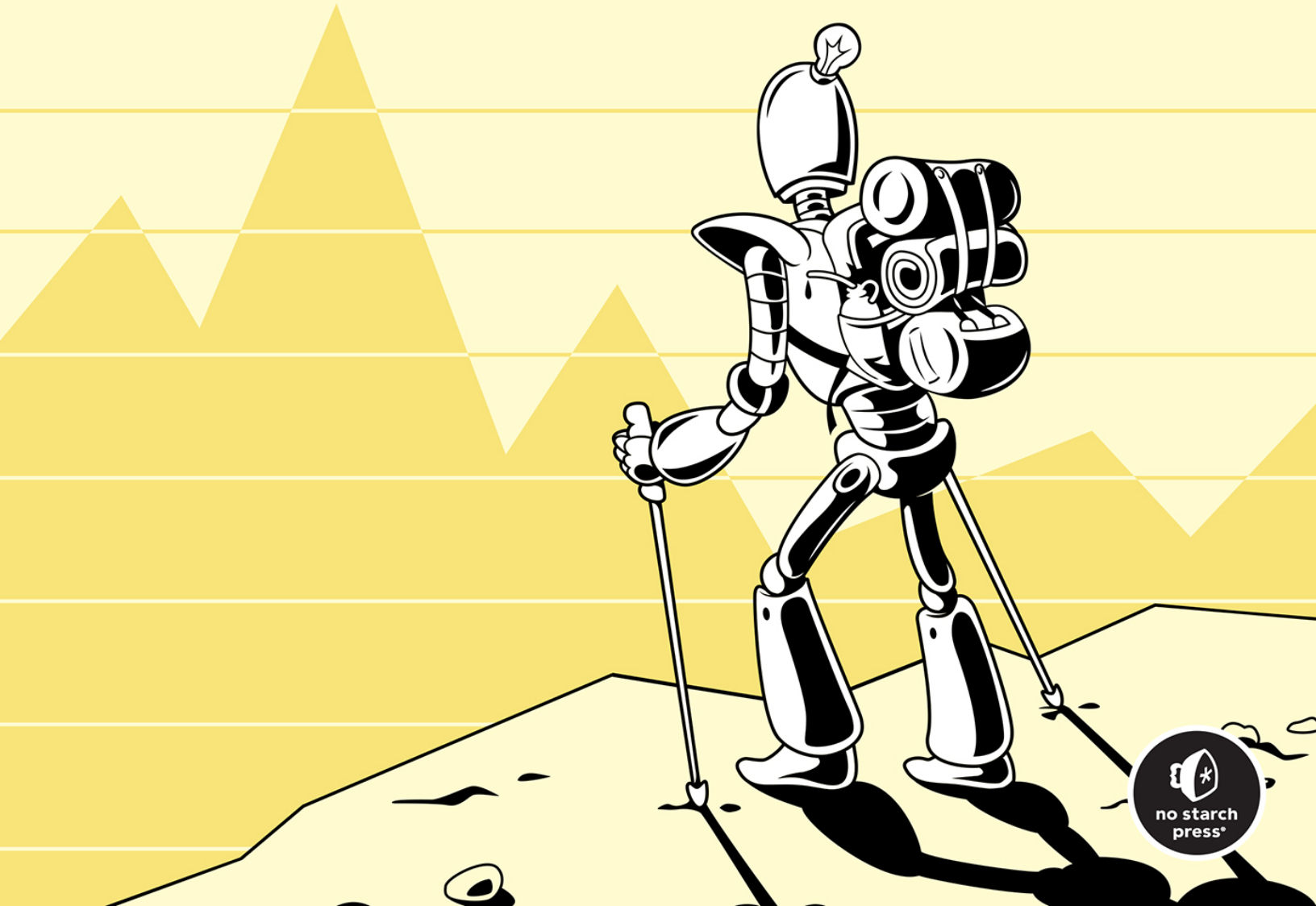
SECOND EDITION

THE BOOK OF R

A FIRST COURSE IN *PROGRAMMING AND STATISTICS*

TILMAN M. DAVIES



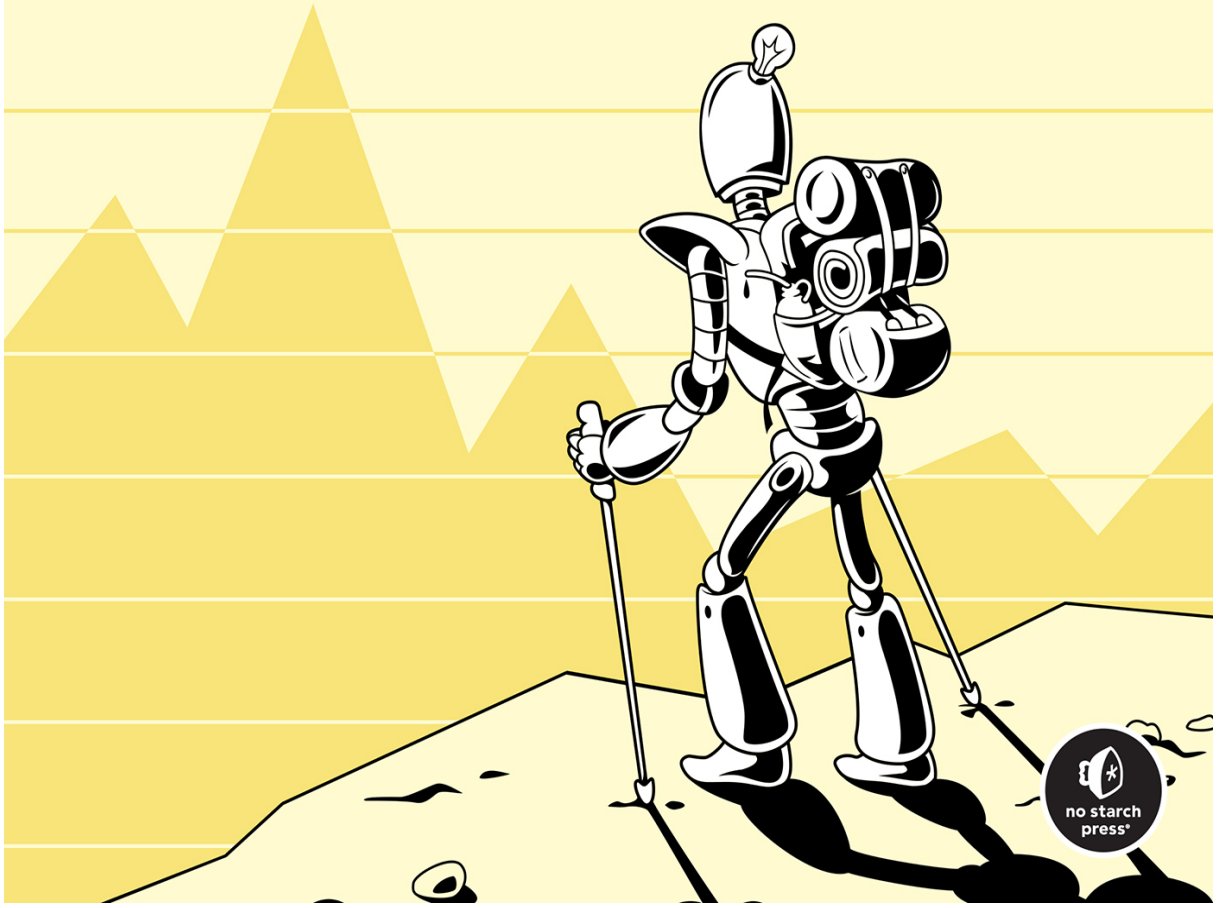
no starch press®

SECOND EDITION

THE BOOK OF R

A FIRST COURSE IN
PROGRAMMING AND STATISTICS

TILMAN M. DAVIES



no starch press®

THE BOOK OF R 2ND EDITION

A First Course in Programming and Statistics

by Tilman M. Davies



no starch press®

San Francisco

THE BOOK OF R, 2ND EDITION. Copyright © 2026 by Tilman M. Davies.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

29 28 27 26 25 1 2 3 4 5

ISBN-13: 978-1-7185-0368-7 (print)
ISBN-13: 978-1-7185-0369-4 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA
94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Cover Illustration: Josh Ellingson
Developmental Editor: Eva Morrow
Technical Reviewer: Debbie Leader
Proofreader: Scout Festa
Indexer: BIM Creatives, LLC

The Library of Congress has catalogued the first edition as follows:

```
Names: Davies, Tilman M., author.
Title: The book of R : a first course in programming and statistics / by Tilman
M. Davies.
Description: San Francisco : No Starch Press, [2016] | Includes bibliographical
   references and index.
Identifiers: LCCN 2015035305| ISBN 9781593276515 | ISBN 1593276516
Subjects: LCSH: R (Computer program language) | Computer programming. |
   Statistics--Data processing.
Classification: LCC QA76.73.R3 D38 2016 | DDC 519.50285--dc23
LC record available at http://lccn.loc.gov/2015035305
```

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Prof. Dr. Dr. Jochen Weniger, 1925–2015

Ein Wissenschaftler. Mein Opa.

About the Author

Tilman M. Davies is an academic at the Department of Mathematics and Statistics at the University of Otago, New Zealand, where he lectures on statistics and R at all university levels. His research, which lies primarily in spatial statistics, is regularly published in leading scholarly journals and has to date been awarded several prestigious Marsden Fund grants from the Royal Society of New Zealand. He is the lead handling editor of the Statistical Computing section of the *Australian and New Zealand Journal of Statistics*, and *The Book of R*, 2nd edition, is a reflection of his ongoing enthusiasm for the language as the most popular tool for statistical modeling and data science worldwide.

About the Technical Reviewer

Dr. Debbie Leader is a senior tutor in statistics at Massey University in New Zealand. She is passionate about teaching the fundamentals of statistics, including guiding students to develop experience with R so that they can appreciate it as a valuable component in their statistical toolbox. Debbie has been using and teaching R for over 15 years.

BRIEF CONTENTS

Preface to the Second Edition
Preface to the First Edition
Acknowledgments
Introduction

PART I: THE LANGUAGE

Chapter 1: Getting Started
Chapter 2: Numerics, Arithmetic, Assignment, and Vectors
Chapter 3: Matrices and Arrays
Chapter 4: Non-Numeric Values
Chapter 5: Lists and Data Frames
Chapter 6: Special Values, Classes, and Coercion
Chapter 7: Basic Plotting
Chapter 8: Reading and Writing Files

PART II: PROGRAMMING

Chapter 9: Calling Functions
Chapter 10: Conditions and Loops
Chapter 11: Writing Functions
Chapter 12: Exceptions, Timings, and Visibility

PART III: STATISTICS AND PROBABILITY

Chapter 13: Elementary Statistics
Chapter 14: Basic Data Visualization
Chapter 15: Probability
Chapter 16: Common Probability Distributions

PART IV: STATISTICAL TESTING AND MODELING

Chapter 17: Sampling Distributions and Confidence
Chapter 18: Hypothesis Testing
Chapter 19: Analysis of Variance
Chapter 20: Simple Linear Regression
Chapter 21: Multiple Linear Regression
Chapter 22: Linear Model Selection and Diagnostics

PART V: ADVANCED GRAPHICS

Chapter 23: Advanced Plot Customization
Chapter 24: Going Further with the Grammar of Graphics
Chapter 25: Defining Colors and Plotting in Higher Dimensions
Chapter 26: Interactive 3D Plots

Appendix A: Installing R and Contributed Packages
Appendix B: Working with RStudio

References
Index

CONTENTS IN DETAIL

PREFACE TO THE SECOND EDITION

PREFACE TO THE FIRST EDITION

ACKNOWLEDGMENTS

INTRODUCTION

A Brief History of R
About This Book
What's New in the Second Edition
Conventions and Styles
    Coding
    Math and Equation References
    Exercises
For Students
For Instructors
Web Resources

PART I
THE LANGUAGE

1
GETTING STARTED

1.1 Obtaining and Installing R from CRAN
1.2 Opening R for the First Time

1.2.1 Base R GUI
1.2.2 RStudio and Other Third-Party Editors
1.3 Workflow Basics
1.3.1 Console and Editor Panes
1.3.2 Code Execution
1.3.3 Comments
1.3.4 Working Directory
1.3.5 R Packages
1.3.6 Help Files and Function Documentation
1.4 Saving Work and Exiting R
1.4.1 Workspaces
1.4.2 Scripts

2
NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS
2.1 R for Basic Math
2.1.1 Arithmetic
2.1.2 Logarithms and Exponentials
2.1.3 E-Notation
2.2 Assigning Objects
2.3 Vectors
2.3.1 Vector Creation
2.3.2 Sequences, Repetition, Sorting, and Lengths
2.3.3 Subsets and Element Extraction
2.3.4 Vector-Oriented Behavior

3
MATRICES AND ARRAYS
3.1 Two-Dimensional Matrix
3.1.1 Matrix Creation

3.1.2 Subsets
3.1.3 Matrix Operations and Algebra
3.2 Multidimensional Arrays
3.2.1 Array Creation
3.2.2 Subsets, Extractions, and Replacements

4

NON-NUMERIC VALUES

4.1 Logical Values
4.1.1 TRUE or FALSE?
4.1.2 Relational Operators
4.1.3 Logical Operators
4.1.4 Logicals Are Numbers!
4.1.5 Logical Subsetting and Extraction
4.2 Characters
4.2.1 String Creation
4.2.2 Concatenation
4.2.3 Escape Sequences
4.2.4 Substrings and Matching
4.3 Factors
4.3.1 Identifying Categories
4.3.2 Defining and Ordering Levels
4.3.3 Dropping Categories
4.3.4 Working with the Numeric Representation of Factors
4.3.5 Collapsing Factors
4.3.6 Combining Factors
4.3.7 Creating Factors from Continuous Values
4.4 Dates and Times
4.4.1 Creating and Formatting Dates
4.4.2 Performing Simple Operations with Dates
4.4.3 Incorporating Time

5
LISTS AND DATA FRAMES
5.1 Lists of Objects
5.1.1 Creating Lists and Accessing Components
5.1.2 Naming
5.1.3 Nesting
5.2 Data Frames
5.2.1 Construction
5.2.2 Basic Subsets
5.2.3 Column Insertions and Data Frame Combinations
5.2.4 Logical Record Subsets
5.2.5 The Explicit subset Command

6
SPECIAL VALUES, CLASSES, AND COERCION
6.1 Special Values
6.1.1 Infinity
6.1.2 NaN: Not a Number
6.1.3 NA: Not Available
6.1.4 NULL
6.2 Types, Classes, and Coercion
6.2.1 Attributes
6.2.2 Object Class
6.2.3 Is-Dot Object-Checking Functions
6.2.4 As-Dot Coercion Functions

7
BASIC PLOTTING
7.1 Using plot with Coordinate Vectors
7.2 Visual Controls and Graphical Parameters

7.2.1 Automatic Plot Types
7.2.2 Title and Axis Labels
7.2.3 Color
7.2.4 Line and Point Appearances
7.2.5 Custom Region Limits
7.2.6 Aspect Ratio
7.3 Adding Points, Lines, and Text to an Existing Plot
7.4 The ggplot2 Package
7.4.1 Plotting a Toy Example
7.4.2 Setting Appearance Constants with Geoms
7.4.3 Aesthetic Mapping with Geoms

8
READING AND WRITING FILES

8.1 R-Ready Datasets
8.1.1 Built-In Datasets
8.1.2 Contributed Datasets
8.2 Reading in External Data Files
8.2.1 The Table Format
8.2.2 Spreadsheet Workbooks
8.2.3 Web-Based Files
8.2.4 Other File Formats
8.3 Writing Out Data Files and Plots
8.3.1 Datasets
8.3.2 Plots and Graphics Files
8.4 Ad Hoc Object Read/Write Operations
8.4.1 save and load
8.4.2 saveRDS and readRDS
8.4.3 dput and dget
8.4.4 Suitability

PART II
PROGRAMMING

9
CALLING FUNCTIONS

9.1 Scoping
- 9.1.1 Environments
- 9.1.2 Search Path
- 9.1.3 Reserved and Protected Names

9.2 Argument Matching
- 9.2.1 Exact
- 9.2.2 Partial
- 9.2.3 Positional
- 9.2.4 Mixed
- 9.2.5 Ellipses

9.3 Pipes
- 9.3.1 Syntax and Examples
- 9.3.2 Should You Pipe?

10
CONDITIONS AND LOOPS

10.1 if Statements
- 10.1.1 Stand-Alone Statement
- 10.1.2 else Statements
- 10.1.3 Element-Wise Checks with ifelse
- 10.1.4 Nested and Stacked Statements
- 10.1.5 The switch Function

10.2 for Loops
- 10.2.1 Definition and Basics
- 10.2.2 Via Index or Value

10.2.3 Nests
10.3 while Loops
10.3.1 Definition and Basics
10.3.2 Condition Object Modification
10.4 Implicit Looping
10.4.1 The apply Function
10.4.2 Some apply Variants
10.4.3 When to Use Implicit Loops
10.5 Other Control Flow Mechanisms
10.5.1 break
10.5.2 next
10.5.3 repeat

11
WRITING FUNCTIONS
11.1 The function Command
11.1.1 Function Creation
11.1.2 The return Statement
11.2 Arguments
11.2.1 Considering Lazy Evaluation
11.2.2 Setting Defaults
11.2.3 Checking for Missing Arguments
11.2.4 Dealing with Ellipses
11.3 Specialized Functions
11.3.1 Helper Functions
11.3.2 Anonymous Functions
11.3.3 Recursive Functions

12
EXCEPTIONS, TIMINGS, AND VISIBILITY

12.1 Exception Handling
12.1.1 Errors and Warnings
12.1.2 try Statements
12.2 Progress and Timing
12.2.1 Tracking Progress: Are We There Yet?
12.2.2 Measuring Completion Time: How Long Did It Take?
12.2.3 Profiling Your Code: Where's the Bottleneck?
12.3 Masking
12.3.1 Function and Object Distinction
12.3.2 Data Frame Variable Distinction

PART III
STATISTICS AND PROBABILITY

13
ELEMENTARY STATISTICS

13.1 Describing Raw Data
13.1.1 Numeric Variables
13.1.2 Categorical Variables
13.1.3 Examples
13.1.4 Univariate and Multivariate Data
13.1.5 Parameter or Statistic?
13.2 Summary Statistics
13.2.1 Measures of Centrality
13.2.2 Counts, Percentages, and Proportions
13.2.3 Rounding, Floors, Ceilings, and Absolute Values
13.2.4 Quantiles, Percentiles, and the "Five-Number Summary"
13.2.5 Measures of Spread
13.2.6 Relationships Between Numeric Variables

13.2.7 Outliers

14

BASIC DATA VISUALIZATION

14.1 Bar Plots and Pie Charts
14.1.1 Bar Plot Creation
14.1.2 In ggplot2
14.1.3 A Quick Pie Chart
14.2 Histograms
14.2.1 A Simple Example
14.2.2 Bin Width and Customization
14.2.3 In ggplot2
14.3 Density Plots
14.3.1 Construction
14.3.2 Bandwidths and Customization
14.3.3 When Multiple Densities Overlap
14.3.4 In ggplot2
14.4 Box Plots
14.4.1 Stand-Alone
14.4.2 Side-by-Side
14.4.3 In ggplot2
14.5 Violin Plots
14.5.1 Design and Interpretation
14.5.2 Side-by-Side Violins
14.5.3 In ggplot2
14.6 Scatterplots and Coordinate-Based Plots
14.6.1 Single Plot
14.6.2 Matrix of Plots
14.6.3 In ggplot2
14.6.4 Plots of Spatial Coordinates

15

PROBABILITY

15.1 What Is a Probability?
- 15.1.1 Events and Probability
- 15.1.2 Conditional Probability
- 15.1.3 Intersection
- 15.1.4 Union
- 15.1.5 Complement

15.2 Random Variables and Probability Distributions
- 15.2.1 Realizations
- 15.2.2 Discrete Random Variables
- 15.2.3 Continuous Random Variables
- 15.2.4 Shape, Skew, and Modality

16

COMMON PROBABILITY DISTRIBUTIONS

16.1 Common Probability Mass Functions
- 16.1.1 Bernoulli Distribution
- 16.1.2 Binomial Distribution
- 16.1.3 Poisson Distribution
- 16.1.4 Other Mass Functions

16.2 Common Probability Density Functions
- 16.2.1 Uniform
- 16.2.2 Normal
- 16.2.3 Student's t-Distribution
- 16.2.4 Exponential
- 16.2.5 Chi-Squared
- 16.2.6 F-Distribution
- 16.2.7 Other Density Functions

PART IV
STATISTICAL TESTING AND MODELING

17
SAMPLING DISTRIBUTIONS AND CONFIDENCE

17.1 Sampling Distributions
- 17.1.1 For a Sample Mean
- 17.1.2 For a Sample Proportion
- 17.1.3 For Other Statistics

17.2 Confidence Intervals
- 17.2.1 An Interval for a Mean
- 17.2.2 An Interval for a Proportion
- 17.2.3 Other Intervals
- 17.2.4 Comments on the Interpretation of a CI

18
HYPOTHESIS TESTING

18.1 Components of a Hypothesis Test
- 18.1.1 Hypotheses
- 18.1.2 Test Statistic
- 18.1.3 p-Value
- 18.1.4 Significance Level
- 18.1.5 Criticisms of Hypothesis Testing

18.2 Testing Means
- 18.2.1 Single Mean
- 18.2.2 Two Means

18.3 Testing Proportions
- 18.3.1 Single Proportion
- 18.3.2 Two Proportions

18.4 Testing Categorical Variables

18.4.1 Single Categorical Variable
18.4.2 Two Categorical Variables
18.5 Errors and Power
18.5.1 Hypothesis Test Errors
18.5.2 Type I Errors
18.5.3 Type II Errors
18.5.4 Statistical Power

19
ANALYSIS OF VARIANCE
19.1 One-Way ANOVA
19.1.1 Hypotheses and Diagnostic Checking
19.1.2 Table Construction
19.1.3 ANOVA Table Construction with the aov Function
19.2 Two-Way ANOVA
19.2.1 A Suite of Hypotheses
19.2.2 Main Effects and Interactions
19.3 Kruskal–Wallis Test
19.3.1 Hypotheses
19.3.2 Example and Interpretation

20
SIMPLE LINEAR REGRESSION
20.1 An Example of a Linear Relationship
20.2 General Concepts
20.2.1 Defining the Model
20.2.2 Estimating the Intercept and Slope Parameters
20.2.3 Fitting Linear Models with lm
20.2.4 Illustrating Residuals
20.3 Statistical Inference

20.3.1 The summary Command
20.3.2 Regression Coefficient Significance Tests
20.3.3 Coefficient of Determination
20.3.4 Other summary Output
20.4 Prediction
20.4.1 Confidence Interval or Prediction Interval?
20.4.2 Interval Interpretation
20.4.3 Interval Plots
20.4.4 Interpolation vs. Extrapolation
20.5 Understanding Categorical Predictors
20.5.1 Working with Binary Variables: k = 2
20.5.2 Working with Multilevel Variables: k > 2
20.5.3 Changing the Reference Level
20.5.4 Treating Categorical Variables as Numeric
20.5.5 Checking for Equivalence with One-Way ANOVA

21
MULTIPLE LINEAR REGRESSION

21.1 Terminology
21.2 Theory
21.2.1 Extending the Simple Model to a Multiple Model
21.2.2 Representing in Matrix Notation
21.2.3 Reviewing a Basic Example
21.3 R Implementation and Interpretation
21.3.1 Adding Predictors
21.3.2 Interpreting Partial Effects
21.3.3 Visualizing the Multiple Linear Model
21.3.4 Finding Confidence Intervals
21.3.5 Computing the Omnibus F-Test
21.3.6 Predicting from a Multiple Linear Model
21.4 Transforming Numeric Variables

