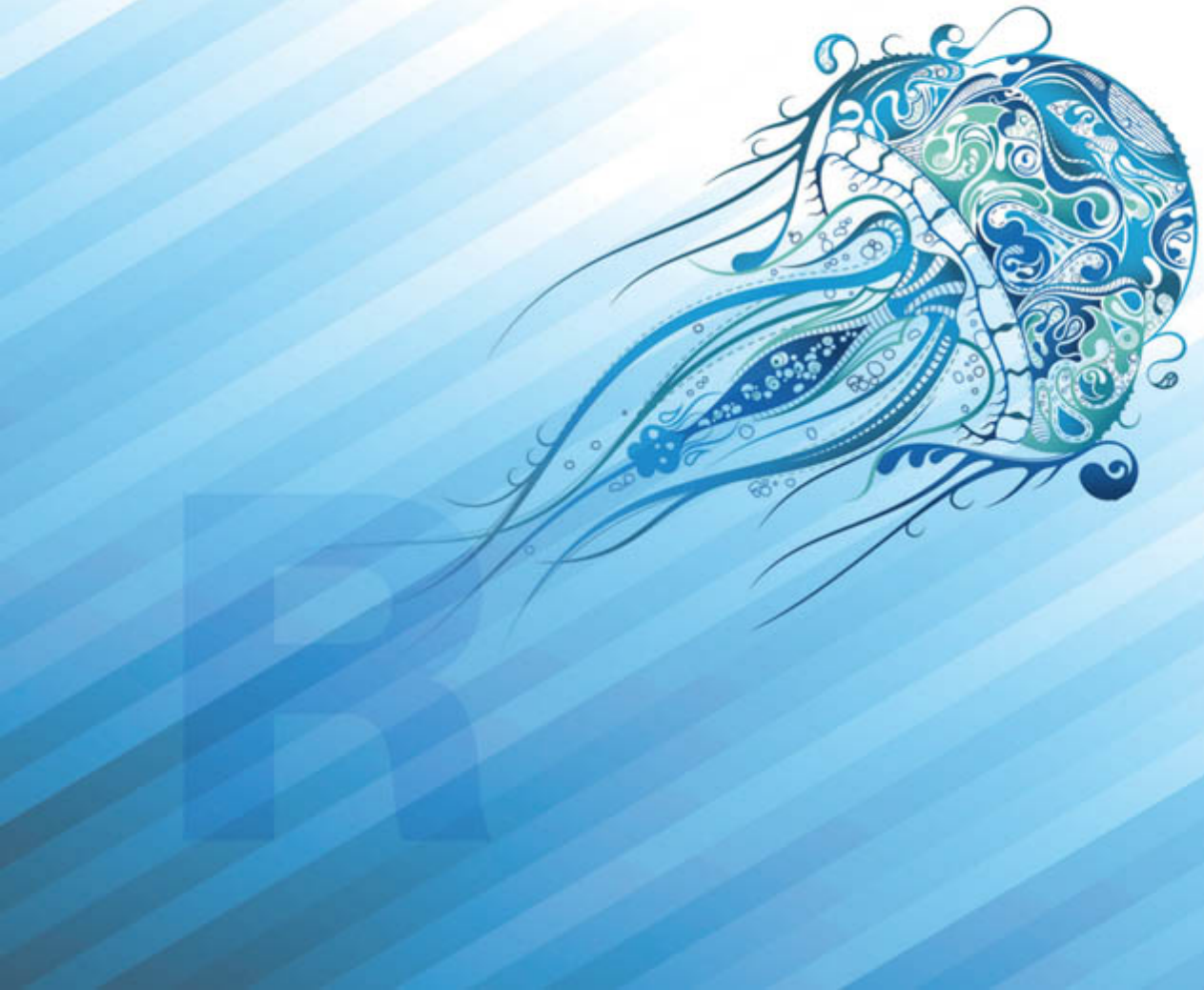
OXFORD
BIOLOGY

GETTING STARTED WITH *R*

An Introduction for Biologists

ANDREW P. BECKERMAN
& OWEN L. PETCHEY



Getting Started with R

This page intentionally left blank

Getting Started with R

An Introduction for Biologists

ANDREW P. BECKERMAN
& OWEN L. PETCHEY

Department of Animal and Plant Sciences
University of Sheffield
&
Institute of Evolutionary Biology and
Environmental Studies
University of Zürich

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First Edition published in 2012

Impression: 1

British Library Cataloguing in Publication Data

Data available

Library of Congress Cataloging in Publication Data

Library of Congress Control Number: 2011945448

ISBN 978–0–19–960161–5 (hbk.)
978–0–19–960162–2 (pbk.)

Printed in China by
CC Offset Printing Co. Ltd

Table of Contents

Preface

What this book is about

This is a book about how to use R, an open source programming language and environment for statistics. It is not a book about statistics *per se*, but a book about getting started using R. It is a book that we hope will teach you how using R can make your life (research career) easier.

We love R. We use statistics in our everyday life as researchers and we use R to do this. We are first and foremost evolutionary ecologists, but over the past 10 years we have developed, at first in parallel and then together, an affinity for R. We want to share our 20 years of combined experience using R to show you how easy, important, and exciting it can be. This book is based on a 3–5 day course we have given in various guises around the world. The course is designed to give students and staff alike a boost up the steep initial learning curve associated with R. We assume that course participants already use some spreadsheet, statistical, and graphing programs (such as Excel, GLIM, SAS, JMP, Statistica, and SigmaPlot). Most participants have some grasp of common statistical methods, including chi-square tests, the t-test, and ANOVA. In return for a few days of their lives, we give participants knowledge about how to easily use R, and R only, to manage data, make figures, and do statistics. R changed our research lives, and many participants agree that it has done the same for them.

The effort we put into developing the course and this book are, however, miniscule compared to the effort of the R Core Development Team. Please

remember to acknowledge them when you use R to analyse and publish your amazing results.

What you need to know to make this book work for you

There are a few things that you need to know to make this book, and our ideas, work for you. Many of you already know how to do most of these things, having been in the Internet age for long enough now, but just to be sure:

1. You need to know how to download things from the Internet. If you use Windows, Macintosh, or Linux, the principles are the same, but the details are different. Know your operating system. Know your browser and know your mouse (both buttons!).
2. You need to know how to make folders on your computer and save files to them. This is essential for being organized and efficient.
3. You need to understand what a "PATH" is on your computer. This is the address of the folders and files (i.e. the path to a file). On Windows, depending on the type you are using, this involves a drive name, a colon (:) and slashes (\\ 's or / 's). On a Macintosh and Linux/Unix, this requires the names of your hard drive, the name of your home directory, the names of folders, and slashes (/).
4. You need to be able to type instructions that your computer (R, actually) will attempt to carry out. Menus and the mouse are not very important in R! You'll eventually love this feature of R. You might not like it in the beginning.
5. You need to know how to use Microsoft Excel or a program that saves rows and columns of data as a "comma separated values file" (".csv"). While there are other ways of getting data into R, our method is based on storing a copy of your **raw** data in a spreadsheet, specifically as a .csv file (comma separated). Again, this is a convention we take, and no more than that.
6. Finally, you need to know how to do, and why you are doing, statistics. We recommend that you know the types of questions a

t-test, a chi-square test, linear regression, ANOVA, and ANCOVA are designed to help you answer BEFORE you use this book. As we said, we are not aiming to teach you statistics *per se*, but how to do some of the most common plotting and frequent statistics in R, and understand what R is providing as output.

How the book is organized

In this book, we will show you how to use R in the context of every day research in Biology. Our philosophy assumes that you have some data and would like to derive some understanding from it. Typically you need to manage your data, explore your data (e.g. by plotting it), and then analyse your data. Before any attempt at analysis, we suggest that you always plot your data. As always, analysing (modelling) your data involves first developing a model and testing critical assumptions associated with the statistical method (model). Only after this do you attempt interpretation. Our focus is on developing a rigorous and efficient routine (workflow) and a template for using R for data exploration, visualization, and analysis. We believe that this will give you a *functional* approach to using R, in which you always have the goal (understanding) in mind.

We start by providing in-depth instruction for how to get data into R, manipulate and summarize your data, and make a variety of informative, publication-quality figures common to our field. We then provide an overview of how to do some common analyses. In contrast to other books, we spend most of our time helping you to develop a workflow for analysis and an understanding of how to tell R what to do. We also help you identify core pieces of R output that are reported regularly in our field. This is important because the output of all statistic packages is different.

Chapter 1 is titled Why R? It is an (our) overview of why you might spend a few days (and more!) of your valuable time converting your data management, graphics, and analysis to R. There are many reasons, though

we advise all readers to make a careful decision about whether the investment of time and effort will give sufficient return.

Chapters 2–4 are based on our tried and tested **Import, Explore, Graph**. We walk you through one of the most difficult stages in using R—getting your data into R and producing your first figure in R. Then we show you how to explore your data, summarize it in various forms, and plot it in various formats. Visualizing your data before you do statistics is vital. These chapters also introduce you to the script—a permanent, repeatable, annotated, shareable, cross-platform record of your analysis.

Chapter 5 introduces you to our workflow for implementing and interpreting t-tests, chi-square tests, and general linear models. General linear models are a flexible set of methods that include the more well-known concepts of regression, ANOVA, and ANCOVA. In the spirit of functionality and making R work for you, our objective is to help you develop a repeatable and reliable workflow in R. We focus on helping you produce interesting, appealing, and appropriate figures, interpreting the output of R and crafting sensible descriptions of the methods and results for publication. Our focus is the workflow.

Throughout the book, we highlight where you can work along with us, on your own computer, using R through the use of this symbol at the left. Throughout the book, we use syntax (code) colouring as found in the OSX R script editor. This should help visualize the instructions you need to give R. We believe this book can be used as a self-guided tutorial. All of the datasets we use are available online at http://www.r4all.org. There are also 13 boxes embedded in the book with extra detail about certain topics. Take your time and learn the magic of R.

Why R?

Some of you will have established research careers based around using a variety of statistical and graphing packages. Some of you will be starting with your research career and wondering whether you should use some of the packages and applications that your supervisor/research group uses, or jump ship to R. Perhaps your group already uses R and you are just looking for that "getting started" book that answers what you think are embarrassing questions. Regardless of your stage or background, we think a formal introduction to an approach to, and routine for using R, will help. We begin by reviewing a core set of features and characteristics of R that we think make it worth using and worth making a transition to from other applications.

First, we think you should invest the effort because it is freely available and cross-platform (e.g. it works on Windows, Macs [OS X], and Linux). This means that no matter where you are and with whom you work, you can share data, figures, analyses, and most importantly the instructions (also known as scripts and code) used to generate the figures and analyses. Anyone, anywhere in the world, with any kind of Windows, Macintosh, or Linux operating system, can use R, without a license. If you, or your

department, or your university invest heavily in multiple statistical packages, R can save a great deal of money. When you change institutions, R doesn't become inaccessible, get lost, or become un-usable.

Second, R is a command line programming language. It does not involve extensive use of menus. As a result you have to know what to ask R, know why you are asking R for this, and know what to expect from R. You can't just click on menus and get some results. This means that by using R, you continually learn a great deal about statistics and data analysis.

Third, it is free. Oh, we said that already. Actually, it's more accurate to state that it's freely available. Lots of people put an awful lot of effort into developing R…that effort wasn't free. Please acknowledge this effort by citing R when you use it.

Fourth, we believe that R can replace common combinations of programs that you might use in the process of analysing your data. For example, we have, at times, used two to three of Excel, Minitab, SAS, Systat, JMP, SigmaPlot, and CricketGraph, to name a few. This results in not only costly licensing of multiple programs, but software specific files of various formats, all floating around in various places on your computer (or desk) that are necessary for the exploration, plotting, and analysis that make up a research project. Keeping a research project organized is hard enough without having to manage multiple files and file types, proprietary data formats, and the tools to put them all together. Furthermore, moving data between applications introduces extra steps into your workflow. These steps are removed by investing in R.

Fifth, you can make publication-quality figures in R, and export them in many different formats, including pdf. We now use only R for making graphs, and when submitting manuscripts to journals we usually send only pdf files generated directly from R. One of the nice things about pdfs is that they are resolution independent (you can zoom in as far as you like and they don't get blocky). This means that publishers have the best possible version of your figure. And if the quality is poor in the published version of your paper, you know it is down to something the publishers have done!

Finally, and quite importantly, R makes it very easy to write down and save the instructions you want R to execute—this is called a **script** in R. In fact, the script becomes a *permanent, repeatable, annotated, cross-platform, shareable record of your analysis*. Your entire analysis, from transferring your data from field or lab notebook, to making figures and performing analyses, is all in one, secure, repeatable, annotated place.

So, if you have not already done so, go get R. Follow this link and locate the server closest to you holding R:

http://cran.r-project.org/mirrors.html

Then click on the operating system (Mac OS X, Windows, and Linux) you use. This will take you to a page specific to your operating system. For Mac OS X users, look down until you see a link to something like R-2.13.pkg. Click on this, and then install the downloaded file. For Windows users, you get taken to a page where you click on the "base" link (you want the "base" version of R); from here the on-screen instructions make it clear what to do next. If you get stuck here, please read the instructions about installation in the FAQs, linked at the bottom of every R homepage. Linux users, and those of other operating systems, probably know how to get and download R. If any of you are stuck at this stage, take a look at **Box 2.1 and Box 2.2** for more information on setting up your computer and getting and installing R. If you're still stuck, email one of us. Really.

This page intentionally left blank

Import, Explore, Graph I

Getting Started

One of the most frequent stumbling blocks for new students and seasoned researchers using R is actually just getting the data into R. This is really unfortunate, since it is also most people's first experience with R! Let's make this first step easy.

Assume you have some datasheets on your clipboard or in your lab book. Perhaps these data are measures of the abundances of various species in various places, or morphological measurements of individual organisms. Obviously, you need to put this data into your computer in order that you can then get it into R. And though we've implied that R is good for everything, it is not so good for entering large amounts of data.

Instead, use a spreadsheet application such as Microsoft Excel, Open Office, or Numbers to enter your data. Make the first row of your spreadsheet the names of your variables (column names). Keep these variable names informative, brief and simple. Try not to use spaces or special symbols, which R can deal with but may not do so particularly nicely. Also, if you have categorical variables, such as sex (male, female) use the category names rather than codes for them (e.g. don't use 1 = male, 2 = female).

R can easily deal with variables that contain words, and this will make subsequent tasks in R much more efficient.

Finally, we highly recommend that you enter data so there is one observation per row. That is, make a dataset with column names like treatment1, treatment2, replicate, response_variable. This will make your R-life much, much easier, since many of the functions in R prefer data this way. If you, for example, made a dataset with columns treatment1, treatment2, response of replicate 1, response of replicate 2, and response of replicate 3, you would have to do some data manipulation to get the data into the correct shape for many R functions.

Next, print your entered data onto paper, and check that this copy of your data matches the data on your original datasheets. Correct any mistakes you made when you typed your data in.

Now, don't save your file as an .xls, .xlsx, .oo, or .numbers file. Instead, save it as a "comma separated values" file (.csv file). In Excel, Open Office or Numbers, after you click Save As… you can change the format of the file to "comma separated values', then press Save. Excel might then, or when you close the Excel file, ask if you're sure you'd like to save your data in this format. Yes, you are sure! At this point in our workflow, you have your original datasheets, a digital copy of the data in a .csv file, and a printed copy of the data from the .csv file.

One of the remarkable things about R is that once a copy of your "raw data" is established, the use of R for data visualization and analysis will never require you to alter the original file any way (unless you collect more data!). Therefore keep it very safe! In many statistical and graphing programs, your data spreadsheet has columns added to it, or you manipulate columns or rows of the spreadsheet before doing some analysis or graphic. With R, your original can always remain unmanipulated. If any adjustments do occur, they occur in an R copy only and only via your explicit instructions.

But how do you do actually get the data in this .csv file into R? How do you go from making a dataset in a .csv file to having a working copy in R's brain to use for your analysis?

2.1 *Where* to put your data

Part of keeping your data safe is putting it somewhere safe. But where? We like to be organized and we want you to be organized too.

Make a new folder in the main location/folder in which you store your research information—perhaps you have a Projects folder in MyDocuments (PC) or in the Documents folder (Mac and e.g. Unbuntu); create a folder inside this location. Give the folder an informative name—perhaps "MyFirstAnalysis".

Understanding where your data is stored is very important for learning how to use R. The location, or address, for information on a computer is known as the **PATH**. How do you find out the **PATH** to a folder or dataset on a computer? First, note that there are some really easy ways to get the **PATH** to your raw data file without having to type it in. This is important, since typing in PATHs incorrectly is one of the most common early stumbling blocks, and a very frustrating one. Essentially, the trick for getting paths is to copy them from the Finder (OS X) or the Explorer (Windows) (see Box 2.1 for details). You shouldn't ever have to type a PATH in, though sometimes you might choose to. In case you do, here's what you need to know.

In Windows, the location of things historically starts with a drive letter followed by a colon and some slashes, for example "C:\\MyDocuments\\project1\\raw data\\file.csv". The conventions for the direction of the slashes in Windows varies, depending on how old your version of Window's is—they are either presented as single or double backslashes. On a Macintosh or Linux/Unix machine, the location will usually be in some folder in the path from your home directory, such as "Users/username/Documents/project1/raw data/file.csv" or "~/Documents/project1/raw data/file.csv".

In R, the convention for writing a **PATH** is always a single forward slash. So, whether you are on a PC or a MAC or a Linux/Unix box, the way you describe where files are within your folders is consistent: one forward slash. However, if you copy PATH information from Windows, you will probably need to change backward slashes to forward slashes.

Box 2.1: The PATH: Where are your files?

Linux/Unix

Our guess is that if you are using Linux/Unix to run R, you understand the PATH.

Macintosh

To make OSX show the PATH in the title of the Finder, you need to invoke a command in the Terminal.

Go to Applications/Utilities/Terminal.app and double click.

At the prompt, type the following:

```
defaults write com.apple.finder _FXShowPosixPathInTitle -bool YES
```

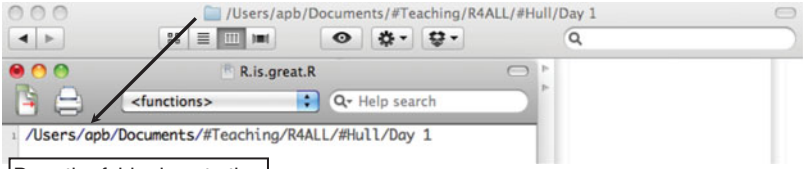
Next, type the following (this restarts the finder).

```
killall Finder
```

Now, the top of your finder window should show the location where you are—the PATH.

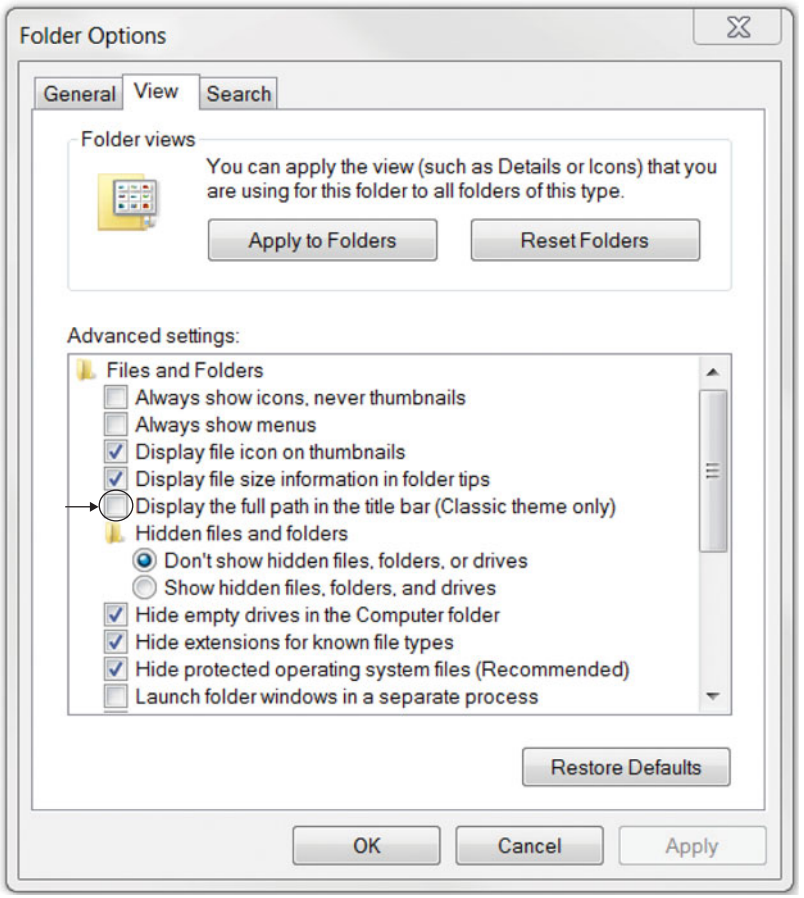
Note that Linux/Unix systems and Macintosh systems use the same convention for describing the PATH—a forward slash separates all folders and documents in the final folder.

One feature of OSX is that you can actually drag this information to another document. If you click and hold the folder symbol, you can drag this information to any text editor, including an R script. In fact, dragging ANY folder or document to the script editor for the Macintosh will paste the path.



Windows

To make Windows show the path in the explorer, you need to set the preferences in the Tools menu for the explorer. Tools->Folder Options->View Tab will take you to a list of tick boxes. There should be a tick box labelled "Display full path…" which will show the path to files in the address bar and the title bar.



IMPORTANT NOTE: In contrast to Linux/Unix and the Macintosh operating system, Windows tends to present paths using a single backslash. The even older convention is two backslashes. R accepts either ONE forward slash or TWO backslashes to read files from the hard drive into R. Even if you copy the path from the Explorer into R, you will need to change the "slashes" to ONE forward, or TWO backwards.

If you're confused by any of that, don't forget you can copy and paste paths, and **Box 2.1** provides a bit more detail about how to do this. We assume that Linux/Unix users are familiar with **PATHs**.

2.2 Make a folder for your instructions (code, script)

Now, inside that folder you just made, make another folder called *analyses* (i.e. "MyFirstAnalysis/analyses"). And perhaps another folder called *manuscript*. Perhaps another one called *Important PDFs*. We think you might be getting the point: use folders to organize separate parts of your project. Your file of instructions for R—your **script** file—will go in the *analyses* folder. We'll build one soon. Be patient. ☺

What is the **script** file? As we noted in the Preface, R allows you to write down and save the instructions that R uses to complete your analysis. Trust us, this is a desirable feature of R. As a result of making a **script**, you, as a researcher, end up with a *permanent, repeatable, annotated, shareable, cross-platform archive* of your analysis. Your entire analysis, from transferring your data from notebook, to making figures and performing analyses, is all in one, secure, repeatable, annotated place. We think you can see the value.

2.3 How to get your data into R and where it is stored in R's brain

Now comes the fun part. You've organized your life a bit. You know where your data is, and you know where you will store things associated with a particular project/experiment/analysis. Let's use R.

Wait! R? Where is it? If you have not installed it yet, **Box 2.2** provides detailed instructions on how to install R on your machine—be it a PC, Macintosh, or Linux. It's not hard, but there are some helpful hints in there.

Now, assuming the installation has been successful, start R. Click the icon on your desktop, navigate to the start menu, find the applications folder, click on the icon in the dock . . . you know what to do. You are clever. You know how to start an application on your computer!

