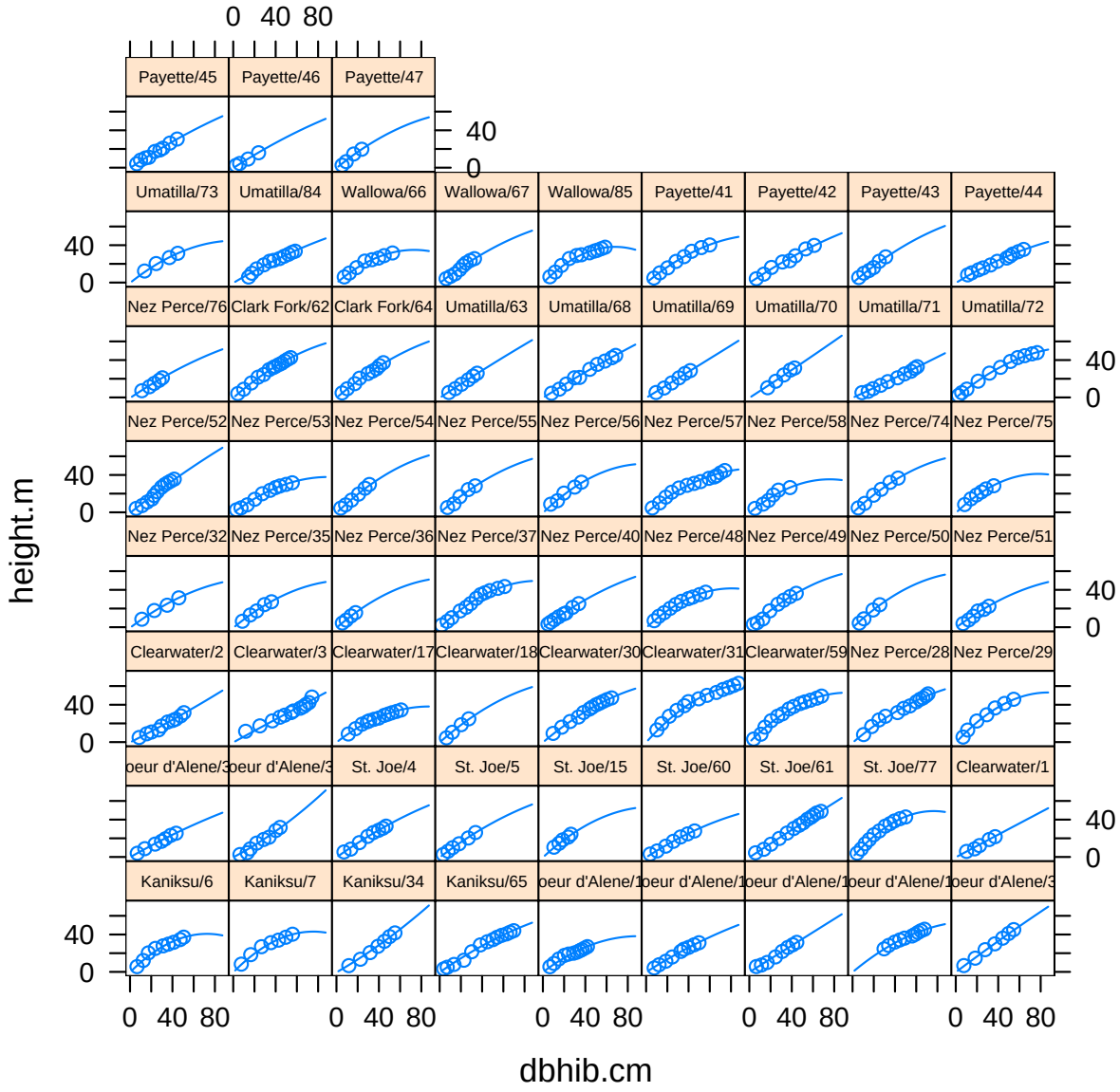

Figure 8.18: Height against diameter by tree, augmented with predicted lines.

```
Random effects:
 Formula: ~dbhib.cm + I(dbhib.cm^2) | Forest.ID
 Structure: General positive-definite, Log-Cholesky parametrization
               StdDev      Corr
(Intercept)   0.275624217 (Intr) dbhb.c
dbhib.cm      0.079258347 -0.563
I(dbhib.cm^2) 0.001170039  0.575 -0.989

 Formula: ~dbhib.cm + I(dbhib.cm^2) | Tree.ID %in% Forest.ID
 Structure: General positive-definite, Log-Cholesky parametrization
               StdDev      Corr
```

```
(Intercept)   0.002125098 (Intr) dbhb.c
dbhib.cm      0.172264549 -0.518
I(dbhib.cm^2) 0.002924372  0.071 -0.800
Residual      1.423900827

Correlation Structure: Continuous AR(1)
 Formula: ~1 | Forest.ID/Tree.ID
 Parameter estimate(s):
      Phi
0.6670859
Fixed effects: height.m ~ dbhib.cm + I(dbhib.cm^2)
                  Value  Std.Error  DF   t-value p-value