21.4.1 Polynomial
21.4.2 Logarithmic
21.4.3 Other Transformations
21.5 Interactive Terms
21.5.1 Concept and Motivation
21.5.2 One Categorical, One Continuous
21.5.3 Two Categorical
21.5.4 Two Continuous
21.5.5 Higher-Order Interactions

22

LINEAR MODEL SELECTION AND DIAGNOSTICS

22.1 Goodness-of-Fit vs. Complexity
22.1.1 Principle of Parsimony
22.1.2 General Guidelines
22.2 Model Selection Algorithms
22.2.1 Nested Comparisons: The Partial F-Test
22.2.2 Forward Selection
22.2.3 Backward Selection
22.2.4 Stepwise AIC Selection
22.2.5 Other Selection Algorithms
22.3 Residual Diagnostics
22.3.1 Inspecting and Interpreting Residuals
22.3.2 Assessing Normality
22.3.3 Illustrating Outliers, Leverage, and Influence
22.3.4 Calculating Leverage
22.3.5 Calculating Cook's Distance
22.3.6 Graphically Combining Residuals, Leverage, and Cook's Distance
22.4 Collinearity
22.4.1 Potential Warning Signs

22.4.2 Correlated Predictors: A Quick Example

PART V
ADVANCED GRAPHICS

23
ADVANCED PLOT CUSTOMIZATION

23.1 Handling the Graphics Device
23.1.1 Manually Opening a New Device
23.1.2 Switching Between Devices
23.1.3 Closing a Device
23.1.4 Controlling Multiple Plots in One Device
23.2 Plotting Regions and Margins
23.2.1 Default Spacing
23.2.2 Custom Spacing
23.2.3 Clipping
23.3 Point-and-Click Coordinate Interaction
23.3.1 Retrieving Coordinates Silently
23.3.2 Visualizing Selected Coordinates
23.3.3 Placing Ad Hoc Annotations
23.4 Customizing Traditional R Plots
23.4.1 Graphical Parameters for Style and Suppression
23.4.2 Box Customization
23.4.3 Axis Customization
23.5 Specialized Text and Label Notation
23.5.1 Font
23.5.2 Greek Symbols
23.5.3 Mathematical Expressions
23.6 Extended Example: A Custom Scatterplot

24

GOING FURTHER WITH THE GRAMMAR OF GRAPHICS

24.1 A Quick Recap
24.2 Smoothing and Shading
    24.2.1 Adding LOESS Trends
    24.2.2 Constructing Smooth Density Estimates
24.3 Multiple Plots and Variable-Mapped Facets
    24.3.1 Independent Plots
    24.3.2 Facets Mapped to a Categorical Variable
24.4 Interactive Tools in ggvis

25

DEFINING COLORS AND PLOTTING IN HIGHER DIMENSIONS

25.1 Representing and Using Color
    25.1.1 Red-Green-Blue Hexadecimal Color Codes
    25.1.2 Built-in Palettes
    25.1.3 Custom Palettes
    25.1.4 Approaches to Color Indexing
    25.1.5 Color Legends
    25.1.6 Opacity
    25.1.7 RGB Alternatives and Further Functionality
25.2 3D Scatterplots
    25.2.1 Basic Syntax
    25.2.2 Visual Enhancements
25.3 Preparing a Surface for Plotting
    25.3.1 Constructing an Evaluation Grid
    25.3.2 Constructing the z-Matrix
    25.3.3 Conceptualizing the z-Matrix
25.4 Contour Plots
    25.4.1 Contour Lines

25.4.2 Color-Filled Contours
25.5 Pixel Images
25.5.1 One Grid Point Equals One Pixel
25.5.2 Surface Truncation and Empty Pixels
25.6 Perspective Plots
25.6.1 Interpreting Basic Plots and Adjusting Angles
25.6.2 Coloring Facets
25.6.3 Rotating with Loops

26
INTERACTIVE 3D PLOTS

26.1 Point Clouds
26.1.1 Basic 3D Cloud: Points and Spheres
26.1.2 Color and Legends
26.1.3 Alternate 3D Point Characters
26.1.4 3D Text
26.1.5 3D Lines, Axes, Grids, and Other Annotations
26.1.6 Extended Example: A Custom, Interactive 3D Scatterplot
26.2 Bivariate Surfaces
26.2.1 Plotting Basic Perspective Surfaces
26.2.2 Adding Components
26.2.3 Coloring by z Value
26.2.4 Dealing with the Aspect Ratio
26.3 Trivariate Surfaces
26.3.1 Evaluation Coordinates in 3D
26.3.2 Isosurfaces
26.3.3 Example: Nonparametric Trivariate Density
26.4 Handling Parametric Equations
26.4.1 Simple Loci
26.4.2 Mathematical Abstractions

A

INSTALLING R AND CONTRIBUTED PACKAGES

A.1 Downloading and Installing R
A.2 Using Packages
    A.2.1 Base Packages
    A.2.2 Recommended Packages
    A.2.3 Contributed Packages
A.3 Updating R and Installed Packages
A.4 Using Other Mirrors and Repositories
    A.4.1 Switching CRAN Mirror
    A.4.2 Reviewing Other Package Repositories
A.5 Citing and Writing Packages
    A.5.1 Citing R and Contributed Packages
    A.5.2 Writing Your Own Packages

B

WORKING WITH RSTUDIO

B.1 Basic Layout and Usage
    B.1.1 Editor Features and Appearance Options
    B.1.2 Pane Customization
B.2 Auxiliary Tools and Features
    B.2.1 Projects
    B.2.2 Package Manager
    B.2.3 Support for Profiling
    B.2.4 Support for Debugging
B.3 Markup, Document, and Graphics Tools
    B.3.1 Sweave
    B.3.2 knitr
    B.3.3 R Markdown
    B.3.4 Quarto
    B.3.5 Shiny

REFERENCES

INDEX

PREFACE TO THE SECOND EDITION

It is entirely unsurprising that since the first edition of this book was published, R has continued to both (a) evolve as a language and (b) gain even greater popularity in science and industry. A straightforward marker of this is in the number of contributed software packages of statistical methodology and other functionality available to users (see Section A.2), which has gone from approximately 7,000 at the time of the first edition to now well over 21,000 (see *https://cran.r-project.org/web/packages/*). This growth in use and functionality, alongside changes to aspects of base behavior and syntax of the language, necessitates keeping introductory texts and tutorials up-to-date and hence forms the motivation for the second edition of *The Book of R: A First Course in Programming and Statistics*.

The broad aims of this edition mirror those I discussed in the preface to the first edition. This second edition contains a wealth of new content and topics, expands upon those originally discussed, and includes dozens of new examples and exercises. In addition, a host of other improvements, updates, and corrections are the result of a complete refresh of the preexisting content designed to reflect the latest versions of R and its packages.

I've been pleasantly surprised by the many readers from various backgrounds who have said they've enjoyed *The Book of R* over the years via personal emails containing feedback, suggestions, questions, and errata. This engagement has been a key motivator for this refreshed edition.

The rise of artificial intelligence (AI) has brought about the convenience of tools that can rapidly regurgitate code on request, though this convenience comes at the expense of one's deeper understanding and hence hinders one's ability to apply skills more broadly. To that end, I have tried hard to ensure that this new edition

continues to provide a robust, lesson-driven utility for researchers, data scientists, students, academics, and teachers—indeed, anyone wanting to learn about the wonderful flexibility R offers with respect to manipulating, summarizing, visualizing, modeling, and ultimately *understanding* data.

Tilman M. Davies
Dunedin, New Zealand, 2025

PREFACE TO THE FIRST EDITION

The aim of *The Book of R: A First Course in Programming and Statistics* is to provide a relatively gentle yet informative exposure to the statistical software environment R, alongside some common statistical analyses, so that readers may have a solid foundation from which to eventually become experts in their own right. Learning to use and program in a computing language is much the same as learning a new spoken language. At the beginning, it is often difficult and may even be daunting, but total immersion in and active use of the language is the best and most effective way to become fluent.

Many beginner-style texts that focus on R can generally be allocated to one of two categories: those concerned with computational aspects (that is, syntax and general programming tools) and those with statistical modeling and analysis in mind, often one particular type. In my experience, these texts are extremely well written and contain a wealth of useful information but better suit those individuals wanting to pursue fairly specific goals from the outset. This text seeks to combine the best of both worlds, by first focusing on *only* an appreciation and understanding of the language and its style and then subsequently using these skills to fully introduce, conduct, and interpret common statistical practices.

The target audience is, quite simply, anyone who wants to gain a foothold in R as a first computing language, perhaps with the ultimate goal of completing their own statistical analyses. This includes but is certainly not limited to undergraduates, postgraduates, academic researchers, and practitioners in the applied sciences with little or no experience in programming or statistics in general. A basic understanding of elementary mathematical behavior (for example, the order of operations) and associated operators (for example, the summation symbol $\sum$) is desirable, however.

In view of this, *The Book of R* can be used purely as a programming text to learn the language or as an introductory statistical methods book with accompanying instruction in R. Though it is not intended to represent an exhaustive dictionary of the language, the aim is to provide readers with a comfortable learning tool that eliminates the kind of foreboding many have voiced to me when they have considered learning R from scratch. The fact remains that there are usually many different ways to go about any given task, which holds true for most so-called high-level computer languages. What this text presents reflects my own way of thinking about learning and programming in R, which I approach less as a computer scientist and more as an applied data analyst.

In part, I aim to provide a precursor and supplement to the work in *The Art of R Programming: A Tour of Statistical Software Design*, the other R text published by No Starch Press (2011), written by Professor Norman Matloff (University of California, Davis). In his detailed and well-received book, Professor Matloff comes at R from a computer science angle, that is, treating it as a programming language in its own right. As such, *The Art of R Programming* provides some of the best descriptions of R's computational features I've yet to come across (for example, running external code such as C from R programs, handling and manipulating R's memory allocations, and formal debugging strategies). Noteworthy, however, is the fact that some previous experience and knowledge of programming in general goes a long way to appreciating some of these more advanced features. It is my hope that my text will not only provide this experience but will do so in R itself at a comfortable pace, with statistical analyses as the supplementary motivation.

This text, which serves as a "traveler's guide" as we backpack our way through R country, was born out of a three-day introductory R workshop I began teaching at the University of Otago in New Zealand. The emphasis is on active use of the software, with each chapter containing code examples and practice exercises to encourage

interaction. For those readers not part of a workshop, just fire up your computer, grab a drink and a comfy chair, and start with Chapter 1.

Tilman M. Davies
Dunedin, New Zealand, 2016

ACKNOWLEDGMENTS

First, I must of course acknowledge the creators and maintainers of R, both past and present. If any prominent contributor ever picks up this book (perhaps as a handy stabilizer for a wonky piece of furniture), they should be reminded of the appreciation that many, including myself, have for the incredible role R plays as a data analysis tool worldwide and a glowing example of what can be achieved with free software. R has transformed the way academic and industry statisticians, and many others, make contributions to research and data analysis. This doesn't refer just to their respective pursuits but also to the ability to make computational functionality readily available so that others may perform the same analyses. I'm just one voice of many to echo this, and *The Book of R* essentially represents my own professional love letter to all things R. The best way to profess this sentiment is to encourage and educate both students and professionals in the analytic freedom R provides and hence perpetuate the enthusiasm many already hold for the software.

The team at No Starch Press is undoubtedly an absolute pleasure to work with. If "informal professionalism" is actually a thing, it's exactly how I like to work and exactly how the team guided the progress of what turned out to be a large, time-consuming project. Thanks to Bill Pollock and Tyler Ortman for taking on this project. Thanks especially to those at the coalface of my drafts, specifically Liz Chadwick (first edition), Eva Morrow, and Jill Franklin (second edition) for picking through and streamlining the language and exposition in numerous $\TeX$ files, and to those who turned the collection of these files into a cohesive, polished book, headed by Riley Hoffman (first edition) and Sabrina Plomitallo-González (second edition). Dr. Debbie Leader, a senior university tutor in statistics at Massey University (Palmerston North, New Zealand), can't escape my appreciation for the scrutiny she applied to the R code and statistical content, improving the clarity and

interpretation of the core focus of the book. Comments from other reviewers and early readers were also instrumental in shaping the finished product.

A special mention must be made of Professor Martin Hazelton, formerly of Massey University, now at my own institution—University of Otago. Once my undergraduate lecturer during my freshman year at the University of Western Australia; then my graduate and doctoral supervisor in New Zealand, where I followed him; and now my ongoing collaborator, mentor, and friend, Martin is pretty much solely responsible for my interest and career in statistics. Although my scribblings might not include all the big fancy Oxford words he'd use, it's only with this education that I'm now in a position to write this book. With it, I hope I can inspire my own students and any others wanting to learn more about statistical programming in the same way he inspired me.

Last, but certainly not least, my immediate family in Perth and my extended family in Germany must be thanked for their ongoing and unwavering love and support through many years. My New Zealand family must also be thanked, especially my wife, Andrea, who would often put up with an Apple logo instead of my face on the other end of the couch as I continued tapping away on the manuscript after work, perhaps pausing to reach for a refreshing sip of Emerson's Hazed and Confused (or if something juicy was happening on *Shortland Street*—Google it). Andrea has been especially busy wrangling our daughter, Quinn (born in 2019, well after the first edition was published), allowing me to undertake the second edition with far fewer interruptions than I deserve. Thanks supermum—good luck with our second one! My venerable cat, Sigma, has been a constant fixture of these moments too, ensuring her insertion in a couple of exercises in this book.

I suppose all this could be summarized in one word, though:

Cheers.

INTRODUCTION



R plays a key role in a wide variety of research and data analysis projects because it makes many modern statistical methods, both simple and advanced, readily available and easy to use. It's true, however, that a beginner to R is often new to programming in general. This, in part, is why R has a bit of a reputation for being "difficult"—but rest assured, that really isn't the case.

A Brief History of R

R is based heavily on the S language, first developed in the 1960s and 1970s by researchers at Bell Laboratories in New Jersey (for an overview, see, for example, Becker et al. 1988). With the goal of embracing open source software, R's founders—Ross Ihaka and Robert Gentleman, who were at the time based at the University of Auckland in New Zealand—released it in the early 1990s under the GNU public license. (The software was named for Ross and Robert's shared first initial.) Since then, R's popularity has grown in leaps and bounds because of its unrivaled flexibility for data analysis and powerful graphical tools, all available for the princely sum of *nothing*. Perhaps the most appealing feature of R is that any researcher can contribute code in the form of packages (or libraries), enabling fast access to

methodological developments in statistics and data science (see Section A.2).

Today, the main source code archives are maintained by a dedicated group known as the R Core Team, and it's a collaborative effort. You can find the names of the most prominent contributors at *https://www.r-project.org*; these individuals deserve thanks for their ongoing efforts, which keep R alive and at the forefront of statistical computing!

Updated versions of R are issued relatively frequently. Substantial changes have been made to the software over time, though neighboring versions are typically similar to one another. In writing the second edition of this book, I've employed versions 4.1.1 to 4.4.2 (the first edition utilized versions 3.0.1 to 3.2.2). All code found herein has been checked against the latest version available at the time of writing. You can find out what's new in the latest version by following the News link on the relevant download page (see Appendix A).

About This Book

The Book of R is a resource intended to help you become proficient in R as a first programming language, motivated by the statistical analysis of data. The goal is to lay a comprehensive foundation of the computational aspects of modern data science.

The book is structured by first focusing on R as a computational and programming tool and then shifting gears to discuss using R for probability, statistics, and data exploration and modeling. You'll build your knowledge progressively, and most chapters end with a table summarizing important commands and operators for quick reference.

Part I: The Language

Part I, which covers the fundamental syntax and object types used across all aspects of R programming, is essential for beginners.

Chapter 1: Getting Started Provides instruction on installing R and explains the default appearance of the software's graphical user interface. Basic workflow is explained, as is the common step of installing a contributed package of code. The coverage is purposefully kept concise; topics are expounded upon in the appendices.

Chapter 2: Numerics, Arithmetic, Assignment, and Vectors Introduces the basics of simple arithmetic, as well as the creation and assignment of objects in the workspace. The key object type of *vectors* and their behavior is explored.

Chapter 3: Matrices and Arrays Extends the discussion of vectors to encompass higher-dimensional object types. Matrices are defined, in terms of both their representation in R and their mathematical properties, as is the more general class of array-type objects. Common tasks, such as subsetting these objects, are explored in detail.

Chapter 4: Non-Numeric Values Explores important data types such as logical (true/false) values, character strings, factors, and date-times. A comprehensive treatment of each of these, including typical operations and use-cases, is given.

Chapter 5: Lists and Data Frames Details the larger, more general object types of lists and data frames. Data frames in particular are the main type of object used for storage of datasets, exploratory data analysis, data visualization, and subsequent statistical modeling.

Chapter 6: Special Values, Classes, and Coercion Covers how R represents important special values such as missing observations. The chapter also examines how to interrogate a particular object for its type, as well as how to distinguish between different object types.

Chapter 7: Basic Plotting Provides a primer on plotting, using both built-in and contributed functionality via the `ggplot2` package. These basic skills lay the groundwork for creating common

statistical plots and more advanced graphical design, both of which are detailed in later parts of the book.

Chapter 8: Reading and Writing Files Teaches the fundamentals of reading in data from external files, essential for analyzing your own data. You'll also consider how to write out your data to external files, such as plaintext files or comma-separated, spreadsheet-style files, as well as write out specific R objects that can subsequently be used in a different R session.

Part II: Programming

Part II focuses on getting you familiar with the syntax and behavior of R's programming mechanisms.

Chapter 9: Calling Functions Introduces basic technicalities to help you understand how R interprets your commands. Topics include environments, the search path, and argument matching.

Chapter 10: Conditions and Loops Explores fundamental programming structures, such as the conditional `if...else` statements used to control the flow of your code and different types of loops for repeating specified operations.

Chapter 11: Writing Functions Covers writing your own executable R functions, facilitated by skills learned in the immediately preceding chapters. You'll learn how to handle arguments and formally return results. Specialized functions, such as anonymous no-name functions and those that recursively call themselves, are also discussed.

Chapter 12: Exceptions, Timings, and Visibility Deals with additional topics relevant to executing R functions. These include handling warnings and errors, measuring function execution time, and profiling code to determine where computational bottlenecks might lie.

Part III: Statistics and Probability

With a firm handle on R as a language, you'll shift your attention to statistical thinking in Part III.

Chapter 13: Elementary Statistics Introduces you to basic statistical and data-related tasks. You'll look at important terminology used to describe variables; elementary summary statistics such as the mean, variance, quantiles, and correlation; and how these statistics are implemented in R.

Chapter 14: Basic Data Visualization Combines knowledge from the preceding chapter with important code from Chapter 7 as a foundation for visually exploring your data with both base R and `ggplot2` functionality. You'll learn when and how to use and customize common statistical plots such as histograms, box plots, and scatterplots.

Chapter 15: Probability Gives a primer on key concepts related to the statistical understanding of probability, including the ideas of random variables and their probability distributions.

Chapter 16: Common Probability Distributions Extends the fundamental definitions of the preceding chapter to look at the R implementation and statistical interpretation of some well-known probability distributions.

Part IV: Statistical Testing and Modeling

Part IV introduces statistical uncertainty, hypothesis testing, and linear regression models. Linear regression represents just one class of parametric models and is a natural starting point for learning about statistical regression. Similarly, the R syntax and output used to fit, summarize, predict from, and diagnose linear models of this kind are much the same for other regression models, so once you're comfortable with these chapters, you'll be ready to tackle the R implementation of morecomplicated models covered in more-advanced texts with relative ease.

Chapter 17: Sampling Distributions and Confidence Takes the general idea of a probability distribution and lends it to statistics estimated from data. You are thereby introduced to sampling distributions—the fundamental building block of statistical inference—and corresponding confidence intervals.

Chapter 18: Hypothesis Testing Details the fundamentals of hypothesis testing from an applied, example-driven perspective. It explains the meaning of *p*-values and demonstrates implementation and interpretation using R.

Chapter 19: Analysis of Variance Discusses basics of the common analysis of variance (ANOVA) procedure and covers design of the corresponding hypotheses, their interpretation, and implementation of ANOVA in R.

Chapter 20: Simple Linear Regression Introduces regression models, using the simplest setting in which you aim to model (or predict) the value of one variable based on the value of another. Basic theory, notation, and terminology are covered, as is the implementation and interpretation of such models.

Chapter 21: Multiple Linear Regression Extends the discussions of the preceding chapter to examine linear regression models with more than one predictor variable. This involves new ideas around interpretation of the fitted model, dealing with variable transformation, as well as capturing more-complex relationships in your data via interactive effects.

Chapter 22: Linear Model Selection and Diagnostics Discusses methods for selecting an appropriate linear model and assessing the validity of that model with various diagnostic tools.

Taken together, Parts III and IV represent much of what you'd expect to see in first- and second-year college statistics courses. My aim is to keep mathematical details to a minimum and focus on implementation and interpretation. I'll provide references to other

resources where necessary if you're interested in looking more closely at the underlying theory.

Part V: Advanced Graphics

The final part looks at moreadvanced graphing skills. Though not strictly necessary, it's helpful to have some familiarity with the linear regression methods discussed in Part IV before tackling Part V, since some of the examples in this last part use fitted linear models.

Chapter 23: Advanced Plot Customization Shows you how to customize traditional R graphics, from handling the graphics devices themselves to controlling the finer aspects of your plot's appearance, such as dealing with margins and displaying special textual characters.

Chapter 24: Going Further with the Grammar of Graphics Explores the popular `ggplot2` package further, looking at more advanced features such as adding smooth scatterplot trends and producing multiple plots via faceting. You'll also examine the interactive capabilities of the `ggvis` package.

Chapter 25: Defining Colors and Plotting in Higher Dimensions Concentrates on higher-dimensional plotting in R. Covers color handling and 3D surface preparation before discussing contour plots, perspective plots, and pixel images with the aid of multiple examples.

Chapter 26: Interactive 3D Plots Focuses on creating interactive plots (rotated by a mouse) via the `rgl` package. You'll also see simple ways to use `rgl` for plotting multivariate parametric equations.

What's New in the Second Edition

The second edition of *The Book of R* has been revised from start to finish, comprising nearly 100 additional pages of new content and

dozens of new exercises. The following are features of note:

- Updated details on the behavior and handling of factor objects in the latest versions of R
- New content on date-time objects
- New data plots, including density plots and violin plots
- New explanations of handling R graphics, such as dealing with aspect ratios and plotting geographical coordinates
- Updated content on ways to read and write external files
- New content on profiling code (finding computational bottlenecks by using built-in tools)
- New statistical content, including dealing with numerical operations like rounding, floors, and ceilings; and newly covered probability distributions, like the chi-squared distribution and the F-distribution
- New content on using pipes to string commands together
- New and revised exercises to reflect added or improved content
- Updates, revisions, and corrections throughout to reflect the latest versions of the software and improve explanations and readability for contemporary students of R

Conventions and Styles

I adhere to a few conventions in this book when it comes to the presentation of code, math, and general typesetting.

Coding

As mentioned, when you code with R, you execute the code in the console, possibly after writing the script in the editor first. Note the following:

- R code that's entered directly into the console for execution is shown preceded by the `R>` prompt, typeset in bold font, and followed by any output displayed in the console. For example, this simple division of 14 by 6 from Section 2.1.1 looks like this:

```
R> 14/6
[1] 2.333333
```

  To copy and paste console-executed code directly from the text of the book, omit the `R>` prompt. All R code in this book, both console executed and editor written, is available in carefully organized R script files that you can download from the book's web page at *https://nostarch.com/book-of-r-2nd-edition*. I recommend using these script files if you're following the book with R open on your computer.