Starting R does the same thing on Windows and Macintosh machines. The most obvious is the creation of at least one window, called the **R CONSOLE**. This is the window into the engine of R. This is the window which accepts commands from you, sends these commands to the little people inside R that are really smart and talk in a language of 1s and 0s, and then returns an answer.

The second type of window in R is called the **SCRIPT**. Not to labour the point, but the **SCRIPT** is a *permanent, repeatable, annotated, shareable, cross-platform archive* of your analysis. It is a bit like a word-processing window that you write stuff in, but it tends to have at least one major feature that word processors don't—there are combinations of keyboard strokes that send information from the **SCRIPT** to the **R Console**, automatically.

To make a new **SCRIPT**, simply go to the File menu and choose *New Document* (OS X) or New script (Windows). **Box 2.3** provides some added insight into the built-in script editors in the Windows and Macintosh versions of R.

Now you are ready to use R. If you have been working along with us, you will have a data file residing in a folder, R running as an application, and two windows visible on your desktop—the **CONSOLE** and a **SCRIPT** window.

2.4 Working with R—hints for a successful first (and more) interaction

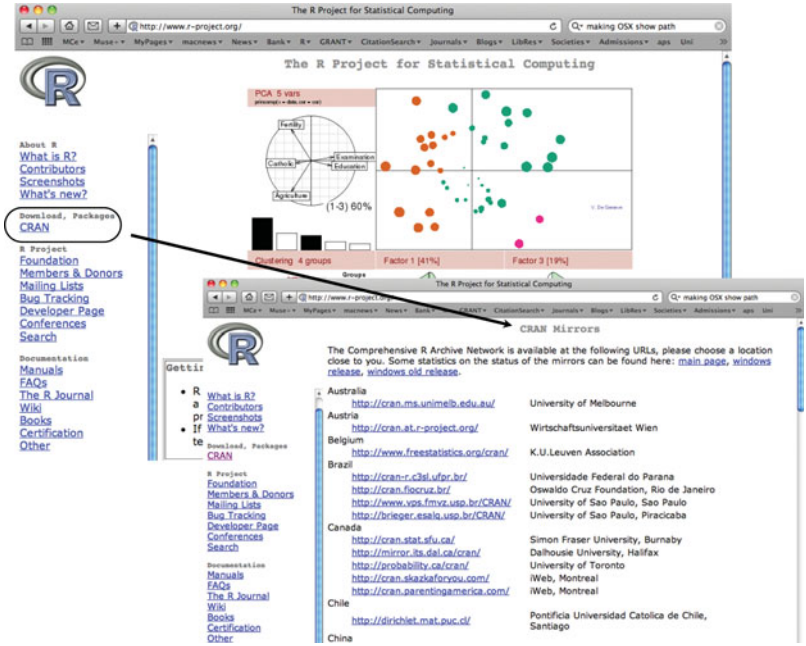
First, if you are a Windows users, consider downloading Rstudio (http://www.rstudio.org/) and using it to run R. We're pretty sure that you'll like it more than the Windows R application, simply for the script functionality. Among other things, it does a much better job of keeping your different R windows organized. And it has a nice script editor. You can use Rstudio on a Macintosh, but there is a nice script editor already built in.

Second, we'd like to introduce a very special keyboard character. It is the # character. It is important to find this character on your keyboard. It may not be obvious on European keyboards, especially European Macintosh

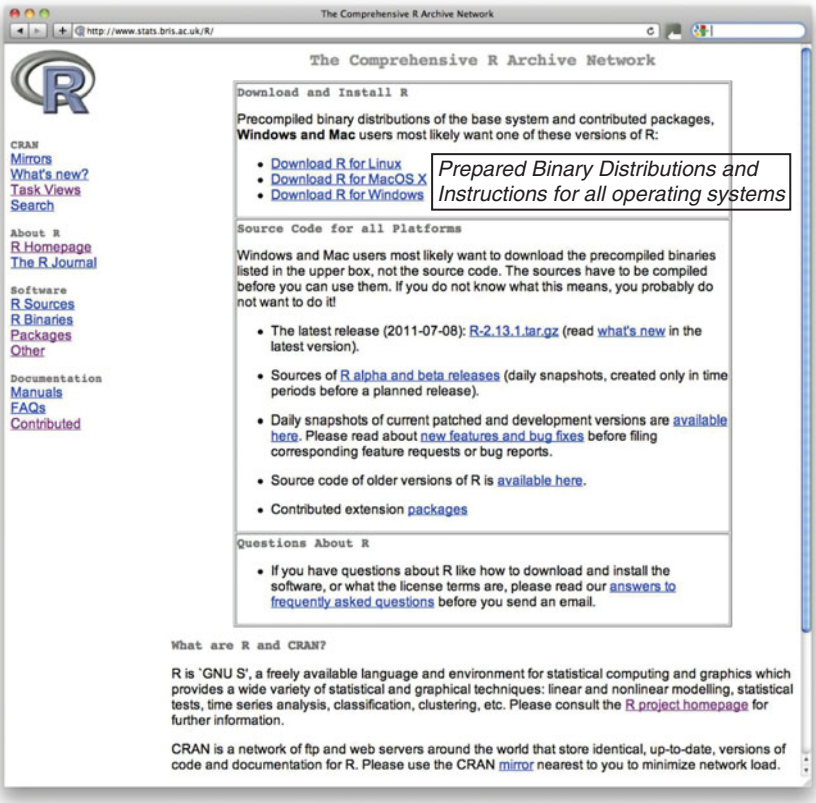
Box 2.2: Downloading and installing R

R software is maintained at the Comprehensive R Archive Network—CRAN. The R Project Home Page is http://www.r-project.org/. However, there are many mirrors providing local access to all software and documentation—physical locations around the world with up to date copies of the latest versions of R and its associated packages.

We suggest that you locate the mirror closest to you:



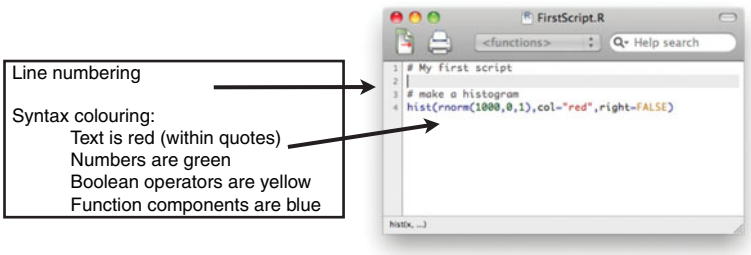
You can bookmark your favourite mirror in your web browser. The mirror provides all the files and instructions necessary for installing R on your computer. Here we show the mirror at Bristol University, UK. Simply click on the link for your operating system, READ THE INSTRUCTIONS, and proceed to enjoy R. There is a section at the bottom of the boxes called "Questions About R". If you think you might ignore these, you are exactly the people they are written for.


The Comprehensive R Archive Network
http://www.stats.bris.ac.uk/R/
The Comprehensive R Archive Network
CRAN
Mirrors
What's new?
Task Views
Search
About R
R Homepage
The R Journal
Software
R Sources
R Binaries
Packages
Other
Documentation
Manuals
FAQs
Contributed
Download and Install R
Precompiled binary distributions of the base system and contributed packages, Windows and Mac users most likely want one of these versions of R:
Download R for Linux
Download R for MacOS X
Download R for Windows
Prepared Binary Distributions and Instructions for all operating systems
Source Code for all Platforms
Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!
The latest release (2011-07-08): R-2.13.1.tar.gz (read what's new in the latest version).
Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
Source code of older versions of R is available here.
Contributed extension packages
Questions About R
If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.
What are R and CRAN?
R is `GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.
CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Box 2.3: Script editors

Macintosh

The Macintosh version of R comes with a feature-rich script editor. It provides line numbering, syntax colouring, and a keyboard combination for sending information from the script to the console.



To "send" this information from the script to the console, you may use the keyboard combination:

cmd + return

or the leftmost icon in the script, with the red arrow (send this to the console).

If you simply press cmd + return, the line on which the cursor is on will be sent to the console. Alternatively, you can select larger portions of the script, and then press cmd + return. This will send the entire selection to R.

Because the Mac is now based on UnixBSD, it also has access to a variety of Unix/Linux script editors including vi, emacs, and so on. The Mac version of emacs can be fully functional and integrated with R (using Emacs Speaks Statistics).

Windows

The built in Windows script editor is currently as basic as it gets. It is simply a text editor. It contains no line numbering or syntax highlighting. However, it does possess a keystroke combination for sending lines or selections from the script to the console:

control + R

Despite the lack of features for the built in editor, there are numerous Windows editors that interface with R very well. The two most popular with our students seem to be TinnR and Rstudio (which is actually available for Windows, Macintosh and Linux):

http://www.sciviews.org/Tinn-R/
http://rstudio.org/

General

There are innumerable scripting applications available for many platforms. A small repository can be found here: http://www.sciviews.org/_rgui/index.html.

keyboards. If it is not visible, it is often associated, somehow, with the number 3 (for example, alt+3 on many Macintosh keyboards). Why is this important? In your script, you will type commands telling R to, for example, make a plot or run a t-test. However, the script is also an *annotated archive* of your instructions to R. **Annotation** (commenting) is a process by which you add notes, interpretation, explanation, and any other information to your script. These notes help you remember what you were thinking when you wrote the script. This could be information you need/want to remember every time you come back to the script. It might be information or an explanation about a particularly clever or complicated set of instructions. It may just be information that lets you and your colleagues understand the process by which you are developing graphics and making models. Importantly, annotations (comments) are not for R…just for us humans.

Any information you don't want R to read and attempt to process should be preceded by a #. That is, you start a line in the **SCRIPT** with # and then write something after it. When R sees a #, it does not try and get the little people inside to do anything. It ignores this information. This information is for you, not for R.

2.5 Make your first script file

So, at this stage, we suggest you write something like the following at the top of the script file. In fact, it is good practice to start every script you make with this information, or some variant of it.

```
# Amazing R. User (your name)
# 12 January, 2021
# This script is for the analysis of coffee consumption and
# burger eating
```

Now, save this script to your analysis folder. You can either choose *File ->Save* or use keystrokes like CTRL + S (Windows) or cmd + S (Macintosh). Again, provide an informative name for the script.

A note about the three letter combination on the end of file names…otherwise known as file extensions (such as .exe, .csv, .txt). These tell your operating

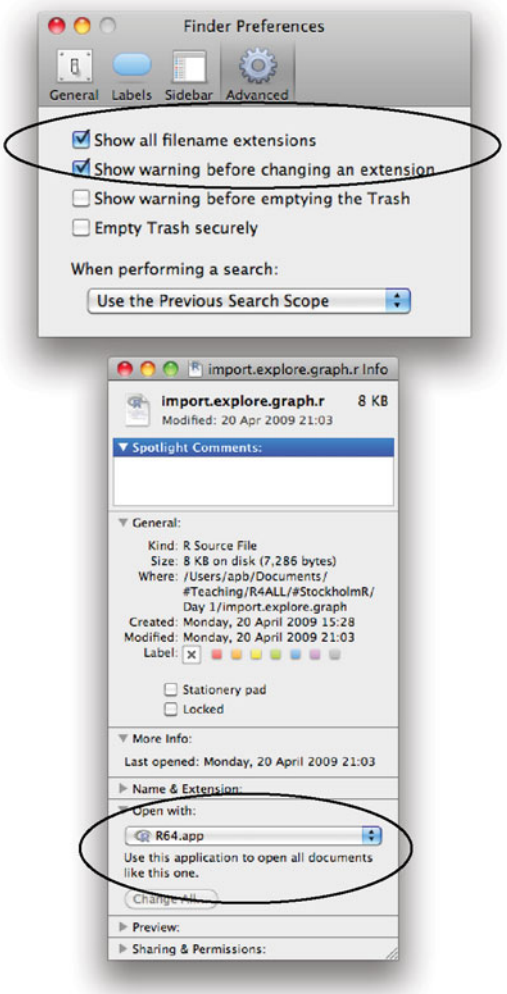
Box 2.4: File extensions and operating systems

Macintosh

To ensure that OSX shows you file extensions, you need to adjust the Finder preferences:

Finder -> Preferences -> Advanced

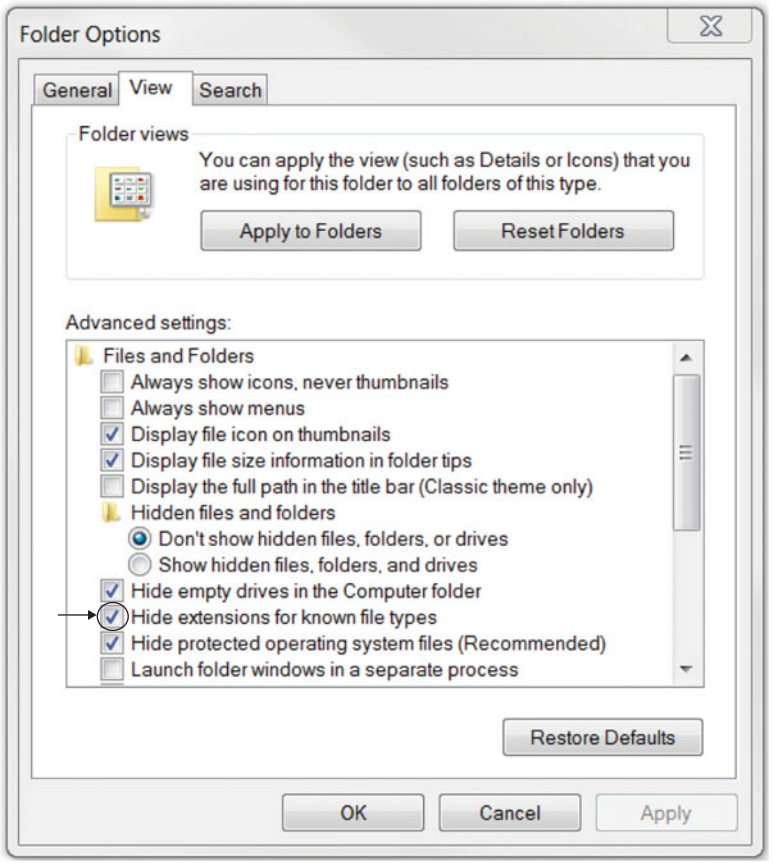
Tick "Show all filename extensions"



You may need to tell OSX to recognize the .R filename extension as an R application file. To do this, simply select an R script icon (1 click) and press cmd + i. This will open the info for this file. At the bottom of this box, you can select the program with which to always open these files. Click on the drop-down menu and select R. Just below this you can see the instruction to use this application for all documents like this one (i.e. with a ".R" file extension). Press the button to confirm.

Windows

To ensure that Windows presents file extensions, navigate in an Explorer window, via the Tools menu to Folder Options. Tools->Folder Options->View Tab. Ten tick boxes/circles down, you should see an option stating "Hide extensions for known file types". De-select this, if it is selected, and Windows should show all file extensions.



system what application to open a file in, if you double click the file. On a Mac and in Windows the default is for hidden file extensions (you don't see them). However, it can be quite useful to see them. How to change your settings, so you can see file extensions, is described in **Box 2.4**. If you have visible file extensions, you can easily see if your script file has the .r (or .R) extension. If it doesn't you should probably add it. If double clicking on your script file doesn't open it in R, then you need to add the .R file extension, or perhaps tell your operating system to open .R files in R. See **Box 2.4** for more details about this.

Great. Let's recap a few things. You've created a .csv file of your data. You've installed R, and opened it up. You've opened a **SCRIPT**, alongside a **CONSOLE**. You've added **annotation** at the top of the script using the # symbol. You've saved that file to your analysis folder for this project. That's brilliant. You are now ready to use R to do some work!

2.6 Starting to control R

But before we let you loose we would like to introduce a few more conceptual ideas to you before we work through a practical session.

R only does things you ask it to do. Asking R to do things requires using R **functions**. R uses **functions** to do all kinds of things and return information to you. There are four really helpful **functions** in R that you can use to accomplish four fundamental tasks at the start of an analysis/script: clear out R's brain, look inside R's brain, find out where in your computer R is looking, and tell R where to look in your computer.

To clear R's brain, we use two functions at once, **rm** and **ls. rm** stands for remove, and **ls** stands for list. We combine them in the following way to make R "clear" its brain.

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

It is best to read this from the inside out. **ls()** requests all of the objects in R's brain. **list=ls()** asks R to make a list of these objects. **rm()** asks R to remove all of these objects in the list. You will have noticed that you now know a function that gives a list of all of the objects in R's brain: **ls()**. This is a handy function.

Our discussion of the PATH above suggests that we need to be able to tell R where to "look" to find our data, for example. To find out where R is "currently" looking, you can use the function **getwd. getwd** stands for "get working directory". This is the PATH on your computer where R is currently expecting to get things from, if you ask it to. You will also be overjoyed to find out that you can tell R where to look by using the command **setwd**, or "set working directory". So, if you want R to find your data really easily, you can actually direct R to look at the raw data folder that you set up.

Box 2.5 shows a hypothetical script demonstrating a very useful way to start every script you make. It contains **annotation**, a request to "reset" R using **rm(list=ls())**, a request to find out where R is looking using **getwd()**, and a request to tell R where we actually want it to be looking for our data, using **setwd()**.

You will notice that after initial annotation in the script, the very first thing we suggest is the "clearing" of R's brain. We advocate making sure that R's brain is empty before you start a new analysis. Why? Well, imagine that you are analysing body size of your special species in two different experiments. And, not surprisingly, you've called it body size in both datasets. You finish one analysis, and then get started on the next. By clearing R's brain before you start the second analysis, you are ensuring that you don't use the wrong body size data for an analysis.

It is also a very good idea to save your script now, again, please. Just to be sure. There are only two things you need to keep safe (and keep them very safe): your raw data and your script!

2.7 Making R work for you—developing a workflow

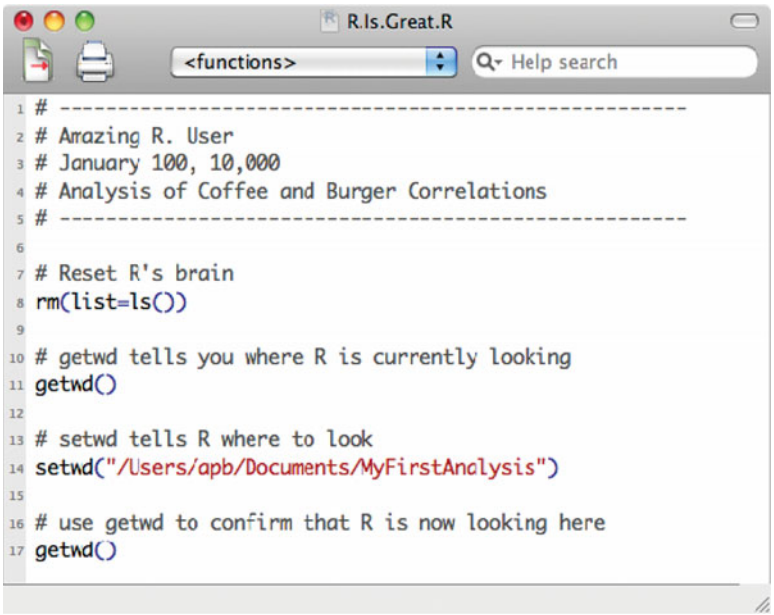
To make this section most effective, make sure you've created the script file as described in **Box 2.5** and above. And if you have not done so yet, save it in a new folder called "MyFirstAnalysis" (yes, with no spaces) on your Desktop.

Box 2.3 introduced how to make R actually read and execute instructions you've put into the script. Computers these days are quite smart. You

Box 2.5: Developing a script

The script below presents a structure and initial functions that we have found very useful. It starts with some annotation about the script. This is followed by the resetting command

rm(list=ls())



This is followed by a set of functions, getwd and setwd, that allow you to tell R where to look for your data for this analysis session.

When you make this script, REMEMBER that the PATH in setwd() MUST be specific to your platform (see **Box 2.1 for platform specific instructions**).

Obviously, as you get more proficient, the getwd() function can be omitted from the script.

know how to select text and lines in a word-processing program, either with a mouse or the arrow keys and various other keyboard keys. If you select text in the **SCRIPT**, you can make R read AND execute this by typing **CTRL + R** (Windows) and **cmd + Return** (Macintosh).

Now, look at the **CONSOLE** window. You'll see the following:

>

with a cursor flashing after it. This is known as the PROMPT.

Now, shift focus to your **SCRIPT**. Try selecting (with the mouse) your first set of comments (#s) in your **SCRIPT**. Selecting them should highlight them. Let go of the mouse and press the appropriate keystrokes for your operating system (**CTRL + R** (Windows) and **cmd + Return** (Macintosh)). Now, look at the **CONSOLE**. You'll see that R has sent the annotation from the script to the **CONSOLE**, returned the text to you and effectively ignored the message. We know that it has done this because R has not returned any errors, and by showing the **PROMPT**, it is ready again.

Now select the **rm(list=ls())** and **getwd()** function/lines in the SCRIPT, press the appropriate keystrokes, and look at the CONSOLE again. Magic!

You've asked R to clear its head AND tell you where it is looking, and it's telling you where, using the syntax (**PATH**) we introduced previously!

Now, select the **setwd()** line. If you select this and execute it, R will change where R is looking. How can you make sure it has changed? You guessed it, select the **getwd()** line again and confirm.

At this point, you should be getting more comfortable with R. There are (for now) two windows to work with, and your script is the most important document of your life! Annotation is vital. You are now ready to work with some real data. We suggest that you now download our datasets folder from the www.r4all.org . Don't forget to move it from your downloads location to a sensible place, such as a folder in your "Learn R" folder.

2.8 And finally . . .

Save your script and quit R. You should find that R asks if you'd like to save the workspace. Because you saved your script, and because your raw data

is safe, there is no need to save the workspace. So we never save it. Well, there are always exceptions, like if you had some really involved analyses that took R much more than a few minutes to complete. Then you could save the workspace; but better to do so using the save() function than waiting for R to ask you as you're packing up for the day and quitting R. Just to be sure, the two most important things in your R-life are your raw data and your script. Keep those safe and you're sorted.

Import, Explore, Graph II

Importing and Exploring

3.1 Getting your data into R

By now you must be quite excited that simply by typing some instructions, you can get R to do stuff. But what you really want to do is get your data into R's brain. It is empty—R's brain, not yours. To do this, we need to introduce two things. The first is a function that R uses to read "comma separated values" files (like the one we described earlier), and the second is the idea of an object in R's brain.

At this point, we would like you to open the script you started AND copy the compensation.csv file found in our online datasets into the MyFirstAnalysis folder (make this if you have to).

Now, not surprisingly (are you tired of us saying that yet?), the function that reads ".csv" files into R is **read.csv()**. So, if we want R to read a file called "compensation.csv" file from the MyFirstAnalysis folder, we might add the following line to our script.

```
read.csv("compensation.csv")
```

Note that we DO NOT need to specify the whole PATH to "compensation.csv". Why? Because we used **setwd()** to tell R where to look (see **Box 2.5**).

R is already looking inside the MyFirstAnalysis folder, so we need only specify the file to get.

This is a good start, but not totally effective, because reading a file is different to committing it to memory. If we were to execute this line of code—go for it—R will return the dataset to you printed in the R Console. We actually want to be able to use the dataset for further analysis though. To make R "remember" or hold onto the compensation.csv data, we need to assign the data to an object. That is really just a fancy way of saying that we need to give the dataset a name, inside R, for R to use. By giving it a name in R, we commit the data to R's memory.