(Intercept)   -0.4751447 0.27646715 474 -1.718630  0.0863
dbhib.cm       0.8840097 0.04073775 474 21.700013  0.0000
I(dbhib.cm^2) -0.0031469 0.00066374 474 -4.741178  0.0000
 Correlation:
              (Intr) dbhb.c
dbhib.cm      -0.493
I(dbhib.cm^2)  0.437 -0.915

Standardized Within-Group Residuals:
        Min          Q1         Med          Q3         Max
-2.83345241 -0.48312402 -0.01318825  0.42506163  3.29282001

Number of Observations: 542
Number of Groups:
             Forest.ID Tree.ID %in% Forest.ID
                     9                    66
```

8.4.2 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")
```

In order to assess whether we would be better served by adding habitat type to the model, we can construct a graphical summary, thus:

```
> print(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)
```

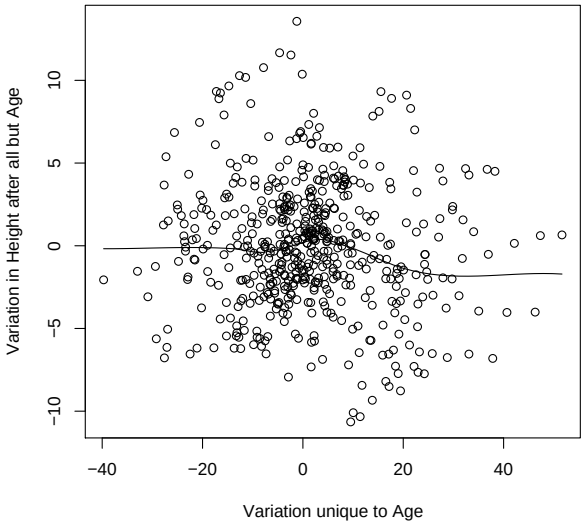

Figure 8.19: Added-variable plot for Age against the ratio of Height over Diameter.