- For code that should be written in the editor before it's executed in the console, I indicate as such in the text, and that code is presented *without* the prompt and in a non-bold typeface. The following example comes from Section 10.2.1:

```
for(myitem in 5:7){
    cat("--BRACED AREA BEGINS--\n")
    cat("the current item is",myitem,"\n")
    cat("--BRACED AREA ENDS--\n\n")
}
```

  My preferred coding style for arranging and indenting chunks like this will become clearer as you progress through Part II.

- Long lines of code (either executed directly in the console or written in the editor) are split and indented at an appropriate place to fit on the printed page. For example, take this line from Section 6.2.2:

```
R> ordfac.vec <- factor(x=c("Small","Large","Large","Regular",
                        "Small"), levels=c("Small","Regular","Large"),
                        ordered=TRUE)
```

Although this can be written out as a single line when using R, you can also break the line at a comma (in this case, the comma splits the arguments to the `factor` function). In this book, a broken line is indented to the level of the opening parenthesis of the relevant command. Both forms, single line or split, will work when executed in R.

- Lastly, any lengthy console output that is not essential to your understanding of the immediate content is suppressed for the sake of space. I indicate these instances in the text, and you'll see the designation `--snip--` in the affected chunk of code.

Math and Equation References

Mathematics and equations are kept to a minimum in this book (mainly in Parts III and IV), but going into a little mathematical detail is sometimes necessary in certain sections.

Important equations are presented on their own lines as follows:

$$y = 4x \qquad (1)$$

Equations are numbered in parentheses, and text references use these numbers, preceded by *Equation*. For example:

- As per Equation (1), $y = 8$ when $x = 2$.
- Inversion of Equation (1) yields $x = y/4$.

When numeric results are rounded to a certain level, I indicate the number of decimal places, abbreviated to *d.p.* Here are some examples:

- The famous geometric value pi is given as $\pi = 3.1416$ (rounded to 4 d.p.).
- Setting $x = 1.467$ in (1) results in $y = 5.87$ (2 d.p.).

Exercises

Exercise questions in the chapters appear in a rounded box:

Exercise 0.1

a. Say the word *cat* aloud.

b. Using nothing but your brain, find the solution to 1 + 1.

These exercises are optional. If you choose to tackle them, they are intended to be completed as and when they appear in the text to help you practice and understand the specific content and code in the sections that immediately precede them.

All the datasets you'll use in this book for coding and plotting examples are available either as built-in R objects or as part of one of the contributed packages you'll install. These packages are noted as needed in the relevant part of the text (for a list of them, see Section A.2.3).

In addition to all lines of code as shown in this book, complete suggested solutions to *all* practice exercises are also available as runnable *.R* script files at *https://www.nostarch.com/book-of-r-2nd-edition*. You should think of these solutions (and any accompanying commentary) as *suggested* because there are often multiple ways to perform a certain task in R, which may not necessarily be any better or worse than those supplied. Try to resist the urge to peek at the solutions before attempting the exercises!

For Students

Like many, I first started becoming proficient in R programming and the associated implementation of various statistical methods when I began my graduate studies (at Massey University in Palmerston North, New Zealand). Building on little more than the odd line or two of code I'd encountered during my undergraduate years in Australia, being

"thrown in the deep end" had both benefits and drawbacks. While the immersion accelerated my progress, not knowing what to do when things didn't work properly was incredibly frustrating.

The Book of R thus represents the introduction to the language that I wish I'd had when I began exploring R, combined with the first-year fundamentals of statistics as a discipline, implemented in R. With this book, you'll be able to build a well-rounded foundation for using R, both as a programming language and as a tool for statistical analyses.

This book was written to be read cover to cover, like a story (albeit with no plot twists!). As mentioned, ideas are built up progressively within each part of the book, so you can choose to begin either right at the start or where you feel your level of knowledge currently stands. With that in mind, I offer the following recommendations to students of R:

- Try not to be afraid of R. It will do exactly what you tell it to—nothing more, nothing less. When something doesn't work as expected or an error occurs, this literal behavior works in your favor. Look carefully at the commands line by line and try to narrow down the instructions that caused the fault. Third-party editors like RStudio (see Appendix B) can help flag syntactical mistakes quickly.
- Attempt the practice exercises in this book and check your responses by using the suggested solutions; these are available as R script files on the book's website (see page xl). Download the *.zip* file and extract the *.R* files, one for each part of the book. Open these in your R session, and you can run the lines as you would any R code to see the output. The short practice exercises are intended to be exactly that—practice—as opposed to being hard or insurmountable challenges. Everything you need to know to complete them is contained in the preceding sections of that chapter.

- Especially in your early stages of learning, when you're away from this book, try to use R for everything, even simple tasks or calculations you might usually do elsewhere. This will force your mind to switch to "R mode" more often and will help you become comfortable with the environment more quickly.

For Instructors

This book was originally designed from a three-day workshop, Introduction to R, that I wrote at my current institution—the Department of Mathematics and Statistics at the University of Otago in New Zealand—as part of our Statistics Workshops for Postgraduates and Staff (SWoPS). Succeeded by the SWoPS class Statistical Modelling 1 run by two of my colleagues, the aim of Introduction to R is, as the title suggests, to address the programming side of things. Your coverage will naturally depend on your target audience.

Here, I provide some recommendations for using the content in *The Book of R* for workshops of similar length to our SWoPS series. Particular chapters can be added or dropped depending on your target workshop duration and students' existing knowledge:

Programming Introduction Parts I and II. Selected material from Part V, especially Chapter 23, might also suit the scope of such a course.

Statistics Introduction Parts III and IV. If a brief introduction to R is warranted beforehand, consider dropping, for example, Chapter 13 from Part III and Chapters 17 through 19 in Part IV and building an initial foundation from content in Part I.

Intermediate Programming and Statistics Parts II and IV. Consider dropping Chapters 17 through 19 from Part IV to include content from Part V if the audience is interested in developing plotting skills.

R Graphics Parts I and V. Depending on audience knowledge, material from Part I may be dropped so that Chapter 14 in Part III can be included.

If you're planning to go even further and structure a longer course around this book, the practice exercises make particularly good lecture-specific homework to keep students abreast of the skills in R and statistics as they're developed. The main points of the sections making up each chapter are relatively easy to translate into slides that can be initially structured with help from the Contents in Detail.

Web Resources



https://www.nostarch.com/book-of-r-2nd-edition

Here, you'll find the following content:

- R script files for each part of the book, replicating all example code section by section, mirroring the layout of the book. You can use these to work in R alongside the text.
- R script files containing suggested solutions for all exercises in each part of the book.
- Example files used in Chapter 8 to demonstrate reading in external datasets and R objects.
- A list of errata and corrections that will be updated as necessary.

PART I
THE LANGUAGE

1
GETTING STARTED



R provides a wonderfully flexible programming environment favored by many researchers who do some form of data analysis as part of their work. In this chapter, I'll lay the groundwork for learning and using R, and I'll cover the basics of installing R and other useful details to know before you begin.

1.1 Obtaining and Installing R from CRAN

R is available for Microsoft Windows, Apple macOS, and Linux/Unix platforms. You can find the main collection of R resources online at the Comprehensive R Archive Network (CRAN). If you go to the R project website at *https://www.r-project.org*, you can navigate to your local CRAN mirror and download the installer relevant to your operating system. Section A.1 provides step-by-step instructions for installing the base distribution of R.

1.2 Opening R for the First Time

R is an interpreted language that's strictly case and character sensitive, which means that you enter instructions that follow the specific syntactic rules of the language into a console or command line interface. The software then interprets and executes your code and returns any results.

R is a *high-level* programming language. *Level* refers to the level of abstraction away from the fundamental details of computer execution. That is, a *low-level* language will require you to do things such as manually manage the machine's memory allotments, but with a high-level language like R, you're fortunately spared these technicalities.

1.2.1 Base R GUI

When you open the base R application, you're presented with the R console; Figure 1-1 shows a Windows instance, accompanied by an empty code editor window (we'll learn more about these in Section 1.3). This built-in graphical user interface (GUI) is a typical way base R is used.

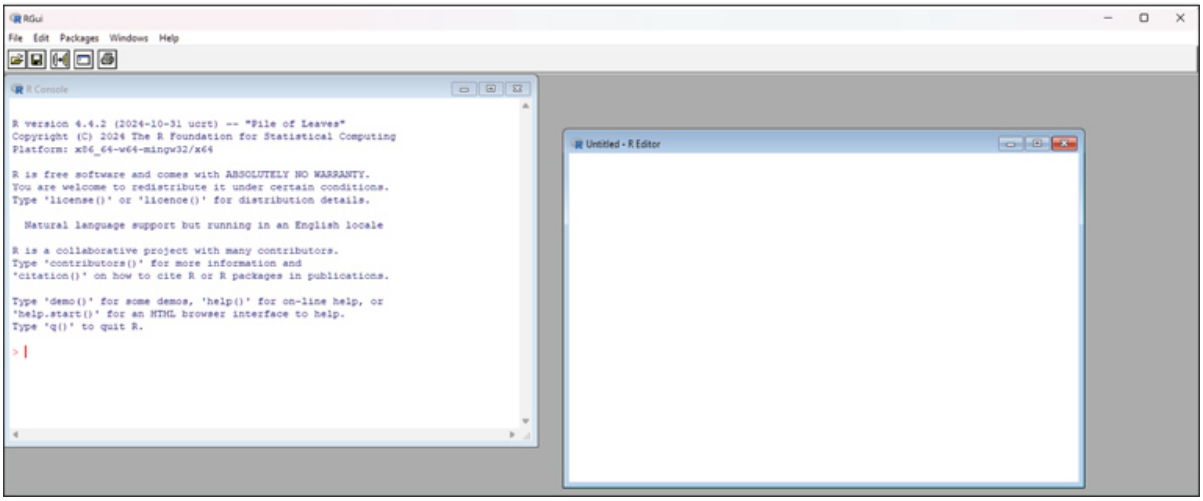


Figure 1-1: The R GUI application (default configuration) in Windows

Figure 1-2 shows an example of the same two windows in macOS.

The functional, no-frills appearance of the console, which in my experience has struck fear into the hearts of many an undergraduate, stays true to the very nature of the software: a blank statistical canvas

that can be used for any number of tasks. Note that macOS versions use separate windows for the console and editor, though the default behavior in Windows is to contain these panes in one overall R window (you can change this in the GUI preferences if desired; see Section 1.3.1). Many users prefer to additionally install a popular third-party editor called *RStudio*, which provides a friendlier user interface; I discuss this software briefly in Section 1.2.2 and in greater detail in Appendix B.

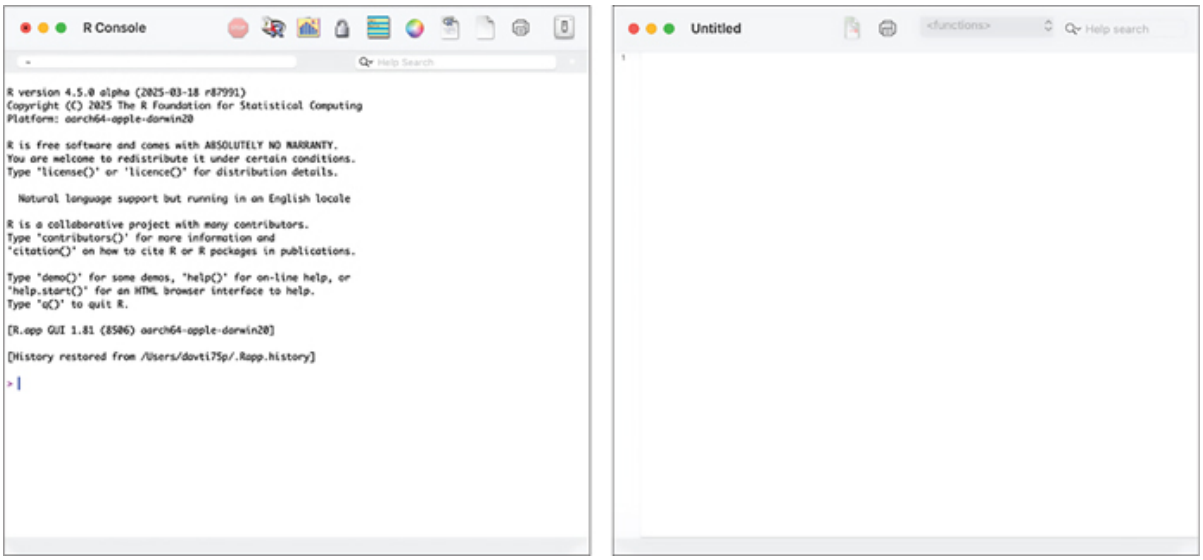


Figure 1-2: The base R GUI console pane (left) and a newly opened instance of the built-in editor (right) in macOS

1.2.2 RStudio and Other Third-Party Editors

The popularity of R has led to the development of several third-party code editors, or compatible plug-ins for existing code-editing software, which can enhance the experience of coding in R.

One noteworthy contribution is RStudio (Posit team 2024). This is an integrated development environment (IDE) available free for Windows, macOS, and Linux/Unix platforms at *https://posit.co/downloads/*; it requires an existing installation of the base R language as described in Section 1.1.

RStudio includes a direct-submission code editor; separate point-and-click panes for tasks such as file, object, and project management; as

well as the display of plots and help files and the creation of markup documents incorporating R code. Given RStudio's popularity, I use Appendix B to discuss the IDE and its capabilities in more detail alongside various screenshots.

Use of any third-party editor, including RStudio, is by and large a personal choice. In this book, our focus is on the language and on coding skills, and I do not assume use of any particular editor.

1.3 Workflow Basics

Here, I'll introduce some high-level features of the R software that make up a large part of the overall experience when coding.

1.3.1 Console and Editor Panes

As you saw in Figures 1-1 and 1-2, R uses two main window types for programming R code and viewing output. The console, or command line interpreter, that you've just seen is where all execution takes place and where all textual and numeric output is provided. You would typically use the R console directly only for short, one-line commands, such as for calculations or plotting.

By default, the R *prompt* that indicates R is ready and awaiting a command is a `>` symbol, after which a text cursor appears. To avoid confusion with the mathematical symbol for *greater than*, >, I prefer to modify this with `R>`, which you can set as follows:

```
> options(prompt="R> ")
R>
```

With the cursor placed at the prompt, you can use the keyboard up arrow (↑) and down arrow (↓) to scroll through any previously executed commands; this is useful when making small tweaks to earlier commands.

For longer chunks of code and function authoring, it's more convenient to first write your commands in an editor and execute them

in the console only when you're done. R has a built-in code editor for this purpose. The R *scripts* you write in the code editor are essentially plaintext files with a *.R* extension.

You can open a new instance of the editor by using the R GUI menus (for example, File ▸ New Script in Windows or File ▸ New Document in macOS). Those using RStudio can navigate to File ▸ New File ▸ R Script.

You can tailor aesthetics of both the console and the editor, such as coloring and character spacing, to a certain extent by accessing the relevant GUI preferences. Figure 1-3 shows the R GUI preferences in Windows (Edit ▸ GUI Preferences) and macOS (R ▸ Preferences).



Figure 1-3: The R GUI preferences in Windows (left) and macOS (right)

A nice feature of the macOS version of the base R GUI is the code-coloring and bracket-matching features of the bare-bones editor, which can improve the authoring and readability of large sections of code. That said, if you're looking for the greatest flexibility in customizing editor aesthetics, then RStudio is a good option (see Section B.1.1).

1.3.2 Code Execution

Both the built-in R GUI editor and the RStudio IDE feature useful shortcuts to efficiently send code to the console for evaluation. Specific menu items or keystrokes depend on the operating system, but the

general action of editor-to-console is the same regardless. Here are some options to execute code you've written in the editor:

- Copy and paste the selected code from the editor directly into the console.
- Using the base R GUI, choose Edit ▸ Run Line or Selection in Windows; choose Edit ▸ Execute in macOS.
- Using RStudio, choose Code ▸ Run Selected Line(s).
- Use the keystroke shortcuts such as CTRL-R in Windows or ⌘-RETURN in macOS. The latter keystroke remains the same for RStudio users on a Mac, though RStudio users in Windows will need the CTRL-ENTER keystroke.

Use of a keystroke shortcut is by far the most common way to execute editor-based code. Depending on your operating system and choice of editor, this action will quickly become automatic. These keystrokes allow you to send the single, non-highlighted line upon which the cursor sits, a highlighted line, a highlighted part of a line, or a highlighted chunk of code. Having multiple editor panes open at once is permitted; keystroke code submissions simply operate with respect to the currently selected editor pane.

1.3.3 Comments

In R, you can annotate your code with comments. Just preface the line with a hash mark (#), and anything that comes thereafter will be ignored by the interpreter. For example, executing the following in the console simply returns you to the prompt:

```
R> # This is a comment in R...
```

Comments can also appear after valid commands:

```
R> 1+1 # This works out the result of one plus one!
[1] 2
```

If you're writing large or complicated chunks of code in the editor, this kind of annotation can be helpful to others (and your future self!) who want to understand what your code is doing.

1.3.4 Working Directory

An active R session always has a working directory associated with it. Unless you explicitly specify a filepath when saving or importing data files, R will use this working directory by default. To check the location of the working directory, use the `getwd` function. Here is what the result looks like on my Mac:

```
R> getwd()    # This is on my Mac
[1] "/Users/tdavies"
```

And on my Windows machine, here's what the same command returns:

```
R> getwd()    # This is on my Windows PC
[1] "C:/Users/tdavies/Documents"
```

Notice that full filepaths on Windows PCs require the appropriate drive letter designation (revealed as `C:` in this example).

Filepaths are always enclosed in double quotation marks, which R uses to define character strings (you'll learn about these throughout Section 4.2). Furthermore, note the single forward slashes (not single backslashes) in the presentation of the folder hierarchies. (This is because single backslashes are used to manipulate character strings for certain printing behaviors; see Section 4.2.3.) Using a *double* backslash (`\\`) for this purpose is nevertheless valid, and you may come across this presentation as well.

You can change the default working directory by using the `setwd` function. For instance, on my Mac I can execute

```
R> setwd("/Users/tdavies/Book/SECOND_EDITION")
```

to instruct R to set the working directory of my current session to be the folder where I store the development files for this book. (Naturally, this particular line won't work on your own machine unless you happen to have my exact folder structure!)

You may provide your filepath relative to the current working directory or fully (in other words, from a system root drive). The previous call to `setwd` on my Mac is actually a little wasteful, since my preexisting working directory was already */Users/tdavies* (as revealed by my previous use of `getwd`). Thus, simply executing `setwd("Book/SECOND_EDITION")` would've had an equivalent effect. Either way, it's important to remember the case-sensitive nature of R; you must match the naming and punctuation of any folder names exactly, or an error will be thrown. For this reason, it's helpful to avoid long or complicated folder names and filenames when intending to use these in conjunction with R.

Your working directory doesn't matter that much if you're not working with any external files (datasets, for example). However, in many cases you will be. For a given project, setting your R working directory to be the folder where all your necessary external files are stored is convenient if you want to avoid having to specify filepaths each time you read or write a file (you'll learn about all that in Chapter 8). This is an automatic feature of Projects in the RStudio IDE (see Section B.2.1).

1.3.5 R Packages

The base installation of R comes ready with a plethora of built-in commands for numeric calculations, common statistical analyses, and plotting and visualization. These commands can be used right from the outset and needn't be loaded or imported in any way. I refer to these functions as *built in* or *ready to use* in this text.

Slightly more specialized techniques and datasets are contained within packages (also referred to as *libraries*) of code. Using contributed packages is common, and you'll be doing so throughout this book, so it's

important to get comfortable with installing and loading the required libraries.

Section A.2 covers the relevant details concerning package download and installation from CRAN, but I'll provide a brief overview here.

Loading

A small number of recommended packages are included with the base distribution of R (listed in Section A.2.2). They don't need to be installed separately, but to use them, you need to load them by calling `library`. For example, you'll use the `MASS` package throughout this book (Venables & Ripley 2002). To load it (or any other installed package) and gain access to its functions and datasets, simply execute `library` at the prompt as follows:

```
R> library("MASS")
```

Note that calling `library` provides access to a package's functionality only for the running R session. When you close R and reopen a fresh instance, you'll need to reload any packages you want to use.

Installing

Thousands of contributed packages are not included with the typical R installation; to make them loadable in R, you must first download and install them from a repository (usually CRAN). The easiest way to do this is by using the `install.packages` function directly at the R prompt (you'll need an internet connection).

For example, one such package is `ks` (Duong 2007, 2024), which you'll use in Chapter 26. Executing the following will attempt to connect to your local CRAN mirror and download and install `ks`, as well as several packages upon which it relies (called *dependencies*):

```
R> install.packages("ks")
```

The console will show running output as the procedure completes. You need to install a package only once; thereafter, it will be available

for your R installation. You can then load your installed package (like `ks`) in any newly opened instance of R with a call to `library`, just as you did for `MASS`. Further detail on package installation is provided in Section A.2.3.

Updating

The maintainers of contributed packages periodically provide version updates to fix bugs and add functionality. Every so often, you might want to check for updates to your collection of installed packages.

From the R prompt, a simple execution of the following will attempt to connect to your set package repository (defaulting to CRAN), looking for versions of all your installed packages that are later than those you currently have:

```
R> update.packages()
```

Section A.3 offers more details about updating packages, and Section A.4 discusses alternate CRAN mirrors and repositories.