The conventions for this process involves a **name**, the <- symbols and the **read.csv()** function applied to a dataset in a location (i.e. the **PATH**). For example, we might name the dataset "comp.dat" and assign the data to it as follows.

```
comp.dat <- read.csv("compensation.csv")
```

Let's just interpret this. comp.dat is simply a set of letters that make a word that we made up. The <- symbol assigns the stuff on the right to the name on the left, creating an object with value. What is the stuff to the right? **read.csv**() is the **function** that takes a **PATH** as its **argument** and reads the data associated with it, specifically comma separated data. We assign the values returned by **read.csv**() to the object comp.dat. **Box 3.1** provides an in depth look at functions, their anatomy, and the process of assigning a value returned by a function—in this case, a .csv file returned by **read.csv**()—to an object, in this case, the word comp.dat.

Once you have reviewed this, type the code into your **SCRIPT** and save the script. Then, select this line and execute it. Now, look with horror at the **CONSOLE**. It looks as if nothing has happened! All you see is that line from your **SCRIPT** and below that the prompt:

>

But nothing could be further from the truth. Remember how to look in R's brain? In the CONSOLE, try typing:

```
ls()
```

Voila! There it is: comp.dat, in all of its glory.

Box 3.1: Function conventions and the anatomy of object assignment

The second line of your script will typically be a request to import data and store that data as an object, with a name, in R's brain. It will look like this:

WINDOWS

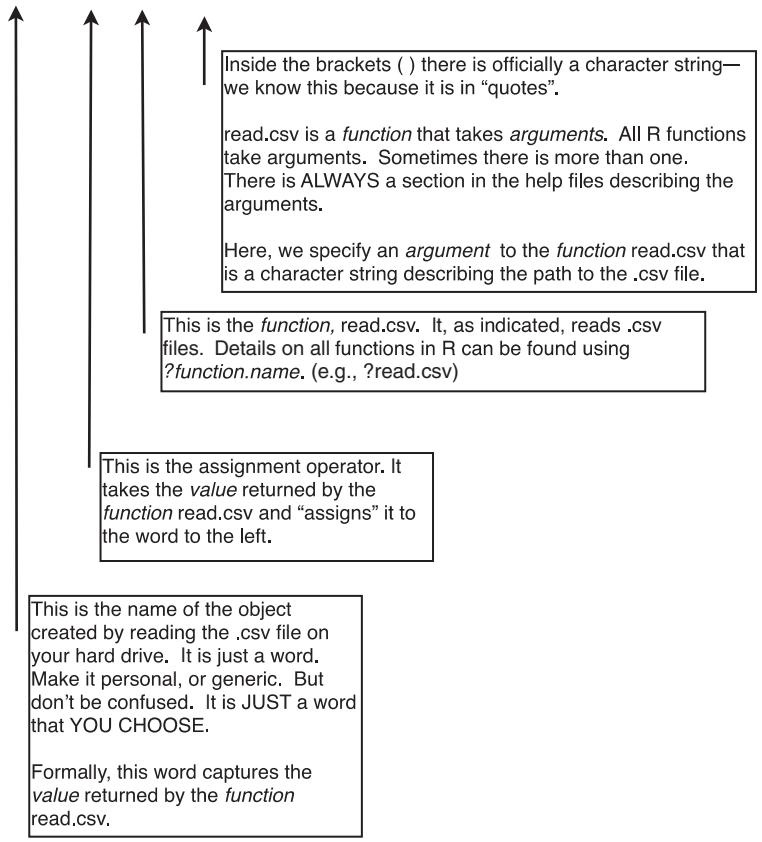
```
mydata <- read.csv("C:\\path\\to\\the\\file\\file.csv")
```

OR

```
mydata <- read.csv("C:/path/to/the/file/file.csv")
```

MACINTOSH/LINUX/UNIX

```
mydata <- read.csv("~/path/to/the/file/file.csv")
```



3.2 Checking that your data is your data

A very sensible next step is to make sure the data you just asked R to commit to its brain is actually the data you wanted. Never trust anyone, let alone yourself. **Box 3.2** shows our expanding script and introduces a number of different functions that allow you to see various pieces of your dataset (e.g. the first six rows), or various attributes of columns and rows—for example how many rows or columns are factors/categorical or numeric/continuous variables. A few very sensible functions to commit to your memory are:

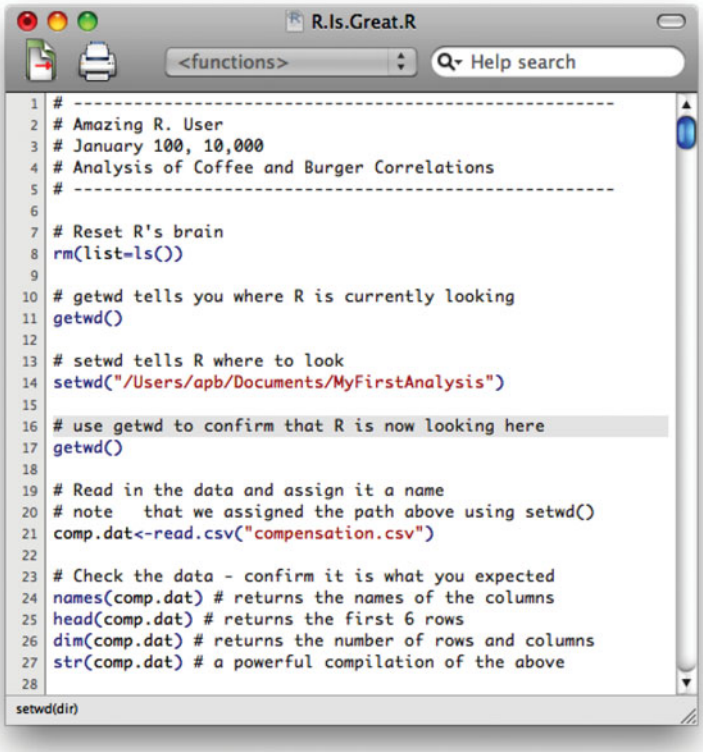
```
names()
head()
dim()
str()
```

By placing the name of your data object—for example comp.dat—inside the brackets () of the functions above, R will execute the function on this object. Not surprisingly, **names**(comp.dat) tells you the names you assigned each column (i.e. your variable names, the column names in your .csv files, which you typed into excel or wherever). **head**(comp.dat) returns the first six rows of the dataset (guess what **tail**(comp.dat) does?). **dim**(comp.dat) tells you the numbers of rows and columns—the **dimension**—of the dataset. Finally, **str**(comp.dat) returns the **structure** of the dataset.

str() is a very useful function, returning several pieces of information about objects. For a **data frame** (spreadsheet like dataset), it returns in its first line a statement saying that you have a data frame type of object and the number of observations (rows) and variables (columns). Then there is a line for each of your variables, giving the variable name, variable type (numeric, factors, or integers, for example), and the first few values of that variable. Thus, **str**() it is an outstanding way to ensure that the data you have imported is what you intended to import, and to remind yourself about features of your data.

Box 3.2: The expanding script

The script now contains a function to read in the data from your hard drive and a number of functions to verify that the data you've imported into R is what you expected. Note the extensive use of annotation.



Of course, you could also just type the name of the dataset—comp.dat. This can be pretty fun/infuriating if you forget that there are 10,000 rows of data.…

Tip: If you find that the data you imported has more columns and or rows than it should, this could be because Excel has saved some extra columns and or rows (goodness knows why!). To see if this is the problem, open your data file (.csv file) using Notepad (Windows) or TextEdit (Mac). If you see any extra commas at the end of rows, or lines of commas at the bottom of your data, this is the problem. To fix this, open your .csv file in Excel, select only the data you want, copy this, and paste this into a new Excel file. Save this new file as before (and delete the old one). It should not have the extra rows/columns.

3.3 Summarizing your data—quick version

summary() is a R function that works on many types of objects. When you specify a data frame as an argument to **summary**(), it will return the median, mean, inter-quartile range, minimum, maximum for all numeric columns (variables), and the levels and sample size for each level of all categorical columns (variables).

3.4 How to isolate, find, and grab parts of your data—I

The two classes of data containers that we use most frequently are **data frames** and **matrices**. A data frame can be thought of as an R interpretation of a spreadsheet. It is unique in that it can accommodate continuous and categorical variables (and other types). A **matrix** is slightly different, and while a data frame and a matrix might have similar **dimensions**, a matrix can only accommodate a single type of data, for example numbers (numeric data).

There will be numerous times in your career when you will want to work with parts of your dataset. That is, you will want to **subset** your data, perhaps by selecting specific columns (variables) to use in a **regression** model, or

particular rows that, perhaps, omit some treatment or values. Both data frames and matrices have implicit row and column numbers that allow you to subset parts of your data. These are associated with a technical term called ***indexing***. We can use this indexing to grab any portion of a data frame or matrix.

There are numerous ways to subset data in R. We start with a method applicable to a data frame and a matrix. The important symbols on your keyboard for grabbing parts of a data frame or matrix are:

```
[] (square brackets)
```

and

```
, (the comma).
```

An effective way to understand the use of these symbols is by example, and if the compensation.csv dataset is imported into R as comp.dat on your computer, you can simply type the following examples into the console, or put them into a script and annotate them for future knowledge.

The square brackets follow the name of your data frame or matrix. Inside the brackets you use numbers or words, and the comma, to tell R what to do. For example:

```
comp.dat[,1]
```

returns the first column of comp.dat, because the comma is to the left of the 1.

```
comp.dat[1,]
```

returns the first row, because the comma is to the right of the 1. It might help to commit to memory *rows-then-columns-always*.

Of course, you can ask for more than one row or column as

```
comp.dat[,1:3]
```

which will return the first three columns while

```
comp.dat[1:3,]
```

will return you the first three rows. And of course, you can ask for both:

```
comp.dat[1:3,1:3]
```

will return the first three rows of the first three columns.

As mentioned before, you should give your variables sensible names when you enter your raw data. You can then use these names to refer to columns, instead of having to know which variable is in which column number. R allows us to replace column numbers with an actual column name (word), in quotes. For example:

```
comp.dat[,"Fruit"]
```

Compare this with

```
comp.dat[,1]
```

to prove to yourself what is going on.

Finally, we can isolate a column in a data frame using the following syntax employing the symbol $:

```
comp.dat$Grazing
```

This is interpreted as "the Grazing column in the comp.dat data frame"; this $ method works for data frames but not matrices.

So there are three ways to refer to columns. And remember, rows-then-columns-always.

3.5 How to isolate, find, and grab parts of your data—II

To finish this "subsetting" section, we will also introduce to you the function **subset**(). It is powerful, logical, and allows you to subset your data by specific columns and values in those columns. This is a very common practice, as you'll recognize in the example below. The syntax is:

```
subset(data.frame.name, column.name == value)
```

where "column.name" is the name of the column (variable) and "value" is the value in the rows that you'd like to keep. This value would be either a value/number if the column contains continuous data, or a factor level, in quotes, if the column contains categorical data. And yes, it is double equal "==". You can, of course, also use other logical operators within the subset. In R, we use "&" for AND, "|" (the pipe) for OR, "!" for not, and "!=" for not equal.

The comp.dat data has a column called "Fruit" (continuous measure of Fruit Production), a column called "Root" (continuous measure of initial root diameter), and a column called "Grazing" (a factor, categorical, variable with two levels—Grazed and Ungrazed). Imagine you want only the data on the Grazed treatment:

```
subset(comp.dat, Grazing == "Grazed")
```

will return a subset of the data (all three columns) with rows corresponding only to "Grazed". Try changing the column name and the value to see how it works.

3.6 Aggregation and how to use a help file

At this point you should be feeling pretty confident. You can import and explore the structure of your data and you can access various part of it using [ ]'s or the function subset(). What next? In this section, we introduce to you functions that help you generate custom summaries of your data—for example the means of some measure of fitness at all levels of your experiment.

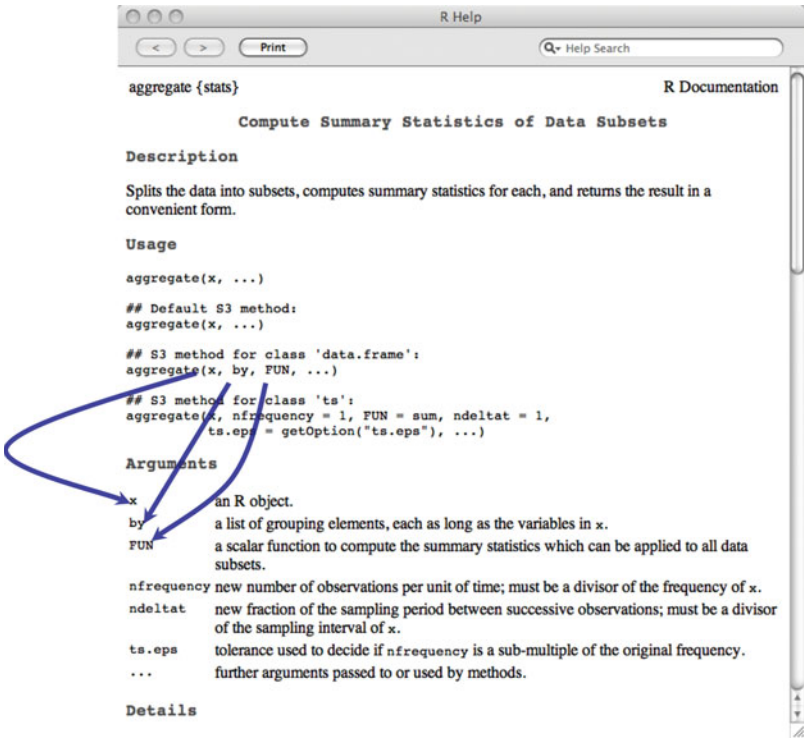
The two key functions for this are **aggregate** and **tapply** (though the only difference between them is what they give you back). We also introduce the functions for **mean** and **sd** (standard deviation) and introduce to you the anatomy of an R help file in detail.

aggregate is a function that takes, as arguments, one (or more) column of the data frame that you want to aggregate, a **list** of other columns to structure the aggregation, and a mathematical function that generates aggregated values. If you type ?aggregate into the console, the help file will be generated by your operating system. **Box 3.3** provides insight into how to read the **aggregate** help file. Take the time to go through this. Our interpretation of what aggregate does, as you will see, is very much a translation!

Box 3.3: Using a help file—aggregate

Help files are very important and very useful in R. They have a formal structure that never varies. They always start with the name of the function and the package to which it belongs and a short description. They always end with some examples. In the middle, they always contain sections called Usage, Arguments, Details, Value, Authors, References, and See Also.

Here is the top of the aggregate help file, accessed by typing ?aggregate in the console.



We can see that aggregate is a function in the stats package of R (upper left). We can use it to "Compute Summary Statistics of Data Subsets". The description is followed by Usage and Arguments. Note that there is specific usage for a data.frame. There are three main *arguments* for the method. x is an R object, by is the grouping variable, and FUN is the function being used to aggregate the object, x, by the levels of the grouping variable.

If you scroll down to the bottom of the help file, you will see some additional sections. There is information on the Values returned by aggregate for different types of data, some related functions (e.g. tapply), some information on the Authors and information on References that influenced the development of the function. Importantly, there are also Examples. Note the prolific use of annotation in the examples. Note as well that you can copy any of the examples and paste them into your console and they will run. This allows you to dissect the use and anatomy of this function.

Value

For the time series method, a time series of class `"ts"` or class `c("mts", "ts")`.

For the data frame method, a data frame with columns corresponding to the grouping variables in `by` followed by aggregated columns from `x`. If the `by` has names, the non-empty times are used to label the columns in the results, with unnamed grouping variables being named `Group.i` for `by[[i]]`.

Note: prior to R 2.6.0 the grouping variables were reported as factors with levels in alphabetical order in the current locale. Now the variable in the result is found by subsetting the original variable.

Author(s)

Kurt Hornik, with contributions by Arni Magnusson.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

apply, lapply, tapply.

Examples

```
## Compute the averages for the variables in 'state.x77', grouped
## according to the region (Northeast, South, North Central, West) that
## each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)

## Compute the averages according to region and the occurrence of more
## than 130 days of frost.
aggregate(state.x77,
          list(Region = state.region,
               Cold = state.x77[,"Frost"] > 130),
          mean)
## (Note that no state in 'South' is THAT cold.)
```

The example below will use the function **aggregate** to generate the mean, standard deviation and sample size of Fruit Production in our comp.dat data. comp.dat has three columns: a column of Fruit Production, Initial Root Diameter, and a categorical variable called Grazing. Here we use **aggregate** to get the mean Fruit Production, in each Grazing level:

```
aggregate(comp.dat$Fruit, list(comp.dat$Grazing), mean)
```

This says to R, FIRST get the Fruit column from comp.dat, THEN divide it up by the Grazing treatment levels (Grazed and Un-Grazed) and FINALLY calculate the mean Fruit Production in grazing treatment. Very handy. The function aggregate returns a new data frame with the first column corresponding to the levels of Grazing and a second column corresponding to mean Fruit production.

```
   Group.1       x
1   Grazed 67.9405
2 Ungrazed 50.8805
```

Of course, if you want to use the values returned by **aggregate**, you must use the <- symbol and assign the result of aggregate to a new object. Perhaps you'll call it mean.fruit.

tapply (definition: apply a function over a ragged array; see ?tapply) returns the same values that aggregate does, using a very similar structure to the arguments in aggregate but, instead of returning a data frame, it returns a matrix. We will see in the next section on graphing why this can be so very handy. Use **tapply** as follows:

```
tapply(comp.dat$Fruit, list(comp.dat$Grazing), mean)
```

Again, the arguments are as we used for aggregate, with the following caveats: aggregate's first argument, which is called "x" in the help file, is replaced by, yes, the big "X" in tapply; aggregate's second argument, labelled as "by" in the help file, is replaced with "INDEX" in tapply. Go on, compare the help files. The interpretation is the same: FIRST get the Fruit column from comp.dat, THEN divide it up by the Grazing treat-

ments (Grazed and Un-Grazed), and FINALLY calculate the mean Fruit Production in each treatment. It produces the following:

```
 Grazed Ungrazed
67.9405  50.8805
```

Of course, if you want to use the values returned by **tapply**, you must use the <- symbol and assign the result of **tapply** to an object.

tapply and **aggregate** are wonderful functions. The by/INDEX argument in each function is a list() and can accommodate as many categorical data columns as you want, creating fast and efficient summaries of your data over many dimensions. The third argument can take many default R functions, including **mean**, **sd** (standard deviation), **median**, and even functions that you make yourself. Examples of where you might be able to use this method include calculating:

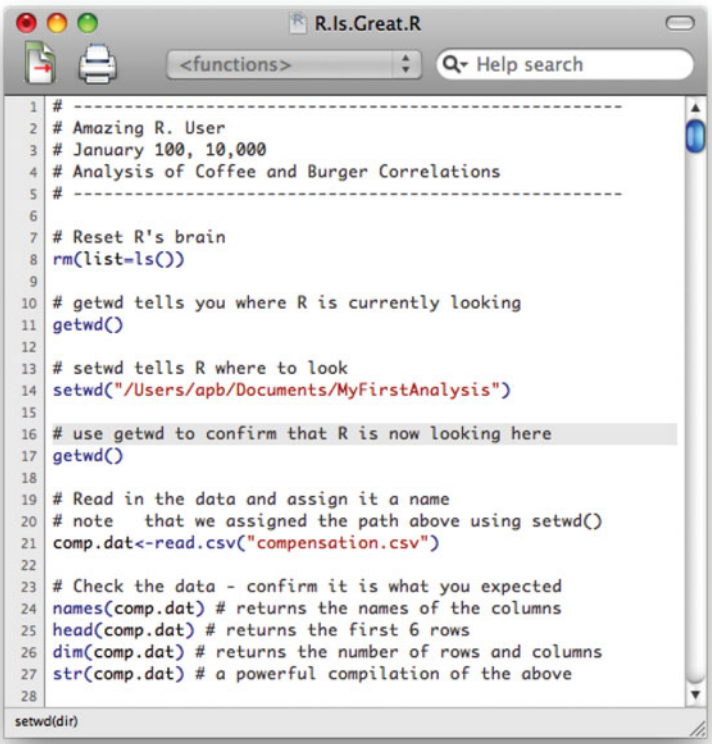
1. the mean phenotype score among genotypes in ponds
2. the standard deviation of biomass in nitrogen treatments
3. the mean species richness in quadrats.

3.7 What your first script might look like (what you should now know)

One of the hardest parts of getting started using R is getting your data from your workbook/spreadsheet into R. You've done it. Practice this. Use the example script (**Box 3.4**) as a template for your own analyses. It is a good idea to try and import one of your own datasets—make a script to import and explore your data. Use aggregate or tapply to calculate some means. If you are not convinced that you've learned something, the table below (Table 3.1) contains a list of the functions and operators we have introduced to you. Take some time and fill in the definitions on the right. Commit to memory what each of them do and which ones are connected. Now, get ready for making some of the best, publication-quality figures you've ever seen.

Box 3.4: A template script for starting any analysis

Use this script as a template to start any analysis. Note that you don't need to change much in the script to make it work for another dataset. Change the PATH, and specify the dataset's name and column names. That's it.




```
# ----------------------------------------------------
# Amazing R. User
# January 100, 10,000
# Analysis of Coffee and Burger Correlations
# ----------------------------------------------------

# Reset R's brain
rm(list=ls())

# getwd tells you where R is currently looking
getwd()

# setwd tells R where to look
setwd("/Users/apb/Documents/MyFirstAnalysis")

# use getwd to confirm that R is now looking here
getwd()

# Read in the data and assign it a name
# note   that we assigned the path above using setwd()
comp.dat<-read.csv("compensation.csv")

# Check the data - confirm it is what you expected
names(comp.dat) # returns the names of the columns
head(comp.dat) # returns the first 6 rows
dim(comp.dat) # returns the number of rows and columns
str(comp.dat) # a powerful compilation of the above

```

Table 3.1 The R commands that have so far appeared. Fill in your own description of what they do.

Function/ operator	Definition/usage
#	
rm	
ls	
setwd	
getwd	
read.csv	
names	
head	
dim	
str	
summary	
[ ]	
subset	
$	
"==", &, |	
aggregate	
tapply	
mean	
sd	
median	

This page intentionally left blank

Import, Explore, Graph III

Graphs

You've made a .csv file. You've made a script. You've imported data into R. You've made sure the data is the right data. You've calculated some basic statistics. You are beginning to see the patterns in your data that you expected. Most importantly, from an R-data analysis/data management perspective, you have a permanent, repeatable, annotated, shareable, cross-platform executable record of what you are doing–the script.

4.1 The first step in data analysis—making a picture

We advocate a very fundamental rule for data analysis. NEVER start an analysis with statistical analysis. ALWAYS start an analysis with a picture. Why? If you have done a replicated experiment, conducted a well organized, stratified sampling program, or generated data using a model, it is highly likely that some theoretical relationship underpinned your research effort. You have in your head an EXPECTED pattern in your data.

Plot your data first. Make the picture that should tell you the answer—make the axes correspond to the theory. If you can do this, AND see the pattern you EXPECTED, you are in great shape. You will possibly know the answer!

There are two major figure types we introduce. The first is a bar graph ± error bars; the second is a scatterplot. These are two of the most common types of graphs people produce in ecology and evolutionary biology and understanding them sets the stage for any other type of figure you might want to make.

4.2 Making a picture—bar graphs

Bar graphs require mean values of some variable to generate the heights of the bars and then some measure of variability to add error bars. Many researchers love them, many despise them. Regardless, it is instructive to know how to make one in R, and many of the methods used to make one are transferable.