```
+         panel.abline(lm(y ~ x), lty = 3)
+     }))
```

Neither of figures 8.19 or 8.20 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.

8.5 The Model

Let's examine our final model.

$$\begin{aligned} y_{ijk} &= \beta_0 + b_{0i} + b_{0ij} && (8.10)\\ &\quad + (\beta_1 + b_{1i} + b_{1ij}) \times x_{ijk} && (8.11)\\ &\quad + (\beta_2 + b_{2i} + b_{2ij}) \times x_{ijk}^2 && (8.12)\\ &\quad + \epsilon_{ijk} && (8.13) \end{aligned}$$

In matrix form, it is still:

$$\begin{aligned} \mathbf{Y} &= \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \boldsymbol{\epsilon}\\ \mathbf{b} &\sim \mathcal{N}(\mathbf{0}, \mathbf{D})\\ \boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, \mathbf{R}) \end{aligned}$$

Here's the structure.

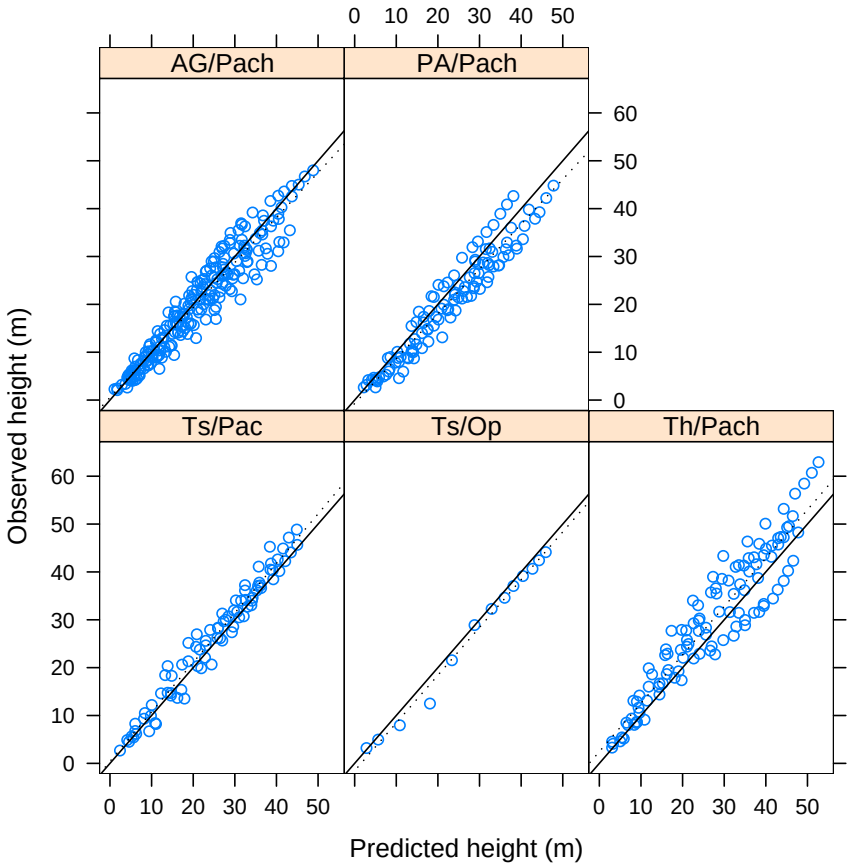

Figure 8.20: Plot of predicted height against observed height, by habitat type. The solid line is 1:1, as predicted by the model. The dotted line is the OLS line of best fit within habitat type.

- 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.

8.5.1 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:

$$\mathbf{Z} = [\mathbf{Z_f} \mid \mathbf{Z_t}] \tag{8.14}$$

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

$$Z_f = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, d_{524}, d_{524}^2, 0, 0, 0, \ldots) \tag{8.15}$$

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.

8.5.2 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).

$$\mathbf{b} = (b_{f10}, b_{f1d}, b_{f1d2}, b_{f20}, b_{f2d}, b_{f2d2}, \ldots, b_{t10}, b_{t1d}, b_{t1d2}, b_{t20}, b_{t2d}, b_{t2d2}, \ldots)' \tag{8.16}$$

The combination of $\mathbf{b}$ and $\mathbf{Z}$ serves to allocate each random effect to the appropriate unit and measurement.

8.5.3 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 $\mathbf{D_t}$.

$$\mathbf{D_f} = \begin{bmatrix} \sigma_{bf0}^2 & \sigma_{bf0d} & \sigma_{bf0d2} \\ \sigma_{bf0d} & \sigma_{bfd}^2 & \sigma_{bfdd2} \\ \sigma_{bf0d2} & \sigma_{bfdd2} & \sigma_{bfd2}^2 \end{bmatrix} \tag{8.17}$$

$$\mathbf{D_t} = \begin{bmatrix} \sigma_{bt0}^2 & \sigma_{bt0a} & \sigma_{bt0d2} \\ \sigma_{bt0d} & \sigma_{btd}^2 & \sigma_{btdd2} \\ \sigma_{bt0d2} & \sigma_{btdd2} & \sigma_{btd2}^2 \end{bmatrix} \tag{8.18}$$

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.

Chapter 9

Extensibility - R Packages

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)

[1] TRUE
```

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

> ip <- installed.packages()
> class(ip)

[1] "matrix"

> ip <- as.data.frame(ip)
> names(ip)