1.3.6 Help Files and Function Documentation

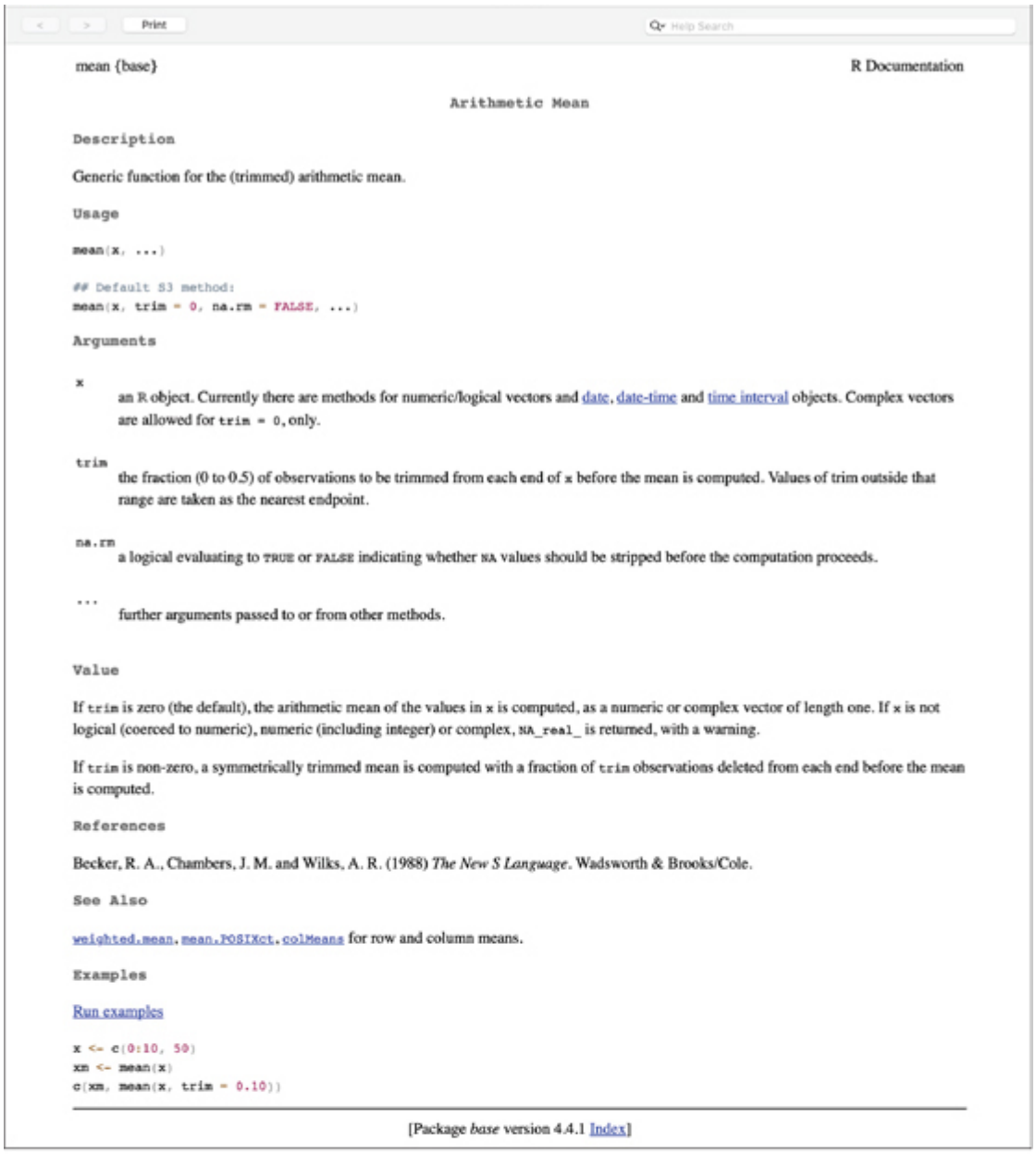
R comes with a suite of *help files*. You can use these to search for particular functionality, see precisely how to use a given function and specify its *arguments* (the values or objects you supply to the function when you execute it), clarify the role of arguments in the operations, learn about the form of any returned objects, find possible examples of using the function, and get details on how to cite any software or datasets.

To access the help file for a given command or other object, use the `help` function at the console prompt or use the convenient shortcut `?`. For example, consider the ready-to-use arithmetic mean function, `mean`:

```
R> ?mean
```

This brings up the file shown in the top image of Figure 1-4.





Print

Help Search

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	an R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical evaluating to TRUE or FALSE indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

References

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

See Also

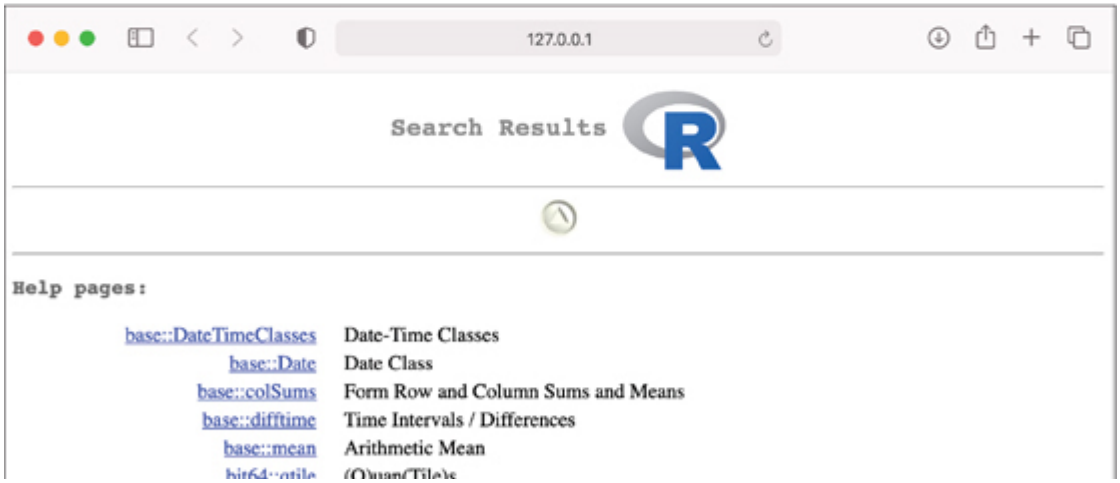
weighted.mean, mean.POSIXct, colMeans for row and column means.

Examples

Run examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

[Package *base* version 4.4.1 Index]



127.0.0.1

Search Results

Help pages:

base::DateTimeClasses	Date-Time Classes
base::Date	Date Class
base::colSums	Form Row and Column Sums and Means
base::difftime	Time Intervals / Differences
base::mean	Arithmetic Mean
bit64::qtile	(Q)uanTile(s

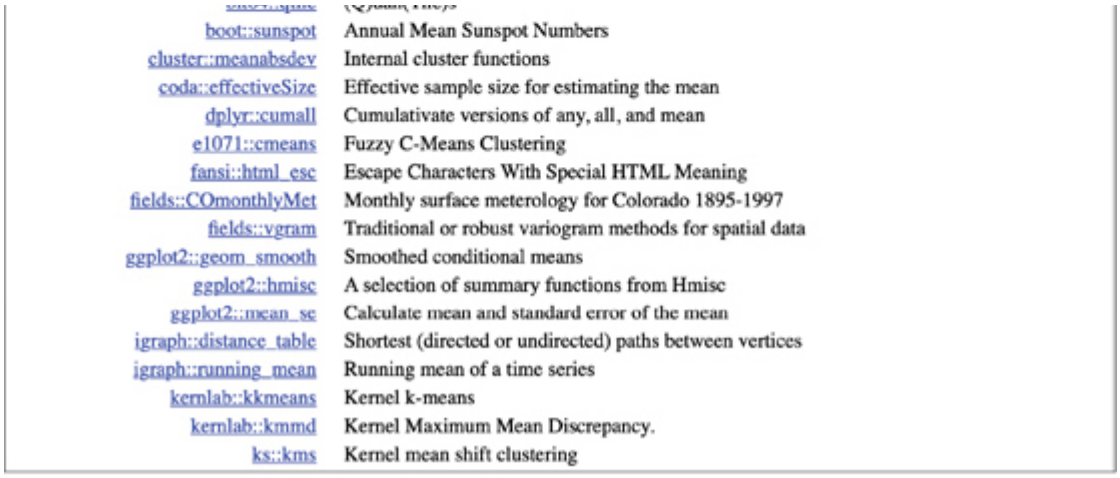


Figure 1-4: The R help file for the `mean` function (top) and the results of a help search for the string `"mean"` (bottom) in macOS

If you're unsure of the precise name of the desired function, you can search the documentation across all installed packages by using a character string (a statement in double quotes) passed to `help.search` (which provides various options as to how the supplied string is searched), or you can simply use `??` as a shortcut:

```
R> ??"mean"
```

This search, performed over all packages in your R installation, brings up a list of functions in a browser window, as shown in the bottom image of Figure 1-4. This list, alphabetically ordered by host package and function name, also briefly describes each item whose help file contains the string of interest. In this example, the fifth entry down in my installation is the arithmetic mean function.

All help files follow the general format shown in the top image of Figure 1-4. The length and level of detail in the file typically reflect the complexity of the operations carried out by the function. Most help files include the first three items listed here; the others are common but optional:

- The Description section provides a short statement about the operations carried out.

- The Usage section specifies the form of the function in terms of how it should be passed to the R console, including the defined order of the arguments and any default values (these are the arguments that are shown being set using =).
- The Arguments section gives more detail about what each argument does as well as its possible values.
- The nature of the object that's returned by the function (if anything) is specified under Value.
- The References section provides relevant citations for the command or the methodology behind the function.
- The help files for related functions are linked under See Also.
- The Examples section provides executable code that you can copy and paste into the console, demonstrating the function in action.

A help file may contain several more fields; functions with longer explanations often contain a Details section after Arguments. Common traps or mistakes made when calling the function are usually placed in a Warnings section, and additional information can be placed in Notes.

Although help files might seem quite technical when you're first starting out, I encourage you to keep looking at them. Even if you already know how a function works, getting comfortable with the layout and interpretation of function documentation is an important part of becoming a skilled R user.

1.4 Saving Work and Exiting R

Say you've spent a few bliss-filled hours coding in R, and now it's time to go home. When saving work in R, you need to pay attention to two details: any R objects that have been created (and stored) in the active session and any R script files written in an editor.

1.4.1 Workspaces

You can use menu items (for example, File ▸ Save in Windows, Workspace ▸ Save Workspace File in macOS, Session ▸ Save Workspace As in the RStudio IDE) to save and load workspace image files. An R *workspace image* contains all the information held in the R session at the time of exit and should be saved with a *.RData* extension. This file will include all objects you've created and stored (in other words, *assigned*) within the session (you'll see how to do this in Chapter 2), including those that may have been present from a previously loaded workspace file.

Essentially, loading a stored *.RData* file allows you to pick up from where you left off. At any point in an R session, you can execute `ls` at the prompt to list all objects, variables, and user-defined functions currently present in the active workspace.

Alternatively, you can use the R commands `save` (or `save.image`) and `load` at the console for handling workspace *.RData* files. These functions expect a `file` argument to which you pass the folder location and name of the target *.RData* file (see the corresponding help files `?save` and `?load` for further information on their usage). You'll see these functions in action in Section 8.4.1.

Note that saving a workspace image in this way doesn't retain the functionality of any contributed packages that were loaded in the previously active R session. As mentioned in Section 1.3.5, you'll also need to use `library` to load any packages required for your work for each new instance of R.

The quickest way to exit the software is to enter `q()` at the prompt:

```
R> q()
```

By default, this will bring up a dialog asking if you'd like to save the workspace image. In this case, choosing to save doesn't open a file browser to name your file but creates (or overwrites) a *no-name* file as one with a *.RData* extension in your working directory (refer to Section 1.3.4).

If an unnamed *.RData* file exists in the default working directory when a new instance of R is opened, the program will automatically load that default workspace; if that has happened, you'll be notified in the console's welcoming text.

NOTE

Alongside the .RData *file, R will automatically save a file containing a line-by-line history of all the commands executed in the console for the associated workspace in the same directory. This* .Rhistory *file allows you to scroll through the previously executed commands by using the keyboard directional arrows, as noted earlier.*

1.4.2 Scripts

For tasks requiring anything more than a handful of commands, you'll usually want to work in the built-in code editor. Saving your R scripts is therefore at least as important as saving a workspace, if not more so.

You save editor scripts as plain text files with a *.R* extension (noted in Section 1.3.1); this allows your operating system to associate these files with the R software by default. To save a script from the built-in editor, ensure that the editor is selected and navigate to File ▸ Save (or press CTRL-S in Windows or ⌘-S in macOS). To open a previously saved script, choose File ▸ Open Script (CTRL-O) in Windows, File ▸ Open Document (⌘-O) in macOS, or File ▸ Open File in RStudio.

Often you won't really need to save a workspace *.RData* file if your script files are saved. Once any required commands in a saved script are re-executed in a new R console, the objects created previously (in other words, those that might otherwise be contained within a saved *.RData* file) are simply created once more. This can be useful if you're working on multiple problems at one time because it can be easy to mistakenly overwrite an object when relying solely on the stand-alone default workspace. Keeping your collection of R scripts separate is therefore a

simple way to organize several projects without needing to worry about inadvertently overwriting anything important.

R also provides various ways to write individual objects, such as datasets and image files of plots, to disk, which you'll look at in Chapter 8.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`options`	Set various R options	Section 1.3.1, p. 6
`#`	A comment (ignored by interpreter)	Section 1.3.3, p. 7
`getwd`	Print current working directory	Section 1.3.4, p. 8
`setwd`	Set current working directory	Section 1.3.4, p. 8
`library`	Load an installed package	Section 1.3.5, p. 9
`install.packages`	Download and install packages	Section 1.3.5, p. 10
`update.packages`	Update installed packages	Section 1.3.5, p. 10
`help` or `?`	Function/object help file	Section 1.3.6, p. 10
`help.search` or `??`	Search help files	Section 1.3.6, p. 12
`q`	Quit R	Section 1.4.1, p. 13

2
NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS



In its simplest role, R functions as a mere desktop calculator. In this chapter, I'll discuss how to perform arithmetic and how to store results for use in subsequent calculations. Then, you'll learn about vectors, the R construct used to handle multiple values at once. You'll examine useful ways to manipulate vectors and take advantage of vector-oriented behavior.

2.1 R for Basic Math

All common arithmetical operations and mathematical functions are accessible at the console prompt. You can perform addition, subtraction, multiplication, and division with the symbols `+`, `-`, `*`, and `/`, respectively. You can create exponents (known as *powers* or *indices*) with the caret character (`^`) and control the order of calculations with parentheses (`()`).

2.1.1 Arithmetic

In R, standard mathematical rules apply and follow the left-to-right order of operations: parentheses, exponents, multiplication, division, addition, subtraction (PEMDAS). Here's an example in the console:

```
R> 2+3
[1] 5
R> 14/6
[1] 2.333333
R> 14/6+5
[1] 7.333333
R> 14/(6+5)
[1] 1.272727
R> 3^2
[1] 9
R> 2^3
[1] 8
```

R has functions for other common operations. For instance, to find the square root of any non-negative number, use the `sqrt` function:

```
R> sqrt(x=9)
[1] 3
R> sqrt(x=5.311)
[1] 2.304561
```

You may often find that you need to translate a complicated arithmetical formula into code for evaluation—for example, when replicating a calculation from a textbook or research paper. The next examples provide a mathematically expressed calculation alongside its execution in R:

$10^2 + \frac{3 \times 60}{8} - 3$

```
R> 10^2+3*60/8-3
[1] 119.5
```

$\frac{5^3 \times (6-2)}{61-3+4}$

```
R> 5^3*(6-2)/(61-3+4)
[1] 8.064516
```

$2^{2+1} - 4 + 64^{-2^{2.25-\frac{1}{4}}}$

```
R> 2^(2+1)-4+64^((-2)^(2.25-1/4))
[1] 16777220
```

$$\left(\frac{0.44\times(1-0.44)}{34}\right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

Note that some R expressions may require extra parentheses that aren't present in the mathematical notation. Missing or misplaced parentheses are common causes of arithmetical errors in R, especially when dealing with exponents. If the exponent is itself an arithmetical calculation, it must always appear in parentheses. For example, in the third expression, `2.25-1/4` must have parentheses around it. You must also use parentheses if the number being raised to some power is a calculation. R considers a negative number a calculation because it interprets, for example, `-2` as `-1*2`. This is why parentheses are needed around the `-2` in the third expression, and indeed why they are needed around the overall calculation in the fourth expression. It's important to highlight these issues early because you can easily overlook them in large chunks of code.

2.1.2 Logarithms and Exponentials

When researchers perform a *log transformation* on data, they rescale numbers according to the *logarithm*. If supplied a given number x and a value referred to as a *base*, the logarithm calculates the power to which you must raise the base to get to x. For example, the logarithm of $x = 243$ to base 3 (written mathematically as $\log_3 243$) is 5 because $3^5 = 243$. Data analysts often use log transformation to stabilize right-skewed distributions (see Section 15.2.4), and its other convenient mathematical properties feature in the definitions of formal statistical models.

For log transformations in R, supply the `log` function with the number to transform, assigned to the argument `x`, and the base, assigned to `base`, as follows:

```
R> log(x=243,base=3)
[1] 5
```

Here are some key rules and facts about the log transformation:

- Both x and the base must be positive.
- The log of any number x when the base is equal to x is 1.
- The log of $x = 1$ is always 0, regardless of the base.

Mathematicians often use a type of log transformation called the *natural log*, where the base is set to *Euler's number*, conventionally written as e and approximately equal to 2.718.

Euler's number gives rise to the *exponential function*, defined as e raised to the power of x, where x can be any number (negative, zero, or positive). The exponential function $f(x) = e^x$ is typically written as $\exp(x)$ and represents the *inverse* of the natural log such that $\exp(\log_e x) = \log_e \exp(x) = x$. The R command for the exponential function is `exp`, as in the following:

```
R> exp(x=3)
[1] 20.08554
```

The default behavior of `log` is to assume the natural log and use the base e, so to use this version of the transformation, simply don't supply a value to the `base` argument:

```
R> log(x=20.08554)
[1] 3
```

2.1.3 E-Notation

When R prints numbers beyond a certain threshold of significant figures, set at seven by default, it displays numbers using the scientific e-notation. The e-notation is typical to most programming languages (and many desktop calculators) to allow easier presentation of extreme values. In e-notation, any number x can be expressed as xey, which represents $x \times 10^y$. Consider the number 2,342,151,012,900. Two possible ways to represent this using this kind of notation are:

- 2.3421510129e12, which is equivalent to writing $2.3421510129 \times 10^{12}$
- 234.21510129e10, which is equivalent to writing $234.21510129 \times 10^{10}$

You can use any value for the power of *y*, but standard e-notation uses the power that places a decimal immediately after the first significant digit. For a *positive* power *+y*, you interpret the e-notation as "move the decimal point *y* positions to the *right*." For a *negative* power *– y*, the interpretation is "move the decimal point *y* positions to the *left*."

This is how R presents e-notation:

```
R> 2342151012900
[1] 2.342151e+12
R> 0.0000002533
[1] 2.533e-07
```

In the first example, R shows only the first seven significant digits and hides the rest. No information is lost, even if R hides digits; the e-notation is purely for ease of readability by the user and R still stores the extra digits, even though they aren't shown.

Finally, note that R must impose constraints on how extreme a number can be before it's treated as either infinity (for large numbers) or zero (for small numbers). These constraints depend on your system, and I'll discuss the technical details a bit more in Section 6.1.1. However, any modern desktop system will be precise enough by default for most computational and statistical endeavors in R.

Exercise 2.1

a. Using R, verify that

$$\frac{6a+42}{3^{4.2-3.62}} = 29.50556$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?

   i. `(-4)^2+2`

   ii. `-4^2+2`

   iii. `(-4)^(2+2)`

   iv. `-4^(2+2)`

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log_e 0.3$.

e. Compute the exponential transform of your answer to (d).

f. Identify R's representation of –0.000000004235463 22 when printing this number to the console.

2.2 Assigning Objects

So far, R has displayed the results of example calculations by printing them to the console. You can also save the results to perform further operations by *assign*ing the results to an *object* in the current workspace. Put simply, you store some item or result under a given name; to access the results later, you just enter that name without having to write out the calculation again. In this book, I use the terms *assign* and *store* interchangeably. Note that some programming books refer to a stored object as a *variable* because of the ability to easily overwrite that object and change it to something different, meaning that what it represents can vary throughout a session. However, I use the term *object* because we'll discuss variables in Part III as a distinct statistical concept.

In R, you may specify an assignment in two ways: using arrow notation (`<-`) or using a single equal sign (`=`). Here are both methods:

```
R> x <- -5
R> x
[1] -5

R> x = x + 1  # this overwrites the previous value of x
R> x
[1] -4

R> mynumber = 45.2

R> y <- mynumber*x
R> y
[1] -180.8
```

Although `=` and `<-` do the same thing, it's best to be consistent. Many users prefer `<-` because of the potential for confusion in using `=` (for example, I didn't mean that *x* is *mathematically* equal to *x* + 1). In this book, I do the same and reserve `=` for setting function arguments (see Section 2.3.2). Though you've used only numeric values so far, the procedure for assignment is universal for all types and classes of objects, which you'll examine in the coming chapters.

You can name objects almost anything as long as the name begins with a letter (in other words, not a number), avoids symbols (excepting underscores and periods), and avoids the handful of R's "reserved" words such as those used for defining special values (see Section 6.1) or for controlling code flow (see Chapter 10). You can find a summary of these naming rules in Section 9.1.2.

Using the `ls` command to examine the contents of the current workspace reveals the names of your stored objects in alphabetical order:

```
R> ls()
[1] "mynumber" "x"        "y"
```

To permanently delete an individual object in your workspace, use the `rm` command, supplying the name of the object targeted for removal in

double quotes:

```
R> rm("mynumber")
```

Calling `ls` again, the workspace now contains only the objects `x` and `y`:

```
R> ls()
[1] "x" "y"
```

Finally, completely clear all objects from the session by issuing a blanket removal like so:

```
R> rm(list=ls())
R> ls()
character(0)
```

The final output of `ls` confirms no more objects are left in the workspace.

Use `rm` with caution, especially when aggressively clearing your entire workspace as I've demonstrated here! You cannot recover removed objects (that is, without re-running the code that created them). While this isn't a big problem for rapidly reproduced results, you don't want to re-run an hours- or even days-long script due to a mistaken execution of `rm`. For such objects, it's safest to write them out to a file for safekeeping; you'll see how to do this in Chapter 8.

Exercise 2.2

a. Create an object that stores the value $3^2 \times 4^{\frac{1}{8}}$.

b. Overwrite your object in (a) by itself divided by 2.33. Print the result to the console.

c. Create a new object with the value -8.2×10^{-13}.

d. Print the result of multiplying (b) by (c) directly to the console.

e. Remove the object created in (c) from your workspace.

2.3 Vectors

You'll often want to perform the same calculations or comparisons upon multiple entities; for example, you might want to double every element when rescaling measurements in a dataset. While you can perform repetitive operations one entry at a time, this isn't ideal. R provides a far more efficient solution with *vectors*.

NOTE

To keep things simple, you'll continue working with only numeric elements, though vectors and their behaviors also apply in the case of non-numeric elements. You'll explore these other kinds of data in Chapter 4.

2.3.1 Vector Creation

The vector is the building block for handling multiple items in R. In a scientific sense, think of a vector as a collection of observations or measurements concerning a single variable—for example, the heights of 50 people or a month-long record of the number of coffees you drink daily. More complicated data structures may consist of several vectors.

To create a vector, you use the `c` function. The desired entries must be provided in parentheses separated by commas:

```
R> myvec <- c(1,3,1,42)
R> myvec
[1]  1  3  1 42
```

Vector entries can be single values, calculations, or previously stored items (including vectors themselves):

```
R> foo <- 32.1
R> myvec2 <- c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
R> myvec2
[1]    3.000000   -3.000000    2.000000    3.450000 1000.000000    8.000000
[7]    2.201271   32.100000
```

This code creates an object `foo` with the value 32.1, then creates a new vector assigned to the object `myvec2`. Some entries are arithmetical expressions, the results of which are stored in the vector. The last element, `foo`, is the numeric object we created first.

Let's look at another example:

```
R> myvec3 <- c(myvec,myvec2)
R> myvec3
[1]    1.000000    3.000000    1.000000   42.000000    3.000000   -3.000000
[7]    2.000000    3.450000 1000.000000    8.000000    2.201271   32.100000
```

This code creates and stores the vector `myvec3`, which contains the entries of `myvec` and `myvec2` appended in that order.

Named vectors, whereby each entry in a vector has been given a name, can provide more context about what values represent. To create a named vector, assign the entries of interest to your choice of name:

```
R> myvec4 <- c(name1=1,name2=3,anothername=1,oncemore=42)
R> myvec4
      name1       name2 anothername    oncemore
          1           3           1          42
```

Names in vectors are little more than cosmetic attributes. They don't affect the result of key operations, like arithmetic, when handling the vector. You can strip away name attributes by using `as.vector`:

```
R> as.vector(myvec4)
[1]  1  3  1 42
```

Name attributes are useful when handling complex data objects like *lists* and *data frames*, where they facilitate subsetting. We'll look at these in Chapter 5.

2.3.2 Sequences, Repetition, Sorting, and Lengths

The `seq`, `rep`, `sort`, and `length` functions are commonly associated with R vectors. These offer basic utilities like creating a vector based on repeating values and figuring out how many elements are contained within a vector object.