Recall from the previous chapter that **tapply** produces a **matrix** of aggregated data. **tapply** can accommodate a range of functions, including the mean, standard deviation, and so on. What you may not know is that **tapply** is the best friend of **barplot**, the function that makes a bar graph. **tapply** produces a matrix. **barplot** loves a matrix. Here we show you one way to make an informative bar graph.

First, start a new script, or simply retrieve the previous script. Annotate it with some relevant information, like your name, and a description of the script (# How to Make a Barplot in R).

Next, download the dataset "compensation.csv" from the website (if you haven't already), create a "barplot in R" folder, and put the .csv file into that folder. Save your script to the very same folder, with the .R extension. Perhaps call it "barplot.example.R". (If you are re-using your script from above, that's fine—you will quickly learn to reuse scripts with copy and paste).

Now, add to your script a line that reads the dataset into R using **read.csv** and gives the dataset a name (See **Box 3.4** for review). Here we call it "mydata". You don't have to. You can call it "tiggy" or "warblefly", or "comp.dat" as we did above. Just annotate it, because you might forget tomorrow. For example:

```
# set the working directory to the folder with all of the stuff in it.
# note ~ for home directory in Mac/Linux/Unix
# /'s show path to "barplot in R" folder
setwd("~/R4All_Examples/barplot in R/")

# Read in the data using read.csv function and assign it to the
# object called mydata
mydata <- read.csv("compensation.csv")

# make sure the data are what I wanted
head(mydata)
str(mydata)
```

If you are following our recipe for success, your script will have #s throughout already. Run the script by selecting all of the text in the script (CTRL+A [Windows] or cmd+A [Macintosh] or use the mouse) and then use the execute keystrokes (CTRL+R [Windows] or cmd+return [Macintosh]) to make R do the work. You should see in the CONSOLE the first six lines of the dataset and then the summary provided by str(). (Get used to pressing cmd+A then cmd+return [Mac] or CTRL+A then CTRL+R [Windows] without letting go of cmd or CTRL. This is a quick way to send all your code to R, and so long as your code isn't too long or tough for R to run, it's a really quick way of running all your code.)

Before we continue, let us reintroduce the data. The compensation dataset is about Fruit Production and Initial Root Diameter in Grazed or Un-Grazed conditions. Grazing reduces above-ground biomass. How is fruit production affected by this reduction? And how is initial root diameter involved?

At this point, let's think hard about what we might want to plot. One idea is to explore mean ± standard deviation of fruit production for grazed and un-grazed conditions. We can make a barplot with standard deviation error bars, and to be fancy, let's place as text inside each bar the sample size used to calculate the mean and standard deviation.

First, we need to calculate the means, the standard deviations, and the sample sizes. We suggest using **tapply**. Why? Because **barplot** LOVES **tapply**.

```
# Calculate mean of each group using tapply, which returns a matrix
mean.fruit <- tapply(mydata$Fruit, list(mydata$Grazing), mean)
# Calculate standard deviation of each group
# using tapply, which returns a matrix–it will be the same
# dimension as mean.fruit
sd.fruit <- tapply(mydata$Fruit, list(mydata$Grazing), sd)
# Calculate sample size in each group
n.fruit <- tapply(mydata$Fruit, list(mydata$Grazing), length)
```

These three lines of code and their annotation should do a number of things. They should help refresh your memory about the function **tapply** and its three arguments—if not, look at the help file again and read a few pages back. Make sure you understand this function. They should also refresh your memory about assigning the value returned by functions to objects. Here, we have assigned the value returned by **tapply** to three different objects, depending on the argument we used to summarize/aggregate our data (in this case another function): **tapply** with **mean** is assigned to mean.fruit, **tapply** with **sd** (standard deviation) is assigned to sd.fruit, and **tapply** with length is assigned to n.fruit.

How do you know that this worked? We can look at any object simply by typing the name in the CONSOLE OR selecting the name only in the script and sending a request to show it to R. This is what you should see.

```
> mean.fruit
Grazed  Ungrazed
67.9405 50.8805
> sd.fruit
Grazed   Ungrazed
24.96081 21.75897
> n.fruit
Grazed Ungrazed
20     20
```

Before we move on, we'd like to convince you that **tapply** has produced a matrix for each of these objects. The dimension of the matrix should be 1 row x 2 columns. The column names are Grazed and Ungrazed (the "levels" of the grazing column) and the row names (1 row) are NULL. Try

proving this with the **dim**() function. Also you can ask R directly if an object is a matrix, by using is.matrix(mean.fruit).

Now, the exciting part. To make a **barplot** of the means, you simply use the mean.fruit object as the argument to **barplot**.

Add this to your script and execute it and look at the GRAPHICS window (Fig. 4.1):

```
# Make a barplot using the means returned from tapply
barplot(mean.fruit)
```

Fantastic. This is easy, right? However, we had a plan. The plan was to make a **barplot** with standard deviation error bars, and place the sample size used to calculate the mean for each bar as text inside each bar. We may also want to ensure that the axes have appropriate ranges (minima and maxima) and that the axes labels are informative.

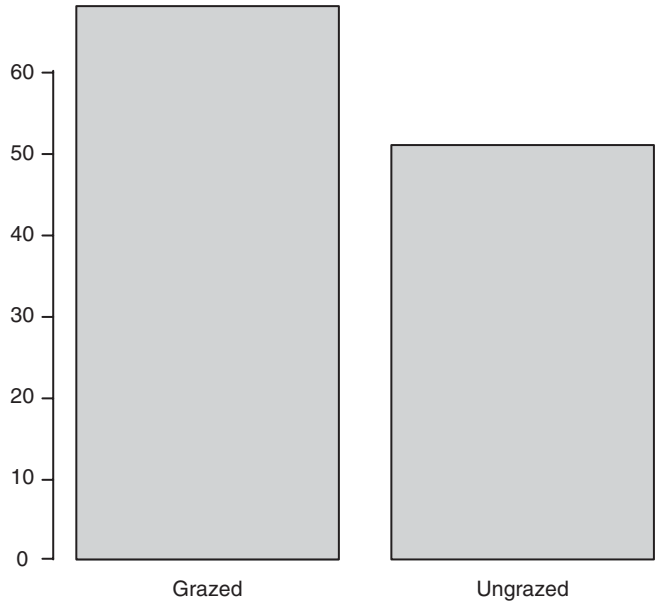


Figure 4.1 Step 1 in making a barplot uses the barplot() function and its main argument, a vector or matrix of bar "heights". In this case, it is the object we have named mean.fruit. Notice that x- and y-axis labels are absent and the Grazed bar extends above the y-axis range.

How to we change an axis' range? How do we add axis labels? How do we add error bars? At this point, we could just tell you what to do (we will). But we implore you to look at the help page for **barplot**. Remember how to find help? Yep, its **?barplot**.

Look at the help file. Examine the arguments. Notice that the *height* argument entry says that it can work with a vector OR a matrix—this is why it is so easy to work with the matrix that **tapply()** returns. Notice in the default "S3 method" description that there are *xlim* and *ylim* arguments, and *xlab* and *ylab* arguments. Scroll down a bit and look at the definitions for these. Get in the habit, particularly at this early stage, of reading the help files. It will be well worth it.

Right. Let's make the figure pretty, then add error bars, and then add our text.

4.2.1 PIMP MY BARPLOT

Let's add informative x- and y-axis labels. We'll use the *xlab* and *ylab* arguments in the barplot function. Simply add the relevant lines to the existing text in your script. No need to re-type everything (Fig. 4.2).

```
barplot(mean.fruit,
   xlab = "Treatment",
   ylab = "Fruit Production")
```

A few things to point out. First, we didn't actually rewrite all of this. We inserted the *xlab* and *ylab* arguments to the script, and then saved it. Second, we used carriage returns and tabs to make the instructions look pretty. You don't have to do this indentation, but we strongly recommend it. As your scripts get bigger and more involved, making life easy for yourself is important. And taking your time to indent and organize your script saves oodles of time and makes it easy for you to read your script and easy for others to read it too.

Now, you're probably thinking that we need to make the y-axis a bit taller. We can use the *ylim* argument in **barplot** to extend it (Fig. 4.3).

```
barplot(mean.fruit,
  xlab = "Treatment",
  ylab = "Fruit Production",
  ylim = c(0,100))
```

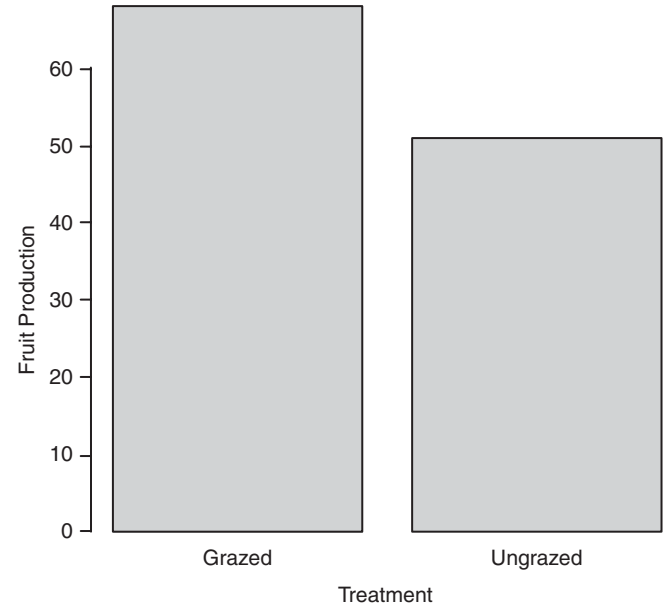


Figure 4.2 Here we add x- and y-axis labels using the *xlab* and *ylab* arguments to barplot.

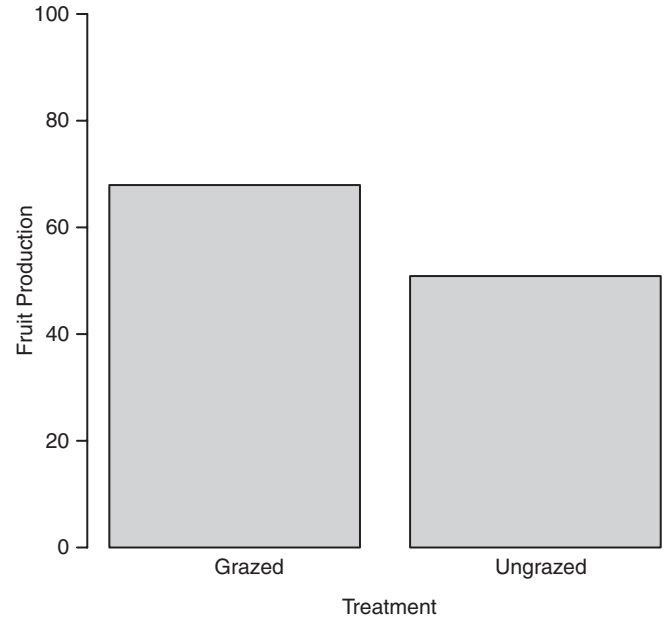


Figure 4.3 Now we extend the y-axis to an upper limit of 100 using the *ylim* argument to barplot.

Again, notice the indentation. And a new letter: **c.**

c is a function in R. It stands for *combine* values into a vector. It essentially makes a collection of things. In this case, the *ylim* argument requires a minimum and maximum for the axis, and we wrap them inside a **c**().

Now, let's add error bars. The **barplot** function does not have an automatic set of tools to do this. There are other packages that do offer this facility. They include Gregory Warnes **gplots**, Frank Harrell's excellent **Hmisc** package, Jim Lemmon *et al.*'s **plotrix** package, and Hadley Wickham's **ggplot2** package. However, you don't need them, and we will ignore them for the moment to demonstrate the logic of error bars and the amazing tools to customize R graphics.

All you need to do is to think of an error bar as a line with stuff at the end of it. What's the first thing that comes to your mind with this description? An arrow, perhaps? Yes, the function we are going to use to put error bars onto a **barplot** is **arrows**. There are other ways. But they are definitely not as fun.

Here is how it works. The arrow (error bar) needs to go from the top of the error bar down to the bottom. The top of the error bar will be at the mean plus the error, and the bottom at the mean minus the error. So these are the y-coordinates of the error bar arrow: mean + error and mean − error. What are the x-coordinates of the error bar? Look at the Fig. 4.3 above to help with a general answer to this question. The x-coordinates of any error bar are the location of the middle of the bars on the x-axis. So all we need to do is get the mid points of the treatments (bars) on the x-axis.

Here is how to do it. We assign the value returned by **barplot** to an object called mids. This is simply convention, established by Ben Bolker (r-help by Ben Bolker on April 16, 2001).

```
mids <- barplot(mean.fruit,
  xlab = "Treatment",
  ylab = "Fruit Production",
  ylim = c(0,100))
```

Take a moment to explore the values returned by mids (type it in the console after running it). You'll notice that the object actually contains the x-axis midpoints of the bars—the location of the bars on the x-axis. Now, check out the help file for **arrows**. This function requires, as arguments, x0, y0, x1, and y1. x0 and y0 are the x–y coordinates of the start of the arrow and x1 and y1 are the x–y coordinates of the endpoints of the arrow. The keen practitioner will recognize that x0 and x1 are the same for an error bar—the x-coordinates for a vertical line are the same. And they are the midpoints of the bar. Thus, we can use mids as x0 and x1.

But what do we use for y0 and y1? As mentioned before, let's start our error bars below the mean and finish above the mean. The point below the mean is the mean | the standard deviation: mean.fruit | sd.fruit. The point above the mean is the mean + the standard deviation: mean.fruit + sd.fruit.

Try typing these (e.g. mean.fruit | sd.fruit) into the console—notice that R performs the calculations for both columns simultaneously and automatically.

In this example, we have two treatments. The matrix of means has two columns and one row, and the matrix of standard deviations is the same dimension. Subtracting the standard deviations from the means results in a matrix of the same dimension and returns two values—the y0 and y1 values for each bar.

```
# Make a barplot using the means returned from tapply
# No need to re-type this.... it's in the script already
mids <- barplot(mean.fruit,
    xlab = "Treatment",
    ylab = "Fruit Production",
    ylim = c(0,100))

# Use Arrows to put error bars on the plot
arrows(mids, mean.fruit - sd.fruit, mids, mean.fruit + sd.fruit,
    code = 3, angle = 90, length = 0.1)
```

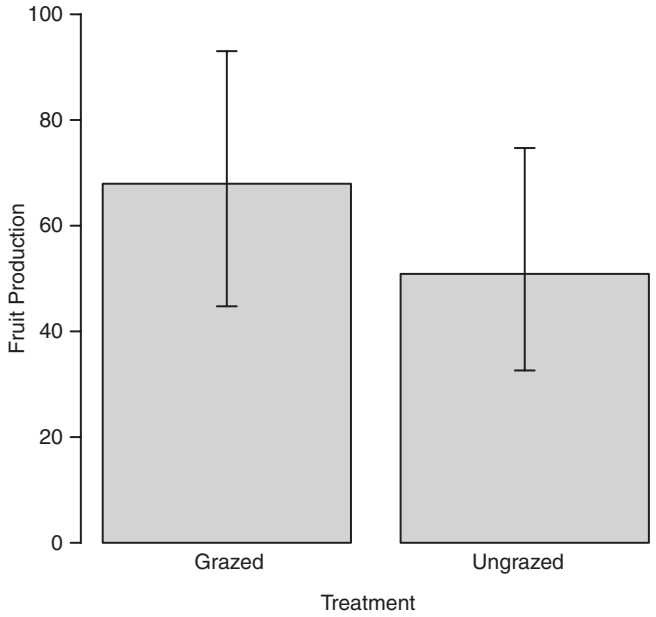


Figure 4.4 Here we have added error bars representing the standard deviation of fruit production. This involves a new function, arrows(), to make "flat ended" arrows that extend from the mean + sd to the mean – sd. See text for details.

There are three *arguments* to explain in **arrows**. *code* = 3 tells **arrows** to draw "arrows" at both ends of the line. *angle* = 90 says make the arrow heads 90 degrees—flat. *length* = 0.1 makes them short (0.1 mm) (Fig. 4.4).

Our final objective was to put some text in the bars to indicate the sample size. Again, not surprisingly, the function to add text to a graph is **text**. Like **arrows**, it takes as an argument a *location* (x, y) and a *text string*. As with arrows, it is important to recognize that an easy location to place the text is centred in the bars—the mids. Thus we can update our script as follows:

```
# Make a barplot using the means returned from tapply
# No need to re-type this.... it's in the script already
mids <- barplot(mean.fruit,
   xlab = "Treatment",
   ylab = "Fruit Production",
   ylim = c(0,100))
```

```
# Use Arrows to put error bars on the plot
arrows(mids, mean.fruit - sd.fruit, mids, mean.fruit + sd.fruit,
  code = 3, angle = 90, length = 0.1)

# Add text at the midpoints (x) and at height 2 on the y-axis
# paste the words n=followed by the n.fruit values
text(mids, 2, paste("n=", n.fruit)
```

Here is what you should see (Fig. 4.5).

The data files for this chapter offer a higher dimensional dataset—growth.csv—where weight gain in cows was measured under a factorial design of four feeding supplements and three different grains. You should produce a **barplot** for this now. Use the help file! Fig. 4.6 is what you are aiming for (script code is available online). Tips for accomplishing this: start with the code you had for your previous barplot and modify this line by line, making sure each modification does exactly what you expect and desire.

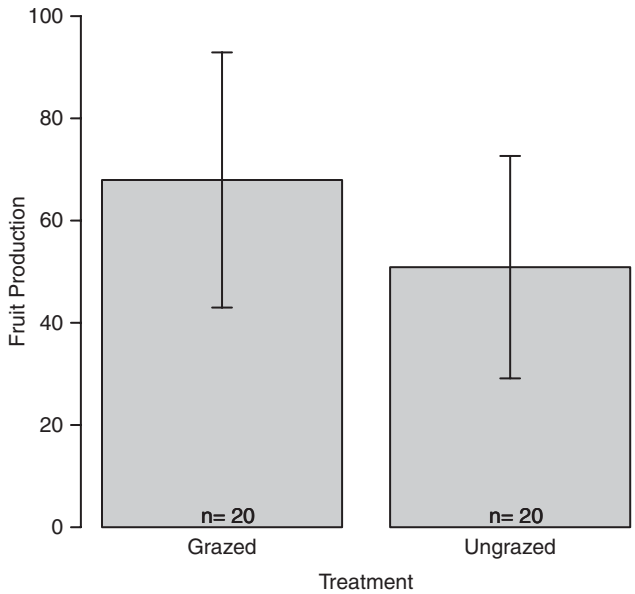


Figure 4.5 We complete this graph by adding sample sizes for each estimate of the mean using the function text() and its arguments specifying the location (x–y coordinates) of the text and the text itself.

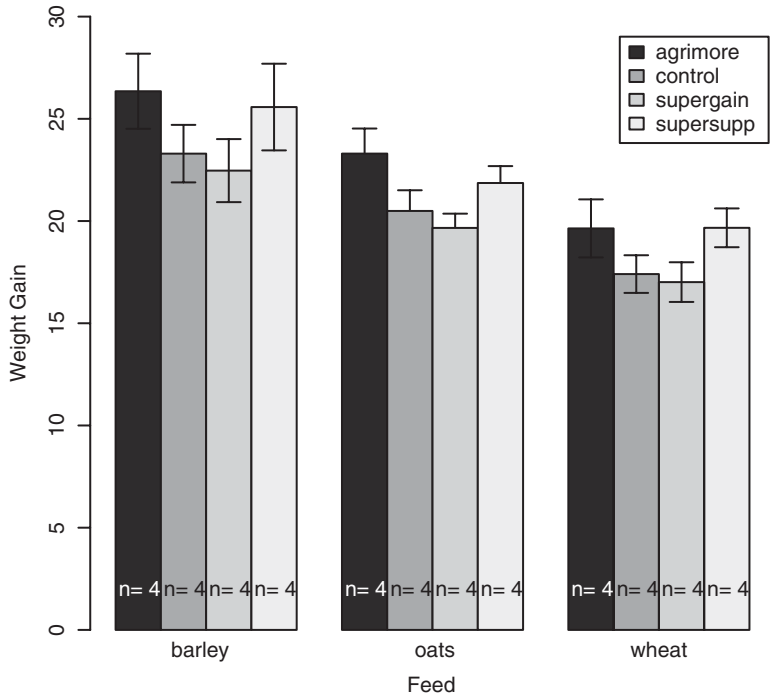


Figure 4.6 In contrast to the grazing data containing two levels (grazed and ungrazed), the weight gain data represents a 3-level (food) x 4-level (supplement) experiment. However, creating an elegant barplot with error bars uses the same simple workflow: combine the creation of a mean, standard deviation, and sample size matrix using tapply() with barplot() and arrows() as in the first example.

4.3 Making a picture—scatterplots

The final example we will run through for plotting involves making a scatterplot. We will teach you the basics for plotting data, customizing the x- and y-axes, customizing the points, and a few key tools for plotting subsets of data. Let's begin with the logical first step: plotting the data.

You won't be surprised by now to find that the function to make a scatterplot is simply **plot**. **plot** is a powerful function that is smart—it can plot x–y scatterplots when given two columns of data; it can also plot correlations among numerous variables when given a **data frame** or **matrix**; and it

can plot numerous diagnostic figures when given a linear model object (see Chapter 5); it can also make a box and whisker plot if you give it a categorical x-variable. Here, we will focus simply on using it to make a scatterplot of two variables from a data frame.

It is worth reviewing at this point how to isolate and use different columns in a data frame. If you've forgotten, take some time and go back a few pages and review Section 3.4 The magic tool is the **$** symbol or []. Making a scatterplot with plot can be done in at least two ways. Examine the help file for **plot**—here you will notice that the standard usage specifies two arguments—x and y, in that order. Plot Method 1 takes x and y columns that you specify from a data frame. We then, as indicated in the help file for **plot** (?plot), use the syntax:

```
plot(data.set$x.column, data.set$y.column)
```

Note that it is x followed by y.

Further perusal of the help file reveals a series of additional plot functions, based on plot—specifically **plot.default** and **plot.formula**—in the "See Also" section below "Usage".

plot.formula is the one we want you to look at. It also requires two arguments, this time a *formula* and a *data frame*. The *formula* is a representation of the data we want plotted on the y- and x-axes - specifically "y ~ x". The data argument is the dataset from which the y- and x-variables come. The full syntax for Plot Method 2 is:

```
plot(y ~ x, data = dataset)
```

Note, in contrast to Plot Method 1, it is y ~ x THEN the name of the dataset. And that is a "~" (tilde/ "squiggle") not a dash/minus symbol.

To make this more clear, let's work with some data. We'll use the fruit and root data from the **barplot** example above again—the compensation.csv dataset. If you'd like, start a new script. Call it scatterplot.tutorial.R. Type as many comments (**#**) as you'd like, and more than you think you need. Don't forget to start it with **rm(list=ls())**.

Now, add the function to import the compensation.csv data (**read.csv**). If you've forgotten how to do this, go back to your **barplot** script. Use **str**(),

names(), and **head**() to make sure the data is the correct data. Again, look back if you've forgotten what these do, or how to use them.

Right. Let's make a scatterplot of fruit production versus root biomass—they are both continuous variables. We'd like fruit production on the y-axis and root production on the x-axis. Regardless of the method we will use (see above, Plot Method 1 or 2), we need to know the name of the dataset. We've called ours "mydata".