 [1] "Package"  "LibPath"  "Version"  "Priority" "Bundle"   "Contains"
 [7] "Depends"  "Suggests" "Imports"  "Built"

> length(ip$Package)

[1] 26

> ip$Package
```

```
 [1] KernSmooth MASS       base       boot       class      cluster
 [7] datasets   foreign    grDevices  graphics   grid       lattice
[13] methods    mgcv       nlme       nnet       rpart      spatial
[19] splines    stats      stats4     survival   tcltk      tools
[25] utils      xtable
26 Levels: KernSmooth MASS base boot class cluster datasets ... xtable
```

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()` 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". Note that 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.

Ah, equivalence needs `boot`. Ok,

The equivalence package is now available for your use. Please note again the use of forward slashes as directory delimiters is platform independent.

Chapter 10

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 10.1) or other languages such as C and Fortran (Section 10.5). Being comfortable writing functions is very useful, as it adds an entirely new dimension to efficient and enjoyable programming.

10.1 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! Of course this is trivial. But as far as R is concerned now, this function is just like any other.

10.2 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.2.area <- function(radius) {
+     x * radius^2
+ }
> radius.2.area(4)

[1] 50.26548
```

10.3 S3 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. Note that there is no such class as `eh`. Let us invent one. Let's say that the `eh` class is just like `integer`, except that when we print out an object of class `eh` we only want the last one.

```
> x <- c(1:10)
> class(x)

[1] "integer"

> class(x) <- "eh"
> class(x)

[1] "eh"
```

Now we write a print function that will be used for all objects of class `eh`. Printing can then be invoked in one of two ways.

```
> print.eh <- function(x) print(x[length(x)])
> print(x)

[1] 10

> x

[1] 10
```

Notice that `x` is still of length 10.

```
> length(x)

[1] 10
```

Also notice that functions of `x` retain the class

```
> x * x

[1] 100

> class(x * x)
```

```
[1] "eh"

> length(x * x)

[1] 10
```

How do we escape now?

```
> class(x) <- "integer"
> x

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

In brief, then, to write a function, say `fu`, that will be automatically invoked to replace an existing `fu` when applied to an object of class `bar`, we need merely call the function `fu.bar`.

10.4 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.

10.5 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 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)
> n
 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 Gaussian; 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.

$$\hat{U}_+ = \binom{n}{2}^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I_+(D_i + D_j) \tag{10.1}$$

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! For 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.

$$\hat{\sigma}^2_{\hat{U}_+} = \binom{n}{2}^{-1} \{2(n-2)[\hat{q}^+_{1(2,3)} - \hat{U}^2_+) + \hat{U}_+ - \hat{U}^2_+\} \tag{10.2}$$

where

$$\hat{q}^{+}_{1(2,3)} = \binom{n}{3}^{-1} \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{3} \left[I_{ij} I_{ik} + I_{ij} I_{jk} + I_{ik} I_{jk} \right] \tag{10.3}$$

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 some R-code,

```
  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.

10.5.1 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. Finally, I wrote the code in XEmacs - it colors C syntactically.
- 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.

10.5.2 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. 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 build -binary <libraryname>`

10.5.3 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.

10.5.4 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.

10.5.5 Benefit

```
> 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.

Bibliography

Laird, N. M., Ware, J. H., 1982. Random-effects models for longitudinal data. Biometrics 38, 963–974.

Pinheiro, J. C., Bates, D. M., 2000. Mixed-effects models in S and Splus. Springer-Verlag.

Robinson, G. K., 1991. That BLUP is a good thing: The estimation of random effects. Statistical Science 6 (1), 15–32.

Schabenberger, O., 2005. Mixed model influence diagnostics. In: SUGI 29. SAS Institute, pp. Paper 189–29.

Schabenberger, O., Pierce, F. J., 2002. Contemporary statistical models for the plant and soil sciences. CRC Press.

Stage, A. R., 1963. A mathematical approach to polymorphic site index curves for Grand fir. Forest Science 9 (2), 167–180.

Venables, W. N., Ripley, B. D., 2000. S programming. Springer-Verlag.

Venables, W. N., Ripley, B. D., 2002. Modern applied statistics with S and Splus, 4th Ed. Springer-Verlag.