First, let's create an equally spaced sequence of increasing or decreasing numeric values. You'll need to do this often, such as when programming loops (see Chapter 10) or plotting data points (see Chapter 7). The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator:

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

You can read `3:27` as "from 3 to 27 (by 1)." This results in a numeric vector as if you had listed each number manually in parentheses with `c`, and the direction of the sequence (that is, either increasing or decreasing) is automatically determined by whether the value to the right of the colon is larger or smaller than the one to the left. As always, you can also provide either a previously stored value or a (strictly parenthesized) calculation when using the colon operator:

```
R> foo <- 5.3
R> bar <- foo:(-47+1.5)
R> bar
 [1]   5.3   4.3   3.3   2.3   1.3   0.3  -0.7  -1.7  -2.7  -3.7  -4.7
[12]  -5.7  -6.7  -7.7  -8.7  -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7
[23] -16.7 -17.7 -18.7 -19.7 -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7
[34] -27.7 -28.7 -29.7 -30.7 -31.7 -32.7 -33.7 -34.7 -35.7 -36.7 -37.7
[45] -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Notice that the colon operator enforces the steps of exactly 1, meaning the final value of `bar` in this example is not exactly –47 + 1.5 = –45.5. Rather, it is the closest value thereto without going beyond it.

Creating Sequences with seq

The `seq` command allows for more flexibility in creating sequences. This ready-to-use function takes in a `from` value, a `to` value, and a `by` value, then returns the corresponding sequence as a numeric vector:

```
R> seq(from=3,to=27,by=3)
[1]  3  6  9 12 15 18 21 24 27
```

This example returns a sequence with intervals of 3 rather than 1.

These kinds of sequences always start at the `from` number but won't necessarily include the `to` number, depending on what you're asking R to increase (or decrease) them by. This is the same as the behavior of the colon operator. For example, if you increase by even numbers and your sequence ends in an odd number, the final number won't be included. Instead of providing a `by` value, however, you can specify a `length.out` value to produce a vector with that many numbers, evenly spaced between the `from` and `to` values:

```
R> seq(from=3,to=27,length.out=40)
 [1]  3.000000  3.615385  4.230769  4.846154  5.461538  6.076923  6.692308
 [8]  7.307692  7.923077  8.538462  9.153846  9.769231 10.384615 11.000000
[15] 11.615385 12.230769 12.846154 13.461538 14.076923 14.692308 15.307692
[22] 15.923077 16.538462 17.153846 17.769231 18.384615 19.000000 19.615385
[29] 20.230769 20.846154 21.461538 22.076923 22.692308 23.307692 23.923077
[36] 24.538462 25.153846 25.769231 26.384615 27.000000
```

By setting `length.out` to `40`, R returns exactly 40 evenly spaced numbers from 3 to 27.

The value given to `by` must be negative for decreasing sequences. Here's an example:

```
R> foo <- 5.3
R> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
R> myseq
 [1]   5.3   2.9   0.5  -1.9  -4.3  -6.7  -9.1 -11.5 -13.9 -16.3 -18.7 -21.1
[13] -23.5 -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

This code uses the previously stored object `foo` as the `from` value and uses the parenthesized calculation `(-47+1.5)` as the `to` value. Given that `foo`

is greater than `(-47+1.5)`, the sequence can progress only in negative steps; here, we set `by` to be `-2.4`.

When using `length.out` to create decreasing sequences, however, it doesn't make sense to specify a "negative length." For the same `from` and `to` values, you can create a decreasing sequence of length 5 as shown here:

```
R> myseq2 <- seq(from=foo,to=(-47+1.5),length.out=5)
R> myseq2
[1]   5.3  -7.4 -20.1 -32.8 -45.5
```

There are shorthand ways of calling these functions, which you'll learn about in Chapter 9, but in these early stages I'll stick with the explicit usage.

Repeating with rep

If you want an object that simply repeats a certain value, use `rep` like so:

```
R> rep(x=1,times=4)
[1] 1 1 1 1
R> rep(x=c(3,62,8.3),times=3)
[1]  3.0 62.0  8.3  3.0 62.0  8.3  3.0 62.0  8.3
R> rep(x=c(3,62,8.3),each=2)
[1]  3.0  3.0 62.0 62.0  8.3  8.3
R> rep(x=c(3,62,8.3),times=3,each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0
[16] 62.0  8.3  8.3
```

Provide the `rep` function with the value or vector of values to repeat to its argument `x`, as well as a value to one or both of the arguments `times` and `each`. The value for `times` stipulates the number of times to repeat `x` in its entirety, and `each` provides the number of times to repeat each element of `x`. In the first line of this code, you repeat a single value four times. The other examples first use `rep` and `times` on a vector to repeat the entire vector, then `each` to repeat each member of the vector, and finally both `times` and `each` to do both at once.

If you specify neither `times` nor `each`, R's default is to treat the values of `times` and `each` as `1` so that a call of `rep(x=c(3,62,8.3))` will just return the originally supplied `x` with no changes.

As with `seq`, you can include the result of `rep` in a vector of the same data type, as shown in the following example:

```
R> foo <- 4
R> c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
[1]  3.00  8.30 32.00 32.00 32.00 32.00 -2.00 -1.25 -0.50  0.25  1.00
```

Here, I've constructed a vector where the third to sixth entries (inclusive) are governed by the evaluation of a `rep` command: The single value `32` `rep`eated `foo` times (where `foo` is stored as `4`). The last five entries are the result of an evaluation of `seq`, namely a sequence from –2 to 1 of length `foo+1` (5).

Sorting with sort

Sorting the elements of a vector in increasing or decreasing order is another operation that crops up in everyday tasks. The conveniently named `sort` function does just that, as shown in the following:

```
R> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1] -10.00  -1.00   2.50   3.44

R> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1]   3.44   2.50  -1.00 -10.00

R> foo <- seq(from=4.3,to=5.5,length.out=8)
R> foo
[1] 4.300000 4.471429 4.642857 4.814286 4.985714 5.157143 5.328571 5.500000
R> bar <- sort(x=foo,decreasing=TRUE)
R> bar
[1] 5.500000 5.328571 5.157143 4.985714 4.814286 4.642857 4.471429 4.300000

R> sort(x=c(foo,bar),decreasing=FALSE)
[1] 4.300000 4.300000 4.471429 4.471429 4.642857 4.642857 4.814286 4.814286
[9] 4.985714 4.985714 5.157143 5.157143 5.328571 5.328571 5.500000 5.500000
```

You supply a vector as the argument `x`, and the second argument, `decreasing`, indicates the direction in which you want to sort. This argument takes a type of value you haven't yet met: one of the all-important *logical* values. A logical value can be only one of two specific, case-sensitive values: `TRUE` or `FALSE`. Generally speaking, logicals indicate the satisfaction or failure of a certain *condition*, and they form an integral part of all programming languages. You'll investigate logical values in R further in Section 4.1. In regards to `sort`, set `decreasing=FALSE` to sort from smallest to largest, and `decreasing=TRUE` to sort from largest to smallest.

Finding a Vector Length with length

The `length` function returns the number of elements in a vector. Pass the vector in as the argument `x` like so:

```
R> length(x=c(3,2,8,1))
[1] 4

R> length(x=5:13)
[1] 9

R> foo <- 4
R> bar <- c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
R> length(x=bar)
[1] 11
```

Note that if you include entries that depend on the evaluation of other functions (in this case, calls to `rep` and `seq`), `length` tells you the number of entries *after* those inner functions have been executed.

Exercise 2.3

a. Create and store a sequence of values from 5 to –11 that progresses in steps of 0.3.

b. Overwrite the object from (a) using the same sequence with the order reversed.

c. Repeat the vector `c(-1,3,-5,7,-9)` twice, with each element repeated 10 times, and store the result. Display the result sorted from largest to smallest.

d. Create and store a vector that contains, in any configuration, the following:

   i. A sequence of integers from 6 to 12 (inclusive)

   ii. A threefold repetition of the value 5.3

   iii. The number –3

   iv. A sequence of nine values starting at 102 and ending at the number that is the total length of the vector created in (c)

e. Confirm that the length of the vector created in (d) is 20.

2.3.3 Subsets and Element Extraction

You may have noticed a curious feature in most of the results printed to the console so far. Immediately to the left of the output is a square-bracketed `[1]`. When the output is a long vector that spans the width of the console and wraps onto the following line, another square-bracketed number appears to the left of the new line. These numbers represent the *index* of the entry directly to the right. The index corresponds to the *position* of a value within a vector, which is why the first value always has a `[1]` next to it (even if it's the only value and not part of a larger vector).

These indexes allow you to retrieve specific elements from a vector, known as *subsetting*. Suppose you have a vector called `myvec` in your workspace. There are exactly `length(x=myvec)` entries in `myvec`, and each entry has a specific position: `1` or `2` or `3`, all the way up to `length(x=myvec)`. To access individual elements, ask R to return the values of `myvec` at

specific locations by entering the name of the vector followed by the desired position in square brackets:

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
R> length(x=myvec)
[1] 10
R> myvec[1]
[1] 5

R> foo <- myvec[2]
R> foo
[1] -2.3
```

You can use other functions with the indexing syntax too. Here, we retrieve the element at the index that's equal to the length of `myvec` by using the `length` function:

```
R> myvec[length(x=myvec)]
[1] -8
```

Because `length(x=myvec)` results in the final index of the vector (in this case, `10`), entering this phrase in the square brackets extracts the final element, `-8`. Similarly, you can extract the second-to-last element by subtracting 1 from the length; let's try that and assign the result to a new object:

```
R> myvec.len <- length(x=myvec)
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

As these examples show, the index may result from an arithmetical operation or from previously stored values. You can assign the result to a new object in your workspace with the `<-` notation. With your knowledge of sequences, use the colon notation with the length of the specific vector to obtain all possible index positions of the vector:

```
R> 1:myvec.len
[1]  1  2  3  4  5  6  7  8  9 10
```

Explicitly creating such a vector of possible index positions is useful when programming loops; you'll see examples of this in Section 10.2.

To omit individual elements, use *negative* versions of the indexes supplied in the square brackets. We'll continue with the objects `myvec`, `foo`, `bar`, and `myvec.len` and remove the first element of `myvec`:

```
R> myvec[-1]
[1]    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0    -8.0
```

Using similar notation, assign the contents of `myvec`, minus its second element, to a new object `baz`:

```
R> baz <- myvec[-2]
R> baz
[1]     5     4     4     4     6     8    10 40221    -8
```

Again, you can also use the result of a calculation in the square brackets to act as the index, like so:

```
R> qux <- myvec[-(myvec.len-1)]
R> qux
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0 -8.0
```

Using the square-bracket operator to extract or delete values from a vector doesn't change the original vector you're subsetting *unless* you explicitly overwrite the vector with the subsetted version. For instance, in this example, `qux` is a new vector defined as `myvec` without its second-to-last entry, but in your workspace, `myvec` itself *remains unchanged*. In other words, subsetting vectors in this way returns the requested elements, which you can assign to a new object, but doesn't alter the original object in the workspace.

Now, as an example that combines several of the aforementioned functions and vector operations, suppose you want to piece `myvec` back together from `qux` and `bar`. You can call something like this:

```
R> c(qux[-length(x=qux)],bar,qux[length(x=qux)])
 [1]     5.0    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0
```

```
[10]    -8.0
```

Here, you use `c` to reconstruct the vector in three parts: `qux[-length(x=qux)]`, the object `bar` defined earlier, and `qux[length(x=qux)]`. For clarity, let's examine each part in turn.

The expression `qux[-length(x=qux)]` returns the values of `qux` except for its last element:

```
R> length(x=qux)
[1] 9
R> qux[-length(x=qux)]
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0
```

Now you have a vector that's the same as the first eight entries of `myvec`.

Earlier, you stored `bar` as the following:

```
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

This is precisely the second-to-last element of `myvec` that `qux` is missing. So, you'll slot this value in after `qux[-length(x=qux)]`.

Finally, you need the last element of `qux` that matches the last element of `myvec`. This is extracted from `qux` (not deleted as earlier) using `length`:

```
R> qux[length(x=qux)]
[1] -8
```

As with most operations in R, you're not restricted to doing things one by one. You can also subset objects by using *vectors of indexes*, rather than individual indexes. Using `myvec` again, pass a vector to the square brackets to get the following:

```
R> myvec[c(1,3,5)]
[1] 5 4 4
```

This returns the first, third, and fifth elements of `myvec` in one go.

Another convenient trick when subsetting involves using the colon operator (discussed in Section 2.3.2) to create a sequence of indexes. Here's an example:

```
R> 1:4
[1] 1 2 3 4
R> foo <- myvec[1:4]
R> foo
[1]  5.0 -2.3  4.0  4.0
```

This provides the first four elements of `myvec` (recall that the colon operator returns a numeric vector, so you don't need to explicitly wrap this using `c`).

The order of the returned elements depends entirely upon the index vector you supply to the square brackets. For example, here I extract the three elements at the end of `foo` from position 4 to position 2, in that order, using the appropriate index vector:

```
R> length(x=foo):2
[1] 4 3 2
R> foo[length(foo):2]
[1]  4.0  4.0 -2.3
```

You can also use `rep` to repeat an index, as shown here:

```
R> indexes <- c(4,rep(x=2,times=3),1,3)
R> indexes
[1] 4 2 2 2 1 3
R> foo[indexes]
[1]  4.0 -2.3 -2.3 -2.3  5.0  4.0
```

This example goes slightly beyond the technical definition of "subsetting"; by using an index vector, you can create an entirely new vector of any length consisting of some or all of the elements in the original vector. Your index vector can contain the desired element positions in any order and permits repetition of indexes.

You can omit some elements of a vector, returning the remaining elements. For example, to create a vector after removing the first and

third elements of `foo`, execute the following:

```
R> foo[-c(1,3)]
[1] -2.3  4.0
```

Note that it isn't possible to mix positive and negative indexes in a single index vector.

Sometimes you need to overwrite certain elements in an existing vector with new values. To do so, first specify the elements to overwrite using square brackets and then use the assignment operator to assign the new values. Here's an example:

```
R> bar <- c(3,2,4,4,1,2,4,1,0,0,5)
R> bar
[1] 3 2 4 4 1 2 4 1 0 0 5
R> bar[1] <- 6
R> bar
[1] 6 2 4 4 1 2 4 1 0 0 5
```

This overwrites the first element of `bar`, `3`, with the new value `6`. When selecting multiple elements to replace, specify a single value to replace them all or enter a vector of values equal in length to the number of elements selected to replace them one for one. Let's try the latter method with `bar`:

```
R> bar[c(2,4,6)] <- c(-2,-0.5,-1)
R> bar
[1]  6.0 -2.0  4.0 -0.5  1.0 -1.0  4.0  1.0  0.0  0.0  5.0
```

Here, you overwrite the second, fourth, and sixth elements with `-2`, `-0.5`, and `-1`, respectively; all else remains the same. By contrast, the following code overwrites elements `7` to `10` (inclusive), replacing them with `100`:

```
R> bar[7:10] <- 100
R> bar
[1]   6.0  -2.0   4.0  -0.5   1.0  -1.0 100.0 100.0 100.0 100.0   5.0
```

This section has focused on just one of the two main methods, or "flavors," of vector element extraction in R. You'll look at the alternative method, using logical flags, in Section 4.1.5.

Exercise 2.4

a. Create and store a vector that contains the following, in this order:
   i. A sequence of length five, from 3 to 6 (inclusive)
   ii. A twofold repetition of the vector `c(2,-5.1,-33)`
   iii. The value $\frac{7}{42} + 2$
b. Extract the first and last elements of your vector from (a), storing them as a new object.
c. Store as a third object the values returned by omitting the first and last values of your vector from (a).
d. Use only (b) and (c) to reconstruct (a).
e. Overwrite (a) with the same values sorted from smallest to largest.
f. Use the colon operator as an index vector to reverse the order of (e), and confirm this is identical to using `sort` on (e) with `decreasing=TRUE`.
g. Create a vector from (c) that repeats the third element of (c) three times, the sixth element four times, and the last element once.
h. Create a new vector as a copy of (e) by assigning (e), as-is, to a newly named object. Using this new copy of (e), overwrite the first, the fifth to the seventh (inclusive), and

the last element with the values 99 to 95 (inclusive), respectively.

2.3.4 Vector-Oriented Behavior

Vectors allow R to carry out operations on multiple elements simultaneously with speed and efficiency. This *vector-oriented*, *vectorized*, or *element-wise* behavior is a key feature of the language that you'll examine here through some examples of rescaling measurements.

First, use the colon operator to create a sequence of six values between 5.5 and 0.5, in decrements of 1:

```
R> foo <- 5.5:0.5
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo-c(2,4,6,8,10,12)
[1]   3.5   0.5  -2.5  -5.5  -8.5 -11.5
```

From this vector, you subtract an entirely new vector containing 2, 4, 6, 8, 10, and 12. R matches up the elements according to their respective positions and performs the given operation—here a subtraction—on each corresponding pair of elements. This results in R subtracting the first element of `c(2,4,6,8,10,12)` from the first element of `foo` (5.5 – 2 = 3.5), subtracting the second element of `c(2,4,6,8,10,12)` from the second element of `foo` (4.5 – 4 = 0.5), and so on. Thus, rather than inelegantly cycling through each element in turn (as you could do by hand or by explicitly using a loop), R permits a fast and efficient alternative with vector-oriented behavior. Figure 2-1 illustrates how this type of calculation works. Notice that the positions of the elements are crucial to the final result; elements in differing positions have no effect on one another.

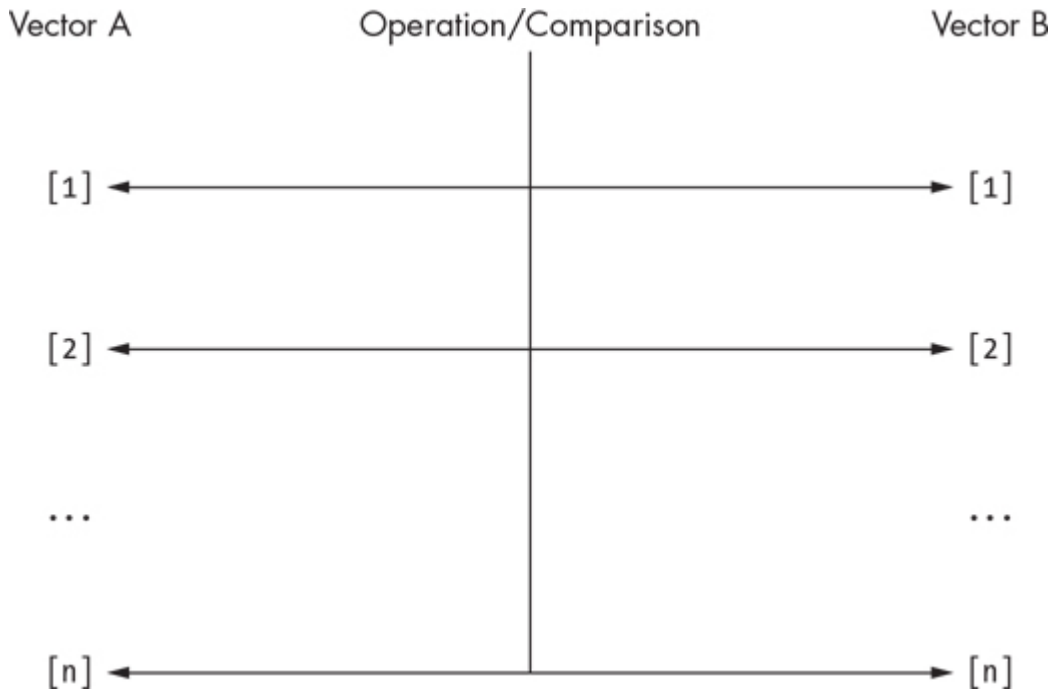


Figure 2-1: The element-wise behavior of a comparison or operation carried out on two vectors of equal length. The operation is performed by matching up the element positions.

Element-wise operations are more complicated when using vectors of different lengths. R essentially attempts to replicate, or *recycle*, the shorter vector as many times as necessary to match the length of the longer vector, before completing the specified operation. As an example, suppose you want to alternate the entries of `foo` shown earlier as negative and positive. You can explicitly multiply `foo` by `c(1,-1,1,-1,1,-1)`, but you don't need to write out the full latter vector. Instead, write the following:

```
R> bar <- c(1,-1)
R> foo*bar
[1]  5.5 -4.5  3.5 -2.5  1.5 -0.5
```

Here, you apply `bar` repeatedly throughout the length of `foo` until completion. This recycling makes sense to R because the length of the longer vector (`foo`) is evenly divisible by the length of the shorter vector (`bar`). The left plot of Figure 2-2 illustrates this particular example.

Now let's see what happens when the vector lengths are *not* evenly divisible:

```
R> baz <- c(1,-1,0.5,-0.5)
R> foo*baz
[1]  5.50 -4.50  1.75 -1.25  1.50 -0.50
Warning message:
In foo * baz :
  longer object length is not a multiple of shorter object length
```

R has matched the first four elements of `foo` with the entirety of `baz`, but it's not able to fully repeat the vector. The repetition has been attempted, with the first two elements of `baz` being matched with the last two of the longer `foo`, though not without a protest from R, which notifies the user of the unevenly divisible lengths (you'll look at warnings in more detail in Section 12.1). The plot on the right in Figure 2-2 illustrates the behavior in this example.

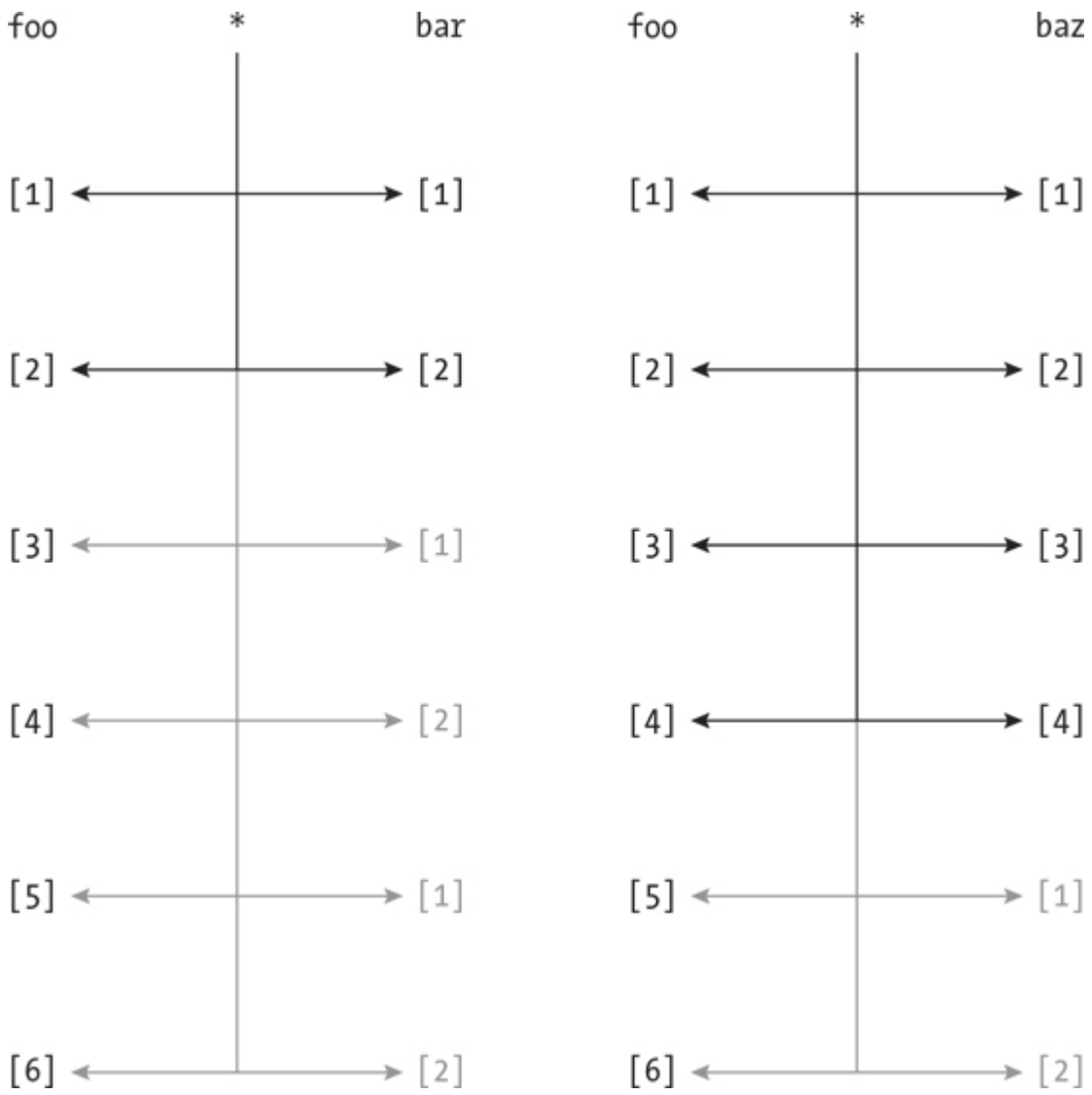


Figure 2-2: An element-wise operation on two vectors of differing lengths. Left: `foo` multiplied by `bar`; lengths are evenly divisible. Right: `foo` multiplied by `baz`; lengths are not evenly divisible and a warning is issued.

As noted in Section 2.3.3, single values are vectors of length 1, so you can use a single value to repeat an operation on all the values of a vector of any length. Here's an example using the same vector `foo`:

```
R> qux <- 3
R> foo+qux
[1] 8.5 7.5 6.5 5.5 4.5 3.5
```

This is far easier than executing `foo+c(3,3,3,3,3,3)` or the more general `foo+rep(x=3,times=length(x=foo))`. Operating on vectors by using a single value in this fashion is quite common, for example, when rescaling or translating a set of measurements by some constant amount.

Another benefit is that vectorized functions can complete potentially laborious tasks. For example, to sum or multiply all entries in a numeric vector, just use a built-in function.

Recall `foo`, shown earlier:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
```

You can find the sum of these six elements with

```
R> sum(foo)
[1] 18
```

and their product with:

```
R> prod(foo)
[1] 162.4219
```

Far from being just convenient, a vectorized operation is faster than a correspondingly coded iterative approach like a loop. Much of R's functionality is designed specifically for certain data structures, ensuring neatness of code as well as optimization of performance.