To make a scatterplot using Plot Method 1 (Fig. 4.7), we specify the two columns explicitly in the arguments to plot. Add this to your script:

```
plot(mydata$Root, mydata$Fruit)
```

Why does root come first? Recall from above that the order of the arguments in this method are x then y. If you can't quickly say what the $ symbol does, add some comments (#).

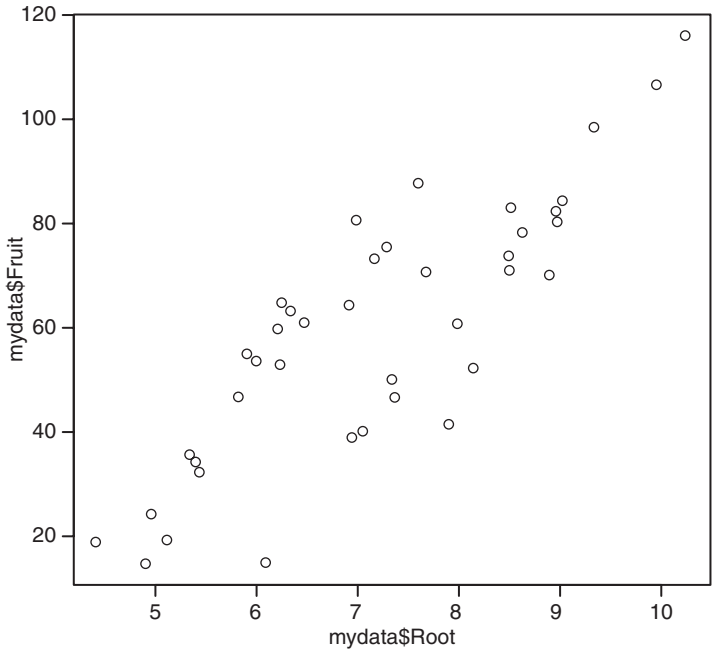


Figure 4.7 A simple and basic scatterplot is produced using the function plot(). Here we make the first argument to plot() the column of our data frame that contains the x-axis values, and the second argument to plot() the column of our data frame that contains the y-axis values.

To make the same picture using Plot Method 2 (Fig. 4.8), we specify the two column names AND the dataset name and we use the formula interface:

```
plot(Fruit ~ Root, data = mydata)
```

Executing this should produce nearly the same figure (Figs 4.7, 4.8). Can you see the difference? Look closely at the axis labels.

Why use the formula interface to plot (i.e., Plot Method 2)? As we will see in the final chapter, this formula is also the way we specify a linear model (e.g. regression, ANOVA, ANCOVA). Thus, plotting data with the formula interface is a logical precursor to specifying the statistical model formula. Plotting y ~ x matches exactly the model we fit: y ~ x. Furthermore, you only have to type the name of the data frame once, and you can even subset the data frame within the plot function. From this point on, we recommend that you use the formula interface, specifying the y- and x-variable in the formula argument and the data frame from which they come.

4.3.1 PIMP MY SCATTERPLOT: AXIS LABELS

As with the **barplot**, the default plot returned by **plot** leaves something to be desired. One might (should) consider this an opportunity. Let's enhance our axes and points first.

To change the axis labels, we use the same arguments found in **barplot**—*xlab* and *ylab*.

```
plot(Fruit ~ Root, data = mydata,
  xlab = "Root Biomass",
  ylab = "Fruit Production")
```

Note the indentation, the use of " " 's to enclose text and the commas. Be particular and fastidious!

Here is how to exercise some fine control of graph features. The above call (using a formula) to plot produces nicer axes labels. What if you want them bigger? *xlab* and *ylab* can accommodate a **list** of instructions too. One of the easiest is the character expansion argument, *cex*. *cex* = 1.5 increases the default font size by 1.5 times. *cex* = 2 is bigger. *cex* = 0.5 is smaller. Try it!

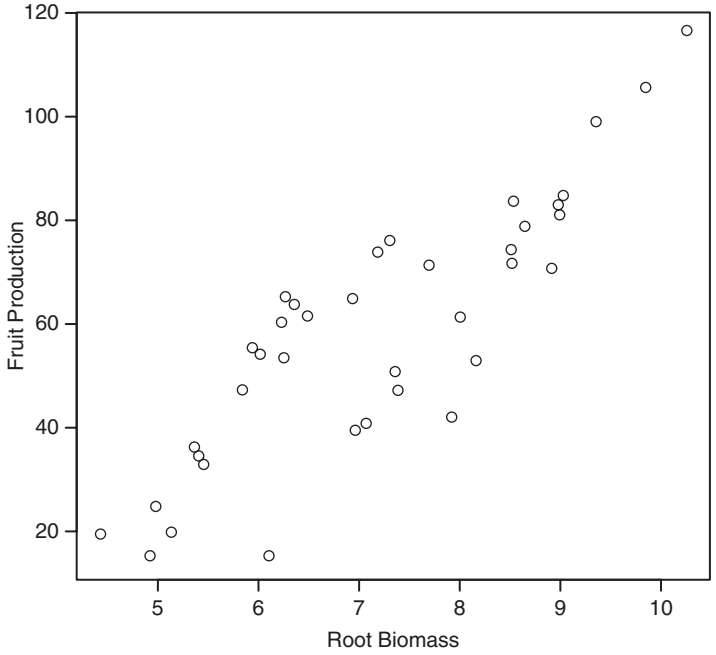


Figure 4.8 We can also produce this plot using the formula interface, where the first argument to plot() is a y ~ x "formula" and the second argument is a data frame. Here we have used plot(Fruit ~ Root, data=mydata). The data argument tells plot where to look for columns labelled Fruit and Root. Notice that we can alter the x- and y-axis labels, and the font size of these labels (see text).

```
plot(Fruit ~ Root, data = mydata,
   xlab = list("Root Biomass",cex = 1.5),
   ylab = list("Fruit Production",cex = 0.5))
```

Again, note the use of **list**() in the call to *xlab* and *ylab*. Note the commas. Note the additional brackets/parentheses you need.

4.3.2 PIMP MY SCATTERPLOT: POINTS

Not good enough you say! SigmaPlot does this better? Let's change some more features of the graph.

One of the most important arguments to the **plot** argument is *pch*—point character. It controls the shape (symbol) of the points. A second

important point argument is *col*—colour. And a third important argument is *bg*—the point background colour. And a fourth...

Wait. How can we find out what we can control? Great question. Part way down the help file for **plot**, in the arguments section, is the answer: **par.**

```
?par
```

par is short for graphical parameters. It is a long list of things in your plot that you can change. It is very worth consulting this list, often, and revising, studying, and even memorizing (or just printing out and highlighting) many of these options. All of your creativity and opportunity for plot enhancement is here.

Our favourite point character is *pch*=21. What? Try this:

```
?points
```

Ahh! Excellent. An entire help page on how to customize points!

As indicated in the help file for **points** (a bit of the way down), *pch*=21 is a filled circle. This means (obviously) that we can specify the circle colour and the background fill colour. The defaults are black ringed circle and empty background. But let's make a grey background of fairly big circles. Here's how.

Enhance your script (remember—don't rewrite everything…just go back and add it to the plot command you've already written, and re-run that selection.

```
plot(Fruit ~ Root, data = mydata,
   xlab = list("Root Biomass",cex = 1.5),
   ylab = list("Fruit Production",cex = 0.5),
   cex = 2, pch = 21, bg = "grey")
```

This last line, containing three arguments, says make the points twice as big as normal (cex = 2), use the filled circle (pch = 21), and fill it with a grey colour (bg = "grey"). Easy, right? This is what you should have now (Fig. 4.9).

OK, now a test of your biological intuition. Can you see two groups of data? What do you think the two "groups" of points represent? Looking at:

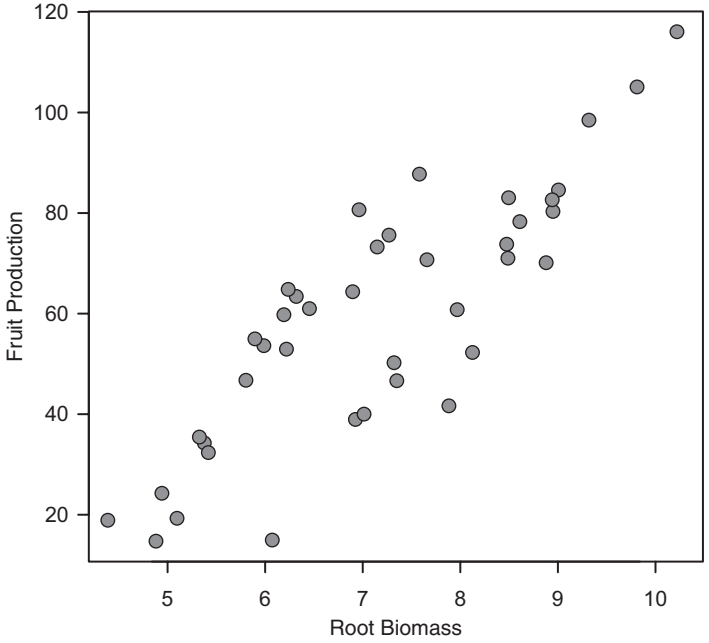


Figure 4.9 Changing the types and colours of the points uses several arguments to plot. Here we specify a point character that is a black ringed circle with a coloured background (pch=21). We specify the background as grey (bg="grey"), and make them larger than the default with the character expansion argument (cex=2).

```
head(mydata)
names(mydata)
```

reminds us of the third column, Grazing, the experimental treatment—the presence or absence of herbivores in the replicate plots.

4.3.3 PIMP MY SCATTERPLOT: COLOURS (AND GROUPS)

The final enhancements to share with you in this section are methods to colour the points according to a categorical variable. We will demonstrate two ways to do this. We will develop this idea later as well, and show you, as is common in R, many ways to achieve the same end point.

First, let's choose come colours—green for grazed plots and blue for ungrazed. One way to colour the points is to first create a list of colour

names that, were we to put it next to the column of grazed/ungrazed in the dataset, would have green next to "grazed" and blue next to "ungrazed". We can make such a list using an **ifelse()** function:

```
culr <- ifelse(mydata$grazing == "Grazed", "green", "blue")
```

The anatomy of this snippet of code is as follows. culr is just a word (actually, it's a rather phonetic collection of letters, isn't it). We assign (<-) to culr the outcome of a rule that operates on the grazing column of mydata—a rule that says if the row of the grazing column is "Grazed" (== asks, is this TRUE), return the text (string) "green", else, return the text (string) "blue".

This produces the following collection of colours, which you can check for yourself, simply by typing the word culr in the console, after adding the above to your script.

```
> culr
[1] "blue" "blue" "blue" "blue" "blue" "blue" "blue"
"blue" "blue" "blue" "blue" "blue" "blue"
[14] "blue" "blue" "blue" "blue" "blue" "blue" "blue"
"green" "green" "green" "green" "green" "green"
[27] "green" "green" "green" "green" "green" "green"
"green" "green" "green" "green" "green" "green"
"green"
[40] "green"
```

How do you use this list? It is rather easy. Again, without retyping everything, update your script as follows:

```
culr <- ifelse(mydata$Grazing == "Grazed", "green", "blue")
plot(Fruit ~ Root, data = mydata,
   xlab = list("Root Biomass",cex = 1.5),
   ylab = list("Fruit Production",cex = 0.5),
   cex = 2, pch = 21, bg = culr)
```

We hope you see the difference in the script—we've simply replaced our *bg*="grey" with *bg*=culr, the collection of colours we made using **ifelse()** list. It produces the figure below (4.10).

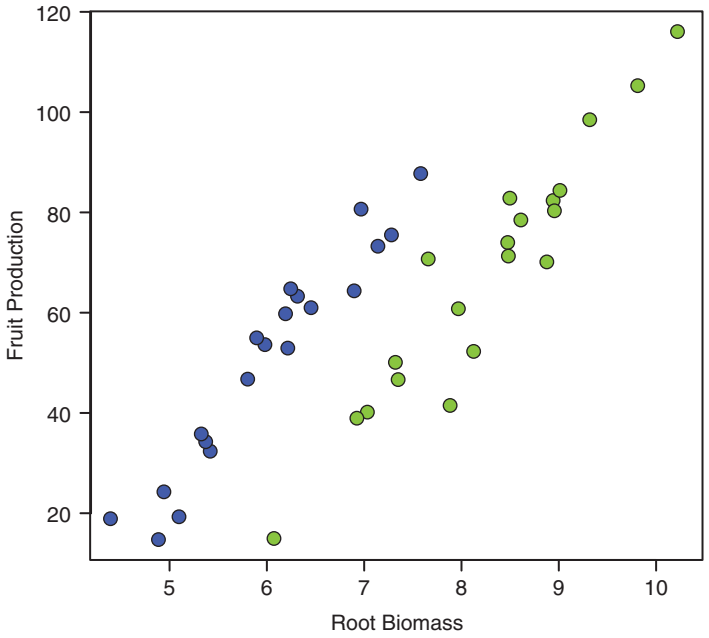


Figure 4.10 Giving the background of the points (bg) as a vector of colours that corresponds to the two groups (grazed and ungrazed; created using ifelse()) produces a scatterplot that highlights the two treatments. All that is needed now is a legend—see the text for how to add this!

The above method is intuitive, and clever, but because we use **ifelse**, it is limited to categorical variables with two levels. Of course, you could use a nested **ifelse**(), but that quickly becomes cumbersome.

The following alternative method requires, again, a choice of colours. In contrast to above, we simply specify these colours in a vector.

```
culr <- c("green","blue")
```

The alternative way to specify the colour for the points, indexed by the levels of Grazing, is as follows:

```
bg = culr[mydata$Grazing]
```

The trick here is the [ ]s used with a factor in your dataset. You should recognize the use of the square brackets—they are used to "index" some-

thing—and that something is the levels or categories of the grazing treatment. The indexing by square brackets assigns the levels of Grazing each to one of the colours—the first colour to the first alphabetical label. As above, this command creates the same list as the **ifelse** method did:

```
> culr[mydata$Grazing]
```

```
[1] "blue" "blue" "blue" "blue" "blue" "blue" "blue"
"blue" "blue" "blue" "blue" "blue" "blue"
[14] "blue" "blue" "blue" "blue" "blue" "blue" "blue"
"green" "green" "green" "green" "green" "green"
[27] "green" "green" "green" "green" "green" "green"
"green"  "green"  "green"  "green"  "green"  "green"
"green"
[40] "green"
```

The important feature of this method for colouring points is that it will work for a categorical variable with more than two levels. Even when the categories do not run in a contiguous order in the dataset. As above, we can modify the script. The figure, however, should not change.

```
culr <- c("green", "blue")

plot(Fruit ~ Root, data = mydata,
   xlab = list("Root Biomass",cex = 1.5),
   ylab = list("Fruit Production",cex = 0.5),
   cex = 2, pch = 21, bg = culr[Grazing])
```

4.3.4 PIMP MY SCATTERPLOT: LEGEND

There is one more feature missing from our plot—a **legend**. Adding a legend is … easy. Try it yourself first by navigating and exploring the help file.

```
?legend
```

Here is one way we might do it. By examining **?legend**, you'll see that it takes a few key arguments: a location (x, y coordinates), a set of *text*, detail

about *points*, detail about *lines*, and details about *colours*. In our figure above, we have *points* and we'd like *text* describing the *colours*.

The location can be specified generically, via "top left", "top right", etc. Or by coordinates on the x- and y-axis. Here are two versions

```
legend("topleft", legend = c("Grazed", "Ungrazed"),
   pch = 21, pt.bg=c("Green", "Blue"), pt.cex=2)
```

OR

```
legend(5, 110, legend = c("Grazed", "Ungrazed"),
   pch = 21, pt.bg=c("Green", "Blue"), pt.cex=2)
```

The *location* in the first example is pinned to the upper left corner of your plot. In the second we specify a *location* a bit to the right and down from this corner. The *legend* argument contains the *text*—the levels of the grazing treatment in our experiment. *pch* specifies the point type, just as we did in the plot function. The legend command requires *pt.bg* and *pt.cex* rather than *bg* and *cex*. Why? Because in the legend function, *bg* and *cex* are arguments for *legend box background colour* and *legend text font size*, so a second set with the "pt." prefix is required.

You should now see a publication-ready plot, if you've added this legend function to your script and saved it and run it (and maybe shared it with a friend, because you are feeling pretty good at R now). Here is the detail that should be in your script. It is a compact, customized, editable, shareable, repeatable record of your plotting.

```
culr <- c("green","blue")

plot(Fruit ~ Root, data = mydata,
   xlab = list("Root Biomass",cex = 1.5),
   ylab = list("Fruit Production",cex = 0.5),
   cex = 2, pch = 21, bg = culr[Grazing])

legend(5, 110, legend = c("Grazed", "Ungrazed"),
   pch = 21, pt.bg=c("Green", "Blue"), pt.cex=2)
```

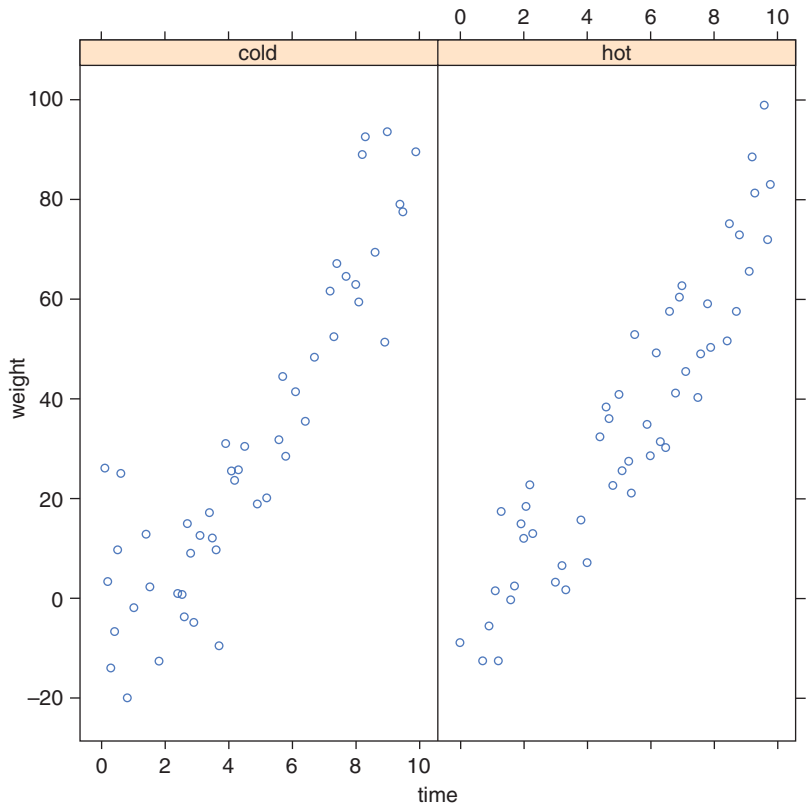
You now have in your aRsenal methods for plotting bars ± errors and scatterplots with colour specific coding for categorical variables. These are powerful, flexible methods. While they require keeping track of some

Box 4.1: Using the layout() and pdf() function to save multiple plots

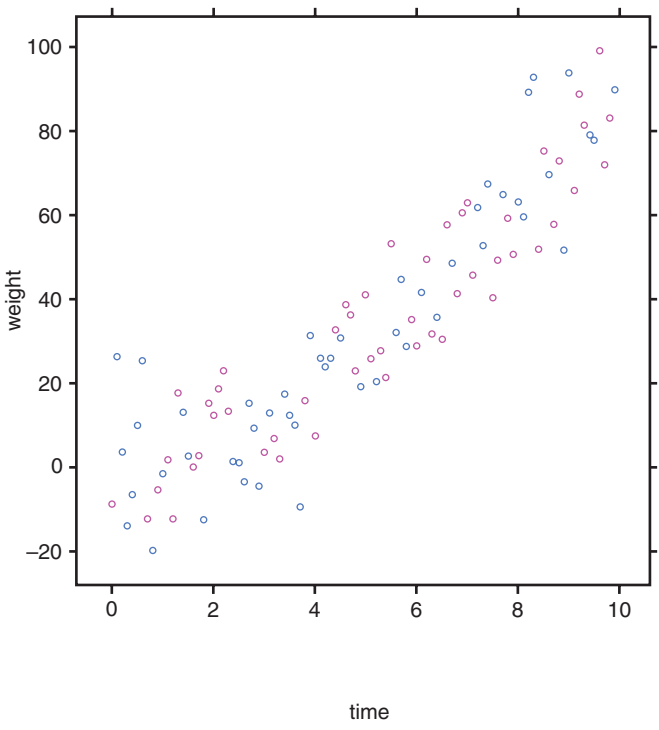
```
# This snippet of code writes a pdf of a multi-panel plot to your hard drive.
# It firsts creates some fake data.
# It then opens a pdf "device" on your computer which prepares your
# working directory to receive a pdf.
# It then executes some plotting code and writes the pdf to the working
# directory with a specific filename you have specified.
# Finally, the dev.off() function closes the connection to the
# working directory, turning off the pdf device; this final step
# ensures that subsequent plotting is written to the graphics window
# Make the data

time <- seq(0, 9.9, 0.1) # a sequence
weight <- (time^2) + rnorm(100, 0, 10) # time^2 with deviation
temperature <- factor(sample(c("cold","hot"),100,
  replace = TRUE))

# Identify your working directory - this is where the PDF will be found

getwd()

# OPEN THE PDF DEVICE—specify a file name and a paper size
pdf(file = "PracticePDF.pdf", paper = "a4")

# set the layout of the plotting
# here we specify a 2 x 2 plot and place the figures into the plot by row

layout(matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE))

# do your plotting—here we add three figures to the layout
# we also use the function mtext() to add letters outside each of the
# boxes as identifiers for a figure legend in your manuscript

plot(weight ~ time, pch = 21, bg = "green")
mtext("A", side = 3, line = 2, adj = 0)

plot(weight ~ temperature)
mtext("B", side = 3, lin e= 2, adj = 0)

plot(weight ~ time, col = c("black", "red")[temperature])
mtext("C", side = 3, line = 2, adj = 0)

# Close the connection to your hard drive
dev.off()

# Now, go to your hard drive and look in the working directory for the pdf.
```

Box 4.2: A very brief introduction to lattice plots, from the lattice library, using xyplot()

```
# This snippet of code shows the basics of the lattice
# library using the function xyplot()
# First, we load the package lattice as a library

library(lattice) # Comes with the base installation of R

# Now, we make up some data

time <- seq(0, 9.9, 0.1) # a sequence
weight <- (time^2) + rnorm(100, 0, 10) # time^2 with deviation
# random assignment of temperature
temperature <- factor(sample(c("cold", "hot"), 100,
  replace = TRUE))
```



```
# Now, we use xyplot in two different ways.
# First we use the panel plot functionality via the "|" symbol
# (also known as a pipe)
# this creates two panels of weight versus time, each panel
# representing the two temperatures
xyplot(weight ~ time|temperature)

# Next we use the groups argument to colour each symbol according
# to the two different temperatures:
xyplot(weight ~ time, groups = temperature)
```



vocabulary, it should be clear that much of the syntax is quite intuitive (plot = plot; col = colour). Once you get to grips with the basics, producing gorgeous figures is easy. And, importantly, by capturing your instructions (code) in a script, you can replicate, adjust, and share the figure with anyone, anywhere, and anytime. You can even produce the same figure, six months later, within seconds.