Lastly, as mentioned earlier, this vector-oriented behavior applies similarly to overwriting multiple elements. Again using `foo`, examine the following:

```
R> foo[c(1,3,5,6)] <- c(-99,99)
R> foo
[1] -99.0   4.5  99.0   2.5 -99.0  99.0
```

Here, you overwrite four specific elements using a vector of length 2, which you recycle in the same familiar fashion. Again, the length of the vector of replacements must evenly divide the number of elements

being overwritten, or you'll get another warning when R cannot complete a full-length recycle.

Exercise 2.5

a. Using a vector of length 3, turn the vector `c(2,0.5,1,2,0.5,1,2,0.5,1)` into a vector of only `1`s.
Hint: Dividing any number by itself will always result in 1).

b. The conversion from a temperature measurement in degrees Fahrenheit F to Celsius C is performed using the following equation:

$$C = \frac{5}{9}(F - 32)$$

Use vector-oriented behavior in R to convert the temperatures 45, 77, 20, 19, 101, 120, and 212 in degrees Fahrenheit to degrees Celsius.

c. Use the vector `c(2,4,6)` and the vector `c(1,2)` in conjunction with `rep` and `*` to produce the vector `c(2,4,6,4,8,12)`.

d. Overwrite the middle four elements of the resulting vector from (c) with the two recycled values `-0.1` and `-100`, in that order.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`+`, `*`, `-`, `/`, `^`	Arithmetic	Section 2.1, p. 15
`sqrt`	Square root	Section 2.1.1, p. 16

Function/operator	Brief description	First occurrence
`log`	Logarithm	Section 2.1.2, p. 17
`exp`	Exponential	Section 2.1.2, p. 17
`<-`, `=`	Object assignment	Section 2.2, p. 19
`ls`	List objects in workspace	Section 2.2, p. 20
`rm`	Delete/remove objects from workspace	Section 2.2, p. 20
`c`	Vector creation	Section 2.3.1, p. 21
`:`, `seq`	Sequence creation	Section 2.3.2, p. 23
`rep`	Value/vector repetition	Section 2.3.2, p. 24
`sort`	Vector sorting	Section 2.3.2, p. 25
`length`	Determine vector length	Section 2.3.2, p. 26
`[ ]`	Vector subsetting/extraction	Section 2.3.3, p. 27
`sum`	Sum all vector elements	Section 2.3.4, p. 34
`prod`	Multiply all vector elements	Section 2.3.4, p. 34

3
MATRICES AND ARRAYS



Now that you have a solid handle on using vectors in R, you can start exploring matrices. A *matrix* is several vectors stored together. While you describe the size of a vector by its length, you determine a matrix's size by the number of rows and columns. You can also create higher-dimensional structures consisting of multiple matrices called *arrays*. In this chapter, you'll begin working with matrices before increasing the dimension to form arrays.

3.1 Two-Dimensional Matrix

The matrix is an important mathematical construct essential to many statistical methods. You typically describe a matrix A as an $m \times n$ matrix; that is, A has m rows and n columns. This means A will have a total of mn entries, where each entry $a_{i,j}$ has a unique position given by its specific row ($i = 1, 2, \ldots, m$) and column ($j = 1, 2, \ldots, n$).

You can therefore express a matrix as follows:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$$

3.1.1 Matrix Creation

To create a matrix in R, use the aptly named `matrix` command, providing the entries of the matrix to the `data` argument as a vector:

```
R> A <- matrix(data=c(-3,2,893,0.17),nrow=2,ncol=2)
R> A
     [,1]   [,2]
[1,]   -3 893.00
[2,]    2   0.17
```

Make sure that the number of entries in this vector matches exactly with the number of desired rows (`nrow`) and columns (`ncol`). You can elect not to supply `nrow` and `ncol` when calling `matrix`, in which case R's default behavior is to return a single-column matrix of the entries in `data`. For example, `matrix(data=c(-3,2,893,0.17))` is identical to `matrix(data=c(-3,2,893,0.17), nrow=4,ncol=1)`.

Filling Direction

Be aware of how R fills up the matrix using the entries from `data`. In the previous example, R fills the 2×2 matrix `A` in a *column-by-column* fashion when reading the `data` entries from left to right, meaning it fills the first column before entering any data into the next, and so on. To control how R fills in the matrix, use the argument `byrow`, as shown explicitly set to its default value in the following example:

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Here, I've instructed R to provide a 2×3 matrix containing the digits 1 through 6. With `byrow=FALSE`, you're telling R to fill this 2×3 structure in a column-wise fashion, by filling each column before moving to the next, reading the `data` argument vector from left to right. As the default behavior, when `byrow` isn't supplied by the user, the software will assume it to be set at `FALSE`. Figure 3-1 illustrates this concept.

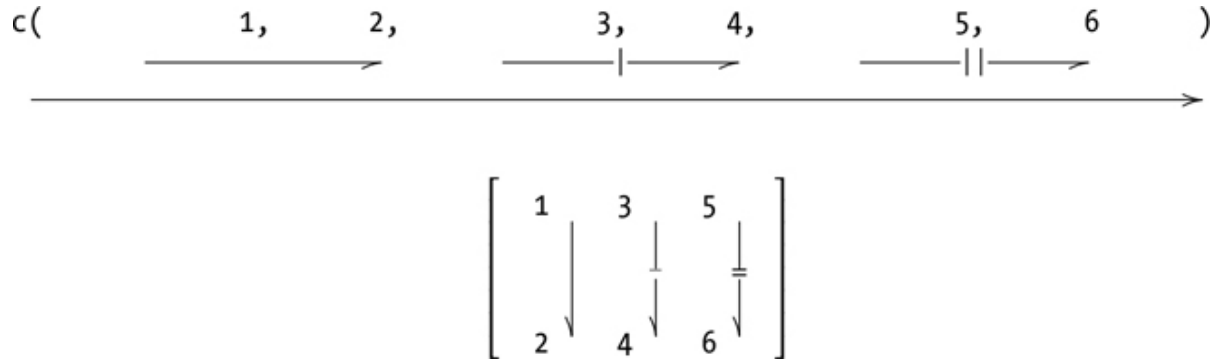


Figure 3-1: Filling a 2×3 matrix in a column-wise fashion with `byrow=FALSE` (R default)

Now let's repeat the same line of code but set `byrow=TRUE`:

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

R fills the resulting 2×3 structure in a row-wise fashion, as Figure 3-2 shows.

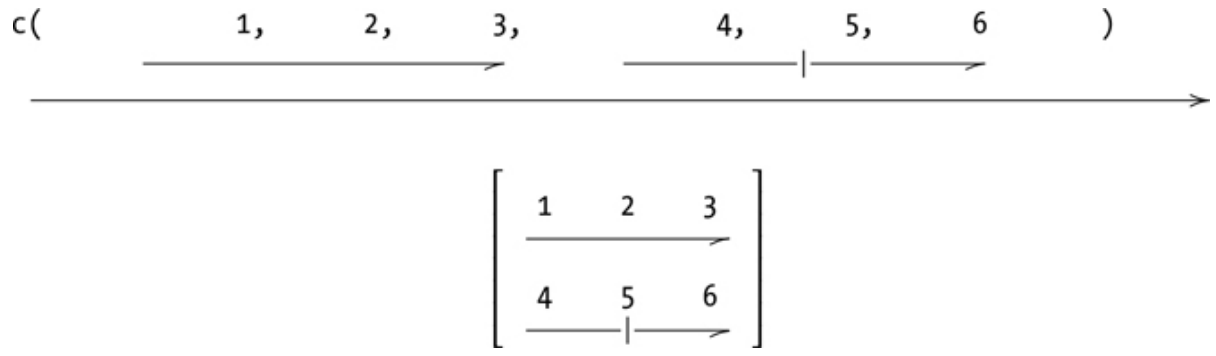


Figure 3-2: Filling a 2×3 matrix row-wise, via `byrow=TRUE`

Like naming elements of stand-alone vectors (see Section 2.3.1), you can stipulate names to label the rows and columns of a matrix. We'll explore this as part of a more formal discussion of R object attributes in Section 6.2.1.

Row and Column Bindings

To quickly build a matrix, bind together vectors of equal length using R's built-in `rbind` and `cbind` functions. You can treat each vector either as a row with `rbind` or as a column with `cbind`. Say you have the two vectors, `1:3` and `4:6`. You can reconstruct the 2×3 matrix in Figure 3-2 using `rbind` as follows:

```
R> rbind(1:3,4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

The `rbind` function has bound together the vectors as two rows of a matrix, with the top-to-bottom order of the rows matching the order of the vectors supplied to `rbind`. You can construct the same matrix using `cbind` as follows:

```
R> cbind(c(1,4),c(2,5),c(3,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, you have three vectors each of length 2. The `cbind` command glues these three vectors together in the order they were supplied, and each vector becomes a column of the resulting matrix.

Matrix Dimensions

The `dim` function provides the dimensions of a matrix stored in your workspace. The following code creates a matrix and queries some information about it:

```
R> mymat <- rbind(c(1,3,4),5:3,c(100,20,90),11:13)
R> mymat
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    4    3
[3,]  100   20   90
[4,]   11   12   13

R> dim(mymat)
[1] 4 3
R> nrow(mymat)
[1] 4
R> ncol(mymat)
[1] 3
R> dim(mymat)[2]
[1] 3
```

Having defined a matrix `mymat` using `rbind`, you can confirm its dimensions with `dim`, which returns a vector of length 2 representing the number of rows and columns; the `dim` function always supplies the number of rows first, then the columns. This example also shows two related functions: The `nrow` function provides only the number of rows, and `ncol` provides the number of columns. In the last command shown, you use `dim` and your knowledge of vector subsetting to extract the same result that `ncol` gives you.

3.1.2 Subsets

Extracting and subsetting elements from matrices in R is similar to extracting elements from vectors, with the only complication being an additional dimension. You still use the square-bracket operator, but now you must provide both a row *and* a column position, given strictly in the order of `[row,column]`. Let's start by creating a 3×3 matrix, which we'll use for the examples in this section:

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A
     [,1]  [,2] [,3]
```

```
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To tell R to "look at the third row of `A` and give me the element from the second column," execute the following:

```
R> A[3,2]
[1] 105.5
```

As expected, you're given the element at position `[3,2]`.

Extracting Rows, Columns, and Diagonals

To extract an entire row or column from a matrix, simply specify the desired row or column number and leave the other value blank. Note that you must still include the comma that separates the row and column numbers, since this is how R distinguishes between a request for a row and a request for a column. The following returns the second column of `A`:

```
R> A[,2]
[1]  91.0   0.1 105.5
```

The following examines the first row:

```
R> A[1,]
[1]  0.3 91.0 -4.2
```

When an extraction (or a deletion, covered in a moment) results in a single value, single row, or single column, R always returns stand-alone vectors comprising the requested values, rather than a matrix.

When you perform more complicated extractions—for example, requesting whole rows or columns, or multiple rows or columns—the result is returned as a new matrix. Consider the following subsets:

```
R> A[2:3,]
     [,1]  [,2] [,3]
[1,]  4.5   0.1  8.2
```

```
[2,] 55.3 105.5 27.9

R> A[,c(3,1)]
     [,1] [,2]
[1,] -4.2  0.3
[2,]  8.2  4.5
[3,] 27.9 55.3

R> A[c(3,1),2:3]
      [,1] [,2]
[1,] 105.5 27.9
[2,]  91.0 -4.2
```

The first command returns the second and third rows of `A`, and the second command returns the third and first columns of `A`. The last command accesses the third and first rows of `A`, in that order, and from those rows it returns the second and third column elements.

You can also identify the values along the diagonal of a square matrix (that is, a matrix with an equal number of rows and columns) using the `diag` command:

```
R> diag(x=A)
[1]  0.3  0.1 27.9
```

This returns a vector with the elements along the diagonal of `A`, starting at `A[1,1]`.

Omitting and Overwriting

To delete or omit elements from a matrix, you use square brackets again, but this time with negative indexes. The following provides all of `A` except its second column:

```
R> A[,-2]
     [,1] [,2]
[1,]  0.3 -4.2
[2,]  4.5  8.2
[3,] 55.3 27.9
```

The following removes the first row from A and retrieves the third and second column values, in that order, from the remaining two rows:

```
R> A[-1,3:2]
     [,1]  [,2]
[1,]  8.2   0.1
[2,] 27.9 105.5
```

The following produces A without its first row and second column:

```
R> A[-1,-2]
     [,1] [,2]
[1,]  4.5  8.2
[2,] 55.3 27.9
```

Lastly, this deletes the first row and then deletes the second and third columns from what's left:

```
R> A[-1,-c(2,3)]
[1]  4.5 55.3
```

Note that this final operation leaves you with only the last two elements of the first column of A, so this result is returned as a stand-alone vector rather than a matrix.

To overwrite particular elements, or entire rows or columns, identify the elements to be replaced and then assign the new values, as you did with vectors in Section 2.3.3. The new elements can be a single value, a vector of the same length as the number of elements to be replaced, or a vector whose length evenly divides the number of elements to be replaced. To illustrate this, let's first create a copy of A and call it B:

```
R> B <- A
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

The following overwrites the second row of `B` with the sequence `1`, `2`, and `3`:

```
R> B[2,] <- 1:3
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  1.0   2.0  3.0
[3,] 55.3 105.5 27.9
```

The following overwrites the second column elements of the first and third rows with `900`:

```
R> B[c(1,3),2] <- 900
R> B
     [,1] [,2] [,3]
[1,]  0.3  900 -4.2
[2,]  1.0    2  3.0
[3,] 55.3  900 27.9
```

Next, you replace the third column of `B` with the values in the third *row* of `B`:

```
R> B[,3] <- B[3,]
R> B
     [,1] [,2]  [,3]
[1,]  0.3  900  55.3
[2,]  1.0    2 900.0
[3,] 55.3  900  27.9
```

To try R's vector recycling in a matrix, let's overwrite the first and third column elements of rows 1 and 3 (a total of four elements) with the two values `-7` and `7`:

```
R> B[c(1,3),c(1,3)] <- c(-7,7)
R> B
     [,1] [,2] [,3]
[1,]   -7  900   -7
```

```
[2,]    1    2  900
[3,]    7  900    7
```

The vector of length 2 replaces the four elements in a *column-wise fashion*. The replacement vector `c(-7,7)` overwrites the elements at positions (1, 1) and (3, 1), in that order, and then repeats to overwrite (1, 3) and (3, 3), in that order.

You can control the way in which R fills in replacement elements using the ordering of the element index. Consider the following example:

```
R> B[c(1,3),2:1] <- c(65,-65,88,-88)
R> B
     [,1] [,2] [,3]
[1,]   88   65   -7
[2,]    1    2  900
[3,]  -88  -65    7
```

The values in the replacement vector have overwritten the four elements I specified by index, again in a column-wise fashion. In this case, because I specified the first and second columns in reverse order, the overwriting proceeded accordingly, filling the second column before moving to the first. Position (1, 2) is matched with `65`, followed by (3, 2) with `-65`; then (1, 1) becomes `88`, and (3, 1) becomes `-88`.

As a final shortcut, you can handle the diagonal elements of a square matrix using the `diag` command, avoiding explicit indexes. Here, we overwrite the diagonal elements with zeros:

```
R> diag(x=B) <- rep(x=0,times=3)
R> B
     [,1] [,2] [,3]
[1,]    0   65   -7
[2,]    1    0  900
[3,]  -88  -65    0
```

Exercise 3.1

a. Construct and store a 4×2 matrix that's filled row-wise with the values 4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, and 6.5, in that order.

b. Confirm the dimensions of the matrix from (a) are 3×2 if you remove any one row.

c. Overwrite the second column of the matrix from (a) with that same column sorted from smallest to largest.

d. What does R return if you delete the fourth row and the first column from (c)? Use `matrix` to convert the result to a single-column matrix, rather than a vector.

e. Store the bottom four elements of (c) as a new 2×2 matrix.

f. Overwrite, in this order, the elements of (c) at positions (4, 2), (1, 2), (4, 1), and (1, 1) with $-\frac{1}{2}$ of the two values on the diagonal of (e).

3.1.3 Matrix Operations and Algebra

Matrices in R have two main usages. The first is as a purely computational tool to store and operate on results, as you've seen so far. Second, you can also use matrices for their mathematical properties in particular calculations, such as expressing regression model equations using matrix multiplication. This distinction is important because the mathematical behavior of matrices is not always the same as the more generic data handling behavior. Here, I'll briefly describe some particular types of matrices, some of the most common mathematical operations involving matrices, and the corresponding functionality in R. If the mathematical behavior of matrices isn't of interest to you, feel free to skip this section and refer to it later as needed.

Transpose

The *transpose* of a matrix is a copy of the original matrix with its columns used as rows and vice versa. That is, for any $m \times n$ matrix A, its transpose, written as $A^\top$, is the $n \times m$ matrix obtained by writing either its columns as rows or its rows as columns.

Here's an example:

$$\text{If } A = \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix}, \text{ then } A^\top = \begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix}$$

In R, the transpose of a matrix is found with the function `t` provided with the name of the matrix. Let's create a new matrix and then transpose it:

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
R> t(A)
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
```

If you transpose the transpose of A, you'll recover the original matrix:

```
R> t(t(A))
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
```

Identity Matrix

The *identity matrix*, written as I_m, is a square $m \times m$ matrix with 1s on the diagonal and 0s elsewhere. Identity matrices crop up in many different areas of mathematics and statistics where matrix algebra is performed.

Here's an example:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can create an identity matrix of any dimension using the standard `matrix` function, but the `diag` function provides a quicker approach. Earlier, I used `diag` on an existing matrix to extract or overwrite its diagonal elements. To build an identity matrix, you use it as follows:

```
R> A <- diag(x=3)
R> A

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Here, `diag` easily produces an identity matrix. The behavior of `diag` depends on what you supply as its argument `x`. If, as earlier, `x` is an existing matrix, `diag` retrieves the diagonal elements of the matrix. If `x` is a single positive integer, as is the case here, then `diag` produces the identity matrix of the corresponding dimension. By supplying 3, R therefore yields a 3×3 matrix. You can find more uses of `diag` on its help page.

Scalar Multiple

A *scalar* value is a single, univariate value. If you multiply any matrix A by a scalar value a, you'll get a matrix in which every individual element

is multiplied by *a*. We call this process, where each operation is performed on each element in turn, *element-wise multiplication*.

Here's an example:

$$2 \times \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 10 & 4 \\ 12 & 2 & 8 \end{bmatrix}$$

R naturally performs this multiplication in an element-wise manner. To perform scalar multiplication of a matrix, use the standard arithmetic `*` operator:

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> a <- 2
R> a*A
     [,1] [,2] [,3]
[1,]    4   10    4
[2,]   12    2    8
```

Here, I create a matrix `A` and multiply it by 2, bound to `a`.

Addition and Subtraction

You can also add or subtract two matrices of equal size, performed in an element-wise fashion. Corresponding elements are added or subtracted from one another.

For example:

$$\begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 8.1 \\ 3 & 8.2 \\ 6 & -9.8 \end{bmatrix} = \begin{bmatrix} 4 & -2.1 \\ 2 & -7.2 \\ -4 & 13.8 \end{bmatrix}$$

Each element in one matrix interacts with its corresponding element in the other. You can add or subtract any two equally-sized matrices with the standard `+` and `-` symbols:

```
R> A <- cbind(c(2,5,2),c(6,1,4))
R> A
```

```
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
R> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
R> B
     [,1] [,2]
[1,]   -2  8.1
[2,]    3  8.2
[3,]    6 -9.8
R> A-B
     [,1] [,2]
[1,]    4 -2.1
[2,]    2 -7.2
[3,]   -4 13.8
```

Multiplication

To *multiply* two matrices A and B of size $m \times n$ and $p \times q$, the column length of the first must be the same as the row length of the second so that $n = p$. The resulting matrix $A \cdot B$ has the size $m \times q$. The elements of a product of two compatible matrices are calculated in a row-by-column fashion, where the value at position $(AB)_{i,j}$ is computed by element-wise multiplication of the entries in row i of A by the entries in column j of B, summing the result.

Here's an example:

$$\begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \times 3 + 5 \times (-1) + 2 \times 1 & 2 \times (-3) + 5 \times (1) + 2 \times 5 \\ 6 \times 3 + 1 \times (-1) + 4 \times 1 & 6 \times (-3) + 1 \times (1) + 4 \times 5 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 9 \\ 21 & 3 \end{bmatrix}$$

In general, multiplication of appropriately sized matrices (say, matrices *C* and *D*) is not commutative; that is, *CD* is not the same as *DC*. We'll delve into this concept shortly.

Unlike addition, subtraction, and scalar multiplication, matrix multiplication is not a simple element-wise calculation, and we cannot use the standard `*` operator. Instead, you must use R's matrix product operator, written with percent symbols as `%*%`. Before trying this operator, let's store the two example matrices and make sure the number of columns in the first matrix matches the number of rows in the second using the `dim` function:

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> dim(A)
[1] 2 3
R> B <- cbind(c(3,-1,1),c(-3,1,5))
R> dim(B)
[1] 3 2
```

This confirms the two matrices are compatible for multiplication, so you can proceed with using the matrix product operator:

```
R> A%*%B
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

We'll show that matrix multiplication is noncommutative using the same two matrices. Switching the order of multiplication gives you an entirely different result:

```
R> B%*%A
     [,1] [,2] [,3]
[1,]  -12   12   -6
[2,]    4   -4    2
[3,]   32   10   22
```

Inversion

Some square matrices can be *inverted*. Matrix inversion is the matrix equivalent of finding the reciprocal of a single value. The inverse of a matrix A is denoted A^{-1}. An invertible matrix satisfies the following equation:

$$AA^{-1} = I_m$$

Here's an example of a matrix and its inverse:

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -0.5 \\ -2 & 1.5 \end{bmatrix}$$

Not all matrices are invertible; those that are not invertible are referred to as *singular*. Inverting a matrix is often necessary when solving equations with matrices and has important practical ramifications. There are several approaches to matrix inversion; these calculations can become extremely computationally expensive as you increase the size of a matrix. We won't go into too much detail here, but if you're interested, see texts such as Golub & Van Loan (1989) or Banerjee & Roy (2014) for formal discussions.

For now, we'll invert a matrix with the `solve` function:

```
R> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
R> A
     [,1] [,2]
[1,]    3    1
[2,]    4    2
R> solve(A)
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

You can also verify the equation above that the product of these two matrices (using matrix multiplication rules) results in the 2×2 identity matrix:

```
R> A%*%solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Exercise 3.2

a. Calculate the following:

$$\frac{2}{7}\left(\begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 7 & 6 \end{bmatrix} - \begin{bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{bmatrix}\right)$$

b. Store these two matrices:

$$A = \begin{bmatrix} 1 \\ 2 \\ 7 \end{bmatrix} \qquad B = \begin{bmatrix} 3 \\ 4 \\ 8 \end{bmatrix}$$

Which of the following multiplications are possible? For those that are, compute the result.

i. $A \cdot B$

ii. $A^\top \cdot B$

iii. $B^\top \cdot (A \cdot A^\top)$

iv. $(A \cdot A^\top) \cdot B^\top$

v. $[(B \cdot B^\top) + (A \cdot A^\top) - 100I_3]^{-1}$

c. For

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

confirm that $A^{-1} \cdot A - I_4$ provides a 4×4 matrix of zeros.

3.2 Multidimensional Arrays

Just as a matrix is the result of increasing the dimension of a vector, we can increase the dimension of a matrix to get more complex data structures. We refer to data structures of more than two dimensions as *arrays*.

Considering a matrix as a collection of vectors of equal length, a three-dimensional array can be considered a collection of equally dimensioned matrices, providing a rectangular prism of elements. You still have a fixed number of rows and a fixed number of columns, as well as a new third dimension called a *layer*. Figure 3-3 illustrates a three-row, four-column, two-layer (3×4×2) array.

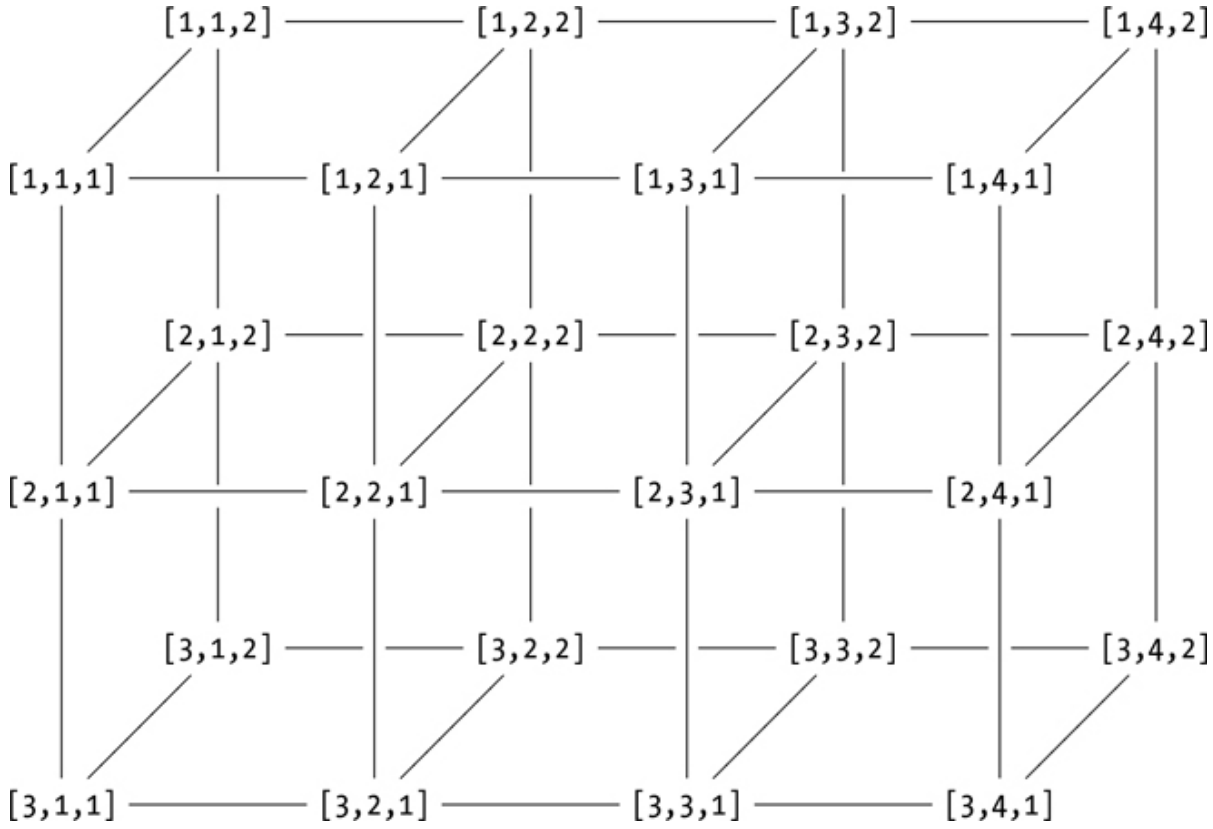


Figure 3-3: A conceptual diagram of a 3×4×2 array, wherein the index of each element in an array is given at the corresponding position

The indexes of an array are provided in the strict order of `[row,column, layer]`.

3.2.1 Array Creation

To create arrays in R, use the `array` function and specify the individual elements in the `data` argument as a vector. Then specify the array's size in the `dim` argument as another vector with a length corresponding to the number of dimensions. Note that `array` fills the entries of each layer with the elements in `data` in a strict column-wise fashion, starting with the first layer.

Consider the following example, which shows creation of a 3×4×2 array:

```
R> AR <- array(data=1:24,dim=c(3,4,2))
R> AR
```

```
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Each of the two layers, denoted by `, , 1` and `, , 2`, constitutes a 3×4 matrix. In this example, note the order of the dimensions supplied to `dim`: `c(rows,columns,layers)`. Just like a single matrix, the product of the dimension sizes of an array will yield the total number of elements.

As you increase the dimension further, you must extend the `dim` vector accordingly. For example, a four-dimensional array is the next step up and can be thought of as *blocks* of three-dimensional arrays. Suppose you had a four-dimensional array consisting of three copies of `AR`, the three-dimensional array just defined. You can store this new array in R as follows (once again, the array is filled column-wise):

```
R> BR <- array(data=rep(1:24,times=3),dim=c(3,4,2,3))
R> BR
, , 1, 1

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

, , 2, 1

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
```

```
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 1, 2

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

, , 2, 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 1, 3

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

, , 2, 3

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

With BR you now have three copies of AR. Each of these copies is split into its two layers so R can print the object to the screen. As before, the rows are indexed by the first digit, the columns by the second digit, and the layers by the third digit. The new fourth digit indexes the blocks.

3.2.2 Subsets, Extractions, and Replacements

High-dimensional objects can be difficult to conceptualize, but fortunately R indexes them consistently. This makes extracting elements from these structures straightforward now that you know how to subset matrices; you just have to keep using commas in the square brackets as separators of the dimensions you're accessing.

Suppose you want the second row of the second layer of the previously created array `AR`. You just enter these exact dimensional locations of `AR` in square brackets. Remember, of the three possible index positions for a three-dimensional array, the first corresponds to rows, the second to columns, and the third to layers:

```
R> AR[2,,2]
[1] 14 17 20 23
```

The desired elements are extracted as a vector of length 4. If you want specific elements from this vector, say the third and first, in that order, you can call the following:

```
R> AR[2,c(3,1),2]
[1] 20 14
```

This literal subsetting method makes dealing with even high-dimensional objects in R manageable.

When the elements you extract result in multiple vectors, R presents the output as columns in the returned matrix. For example, to extract the first rows of both layers of `AR`, you enter this:

```
R> AR[1,,]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

The returned object has the first rows of each of the two matrix layers. However, it has returned each of these vectors as a *column* of the single returned matrix. As this example shows, when extracting multiple

vectors from an array, R returns them as columns by default. This means R will not necessarily return extracted rows as rows.

Turning to the object `BR`, we next extract the single element of the second row and first column of the matrix in the first layer of the three-dimensional array located in the third block:

```
R> BR[2,1,1,3]
[1] 2
```

Again, you just need to look at the position of the index in the square brackets to know which values you're asking R to return from the array. To highlight this, I'll perform a few more extractions:

```
R> BR[1,,,1]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

This returns all the values in the first row of the first block. By leaving the column and layer indexes blank in this subset `[1,,,1]`, I ask R to return all values for all four columns and both layers in that block of `BR`.

The following code returns all the values in the second layer of the array `BR`, composed of three matrices:

```
R> BR[,,2,]
, , 1

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 2

     [,1] [,2] [,3] [,4]
```

```
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 3

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

This last example highlights a feature noted earlier, where R returned multiple vectors from `AR` as a matrix. Broadly speaking, if you have an extraction that results in multiple *d*-dimensional arrays, the result will be an array of the next-highest dimension, *d*+1. In the last example, you extracted multiple (two-dimensional) matrices, and R returned them as a three-dimensional array. I'll demonstrate this again:

```
R> BR[3:2,4,,]
, , 1

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 2

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 3

     [,1] [,2]
[1,]   12   24
[2,]   11   23
```

This extracts the elements at rows 3 and 2 (in that order), column 4, for all layers and for all array blocks. Consider this final example:

```
R> BR[2,,1,]
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    5    5    5
[3,]    8    8    8
[4,]   11   11   11
```

Here, R returns the entire second rows of the first layers of all the arrays stored in `BR`.

Deleting and overwriting elements in high-dimensional arrays follows the same rules as for stand-alone vectors and matrices. You specify the dimension positions the same way, using negative indexes for deletion or using the assignment operator for overwriting.

NOTE

It's possible to use the `array` *function to create one-dimensional arrays (vectors) and two-dimensional arrays (matrices) by setting the* `dim` *argument to be of length 1 or 2, respectively. However, some functions will treat vectors created with* `array` *differently from those created with* `c` *(see the help file* `?array` *for technical details). For this reason, and to make large sections of code more readable, it's more conventional in R programming to use the specific vector- and matrix-creation functions* `c` *and* `matrix`.

Exercise 3.3

a. Create and store a three-dimensional array with six layers of a 4×2 matrix, filled with a decreasing sequence of values between 4.8 and 0.1 of the appropriate length.

b. Extract and store as a new object the fourth- and first-row elements, in that order, of the second column only of all layers of (a).

c. Use a fourfold repetition of the second row of the matrix formed in (b) to fill a new array of dimensions 2×2×2×3.

d. Create a new array consisting of the results of deleting the sixth layer of (a).

e. Overwrite the second and fourth row elements of the second column of layers 1, 3, and 5 of (d) with –99.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`matrix`	Create a matrix	Section 3.1.1, p. 38
`rbind`	Create a matrix (bind rows)	Section 3.1.1, p. 39
`cbind`	Create a matrix (bind columns)	Section 3.1.1, p. 40
`dim`	Get matrix dimensions	Section 3.1.1, p. 40
`nrow`	Get number of rows	Section 3.1.1, p. 40
`ncol`	Get number of columns	Section 3.1.1, p. 40
`[ , ]`	Matrix/array subsetting	Section 3.1.2, p. 41
`diag`	Diagonal elements / identity matrix	Section 3.1.2, p. 42
`t`	Matrix transpose	Section 3.1.3, p. 46
`*`	Scalar matrix multiple	Section 3.1.3, p. 47

Function/operator	Brief description	First occurrence
`+, -`	Matrix addition/subtraction	Section 3.1.3, p. 48
`%*%`	Matrix multiplication	Section 3.1.3, p. 49
`solve`	Matrix inversion	Section 3.1.3, p. 50
`array`	Create an array	Section 3.2.1, p. 52

4
NON-NUMERIC VALUES



So far, you've been working almost exclusively with numeric values. But statistical programming also requires non-numeric values. In this chapter, we'll consider four such data types: logicals, characters, factors, and date-times. These play an important role in effective use of R, especially as we get into more complex programming in Part II.

4.1 Logical Values

Logical values (which we also call *logicals*) are based on a simple premise: A logical-valued object can be either `TRUE` or `FALSE`. We interpret these as yes/no, one/zero, satisfied/not satisfied, and so on. This concept appears across all programming languages. Often, logical values signal whether a condition has been satisfied or whether a parameter should be switched on.

You encountered logical values briefly when you used the `sort` function in Section 2.3.2 and the `matrix` function in Section 3.1. When using `sort`, setting `decreasing=TRUE` returns a vector ordered from largest to smallest, and `decreasing=FALSE` sorts the vector the other way around.

Similarly, when constructing a matrix, `byrow=TRUE` fills the matrix entries row-wise; otherwise, the matrix fills column-wise. Let's take a more detailed look at ways to use logicals.

4.1.1 TRUE or FALSE?

We write logical values in R as `TRUE` and `FALSE`, though we can abbreviate them as `T` or `F`. The abbreviated version has no effect on the execution of the code; for example, using `decreasing=T` is equivalent to `decreasing=TRUE` in the `sort` function. (Note, however, that you shouldn't name other objects `T` or `F` in your workspace if you plan to use this convenience—see Section 9.1.3.)

Assigning logical values to an object is the same as assigning numeric values:

```
R> foo <- TRUE
R> foo
[1] TRUE
R> bar <- F
R> bar
[1] FALSE
```

This gives you one object with the value `TRUE` and one with the value `FALSE`. Similarly, you can fill vectors with logical values:

```
R> baz <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> baz
[1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
R> length(x=baz)
[1] 12
```

You can create matrices (and other higher-dimensional arrays) with these values, too. Using `foo` and `baz`, you might construct something like this:

```
R> qux <- matrix(data=baz,nrow=3,ncol=4,byrow=foo)
R> qux
     [,1]  [,2]  [,3]  [,4]
```

```
[1,] TRUE FALSE FALSE FALSE
[2,] TRUE FALSE  TRUE  TRUE
[3,] TRUE FALSE  TRUE FALSE
```

As expected given the use of `byrow` and the stored value of `foo`, the resulting structure has been filled in a row-wise fashion.

4.1.2 Relational Operators

We commonly use logicals to check relationships between values. For example, to test whether some number *a* is greater than a predefined threshold *b*, use the standard relational operators in Table 4-1, which produce logical values as results.

Table 4-1: Relational Operators

Operator	Interpretation
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

Typically, we use these operators on numeric values (though you'll look at some other possibilities in Section 4.2.1). Here's an example:

```
R> 1==2
[1] FALSE
R> 1>2
[1] FALSE
R> (2-1)<=2
[1] TRUE
```

```
R> 1!=(2+3)
[1] TRUE
```

The results should be unsurprising: `1` being equal to `2` is `FALSE` and `1` being greater than `2` is also `FALSE`, while it's `TRUE` that the result of `2-1` is less than or equal to `2`, and that `1` *isn't* equal to `5` `(2+3)`. These operations are much more useful when used on numbers that are variable in some way, as you'll see shortly.

You're already familiar with R's element-wise behavior when working with vectors. The same rules apply when using relational operators. To illustrate this, first create two vectors and double-check that they're of equal length:

```
R> foo <- c(3,2,1,4,1,2,1,-1,0,3)
R> bar <- c(4,1,2,1,1,0,0,3,0,4)
R> length(x=foo)==length(x=bar)
[1] TRUE
```

Now consider the following four evaluations:

```
R> foo==bar
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
R> foo<bar
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
R> foo<=bar
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
R> foo<=(bar+10)
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The first line checks whether the entries in `foo` are equal to the corresponding entries in `bar`, which is true only for the fifth and ninth entries. As the returned vector contains a logical result for each pair of elements, it's the same length as the vectors we're comparing. The second line compares `foo` and `bar` in the same way, this time checking whether the entries in `foo` are less than the entries in `bar`. Contrast this result with the third comparison, which asks whether entries are less than *or equal to* one another. Finally, the fourth line checks whether `foo`'s

members are less than or equal to `bar`, when the elements of `bar` are increased by `10`. Naturally, all of the results are `TRUE`.

Vector recycling also applies to logicals. Let's use `foo` from earlier, along with a shorter vector, `baz`:

```
R> baz <- foo[c(10,3)]
R> baz
[1] 3 1
```

You create `baz` as a vector of length 2 consisting of the tenth and third elements of `foo`. Now consider the following:

```
R> foo>baz
[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

Here, the 2 elements of `baz` are recycled and checked against the 10 elements of `foo`. Elements `1` and `2` of `foo` are checked against `1` and `2` of `baz`, elements `3` and `4` of `foo` are checked against `1` and `2` of `baz`, and so on. You can also check all the values of a vector against a single value. Here's an example:

```
R> foo<3
[1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

This kind of operation will become familiar when handling and subsetting data frames in Chapter 5.

For now, let's rewrite the contents of `foo` and `bar` as 5×2 column-filled matrices:

```
R> foo.mat <- matrix(foo,nrow=5,ncol=2)
R> foo.mat
     [,1] [,2]
[1,]    3    2
[2,]    2    1
[3,]    1   -1
[4,]    4    0
[5,]    1    3
R> bar.mat <- matrix(bar,nrow=5,ncol=2)
```

```
R> bar.mat
     [,1] [,2]
[1,]    4    0
[2,]    1    0
[3,]    2    3
[4,]    1    0
[5,]    1    4
```

The same element-wise behavior applies here; if you compare the matrices, you get a matrix of the same size filled with logicals:

```
R> foo.mat<=bar.mat
      [,1]  [,2]
[1,]  TRUE FALSE
[2,] FALSE FALSE
[3,]  TRUE  TRUE
[4,] FALSE  TRUE
[5,]  TRUE  TRUE
R> foo.mat<3
      [,1]  [,2]
[1,] FALSE  TRUE
[2,]  TRUE  TRUE
[3,]  TRUE  TRUE
[4,] FALSE  TRUE
[5,]  TRUE FALSE
```

This also applies to arrays of more than two dimensions.

You can use the `any` and `all` functions to quickly inspect a collection of logical values. When examining a vector, `any` returns `TRUE` if any of the logicals in the vector are `TRUE` and returns `FALSE` otherwise. The function `all` returns `TRUE` only if *all* of the logicals are `TRUE`, and returns `FALSE` otherwise. As a quick example, let's work with two of the logical vectors formed by the comparisons of `foo` and `bar` from the beginning of this section:

```
R> qux <- foo==bar
R> qux
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
```

```
R> any(qux)
[1] TRUE
R> all(qux)
[1] FALSE
```

Here, `qux` contains two `TRUE`s, and the rest are `FALSE`, so the result of `any` is `TRUE`, but the result of `all` is `FALSE`. Following the same rules, you get this:

```
R> quux <- foo<=(bar+10)
R> quux
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
R> any(quux)
[1] TRUE
R> all(quux)
[1] TRUE
```

Again, the `any` and `all` functions do the same thing for matrices and arrays of logical values.

Exercise 4.1

a. Store the following vector of 15 values as an object in your workspace: `c(6,9,7,3,6,7,9,6,3,6,6,7,1,9,1)`. Create corresponding logical-valued vectors that identify the following elements with `TRUE`s:
   i. Those equal to 6
   ii. Those greater than or equal to 6
   iii. Those less than 6 + 2
   iv. Those not equal to 6

b. Create a new numeric vector from the one in (a) by deleting its first three elements. With this new vector, fill a

2×2×3 array. Create corresponding logical-valued arrays that identify the following elements with TRUEs:

i. Those less than or equal to 6 divided by 2, plus 4

ii. Those less than or equal to 6 divided by 2, plus 4, *after* increasing every element in the array by 2

c. Confirm with TRUEs the specific locations of elements equal to 0 in the 10×10 identity matrix I_{10} (see Section 3.1.3).

d. Check whether *any* of the values of the logical arrays created in (b) are TRUE. If they are, check whether they are *all* TRUE.

e. By extracting the diagonal elements of the logical matrix created in (c), use `any` to confirm there are no TRUE entries.

4.1.3 Logical Operators

Logicals are especially useful when examining whether multiple conditions are satisfied. Often you'll want to perform certain operations only if a number of different conditions have been met.

The previous section looked at using relational operators to compare the literal values (that is, numeric or otherwise) of stored R objects. Now you'll look at *logical operators*, which you use to compare two logical-valued objects. These operators are based on the statements AND and OR. Table 4-2 summarizes the R syntax and the behavior of logical operators. The AND and OR operators each have a "single" and "element-wise" version; you'll see how they differ in a moment.

Table 4-2: Logical Operators Comparing Two Logical Values

Operator	Interpretation	Results

Operator	Interpretation	Results

`&`	AND (element-wise)	`TRUE & TRUE` is `TRUE`
		`TRUE & FALSE` is `FALSE`
		`FALSE & TRUE` is `FALSE`
		`FALSE & FALSE` is `FALSE`

`&&`	AND (single comparison)	Same as `&`

`|`	OR (element-wise)	`TRUE|TRUE` is `TRUE`
		`TRUE|FALSE` is `TRUE`
		`FALSE|TRUE` is `TRUE`
		`FALSE|FALSE` is `FALSE`

`||`	OR (single comparison)	Same as `|`
`!`	NOT	`!TRUE` is `FALSE` `!FALSE` is `TRUE`

The result of using any logical operator is itself a logical value. An AND comparison is true only if *both* logicals are `TRUE`. An OR comparison is true if at least one of the logicals is `TRUE`. The NOT operator (`!`) returns the opposite of the logical value you use it on. You can combine these operators to examine multiple conditions at once:

```
R> FALSE||((T&&TRUE)||FALSE)
[1] TRUE
R> !TRUE&&TRUE
[1] FALSE
R> (T&&(TRUE||F))&&FALSE
[1] FALSE
R> (6<4)||(3!=1)
[1] TRUE
```

As with numeric arithmetic, there's an order of importance for logical operations in R: An AND statement has higher precedence than an OR statement. It's helpful to place each comparative pair in

parentheses to preserve the correct order of evaluation and make the code more readable. You can see this in the first line of the previous code, where the innermost comparison is the first to be carried out: `T&&TRUE` results in `TRUE`; this result becomes one of the logical values for the next bracketed comparison where `TRUE||FALSE` results in `TRUE`. The final comparison is then `FALSE||TRUE`, and the end result, `TRUE`, prints to the console. The second line reads as "NOT `TRUE` AND `TRUE`," which returns `FALSE`. In the third line, the innermost pair once again evaluates first: `TRUE||F` is `TRUE`, `T&&TRUE` is `TRUE`, and `TRUE&&FALSE` is `FALSE`. The final example evaluates two distinct conditions in parentheses via relational operators, which you then compare using a logical operator. Since `6<4` is `FALSE` and `3!=1` is `TRUE`, you get a logical comparison of `FALSE||TRUE` and a final result of `TRUE`.