4.4 Plotting extras: pdfs, layout, and the lattice package

Before we move on to the next chapter, we'd like to draw your attention to a few extra titbits. First, it is easy to save your figures as a pdf. Both Windows and Mac versions of R allow you to copy, from your screen, the graphics window, and save it as a pdf. **Box 4.1** also provides some code to save figures directly to your hard drive as a pdf. This can be very handy for producing graphics of specific size, orientation, and paper size, as might be required by various journals. By examining

```
?pdf
```

you can see the generous selection of graphic formats in which you can save R figures. Pay special attention to the **dev.off**() function at the end of the plotting!!!! If you don't execute this function, your pdf will appear but you won't be able to open it.

Second, we draw your attention to the lattice package by Deepan Sarker. This package is part of the core set of packages provided with R. "lattice" graphics are a powerful type of graphic that automatically produce "groups" within plots (as we created in **Fig. 4.10** via the use of colours and bg =) and or separate "panels" for each group. **Box 4.2** provides a simple example using the compensation.csv data for each of these types of graphics.

Lattice is very powerful. Its default settings are outstanding for that first phase of data analysis—explorative plotting. It is also infinitely customizable, as are many graphics in R. However, the vocabulary to do this has its own rather sharp learning curve. Thankfully, there is a book, and a very responsive author and team on the R-help forum.

You are ready to move on now. You know how to use R. Trust us. You do.

Doing your Statistics in R

Getting Started

It may seem odd, in a chapter on doing statistical tests, but we are going to start by saying again exactly what we said at the beginning of Chapter 4: we have a very fundamental rule for data analysis. NEVER start an analysis with statistical tests. ALWAYS start an analysis with a picture.

Why? If you have done a replicated experiment, conducted a well organized sampling programme, or generated data using a model, it is highly likely that some theoretical expectations underpinned your research effort. You have in your head an EXPECTED pattern in your data.

Plot it first. Make the picture that should tell you the answer—make the axes correspond to the theory. If you can do this and see a pattern in your data, you are in great shape. Your data will be showing you a pattern to compare to your expectations and guide the development and implementation of your statistical modelling.

What do you do after you plot the data? Our philosophy, translated into a workflow, is as follows. Once you've made a picture, you embark on translating your statistical model into R language to capture the hypothesis you are testing. If you have made an informative picture that reveals the relationships among variables central to your hypothesis, this should be easy (or become easier with experience).

Once you've specified your model in R and made R execute it, we feel that it is vital NOT to then interpret the results. It is instead vital to assess first the assumptions that are important to the type of modelling you have decided to implement. For example, a t-test may assume equal variance among groups; ANOVA and regression assume normally distributed residuals, amongst other things. By assessing these assumptions, you are ensuring that the results returned by the modelling are reliable. If the assumptions are not met, then the predictions from, or interpretation of, the model is compromised. Only once you have assessed the assumptions should you begin to interpret the output of the statistical modelling. This penultimate interpretation step is followed by integrating the modelling results into your original figure—a process some of you may know as adding predicted or fitted lines to your graph.

We will begin this chapter by looking at how to implement a chi-square contingency table analysis. We've chosen this because it allows us to reinforce a few principles from Chapters 1–4 with respect to graphics, and to show you how easy interpreting statistics can be if you can make an informative picture. We will follow this with implementation and interpretation of a two sample t-test and then an introduction to implementing and interpreting general linear models—that class of model that includes regression, ANOVA, and ANCOVA.

If you have never actually used these statistics before, or never taken a course in statistics that has introduced these tools, now is a good time to go and do this. Our instructions will certainly teach you some statistics, but our goal is to teach you how to use (get) R to do these. We are ASSUMING that you understand when and for what type of data you may require a chi-square contingency table analysis, t-test, or ANCOVA. We are ASSUMING that you understand what types of HYPOTHESES can be tested with these different methods.

5.1 Chi-square

We wish to make three points with this introduction to the chi-square contingency table analysis. First, always plot your data first. Second, make

every attempt to understand the hypothesis that you are testing, both biologically and statistically. Finally, barplot loves a matrix.

The chi-square contingency table analysis is an analysis of count data. It is essentially a test of association among two, or more, variables. The need for this sort of analysis arises when you have a set of objects, or events, and for each of these you can classify them by more than one grouping variable (for example, sex: male/female and lifestage: juvenile/mature). Introducing some data will help explain the basics.

Let us assume that we have collected data on the frequency of black and red ladybirds (*Adalia bipunctata*) in industrial and rural habitats. Why might we have collected such data? It has long been thought that industrial and rural habitats, but virtue of the amount of pollution in them, provide different backgrounds against which many insects sit. Contrasting backgrounds to the insect can be bad if contrast is associated with predation risk. Here, we are interested in whether dark morphs are more likely to reside on dark (industrial) backgrounds. These data are available at www.r4all.org.

By performing a chi-square contingency table analysis, we are testing the null hypothesis that there is no association between ladybird colours and their habitat. The alternative is that there is. The test does not allow us to specify the direction of this association, but that is something we will see from the graph we make before performing the test. Biologically, we are trying to answer the question of whether some feature of the habitat is associated with the frequencies of the different colour morphs. Note how generic this statement is—there is no specification of the features of the habitat, or of the direction of potential association.

Here are the data as a 2 x 2 contingency table. The numbers are counts of the numbers of individual ladybirds of each colour type, collected in each of the two habitats.

	Industrial	Rural
Black	115	30
Red	85	70

We can use the matrix() function to enter this table into R.

```
chi.data <- matrix(c(115, 30, 85, 70), 2, 2, byrow = TRUE,
  dimnames = list(c("Black","Red"), c("Industrial","Rural")))
```

Notice that we have used the *dimnames* argument in the **matrix** function to specify the row and column names.

Now, one of the important graphing rules we learned in past chapters is that **barplot** loves a **matrix**. We have just specified our data as a matrix, and a barplot is a good way to visualize the possible relationship between colour morph and habitat. Simply ask barplot to plot the matrix, putting the bars beside each other and colouring them appropriately.

```
barplot(chi.data,beside = TRUE, col = c("Black","Red"),
  ylim = c(0,125),legend = TRUE)
```

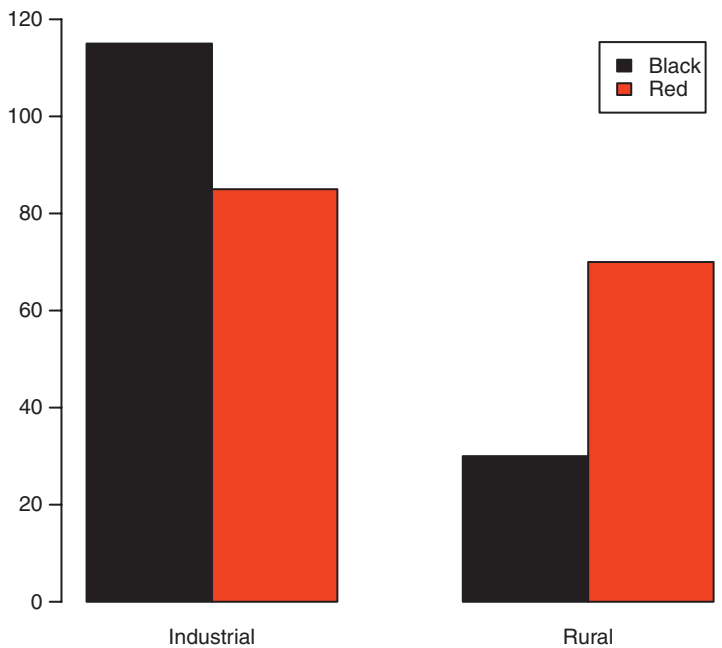


Figure 5.1 barplot() loves a matrix. A contingency table is a matrix, and passing this matrix to barplot produces a lovely barplot. Notice that the colours and axes labels correspond to the column and row names (*dimnames*) we assigned to the matrix.

Notice that our use of **barplot** takes the **matrix** with its dimension names and creates a perfect representation of the data. At this point, we can ask ourselves whether we think the null hypothesis (no association, uniform distribution of counts among classes) is true or not. To actually test this hypothesis, we can use the function chisq.test().

```
> chisq.test(chi.data)
Pearson's Chi-squared test with Yates' continuity correction
data: chi.data
X-squared = 19.1029, df = 1, p-value = 1.239e-05
```

This rather sparse output provides all the information we need in order to present our "result". According to the test, there is a very small probability ($p=1.239 \times 10^{-5} = 0.00001239$) that the pattern we see arose by chance; that is if there really was no association between colour and habitat, and we carried out the sampling process again and again, we would get such a large value of chi-square only once in every 10,000 or so samples. This is a pretty good indication that it probably isn't a chance result. The result allows us to reject the null hypothesis and conclude that there is some association. Our figure suggests that black morphs are more frequent in industrial habitats while red morphs are more frequent in rural habitats.

What might we report in a manuscript? "We tested the hypothesis that there is an association between colour morphs of ladybirds and industrial and rural habitats. Ladybird colour morphs are not uniformly distributed in the habitats ($\chi^2 = 19.1$, df=1, $p<0.001$), with black morphs being more frequent in industrial habitats and red morphs more frequent in rural habitats (Fig. 5.1).

For those of you who are familiar with the mechanics of the chi-square test (row sums, column sums, observed and expected values), you will be please to know that all of these are accessible to you, simply by assigning the values returned by **chisq.test** to a name:

```
 > my.chi <- chisq.test(chi.data)
 > names(my.chi)
[1] "statistic" "parameter" "p.value" "method" "data.name"
[6] "observed" "expected" "residuals"
 > my.chi$expected
        Industrial     Rural
Red       96.66667  48.33333
Black    103.33333  51.66667
```

All of the code for this analysis is provided in **Box 5.1**.

We had three points to make with this introduction to a chi-square contingency table analysis: remind you to plot your data first; implore you to think about how to translate the biological question you are asking into a statistical hypothesis; and that **barplot** loves a matrix. Hopefully you have grasped the importance of all three, and we have reinforced basic details about a chi-square contingency table analysis. Most importantly, this example should have shown you the ease with which you can gain understanding of your data using R.

5.2 Two sample t-test

As above, we wish to make three points with this introduction to the two sample t-test. First, always plot your data, first. Second, checking assumptions associated with the statistical tool is vital for reliable interpretation. Finally, R makes all of this easy.

The two sample t-test is a comparison of the means of two groups. It is appropriate when sample sizes in each group are small. However, it does make some assumptions about the data being analysed. The standard two sample t-test assumes that the data in each group are normally distributed and that their variances are similar.

First, some data. Here we will analyse ozone levels in two gardens, creatively labelled A and B. The data are ozone concentrations in parts per hundred million (pphm) on ten summer days. Ozone in excess of 8 pphm can damage lettuce crops. We are interested in whether there is a difference in the average ozone concentration between the gardens.

Box 5.1: The chi-square contingency table analysis

```
# Chi-square test
# Enter the data, as a matrix, with informative dimension names
chi.data <- matrix(c(115, 30, 85, 70), 2, 2, byrow = TRUE,
  dimnames = list(c("Black", "Red"), c("Industrial",
    "Rural")))
# use barplot to make an informative and helpful figure
barplot(chi.data,beside = TRUE,
  col = c("Black", "Red"),
  ylim = c(0, 125), legend = TRUE)
# perform the chi-square test
chisq.test(chi.data)
# assign the test to an object and examine it,
# retrieving the expected values
my.chi <- chisq.test(chi.data)
names(my.chi)
my.chi$expected
```

As you've learned, the first step is to read these data into R. This requires creating a script. We will assume that you are working along with us here—that you've just started R and opened a new script and are ready to go.

The data can be found at http://www.r4all.org. Download the .csv file to your computer. At this point, start a new script, annotate it with some information and, having downloaded the data, get it into R's brain. Following the convention we established above, we assign the data to a name (object) called ozone. Once it is into R (**read.csv**), examine it using the functions **names()**, **head()**, and **str()**.

```
ozone <- read.csv("path/to/data/ozone.csv")
names(ozone)
head(ozone)
str(ozone)
```

The output from **str**(ozone) reveals that our data are in a data frame format with two columns and that Garden is a factor with three levels. R has assumed (because they are not numbers) that the data in the Garden column should be treated as a factor—a code denoting groups within the dataset.

```
> str(ozone)
'data.frame':20 obs. of 2 variables:
$ Ozone : int 3 4 4 3 2 3 1 3 5 2 ...
$ Garden: Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
```

5.2.1 THE FIRST STEP: PLOT YOUR DATA

As we have repeatedly emphasized, the first step in an analysis of data should be plotting. We have can see that there are two different gardens denoted as A and B. To initiate a comparison of ozone levels between these gardens, we might consider a plotting method that allows us to see the central tendency of the data and the variability in the data, for each garden. A good tool for this is the histogram. If we stack histograms of the ozone levels in each garden on top of each other we should be able to a) see whether the means seem different and b) assess whether the data in each

group appear to be normal and have similar variance. We thus accomplish the process of visualizing our hypothesis and evaluating some assumptions of a two sample t-test, all in one effort.

Here, reinforcing some of the previous exercises, we show two ways to isolate the data for set A and B, and how to plot the histograms.

```
# read in the data
ozone <- read.csv("gardens.csv")

# check the data
head(ozone)
str(ozone)

# subset the data
GA <- ozone[ozone$Garden=="A",]
GB <- subset(ozone,Garden=="B")

# make a single figure with two plots (2 row, 1 column plotting
# grid)

par(mfrow = c(2,1))
hist(GA$Ozone)
hist(GB$Ozone)
```

If you have been successful at importing the data, this code should produce Figure 5.2. We have done something deliberately confusing here. Can you spot the problem with this figure?

To answer that question, let's review the rationale associated with plotting the data this way. One reason was to assess the potential normality of these data, and the equality of variance in each garden—two of the assumptions of the two sample t-test. The figure provides visual representations of the distributions of the data in each sample, and from these it seems reasonable to say that normality and equality of variance has been met. Let us assume that it has. Of course, R has functions to statistically evaluate normality and equality of variance.

The second reason for plotting the data like this was to provide some *a priori* indication of whether our null hypothesis—that there is no difference in ozone levels between the two groups—is true. A cursory glance at the figure that we've produced might suggest to us that we cannot

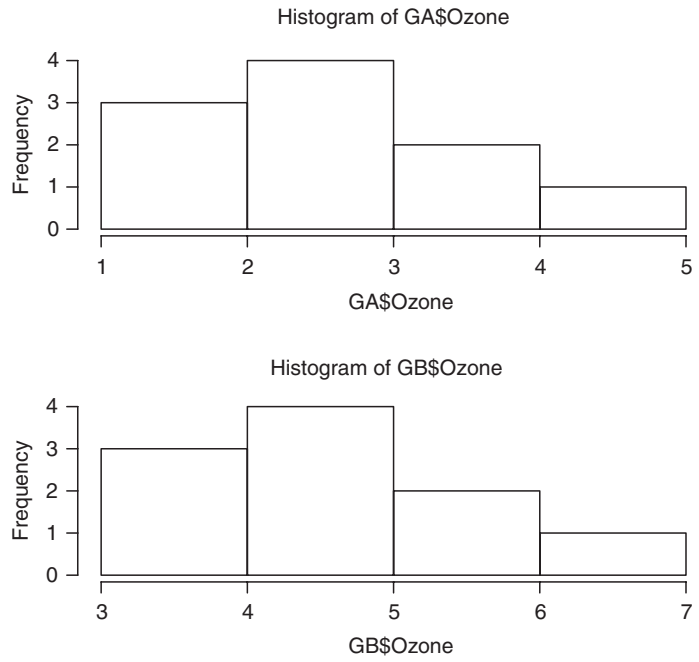


Figure 5.2 A stacked histogram of the ozone data for garden A and garden B. We can use either the par(mfrow())or layout() function to make the 2-row x 1-column graphical window.

reject the null hypothesis—the histograms look the same. But is that actually true? Examine carefully the x-axis of each histogram, and notice that the scales are very different. Garden A's ozone levels range from one to five, while those of Garden B range from three to seven.

This is a lesson: when you just ask for a plot, R has to make some guesses at what you want; while R is very good at creating a picture with minimal instructions, these guesses may not actually be quite what you want. Making informative pictures of your data requires thinking and work. In this case R has guessed wrong—we don't want the scales adjusted separately for each plot, we want them to be comparable. Fixing this is very easy.

Simply update your script:

```
# set the layout
par(mfrow = c(2,1))

# add the histograms with fixed x axis range
hist(GA$Ozone, col="grey40", main="", xlim=c(0,7),
  xlab="A-Ozone")
hist(GB$Ozone, col="grey80", main="", xlim=c(0,7),
  xlab="B-Ozone")
```

We have now told R to use different colours for each garden and exactly what range to use for the x-axis in each plot: xlim=c(0, 7) tells R to make the range of the x-axis extend from 0 to 7. This should produce Figure 5.3.

Now we can see clearly that there is very likely a substantial difference between the ozone measurements in the two different sets of gardens.

At this point, we leave it to the reader to generate some code to calculate the means of each group, and the variance—as one of the main assumptions of the two sample t-test is that variances are equal between the two

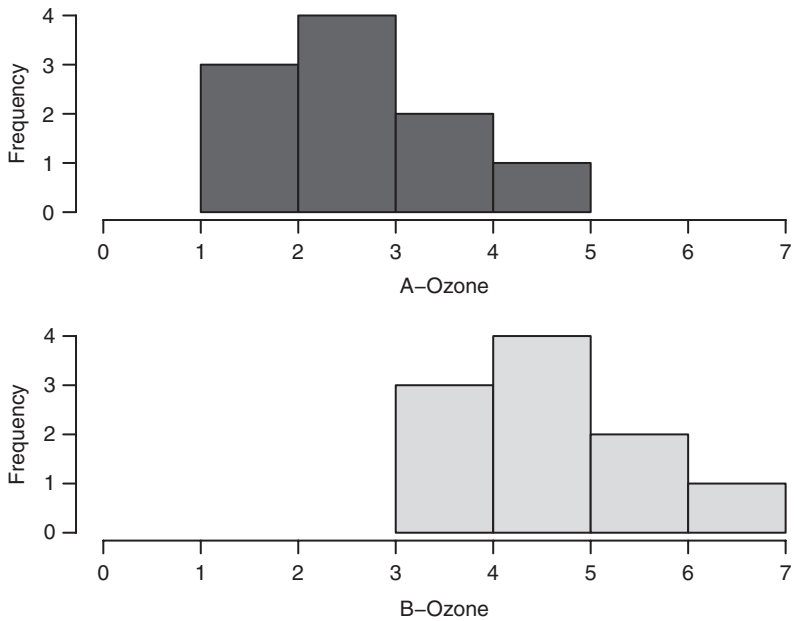


Figure 5.3 Enhanced histograms for the ozone data. The x-axes have been standardized so that the range of data presented in each histogram is the same and colours differentiate the data from each garden.

groups. Consider ?mean and ?var for help files of substantial relevance. Or use the function sd() and some basic statistical theory. The previous chapters have equipped you to do this.

5.2.2 THE TWO SAMPLE t-TEST ANALYSIS

To finish the analysis, you can use the t.test() function in R. Add the following to your script:

```
# Do a t.test now....
t.test(GA$Ozone,GB$Ozone)
```

This should produce the following output:

```
Welch Two Sample t-test
data: GA$Ozone and GB$Ozone
t = -3.873, df = 18, p-value = 0.001115
alternative hypothesis: true difference in means is not
  equal to 0
95 percent confidence interval:
-3.0849115 -0.9150885
sample estimates:
mean of x mean of y
        3         5
```

Let us spend a bit of time reviewing this output as a way to reinforce aspects of our workflow. We will call this the anatomy of the R-output.

Let us begin with the data: we see that the data that have been used are declared—this is a good way to confirm that you've analysed the data you wanted to analyse.

The next line provides the traditional t-test statistic, degrees of freedom, and p-values. We do assume you know what these mean. However, let us use the next few lines of output to make a few clear points about the two sample t-test. The output declares the alternative hypothesis to us: that the true difference in means is not equal to 0. This should help you understand more of what you've been doing. If the difference between two things is 0 then they are the same, and we have no grounds for rejecting the null hypothesis.

Next in the output is a 95% confidence interval. This interval is around the difference between the two means. Keeping in mind that the difference would be 0 if the means were the same, the fact that this interval does not include 0 provides an answer to our initial question: the probability that the true difference between the garden sets is zero, given the data from the two samples, is rather small. We conclude that they are different. This falls in line with the test statistic and associated p-value. Finally, the output provides the means between the two groups.

Finally, look at the first line of the output: it declares that the method is a Welch Two Sample t-test. A quick look in the help files, or on Wikipedia, reveals that this method allows one of the assumptions for a two sample t-test to be overlooked—that of equal variance. While you made the effort to assess the assumption of equal variance above, and you will have found them to be equal, you now know that there are options for when this assumption is not met!

A two sample t-test is often assumed to be easy. If you can collect data in a manner to test a fundamental hypothesis using t-tests, this is certainly a good thing. But don't let the workflow of a good analysis slip just because it is simply a comparison of two groups: always plot your data, evaluate assumptions, and only then interpret your analysis results.

5.3 General linear models

In the following section we introduce the analysis of a dataset combining continuous and categorical variables—an ANCOVA. When analysis needs go beyond two samples, and various other complications arise such as a factorial experimental design and/or categorical and continuous independent variables, techniques such as ANOVA and ANCOVA become useful. All these can be formulated with what are known as general linear models. A general linear model is an analysis tool that captures a set of analyses that includes regression, ANOVA, and ANCOVA.

This example will introduce the basics necessary to define, assess, and interpret a general linear model. Again, if you are not familiar with the

statistical foundation of these tools, you should take the time to become so. Our goal is to introduce to you the formulation and interpretation of such models in R.

The aims for this section are to introduce the implementation of a general linear model and to reinforce again our workflow: plot your data, make an appropriate model, evaluate the assumptions of the model, and make an interpretation. In doing so, we will help you understand exactly how the R output from these models relates to the figures you make and the hypotheses you are testing.

The dataset we will use is from Quinn and Keough's 2002 book *Experimental Design and Data Analysis for Biologists*; it is available on http://www.r4all.org. The data relate egg production by limpets to four density conditions in two seasons. The response variable (y) is egg production (EGGS) and the independent variables (x's) are DENSITY (continuous) and SEASON (categorical). Because we are examining egg production along a continuous density gradient, this is essentially a study of density dependent reproduction. The experimental manipulation of density was implemented in spring and in summer. Thus, a motivation for collecting these data could be "does density dependence in egg production differ in spring and summer"?

5.3.1 ALWAYS START WITH A PICTURE

Let's begin the analysis as we normally do—with a fabulous picture in R. Download the data from http://www.r4all.org and initiate a new script in R. Prepare your script with some annotation about the data (i.e. detail above). Import the data.

Examine the dataset using **str()**. You should notice that the dataset contains three columns, two that are numeric (EGGS and DENSITY) and a third that is categorical (SEASON). Recalling that the response (y) variable is egg production, that DENSITY is the continuous independent variable (x), and that the values are also classified by the category SEASON. This suggests that a plot of egg production, against density, distinguishing the two seasonal categories, is what we need. This is readily achieved using the plot() function and methods for colouring points by a categorical variable (see Chapter 4).

```
# plot window
par(mfrow = c(1,1))

# the plot
plot(EGGS ~ DENSITY, data = limp, pch = 19, cex = 1.5,
    col = c("Black","Red")[limp$SEASON],
    xlab = list("Density", cex = 1.2),
    ylab = list("Eggs Produced", cex = 1.2))

# add a legend
legend(35,3,legend = c("spring","summer"),
    col = c("black", "red"), pch = c(19, 19))
```

The code above uses a number of key ideas we've introduced in previous chapters, the most interesting being the use of [] to index a list of colours against the **levels** of a categorical variable. It produces the following:

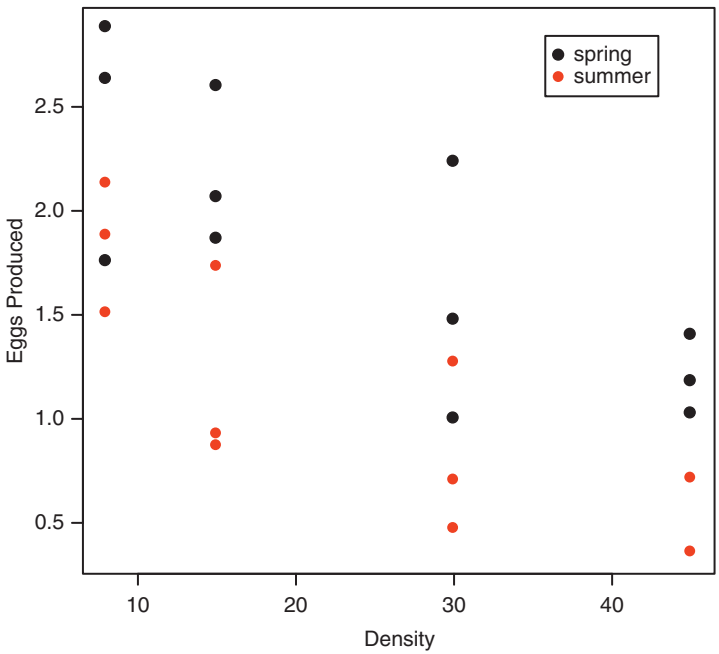


Figure 5.4 Always begin analyses with a picture. Our analysis of egg production by limpets in two seasons and at four limpet densities uses the plot() function with several arguments to produce this two colour plot with a legend showing the distinction between data collected in spring and summer.

Let's consider the pattern that we see in the picture. There is clearly a decline in egg production with increasing limpet density, and it looks as though, for any particular density, there is a tendency for the production to be higher in spring than in summer. So, just from the picture, we have extracted some pretty useful information. Now the challenge is how to test this more formally; in other words how to make a model for the data.

To think about how to do this, we can start by recalling the equation for a straight line: y = b + m*x. On our graph, if we were to describe the relationship between egg production and density by a straight line (ignore season for the moment), then y is egg production, x is density, b is where the line crosses the y-axis (i.e. the egg production at 0 density—admittedly an odd concept!), and m is the slope of the egg ~ density relationship. This slope represents the change in egg production per unit change in density—that is the rate of density dependence. m and b are the parameters of the line.

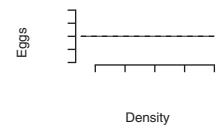
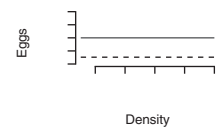
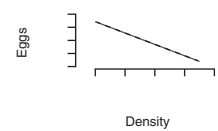
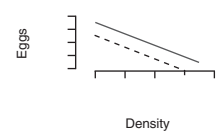
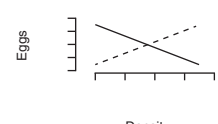
5.3.2 POTENTIAL STATISTICAL AND BIOLOGICAL HYPOTHESES—IT'S ALL ABOUT LINES

With the understanding that all lines are comprised of an intercept and a slope, we can reinterpret our figure assigning biological meaning to intercepts and slopes. First, we can note that overall, egg numbers decline with increasing density—that is, there is a negative slope. Second, we can begin to speculate that there is a seasonal difference—that is that the intercept, the value of egg production at zero density, is different in each season.

Once we start to look at the data plotted on the figure, we immediately start to identify the patterns that stand out, but in order to think about testing these, and interpreting what they mean, it is important to step back and consider all the possible patterns that we might have found, and the different hypotheses, or interpretations they represent. We can do this using the two concepts of slope and intercept of the lines describing the data. Table 5.1 shows the range of possibilities, and describe each in verbal terms, graphically, and as the equivalent model in R.

Looking at our data above (Fig. 5.4), we can walk through the different hypotheses in Table 5.1. We might ignore A and B—it seems clear from the data that there is at least a decline, on average, in egg production with den-

TABLE 5.1 Verbal, mathematical, R, and graphical interpretation of various hypotheses that translate into specific linear models.

	VERBAL HYPOTHESIS	INTERCEPT/ SLOPE	MODEL IN R	GRAPHICAL
A	The number of eggs produced by limpets does not vary with density or season	Common Intercept zero Slope(s)	lm(EGGS~1, data=limp)	
B	The number of eggs produced by limpets does not vary with density but is reduced in the summer season (dashed)	Different Intercepts zero Slope(s) Parallel Horizoltal Lines	lm(EGGS~SEASON, data=limp)	
C	The number of eggs produced declines with density, but the maximum number of eggs (intercept) and the rate (slope) does not vary with season	Same Intercept Same Slope Same Lines	lm(EGGS~DENSITY, data=limp)	
D	The number of eggs produced declines with density and the number of eggs at density zero (intercept) differ between seasons but the rate (slope) does not vary with season	Different Intercepts Same (negative) Slope Parallel lines	lm(EGGS~DENSITY+SEASON, data=limp)	
E	The number of eggs at density zero (intercept) and the rate (slope) vary with season	Different Intercept Different Slopes ‘’Crossing” Lines	lm(EGGS~DENSITY*SEASON, data=limp)	

sity. However, the higher the variation in egg production, the less likely we are to detect such a trend as significant. C is clearly possible; again, more-so if the variation in egg production in each season is high. D and E are compelling. In our data, if the slopes are indeed equivalent, scenario D would be the best explanation. However, if the slopes are different, and the lines for each season cross (even if they are both negative), then scenario E is a compelling, competing hypothesis. Because we can't actually see whether there are different slopes and intercepts, we can use statistics.

For a number of reasons, we'd like you to understand that the difference in model D and E is the presence (E) or absence (D) of one particular term: an interaction term that specifies that the slopes are different. This interaction is embodied in the following sentence: the effect of density on egg production depends on season.

Try looking at the figure and saying this out-loud. "The effect of density on egg production depends on the season." If we deconstruct this sentence, we can reveal the features of a line. "The effect of density on egg production" means the value of the slope(s). "depends on season" indicates that these values may be different depending on season. Of course, if the slope depends on season, we must also be estimating the effect of season, in other words estimating intercepts as well as slopes. In fact, what statistical modelling is doing is asking whether our data justify specifying more than one intercept and more than one slope. Put another way, we are asking: do we get a better description of the data by using more than one intercept and more than one slope?

At this stage we have a good figure (5.4) that reveals a pattern and allows us to speculate on a result—some of you might think that there is a common rate of density dependence (slope) while others might see an interaction—there is the distinct possibility that the effect of density on egg production depends on season.

Before we explore these alternatives, consider one more feature of these data: they were collected as part of a manipulative experiment. Philosophically, we may want to limit the number of competing hypotheses we consider (i.e. A–E) because when one designs an experiment, one usually has a

particular hypothesis in mind; we have an *a priori* hypothesis. In the case of these data, if we assume that the researchers were testing whether "The effect of density on eggs depends on the season", then in fact we are not equally interested in all the hypotheses: we are starting with the belief that density does have an effect on egg production, and we are focusing our investigation on whether this effect differs between seasons; that is we are really interested in whether the data are better described by model E, than by model D.

5.3.3 SPECIFYING THE MODEL

To specify a general linear model (regression, ANOVA, ANCOVA), and one that would capture any of the above statistical hypotheses, we recommend the function **lm**(), the workhorse of basic statistics in R. **lm()** takes as a minimum set of arguments one describing your model and one declaring your dataset. The model is formulated in exactly the same way that we specified the arguments in plot() using Plot Method 2 (see page 51), using the y~x formula format:

```
limp.mod <- lm(EGGS ~ DENSITY*SEASON, data = limp)
```

We have assigned the model returned by **lm()** to the object limp.mod. And we used a formula to specify the relationship between EGGS, DENSITY, and SEASON. The formula's right hand side expression is a sort of shorthand: it specifies, all at once, that we want to include an effect of DENSITY (main effect), an effect of SEASON (main effect), and the potential for the effect of DENSITY to depend on SEASON (interaction). The specification expands to the full model of DENSITY + SEASON + DENSITY*SEASON. Note the explicit argument called *data* and the use of the " * " to specify the model with an interaction.

We could specify a model that doesn't include all these things—the other examples in Table 5.1 illustrate what these would look like. But since we are interested in whether the effect of density depends on season, we need a model where there is, in addition to the effects of density and season on their own, a specific term allowing for their interaction, that is in which the effect of density on egg production "could" depend on season.

Why do we say "could"? We are fundamentally testing the null hypothesis (Table 5.1, E) that this interaction term is not significant: there are no differences in the slopes for each season; there is no extra variation explained by fitting different slopes. The alternative is that, by allowing for separate slopes to be fit, we explain more variation in the data. That's it. Said another way, we are asking if having explained a certain amount of variation with different intercepts and a common slope (i.e. hypothesis D), do we explain any more variation by allowing separate slopes?

5.3.4 PLOT, MODEL, THEN ASSUMPTIONS

If you run the code above, nothing much will seem to happen. All the good stuff has been done, but has been quietly collected and organized for you in the limp.mod object you specified.

As with all of the amazing R functions you have experienced thus far, by assigning the values returned by lm() to an object (limp.mod), we can see what has been collected for us:

```
names(limp.mod)
```

R has stored for us the coefficients (intercept and slope estimates), residuals, fitted values, and a number of other values. Peruse the help file for **lm**, specifically the Value section, to find out what **lm** can return to you.

So, all the results we want are inside limp.mod. The question is: what do we need to look at? What we do NOT do first is explore the modelling result. Instead, we MUST first evaluate the assumptions.

Now, some magic. The assumptions of a general linear model revolve around examining the residuals and fitted values. Two important assumptions that must be met before declaring your results as reliable are as follows: your residuals must be normally distributed and they must not be heteroskedastic. If you don't know what these mean or have relatively little experience with assessing assumptions of a general linear mode, head for the stats books such as Crawley 2005/2007, Faraway 2002, and Dalgaard 2008.

The way to examine these assumptions is to plot the residuals from your analysis in various ways. Fortunately R knows what you need. In fact, the brains behind R have happily embedded in the plot function a special little

"if-then" statement: *if* plot receives a model object—something produced by **lm**—it *then* produces diagnostic plots about the residuals and heteroskedasticity from that model object.

We can see the default 4 (of 6) plots that R produces with the following code:

```
par(mfrow = c(2,2)) # make a four panel plot window
plot(limp.mod) # add the four diagnostic plots to the window
```

This will produce Figure 5.5 for you:

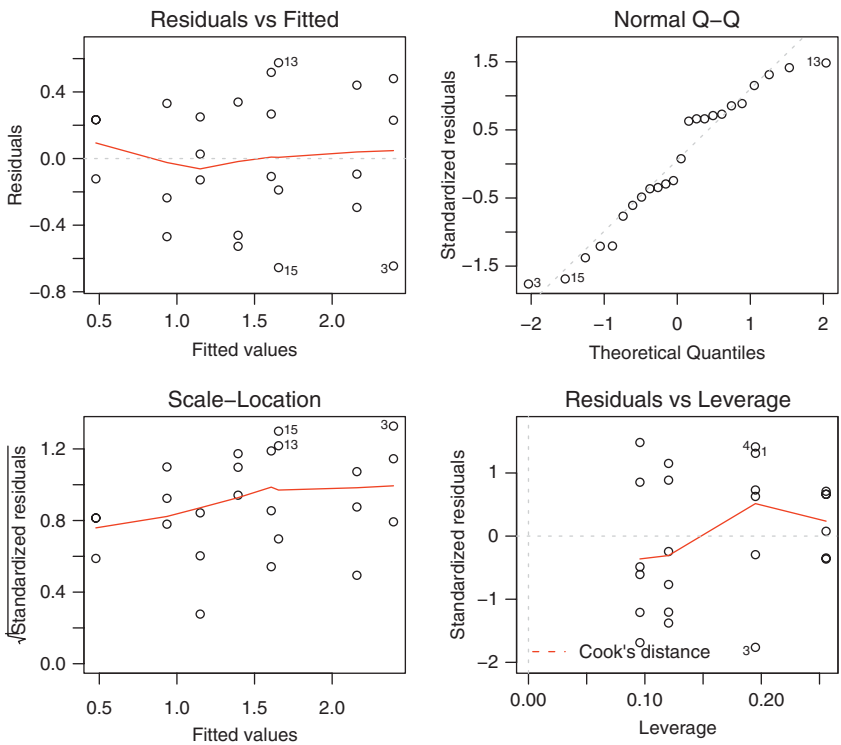


Figure 5.5 The default set of residual diagnostic plots for linear models. We combine the par(mfrow = c(2, 2)) code, producing a four panel plot, with the plot(model.name) code, to generate these four useful diagnostics about our model.

If, in addition, you'd like to examine a histogram of the residuals, your R programming skills should be twitching to write the following:

```
hist(limp.mod$resid)
```

which makes a histogram of the residuals value stored in the limp.mod object. How can you add this to the 4 x 4 plot you made above? Simply adjust the mfrow= part of your code to make a 2 x 3 (6 panel) rather than a 2 x 2 (4 panel) plot and add the above line to your earlier code, below the call to plot.

The diagnostics are quite satisfactory. The upper left panel, showing rather random distribution of the residuals above and below the reference line at 0, suggests that the residuals are homoskedastic—there is no pattern to the residuals. The right hand panel (a Q–Q plot) provides a test of the normality of the residuals. The dashed reference line corresponds to a normal distribution and, if the residuals lay along this line, then we would have no reason to be worried about the normality of the errors.

We will assume that you too find the diagnostics satisfying, illuminating, and downright good. We also hope you see how the plot function, taking a model as an argument, has returned a very helpful set of diagnostic plots. All you needed to do was to assign the values from your **lm** model to an object, and then plot it.

Before we head to our interpretation phase, let's review. We've made an insightful and useful figure representing the data and which helped us to formulate our ideas about how density and season might affect the egg production of limpets. We've made a model to capture our ideas, and more importantly, an experimental design. And we've evaluated the core assumptions associated with a general linear model. Now you're ready for interpretation.

5.3.5 INTERPRETATION

There are two functions to aid in interpretation of a general linear model: **anova()** and **summary()**. **anova()** produces an ANOVA table listing sums of squares, mean squares, F-values, and P-values. It DOES NOT perform an ANOVA. **summary()** does many things, as you might recall. When provided with a model object from **lm** as its argument, summary returns a table of coefficients (slopes and intercepts), standard errors, and t-values.

Together these two tables help us interpret the relationships between egg production and the combined or independent impact of density and season. Sounds easy? If you know how to interpret the output of other packages,

and understand contrasts and statistics deeply, it is. If you've never delved deeply into what a statistical package returns to you, hold on tight!

Let's start with **anova()**.

```
> anova(limp.mod)
Analysis of Variance Table

Response: Eggs
              Df  Sum Sq Mean Sq F value    Pr(>F)
Density        1  5.0241  5.0241 30.1971 2.226e-05  ***
Season         1  3.2502  3.2502 19.5350 0.0002637  ***
Density:Season 1  0.0118  0.0118  0.0711 0.7925333
Residuals     20  3.3275  0.1664
---
Signif.codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1.1
```

This table presents a sequential sums of squares analysis of variance table. This is interpreted as follows: first, we estimate a common slope (Density), using 1 df and this explains a certain amount of variation, captured in the Mean Sq value of 5.0241. Having done this, we then estimate different intercepts to the model (Season), and having done so, explain an additional amount of variation (3.2502). Finally, we allow the slopes to vary (Density:Season), and this explains an additional 0.0118 units of variation. This is the sequence of explanation.

Because we conducted, and are analysing, an experiment, we had a specific hypothesis in mind—that the effect of density on egg production depends on season. Testing this hypothesis is embodied in the Density: Season row.

This table reveals that there is no additional, significant variation in egg number explained by allowing different slopes for each season (i.e. the p-value in for the Density:Season row is 0.79). We accept the null hypothesis that the slopes are the same. We reject the alternative hypothesis that the effect of density on eggs depends on season! Our result could actually be written:

"We tested the hypothesis that the effect of density on egg production in limpets depends on the season in which they are reproducing. Data from a factorial, replicated experiment, manipulating density in spring and summer, provided no evidence that the effect of density on egg production depended on the season ($F=0.0711$, $df=1,20$, $p=0.79$)."

We could stop the analysis there. We have analysed data from an experiment designed to test the hypothesis, and R has provided an answer. However, we probably want more. Perhaps we want to know what the estimate of egg production at low density is in each season (intercepts; density independent egg production). We may also want to know what the rate of density dependence is.

We can use the summary table to identify all of this.

```
> summary(limp.mod)

Call:
lm(formula = EGGS ~ DENSITY * SEASON, data = limpet)

Residuals:
     Min       1Q   Median       3Q      Max
-0.65468 -0.25021 -0.03318  0.28335  0.57532

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     2.664166   0.234118  11.380 3.45e-10 ***
Density        -0.033650   0.008259  -4.074 0.000591 ***
Seasonsummer   -0.812282   0.331092  -2.453 0.023450 *
Density:
Seasonsummer    0.003114   0.011680   0.267 0.792533

---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.4079 on 20 degrees of freedom
Multiple R-squared: 0.7135, Adjusted R-squared: 0.6705
F-statistic: 16.6 on 3 and 20 DF, p-value: 1.186e-05
```

The output from **summary()**, when provided a model object (i.e. what **lm** returns) has four sections. First, it provides a restatement of your model by repeating your usage of **lm**—the call. Next it provides the range, interquartile range, and median of the residuals as a quick, additional check of the residuals. These should not be relied upon as a sole diagnostic and should be used with the more substantial plotting methods described above.

The next section of the output is the coefficients table. This is one of the most interesting tables a statistics package can produce as it provides, one way or another, insight into the estimates for the lines that you have fitted—these are the coefficients that quantify your hypothesis and explain variation in the response variable. If you have used other packages, you will have seen similar tables. Here is how to interpret the table in R.

5.3.6 TREATMENT CONTRASTS AND COEFFICIENTS

First, some background. R works alpha-numerically. That is to say that when presenting things, R will order them alphabetically and numerically. In this current example we should thus expect to see, for example, the season "spring" reported before the season "summer", because spring comes before summer. We like saying that.

R calculates these coefficients using **treatment contrasts**. Contrasts are a way of expressing coefficients from statistical models, and there are many types (see Crawley 2008, Venables and Ripley 2002, and Harrell 2001 books). Understanding what type of contrast is used by a statistical package is required for interpreting the summary table. This is also important when you make comparisons between the results from R and other packages. We suggest that those of you who are transitioning from other packages take a good look at:

```
?contr.treatment
```

as this help file will explain R's contrasts and also reveal methods for making R produce output that matches other statistical packages.

5.3.7 INTERPRETATION

Let's start our interpretation from the bottom, where we will find some generic statistics and a commonly used estimate of variance explained by the model, R^2. The model we have fitted explains 67% of the variation in egg production, and has a significant fit to the data ($F_{3,20}$ =16.6, $p<0.001$), leaving a residual standard error of 0.4079 on the original 20 degrees of freedom corresponding to our overall sample size.

Now the fun part. First, recall that R behaves alpha-numerically. Spring comes before summer. We told you we like saying that. Second, note that R has labelled aspects of rows three and four with the word summer—this should give something away.

Looking closely at the top two rows, we can see two words in the left-most column under the word coefficients: Intercept and Density. The equation for the line we are fitting is Eggs = Intercept (b) + slope (m) * Density. This should be reassuring. R has told us that here is the estimate of an intercept and a slope. And, we know that it is the intercept and slope estimate for the spring data. Spring comes before summer.

Specifically, the model (equation for a line) for egg production in spring is:

$$EGGS_{spring} = 2.66 - 0.033 \times Density$$

To confirm this, we've re-plotted the data and added this spring line (Fig. 5.6). We'll return to adding lines to the plot shortly:

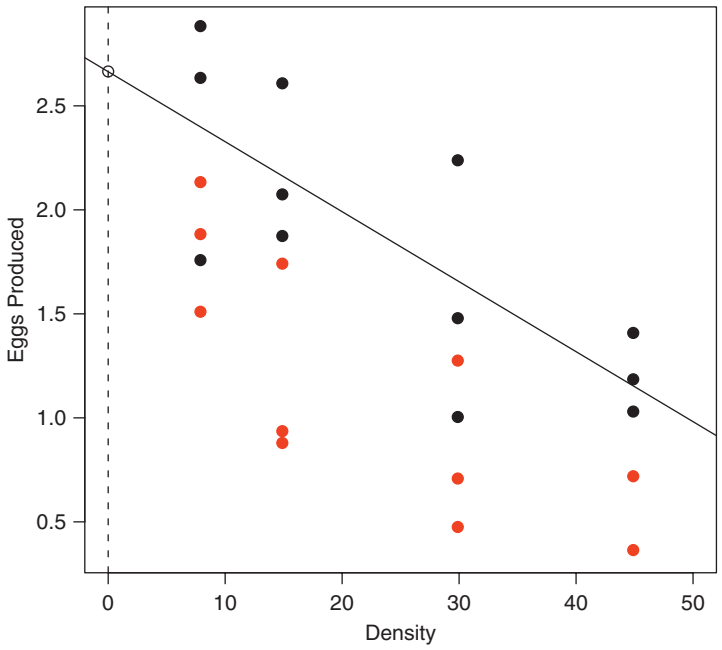


Figure 5.6 Adding a fitted line to the raw data on egg production by limpets. Here we have added a black line with slope and intercept fitted by our model for spring. The dashed vertical line is at x=0 allowing us to see the intercept (open circle).

Looking at the rows three and four, we might wonder what we are supposed to do. The third line, labelled SEASONsummer is very specifically the difference between the spring and the summer intercept. More biologically, it is the change in egg production that arises from shifting the season from spring to summer. The value is – 0.812 eggs. If this is the difference between spring and summer intercepts, then adding this number of eggs to the estimate of the intercept for spring should provide our estimate of the intercept for summer.

Likewise, the fourth line is the difference between the slope for spring and summer. More biologically, it is the change in the rate of density dependence that arises from shifting from spring to summer. The value is 0.003. If this is the difference in slopes, then adding this number to our estimate of the slope for spring should provide our estimate for the summer slope. Here is the maths:

$$EGGS_{spring} = 2.66 - 0.033 \times Density$$
$$EGGS_{spring} = 2.66 + (-0.033) \times Density$$
$$EGGS_{summer} = (2.66 - 0.812) + (-0.033 + 0.003) \times Density$$
$$EGGS_{summer} = (1.84) + (-0.03) \times Density$$
$$EGGS_{summer} = 1.84 - 0.03 \times Density$$

We can add both of these lines to the plot and identify the intercepts (Fig. 5.7).

We also highlight the t-values and p-values in this table. We mentioned above that a t-test essentially evaluates whether two values are different by asking if the difference between the two values differs from zero. Here, the treatment contrasts, comparing for example the difference in intercepts for each season, makes sense. The t-value and p-value reveal whether the season intercepts and slopes differ. If the t-value is small and the p-value large, we are unable to reject the null hypothesis that the two values are the same (there is no difference). Critically, we can reject this null hypothesis for intercepts, but not slopes.

So, via both the summary table and Fig. 5.7, we can draw the following conclusions. Summer reduces egg production in limpets, on average, by 0.812 eggs. In summer, the rate of decline in egg production

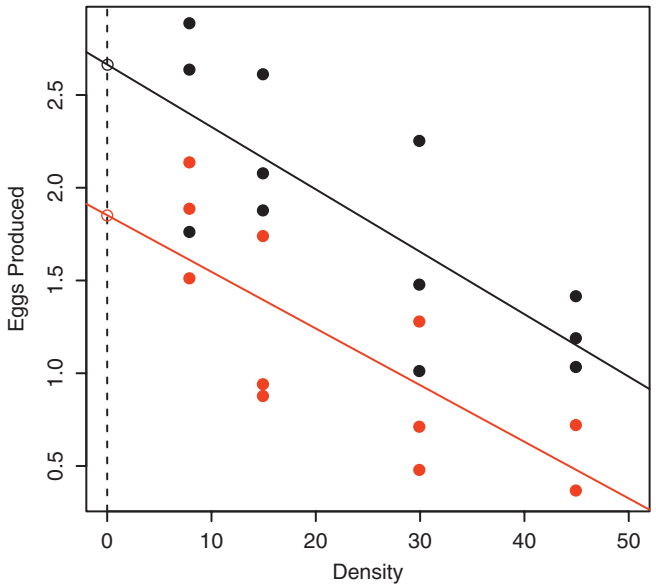


Figure 5.7 Adding the second fitted line to the limpet data. The red line is generated with the fitted intercept and slope for summer. The dashed vertical line is at x=0 allowing us to see the intercept for both seasons (open circles).

with density was slightly less than in spring (+0.003 eggs/density) but there was no way we could conclude that this change was different from 0. Finally, there is no evidence that the effect of density depends on season. Having *a priori* defined the hypothesis we were testing as "the effect of density on egg production depends on season", with an associated null hypothesis that it does not, we find that we cannot reject the null hypothesis. There is no evidence in these data that the effect of density on egg production depends on season.

5.4 Making a publication quality figure

Our final installment centres on two methods to craft a figure that combines the raw data with model fits—that is the lines estimated by the ANCOVA. The method builds on your understanding that R provides you,

in the summary table, with the coefficients for two lines. The second is a much more generic method, useful for models of infinite complexity (i.e. more than two lines, non-linear models etc).

5.4.1 COEFFICIENTS, LINES, AND LINES()

R provides, as one of the values returned by the function **lm**, an object called coefficients. You can find it by looking at the names of your model:

```
> names(limp.mod)
 [1] "coefficients"  "residuals" "effects" "rank"
 [5] "fitted.values" "assign"    "qr"      "df.residual"
 [9] "contrasts"     "xlevels"   "call"    "terms"
[13] "model"
```

You can isolate the coefficients using the function coef(), or the $ to grab a part of the list, or [ ] specifying the number in the list.

```
> coef(limp.mod)
          (Intercept)          DENSITY   SEASONsummer
          2.664166462   -0.033649651   -0.812282493
DENSITY:SEASONsummer
          0.003113571
> limp.mod$coefficients
          (Intercept)          DENSITY   SEASONsummer
          2.664166462   -0.033649651   -0.812282493
DENSITY:SEASONsummer
          0.003113571
```

A quick flick back to section 5.3.5 will confirm that these are the same numbers provided by summary(limp.mod).

Now, what do we do? Let's return to our initial plot:

```
plot(EGGS ~ DENSITY, data = limp, pch = 19, cex = 1.5,
    col = c("Black","Red")[limp$SEASON],
    xlab = "Density", ylab = "Eggs Produced")
```

Now we add some lines. Here we use the function **abline()** to add lines. **?abline** will reveal that this function accepts a list of values that comprise the intercept and slope of a line. You also need to recall the use of [] brack-

ets to identify specific elements in a list, in this case, the list of coefficients gathered from the function **coef()**. We can use this combination of functions to draw the lines.

First, we note that **coef**(limp.mod) returns a list of four coefficients. The first and second of these correspond to the intercept and slope of the line for spring. The first and second coefficients can be obtained with:

```
coef(limp.mod)[1]
```

and

```
coef(limp.mod)[2]
```

Embedding these as arguments in the function **abline** will create the spring line:

```
abline(coef(limp.mod)[1], coef(limp.mod)[2])
```

The third and fourth coefficients returned by **coef** are the differences in intercept and slope, respectively, between summer and spring, so adding them to the coefficients for spring generates the intercept and slope for summer. Make sure you understand this.

Again, embedding these as arguments within the function abline produces the line for summer:

```
abline(coef(limp.mod)[1] + coef(limp.mod)[3], # summer intercept
    coef(limp.mod)[2] + coef(limp.mod)[4], # summer slope
    col="red")
```

This results in Fig. 5.8.

5.4.2 EXPANDED GRIDS, PREDICTION, AND A MORE GENERIC MODEL PLOTTING METHOD

The above method is intuitive (once you embed the equation for a line in your head) and useful, but becomes cumbersome and challenging when there are more than two predictor variables or more than two levels to a categorical factor. It is also less than intuitive if you want to, for example, fix one of the variables at its mean and evaluate the response variable. For example, perhaps we'd like to identify what the

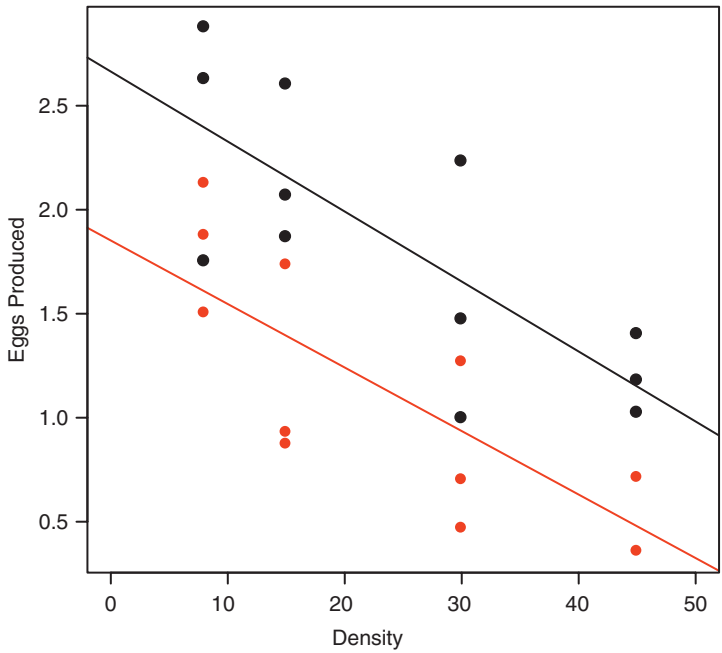


Figure 5.8 Fitted lines for both seasons added to the raw data for egg production by limpets. These lines are produced using the function abline() with the coefficients of the model as the arguments. This produces lines that extend beyond the range of the data, which is not necessarily good.

value of egg production is for spring and summer at, say, the mean density in the experiment.

A more generic method for plotting and evaluating the results of the modelling relies on the following logic. R has fit a model and has in its brain the coefficients for the lines. In order to plot lines, we need to create (predict) some "y" values from a sensible range of "x" values, with the coefficients defining the new y values. To do this, we invoke two functions: **expand.grid()** and **predict()**.

expand.grid() is a function that generates a grid of numbers—essentially it builds a factorial representation of any variables you provide to it. For example:

```
expand.grid(FIRST = c("A","B"), SECOND = c(1,2))
```

returns the following, where the first and second column are each of the variables and the grid is expanded like a factorial design:

```
  FIRST SECOND
1     A      1
2     B      1
3     A      2
4     B      2
```

predict() is a function that generates fitted values from a model. It requires at least one argument, but works most effectively for us with at least two: a model object and a set of "x" (explanatory) values at which we want to know "y" values, generated according to our model.

If we use **predict()** with only a model object as an argument, it returns to us the predicted value of "y" at each of the "x" 's in our original data frame. For our limp.mod model and original data frame of 2 seasons x 4 density treatments x 3 replicates, we should see 24 predicted values returned, with sets of 3 (the replicates) being identical:

```
predict(limp.mod)
       1        2        3        4        5        6        7
2.3949692 2.3949692 2.3949692 1.6075953 1.6075953 1.6075953 2.1594217
       8        9       10       11       12       13       14
2.1594217 2.1594217 1.3938428 1.3938428 1.3938428 1.6546769 1.6546769
      15       16       17       18       19       20       21
1.6546769 0.9358016 0.9358016 0.9358016 1.1499321 1.1499321 1.1499321
      22       23       24
0.4777604 0.4777604 0.4777604
```

However, let's assume that we would specifically like a prediction of the number of eggs at a set of specific densities. First, we create the "new x's" (density) at which we want values of y (egg production). Here we use **expand.grid()** to generate a set of "new x" values, that we take special care to label as the column name in the original dataset, DENSITY. Note the use of **unique()** to grab the unique densities from the experiment: 8,15,30,45.

```
new.x <- expand.grid(DENSITY = unique(limp$DENSITY))
```

```
> new.x
  DENSITY
1       8
2      15
3      30
4      45
```

Let's add SEASON to this grid now. SEASON has two "levels", spring and summer. We can add these levels to the "new x"'s as follows. Note how we use the functions **unique()** and **levels()**, against the dataset, to extract the information we desire:

```
new.x <- expand.grid(DENSITY = unique(limp$DENSITY),
   SEASON = levels(limp$SEASON))
```

```
> new.x
     DENSITY   SEASON
1          8   spring
2         15   spring
3         30   spring
4         45   spring
5          8   summer
6         15   summer
7         30   summer
8         45   summer
```

Notice that **expand.grid()** has now created a factorial representation of DENSITY (4 levels) and SEASON (2 levels), such that each and every DENSITY:SEASON combination is represented. Now, we can embed these "new x"'s into the function **predict()**, in the argument known as…wait… *newdata*.

We will use **predict()** with two arguments: a model and a value for newdata, created using **expand.grid**.

```
> predict(limp.mod, newdata=new.x)
       1         2         3         4         5         6
2.3949692 2.1594217 1.6546769 1.1499321 1.6075953 1.39384280
       7         8
0.9358016 0.4777604
```

Notice that we now have only eight values returned—corresponding to the two seasons x four density factorial expansion we specified with **expand.grid()**. It is important to recognize that we now have predicted values of egg production at each of four densities for each season. It is also vital that you recall that you can assign these numbers to an object, like the word "predictions".

Housekeeping is an important part of using R and at this point in the plotting cycle, we advocate a bit of it. What we suggest is that you combine the predictions (new y's) with the new x's, so that we have a clear picture of what it is you've made. We can do that with the function **data.frame()**. Here we use the code we explained above to create an object of the eight predictions ("predictions") and then combine the with the new x's in a data frame:

```
predictions <- predict(limp.mod, newdata = new.x)
preds.for.plot <- data.frame(new.x, predictions)
```

```
> preds.for.plot
      DENSITY     SEASON     predictions
1           8     spring      2.3949692
2          15     spring      2.1594217
3          30     spring      1.6546769
4          45     spring      1.1499321
5           8     summer      1.6075953
6          15     summer      1.3938428
7          30     summer      0.9358016
8          45     summer      0.4777604
```

So, now you have a new, small data frame that contains the grid of seasons and densities, as well as the predicted values at each of these combinations; we've called it preds.for.plot. You did not need to specify the coefficients, or the equations for the lines. **predict()** does all of that for you.

Before we finish with adding these data to the figure, we want to show how flexible this method is. Let's modify the code so that we get the predicted number of eggs at the mean density for each season.

```
new.x <- expand.grid(DENSITY = mean(ll$DENSITY),
   SEASON = levels(ll$SEASON))
predictions <- predict(limp.mod, newdata = new.x)
preds.for.plot <- data.frame(new.x, predictions)
```

```
> preds.for.plot
    DENSITY   SEASON    predictions
1   24.5      spring    1.83975
2   24.5      summer    1.10375
```

The anatomy of the above code is simple—the only change in our code is in the new.x <- expand.grid() section. The original code was:

```
new.x <- expand.grid(DENSITY = unique(limp$DENSITY),
   SEASON = levels(limp$SEASON))
```

while the new code is:

```
new.x <- expand.grid(DENSITY = mean(limp$DENSITY),
   SEASON = levels(limp$SEASON))
```

Did we rewrite all of this? Nope. All we did was replace the function **unique()** with the function **mean()**.

5.4.3 THE FINAL PICTURE

You are nearly there. You've carried out quite a substantial ANCOVA of an experiment. You've plotted the data, you've made a model, you've checked the assumptions, and interpreted the model. You've got some predictions as well, detailing how the model has fitted lines to your data. The final step, the last effort you may want to make, is to add these lines to the figure. An informative figure is worth a thousand words. If you can provide a figure that a reader looks at and as a result knows the question AND the answer to the question, you will have communicated science effectively.

The functions **lines()** and **points()** are your friends at this stage. We have an original plot, and some additional data and functions to add data to an existing plot.

Here is the last bit of code to do this:

```
# raw data plot (you don't need to write this again...)
plot(EGGS ~ DENSITY, data = ll, pch = 19, cex = 1.5,
    col = c("Black","Red")[ll$SEASON],
    xlab = list("Density", cex = 1.2),
    ylab = list("Eggs Produced", cex = 1.2),
    xlim = c(0,50)

# add lines using preds.for.plot data.frame
lines(predictions ~ DENSITY,subset(preds.for.plot,
  SEASON == "spring"))
lines(predictions ~ DENSITY,subset(preds.for.plot,
  SEASON == "summer"), col = "red")
```

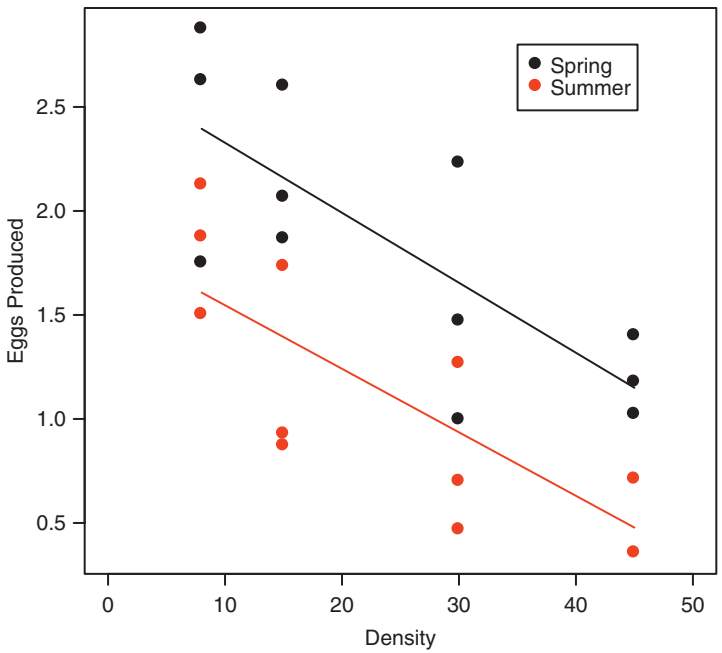


Figure 5.9 Fitted lines for both seasons added to the raw data for egg production by limpets. These lines are produced using a workflow that involves making predictions using the model via the function predict(). This produces lines that do not extend beyond the range of the data, which is good. The workflow to prepare this figure is very general and can be applied to any linear model. See text for details.

You've seen the raw data plot now a few times. The last two lines use the function **lines()** to add a **subset** of the preds.for.plot data frame, where the rows we plot correspond to each Season. We've made the colour of the second line red. This code produces the plot you've seen (Fig. 5.8), with two changes (Fig. 5.9).

In previous versions of the plot, the fitted lines extend to the edges of the figure. Here, they extend only from the lowest to highest value of DENSITY (x-axis). This is more appropriate in the context of general linear models where one is cautioned against extrapolating beyond the range of the raw data. Finally, we've added an informative legend (can you?).

The entire analysis script is presented in **Box 5.2**. The script is short and provides an archived (save it!), annotated (for you and your colleagues), and cross-platform record of your analysis.

5.4.4 AN ANALYSIS WORKFLOW

Hopefully, it is clear that we have introduced you to a workflow for the analysis of your data using R. There are nine steps in this workflow:

1. Enter your data in a spreadsheet program, check it, and save it as a comma separated values file.
2. Start R and wipe its brain of all previous data—**rm(list=ls())**.
3. Import your data into R using **read.csv()** and check this has worked, for example using **str()**.
4. Make a picture that will help you specify your model and answer your question.
5. Specify a statistical model to capture the hypothesis you are testing.
6. Assess whatever assumptions are important to the type of modelling you have decided to implement.
7. Interpret the model with **anova()** and **summary()**.
8. Add fitted lines to the plot; make the figure that is worth 1,000 words.
9. Keep your data file and script file very safe (and backed up in a separate location). With these, you can recreate all your graphs and analyses with a few keystrokes in a few minutes at most.

Box 5.2: The final ANCOVA script

```
# Read the limpet data and check the structure
limp <- read.csv("limpet.csv")
str(limp)

# make the first plot to explore the data
plot(EGGS ~ DENSITY, data = limp, pch = 19, cex = 1.5,
  col = c("Black", "Red")[limp$SEASON],
  xlab = list("Density", cex = 1.2),
  ylab = list("Eggs Produced", cex = 1.2))

# make the ANCOVA model using lm; confirm what values are returned
limp.mod <- lm(EGGS ~ DENSITY*SEASON, data = limp)
names(limp.mod)

# check the diagnostic plots for this model
par(mfrow = c(2, 2)) # prepare 2 x 2 grid to receive 4 default plots
plot(limp.mod)

# use anova() and summary() to interpret the model
anova(limp.mod) # sequential sums of squares table
summary(limp.mod) # coefficients table

# re-make the figure and add fitted lines
# FIRST—new x values—where do we want predicted eggs estimated?
new.x <- expand.grid(DENSITY = unique(limp$DENSITY),
    SEASON = levels(limp$SEASON))

# use predict() and data.frame() to generate the new y's
# collect them (housekeeping) for plotting in object called preds.for.plot
preds <- predict(limp.mod, newdata = new.x)
preds.for.plot <- data.frame(new.x, predictions)
preds.for.plot # make sure it worked

# remake the figure and add lines
plot(EGGS ~ DENSITY, data = limp, pch = 19, cex = 1.5,
  col = c("Black", "Red")[limp$SEASON],
  xlab = list("Density", cex = 1.2),
  ylab = list("Eggs Produced", cex = 1.2),
  xlim = c(0,50))
```

```
# add lines using lines() and plot.these data.frame
lines(predictions ~ DENSITY, subset(preds.for.plot,
  SEASON == "spring"))
lines(predictions ~ DENSITY, subset(preds.for.plot,
  SEASON == "summer"), col = "red")

# add a legend
legend(x = 35, y = 2.8, pch = 19, col = c("black", "red"),
  legend = c("Spring", "Summer"))
```

If you adhere to this basic workflow for data analysis, you will have a very solid and efficient foundation for using R. Remember that nothing beats an effective picture for communicating your findings, and that the assumptions are just as important as the predictions.

Final Comments and Encouragement

So, you really do know how to use R. You have acquired skills that allow you to import data, summarize and explore it, and plot it in a wide range of formats. You have acquired skills as well that allow you to use R to do basic statistics. Throughout, we have ensured that you understand the value of writing down and saving the instructions you want R to carry out in a script. As a result, you end up with *permanent, repeatable, annotated, shareable, archive of your analysis*. Your entire analysis, from transferring your data from notebook, to making figures and performing analyses is all in one, secure, repeatable, annotated place.

We also have introduced you to a very fundamental rule for data analysis. NEVER start an analysis with statistical analysis. ALWAYS start an analysis with a picture. Why? If you have done a replicated experiment, conducted a well organized, stratified sampling programme, or generated data using a model, it is highly likely that some theoretical relationship underpinned your research effort. You have in your head an EXPECTED pattern in your data. Make the picture that should tell you the answer—make the axes correspond to the theory. If you can do this, AND see the pattern you EXPECTED, you are in great shape. You will know the answer!

In the previous chapter, we introduced a 9 step workflow centred on statistical analysis. Here we expand this workflow to 17(!) steps. The additional steps here include such things as snacks and organization.

1. On your computer, create a folder for your project.
2. Enter your data into a spreadsheet (e.g. Excel) with observations in rows and variables in columns. Save it to your new folder.
3. Print a hard copy of the spreadsheet and check it against the original datasheets. Correct any errors in the spreadsheet.
4. Save the spreadsheet in comma separated values (.csv) format.
5. Protect the spreadsheet and csv file by making them read-only files.
6. Take a coffee, tea, or other beverage break.
7. Start up R, open a new script file, and save it to the folder you created in step 1. Write your instructions for R in this script file, and save it regularly (really!). This script file and your data file are all that you need to keep safe (but you need to keep them really safe!).
8. Always put comments in your script.
9. Import your data into R. Check that the number of rows and variables is correct. Check that variable types are correct. Check the number of levels of categorical variables is correct. Check that the range and distribution of numeric variables are sensible. Check for any missing values and make sure you know where they are. Check how many data points per treatment combination. Check everything!
10. Don't forget to put comments in your script.
11. Have a snack break.
12. Explore your data, primarily by using R's awesome plotting functions. Package lattice will be particularly useful at this stage. Guess the answer that your statistical tests should give you. Make some other graphs to further investigate your guesses. Make some more figures, just to be sure.

13. Use a statistical test to check if your guess is right or wrong. If it's right, rejoice and publish. If it's wrong, figure out why.
14. Don't forget to put comments in your script.
15. Communicate your findings to your colleagues, in an email, report, manuscript, poster, web page, etc.
16. Go back and organize the files in your folder and the script you've written. Leave it all in a state that you'll be happy to come back to in six months time—that's a long time from now, so take your time doing this.
17. Retire to your preferred socializing establishment. Try to not bore your non-R friends by talking about R all night.

If you stick to this basic workflow you will have a very solid and efficient foundation for using R. Yes, there are many ways you could go about doing what we described above. However, why not just use one way and keep your R life simple? Make all of your scripts and analyses cover each of these steps. Make sure you annotate your script so that each of these sections is clear. If you do all of this, you will be an efficient and happy R user. And you, your friends and colleagues and editors will appreciate all of the effort when you need to revisit, share, or publish your analysis.

Happy R-ing.

This page intentionally left blank

Appendix: References and Datasets

Suggested key resources and sources of datasets.

Author	Title	Web/Package
Crawley (2005)	*Statistics: An Introduction Using R*	http://www.bio.ic.ac.uk/research/crawley/statistics/
Dalgaard (2008)	*An Introduction to R*	package: ISwR
Maindonald and Braun (2003)	*Data Analysis and Graphics Using R*	package: DAAG
Fox (2010)	*An R Companion to Applied Regression*	package: car
Crawley (2007)	*The R Book*	http://www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm
Harrell (2001)	*Regression Modelling Strategies*	packages: Hmisc; rms
Faraway (2002)	*Linear Models in R*	package: Faraway
Venables and Riply (2004)	*MASS*	pcakage: MASS
Original Datasets		
Crawley	compensation data, ozone data	http://www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm
Quinn and Keogh	limpet data	http://www.zoology.unimelb.edu.au/qkstats/

This page intentionally left blank

Index

Page numbers in *italics* refer to illustrations, those in **bold** refer to boxes