In Table 4-2, there are short (`&`, `|`) and long (`&&`, `||`) versions of the AND and OR operators. Use the short versions in element-wise comparisons, where you have two logical vectors and you want multiple logicals as a result. The long versions, which you've been using so far, are for comparing two individual values and return a single logical value. This is important when programming conditional checks in R (for example, with an `if` statement, which you'll look at in Chapter 10). While it's possible to compare a single pair of logicals using the short version, it's considered better practice to use the longer versions when a single `TRUE`/`FALSE` result is intended.

Let's look at some examples of element-wise comparisons. Suppose you have two vectors of equal length, `foo` and `bar`:

```
R> foo <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> foo
[1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

and

```
R> bar <- c(F,T,F,T,F,F,F,F,T,T,T,T)
R> bar
[1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

The short versions of the logical operators match each pair of elements by position and return the result of the comparison:

```
R> foo&bar
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
R> foo|bar
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Let's see what happens if you erroneously use the long version of the logical operators to attempt the same multi-element comparison:

```
R> foo&&bar
Error in foo && bar : 'length = 12' in coercion to 'logical(1)'
R> foo||bar
Error in foo || bar : 'length = 12' in coercion to 'logical(1)'
```

As of version 4.3.0+ of R, the interpreter will throw an error when you try to do this; in this case, we're chastised when attempting to do the 12 comparisons comprising the entries of `foo` and `bar`.

Exercise 4.2

a. Store the vector `c(7,1,7,10,5,9,10,3,10,8)` as `foo`. By constructing a corresponding logical vector with relational and logical operators, identify the elements greater than 5 OR equal to 2.

b. Store the vector `c(8,8,4,4,5,1,5,6,6,8)` as `bar`. In the same fashion as (a), identify the elements less than or equal to 6 AND not equal to 4.

c. Identify the elements that satisfy (a) in `foo` AND satisfy (b) in `bar`.

d. Store a third vector, `baz`, that is equal to the element-wise sum of `foo` and `bar`. Once more using the appropriate

operators, construct logical vectors that determine the following:

i. The elements of `baz` greater than or equal to 14 but not equal to 15

ii. The elements of the vector obtained via an element-wise division of `baz` by `foo` that are greater than 4 OR less than or equal to 2

4.1.4 Logicals Are Numbers!

Although this chapter is called "Non-Numeric Values," logicals (and, as you'll see in subsequent sections, factors and date-time objects) actually have an underlying numeric representation. Because of their binary nature, we often represent logical values in datasets with `TRUE` as 1 and `FALSE` as 0. In fact, when performing elementary numeric operations on logical values, R will actually treat `TRUE` like `1`, and `FALSE` like `0`:

```
R> TRUE+TRUE
[1] 2
R> FALSE-TRUE
[1] -1
R> T+T+F+T+F+F+T
[1] 4
```

These operations turn out the same as if you'd used the digits 1 and 0. In some situations when using logicals, you can substitute the numeric values (this generally isn't recommended, as it hampers the clarity of your code):

```
R> 1&&1
[1] TRUE
R> 1||0
[1] TRUE
R> 0&&1
[1] FALSE
```

Being able to interpret logicals as 0s and 1s means you can use a variety of functions to summarize a logical vector, which you'll explore further in Part III.

4.1.5 Logical Subsetting and Extraction

You can also use logicals to extract and subset elements in vectors and other objects, in the same way as with index vectors. Rather than entering explicit indexes in the square brackets, you can supply logical *flag* vectors, where an element is extracted if the corresponding entry in the flag vector is `TRUE`. As such, logical flag vectors and the vector that's being accessed should be the same length (though recycling does occur for shorter flag vectors, as you'll see shortly).

At the beginning of Section 2.3.3 you defined a vector of length 10 as follows:

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
```

If you wanted to extract the two negative elements, you could either enter `myvec[c(2, 10)]` or do the following using logical flags:

```
R> myvec[c(F,T,F,F,F,F,F,F,F,T)]
[1] -2.3 -8.0
```

This particular example may seem far too cumbersome for practical use. The core behavior is incredibly powerful, however, when you implicitly create the desired flag vector based on the desired condition (or several conditions). Taking the current example, remember that you can use logicals to find negative elements in `myvec` by applying the condition `<0`:

```
R> myvec<0
[1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

You can use this flag vector to subset `myvec` to get the same result as earlier. As such, the code

```
R> myvec[myvec<0]
[1] -2.3 -8.0
```

offers up the requested elements without having to explicitly know their specific index positions.

As mentioned, R recycles the flag vector if it's too short. To extract every second element from `myvec`, starting with the first, enter the following:

```
R> myvec[c(T,F)]
[1]     5     4     4     8 40221
```

You can perform more complicated extractions using relational and logical operators, such as:

```
R> myvec[(myvec>0)&(myvec<1000)]
[1]  5  4  4  4  6  8 10
```

This returns the positive elements that are less than 1,000. You can also overwrite specific elements using a logical flag vector, as with index vectors:

```
R> myvec[myvec<0] <- -200
R> myvec
[1]     5  -200     4     4     4     6     8    10 40221  -200
```

This replaces all existing negative entries with –200. Note, however, that you can't directly use negative logical flag vectors to delete specific elements; this can be done only with numeric index vectors.

As you can see, logicals are very useful for element extraction. You don't need to know beforehand which index positions to return, since the conditional check can find them for you. This is particularly valuable when you're dealing with large datasets and you want to inspect records or recode entries that match certain criteria.

In some cases, you might want to convert a logical flag vector into a numeric index vector. This is helpful when you *do* need the explicit indexes of elements that were flagged `TRUE`. The R function `which` takes in a logical vector as the argument `x` and returns the indexes corresponding to the positions of any and all `TRUE` entries:

```
R> which(x=c(T,F,F,T,T))
[1] 1 4 5
```

Use `which` to identify the index positions of `myvec` that meet a certain condition; for example, those containing negative numbers:

```
R> which(x=myvec<0)
[1]  2 10
```

You can do this for the other `myvec` selections you experimented with. Note that a line of code such as `myvec[which(x=myvec<0)]` is redundant because you can make that extraction using the condition by itself, that is, via `myvec [myvec<0]`, without using `which`. On the other hand, using `which` lets you *omit* elements based on logical flag vectorsby identifying the numeric indexes you want to delete and rendering them negative. To see what `myvec` looks like without the negative entries, execute the following:

```
R> myvec[-which(x=myvec<0)]
[1]     5     4     4     4     6     8    10 40221
```

The same can be done with matrices and other arrays. In Section 3.1.2, you stored a 3×3 matrix as follows:

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To extract the second and third column elements of the first row of `A` using numeric indexes, execute `A[1,2:3]`. To do the same thing with logical flags, enter the following:

```
R> A[c(T,F,F),c(F,T,T)]
[1] 91.0 -4.2
```

Again, you usually won't explicitly specify the logical vectors. Suppose you want to replace all elements in `A` that are less than 1 with –7. Performing this using numeric indexes is rather fiddly. It's much easier to use the logical flag matrix created with the following:

```
R> A<1
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE  TRUE
[2,] FALSE  TRUE FALSE
[3,] FALSE FALSE FALSE
```

You can supply this logical matrix to the square bracket operators. The replacement is done as follows:

```
R> A[A<1] <- -7
R> A
     [,1]  [,2] [,3]
[1,] -7.0  91.0 -7.0
[2,]  4.5  -7.0  8.2
[3,] 55.3 105.5 27.9
```

This is the first time you've subsetted a matrix without having to list row or column positions inside the square brackets, using commas to separate out dimensions (see Section 3.1.2). This is because the flag matrix has the same number of rows and columns as the target matrix, thereby providing all the relevant structural information.

If you use `which` to identify numeric indexes based on a logical flag structure, you have to be a little more careful when dealing with two-dimensional objects or higher. Suppose you want the index positions of the elements that are greater than 25. The appropriate logical matrix is as follows:

```
R> A>25
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE  TRUE
```

Now ask R the following:

```
R> which(x=A>25)
[1] 3 4 6 9
```

This returns the four indexes of the elements that satisfied the relational check, but they're provided as scalar values. How do these correspond to the row/column positioning of the matrix?

The answer lies in R's default behavior of the `which` function, which essentially treats the multidimensional object as a single vector (laid out column after column) and then returns the vector of corresponding indexes. If you explicitly arrange the matrix `A` as a vector by stacking the columns first through third, using `c(A[,1],A[,2],A[,3])`, then the indexes returned make more sense:

```
R> which(x=c(A[,1],A[,2],A[,3])>25)
[1] 3 4 6 9
```

With the columns laid out end to end, the elements that return `TRUE` are the third, fourth, sixth, and ninth elements in the list. This can be difficult to interpret, though, especially when dealing with higher-dimensional arrays. In these situations, make `which` return dimension-specific indexes using the optional argument `arr.ind` ("array indexes"). By default, this argument is set to `FALSE`, resulting in the vector converted indexes. Setting `arr.ind` to `TRUE`, on the other hand, treats the object as a matrix or array rather than a vector, providing the row and column positions of the elements you requested.

```
R> which(x=A>25,arr.ind=TRUE)
     row col
[1,]   3   1
[2,]   1   2
[3,]   3   2
[4,]   3   3
```

The returned object is now a matrix, where each row represents an element that satisfied the logical comparison and each column provides

the position of the element. Comparing the output here with `A`, these positions correspond to elements where `A>25`.

Both versions of the output (with `arr.ind=TRUE` or `arr.ind=FALSE`) are useful, depending on the application.

Exercise 4.3

a. Store this vector of 10 values: `foo <- c(7,5,6,1,2,10,8,3,8,2)`. Then, do the following:
    i. Extract the elements greater than or equal to 5, storing the result as `bar`.
    ii. Display the vector containing those elements from `foo` that remain after omitting all elements that are greater than or equal to 5.

b. Use `bar` from (a)(i) to construct a 2 × 3 matrix called `baz`, filled in a row-wise fashion. Then, do the following:
    i. Replace any elements that are equal to 8 with the *squared* value of the element in row 1, column 2 of `baz` itself.
    ii. Confirm that *all* values in `baz` are now less than or equal to 25 AND greater than 4.

c. Create a 3×2×3 array called `qux` using the following vector of 18 values: `c(10,5,1,4,7,4,3,3,1,3,4,3,1,7,8,3,7,3)`. Then, do the following:
    i. Identify the dimension-specific index positions of elements that are either 3 OR 4.
    ii. Replace all elements in `qux` that are less than 3 OR greater than or equal to 7 with the value 100.

d. Return to `foo` from (a). Use the vector `c(F,T)` to extract every second value from `foo`. In some situations, you can substitute `0` and `1` for `TRUE` and `FALSE` (see Section 4.1.4). Can you perform the same extraction from `foo` using the vector `c(0,1)`? Why/why not? What does R return in this case?

4.2 Characters

Character strings are a common data type we use to represent text. In R, we often use strings to specify folder locations or software options (see the brief mention in Section 1.2); to supply an argument to a function; and to annotate stored objects, provide textual output, or help clarify plots and graphics. While we can also use strings to define different groups making up a categorical variable, as you'll see in Section 4.3, *factors* are specifically designed (and hence better suited) for that.

NOTE

There are three different string formats in the R environment. The default string format is an extended regular expression*; the other variants are* Perl *and* literal regular expressions. *The intricacies of these variants are beyond the scope of this book, so any mention of character strings from here on refers to an extended regular expression. For technical details about other string formats, enter* `?regex` *at the prompt.*

4.2.1 String Creation

We indicate character strings with double quotation marks (`"`). To create a string, enter text between a pair of quotes:

```
R> foo <- "This is a character string!"
R> foo
[1] "This is a character string!"
```

```
R> length(x=foo)
[1] 1
```

R treats the string as a single entity. In other words, `foo` is a vector of length 1 because R counts only the total number of distinct strings rather than individual words or characters. To count the number of individual characters, use the `nchar` function. Here's an example using `foo`:

```
R> nchar(x=foo)
[1] 27
```

Almost any combination of characters, including numbers, can be a valid character string, as shown here:

```
R> bar <- "23.3"
R> bar
[1] "23.3"
```

In this form, the string has no numeric meaning, and R won't treat it like the number 23.3. Attempting to multiply it by 2, for example, results in an error:

```
R> bar*2
Error in bar * 2 : non-numeric argument to binary operator
```

This error occurs because `*` is expecting to operate on two numeric values (not one number and one string, which makes no sense).

You can compare strings in several ways, the most common being a check for equality:

```
R> "alpha"=="alpha"
[1] TRUE
R> "alpha"!="beta"
[1] TRUE
R> c("alpha","beta","gamma")=="beta"
[1] FALSE  TRUE FALSE
```

Other relational operators work as you might expect. For example, R considers letters that come later in the alphabet to be greater than

earlier letters, meaning it can determine whether one string of letters is greater than another with respect to alphabetical order:

```
R> "alpha"<="beta"
[1] TRUE
R> "gamma">"Alpha"
[1] TRUE
```

Furthermore, R treats uppercase letters as "greater" than lowercase letters:

```
R> "Alpha">"alpha"
[1] TRUE
R> "beta">="bEtA"
[1] FALSE
```

You can use most symbols in a string. The following string is valid, for example:

```
R> baz <- "&4 _ 3 **%.? $ymbolic non$en$e ,; "
R> baz
[1] "&4 _ 3 **%.? $ymbolic non$en$e ,; "
```

One important exception is the backslash `\`, or *escape*. Using a backslash within the quotation marks of a string initiates some simple control over the printing or display of the string itself. You'll see how this works in Section 4.2.3. First, let's look at two useful functions for combining strings.

4.2.2 Concatenation

There are two main functions we use to *concatenate* (or glue together) one or more strings: `cat` and `paste`. The difference between the two lies in how their contents are returned. The first function, `cat`, sends its output directly to the console screen and doesn't formally *return* anything. The `paste` function concatenates its contents and then returns the final character string as a usable R object. This is useful when R needs to pass the result of a string concatenation to another function or

use it in some secondary way, as opposed to just displaying it. Consider the following vector of character strings:

```
R> qux <- c("awesome","R","is")
R> length(x=qux)
[1] 3
R> qux
[1] "awesome" "R"       "is"
```

As with numbers and logicals, you can also store any number of strings in a matrix or array structure.

When calling `cat` or `paste`, you pass arguments to the function in the order you want them combined. The following lines show identical usage yet different types of output from the two functions:

```
R> cat(qux[2],qux[3],"totally",qux[1],"!")
R is totally awesome !
R> paste(qux[2],qux[3],"totally",qux[1],"!")
[1] "R is totally awesome !"
```

Here, you've used the three elements of `qux` as well as two additional strings, `"totally"` and `"!"`, to produce the final concatenated string. In the output, `cat` simply concatenates and prints the text to the screen. This means you can't directly assign the result to a new variable and treat it as a character string. For `paste`, however, the `[1]` to the left of the output and the presence of the double quotes indicate the returned item is a vector containing a character string, which you can assign to an object and use in other functions.

These two functions have an optional argument, `sep`, that's used as a separator between strings as they're concatenated. If you pass `sep` a character string, it places this string between all other strings you've provided to `paste` or `cat`. For example:

```
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="---")
[1] "R---is---totally---awesome---!"
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="")
[1] "Ristotallyawesome!"
```

The same behavior would occur for `cat`. If you don't want any separation, set `sep=""`, an empty string, as in the second example. You can use the empty string separator to achieve correct sentence spacing; note the gap between `awesome` and the exclamation mark in the code where you first used `paste` and `cat`. If the `sep` argument isn't included, R inserts a space between strings by default. For example, using manual insertion of spaces where necessary, write the following:

```
R> cat("Do you think ",qux[2]," ",qux[3]," ",qux[1],"?",sep="")
Do you think R is awesome?
```

Concatenation can be useful when you want to neatly summarize the results from a certain function or set of calculations. You can pass many R objects directly to `paste` or `cat`; the software will attempt to automatically *coerce* these items into character strings. This means R converts the input into a string to include the values in the final concatenated string. This works particularly well with numeric objects, as the following examples demonstrate:

```
R> aa <- 3
R> bb <- 4.4
R> cc <- 10
R> cat("The value stored as 'aa' is ",aa,".",sep="")
The value stored as 'aa' is 3.
R> paste("The value stored as 'bb' is ",bb,".",sep="")
[1] "The value stored as 'bb' is 4.4."
R> cat("The result of aa+bb is ",aa,"+",bb,"=",aa+bb,".",sep="")
The result of aa+bb is 3+4.4=7.4.
R> paste("Is ",aa+bb," less than ",cc,"? That's totally ",aa+bb<cc,".",sep="")
[1] "Is 7.4 less than 10? That's totally TRUE."
```

Here, the values of the non-string objects are placed where you want them in the final string output. The results of calculations can also appear as fields, as shown with the arithmetic `aa+bb` and the relational comparison `aa+bb<cc`. You'll see more details about coercion from one kind of value to another in Section 6.2.4.

The function `paste0` is identical to `paste` in behavior but has `sep=""` by default. If you want a no-space separator between your strings, use this variant of the function. Repeating the most recent example, see that

```
R> paste0("Is ",aa+bb," less than ",cc,"? That's totally ",aa+bb<cc,".")
[1] "Is 7.4 less than 10? That's totally TRUE."
```

didn't require you to explicitly set `sep=""`.

It's worth highlighting that the same kind of vectorized behavior in terms of element-wise operations (refer to Section 2.3.4) also applies to `paste` (and `paste0`). Consider the following vector of eight character strings:

```
R> poetry <- c("Hey","diddle","diddle","the","cat","and","the","fiddle")
R> length(poetry)
[1] 8
```

We can easily augment each of these components in the same way using vector-oriented behavior. For example:

```
R> paste(poetry,"!")
[1] "Hey !"    "diddle !" "diddle !" "the !"    "cat !"    "and !"    "the !"
[8] "fiddle !"
```

Here, an exclamation mark (with a default space separator given our use of `paste` instead of `paste0`) simply pastes onto the end of each string in `poetry`. Provided evenly divisible lengths (as Figure 2-2 illustrates), recycling of the shorter vector occurs in the typical way:

```
R> paste0(poetry,c("?","!"))
[1] "Hey?"    "diddle!" "diddle?" "the!"    "cat?"    "and!"    "the?"
[8] "fiddle!"
```

The two components to be pasted onto the elements of `poetry`, `"?"` and `"!"`, replicate four times to match the longer vector's length prior to pasting (with no space this time, due to use of `paste0`).

Lastly, it's also possible to concatenate an existing vector of multiple character strings together into a single character string with `paste`,

though you need to make use of an optional argument. Remaining with the eight components in `poetry`, suppose you want to 'glue' them all together. Simply calling `paste` on that alone doesn't do anything, leaving the existing vector untouched:

```
R> paste(poetry)
[1] "Hey"    "diddle" "diddle" "the"    "cat"    "and"    "the"    "fiddle"
```

To achieve the desired outcome, use the `collapse` argument, which specifies the character string used to separate the multiple elements of the target vector when they're appended to one another. The following three examples demonstrate this behavior:

```
R> paste(poetry,collapse=" ")
[1] "Hey diddle diddle the cat and the fiddle"
R> paste(poetry,collapse="")
[1] "Heydiddlediddlethecatandthefiddle"
R> paste(poetry,collapse="&-%")
[1] "Hey&-%diddle&-%diddle&-%the&-%cat&-%and&-%the&-%fiddle"
```

4.2.3 Escape Sequences

In Section 4.2.1, I noted that a stand-alone backslash doesn't act like a normal character within a string. Using `\` invokes an *escape sequence*, which lets you enter characters that control the format and spacing of the string, rather than being interpreted as normal text. Table 4-3 describes some of the most common escape sequences; for a full list, enter `?Quotes` at the prompt.

Table 4-3: Common Escape Sequences for Use in Character Strings

Escape sequence	Result
`\n`	Starts a newline
`\t`	Horizontal tab

Escape sequence	Result
\\	Used as a single backslash
\"	Includes a double quote

Escape sequences add flexibility to the display of character strings, which can be useful for summaries of results and plot annotations. Enter the sequence precisely where you want it to take effect; for example:

```
R> cat("here is a string\nsplit\tto new\n\n\tlines")
here is a string
split to new

       lines
```

Note that the exact amount of space imparted by a single use of the tab escape, `\t`, depends on the environment and context in which you're using R. For example, using the base R GUI exhibits a different default tab spacing than the RStudio IDE in macOS, and so the specific appearance of such output can differ between systems.

The signal for an escape is `\` and the signal to begin and end a string is `"`, meaning if you want to include either character in a string, you must use an escape to have R interpret them as a normal character:

```
R> cat("I really want a backslash: \\\nand a double quote: \"")
I really want a backslash: \
and a double quote: "
```

These escape sequences are the reason you can't use a stand-alone backslash in filepath strings in R. As noted in Section 1.3.4 (where you used `getwd` to print the current working directory and `setwd` to change it), folder separation is best designated with a single *forward* slash (`/`). Filepath specification is directly relevant when reading in and writing out files, and you'll explore these activities in greater detail in Chapter 8.

4.2.4 Substrings and Matching

Pattern matching lets you inspect a given string to identify smaller "query" strings within it. For the simplest form of pattern matching, use the `grep` function to inspect a vector of character strings and return the indexes (if any) of the elements that contain the query string. For example:

```
R> poetry <- c("Hey","diddle","diddle","the","cat","and","the","fiddle")
R> result1 <- grep(pattern="iddle",x=poetry)
R> result1
[1] 2 3 8
```

Here, you apply `grep` to the vector of character strings in the `poetry` object from Section 4.2.2. Via the `pattern` argument, you're asking for matches to the query string `"iddle"`. The result, stored as `result1`, is a numeric vector containing 2, 3, and 8—correctly identifying the positions of those strings in `poetry` that contain the pattern of interest. When such a search yields no results, you see the following:

```
R> result2 <- grep(pattern="dog",x=poetry)
R> result2
integer(0)
```

The somewhat curious notation `integer(0)` simply denotes an "empty" vector, evidenced by use of `length` on the object:

```
R> length(x=result2)
[1] 0
```

This is fitting, seeing as there are no strings in `poetry` containing the substring `"dog"`.

A related function is `substr`, which takes a target string `x` and returns the substring that sits between two character positions, indicated with numbers passed to the `start` and `stop` arguments. Let's try it on the object `foo` from Section 4.2.1:

```
R> foo <- "This is a character string!"
R> substr(x=foo,start=21,stop=27)
[1] "string!"
```

Here, you extract the characters between (and including) positions 21 and 27, to get `"string!"`. The function also works on vectors of character strings; the following example returns the first three characters of each entry in `poetry`:

```
R> substr(x=poetry,start=1,stop=3)
[1] "Hey" "did" "did" "the" "cat" "and" "the" "fid"
```

You can also use the function `substr` with the assignment operator to directly substitute in a new set of characters. In this case, the replacement string should contain the same number of characters as the selected area:

```
R> substr(x=foo,start=1,stop=4) <- "Here"
R> foo
[1] "Here is a character string!"
```

If the replacement string is longer than the number of characters indicated by `start` and `stop`, then replacement still takes place, beginning at `start` and ending at `stop`. It cuts off any characters that overrun the number of characters you're replacing. If the string is shorter than the number of characters you're replacing, then replacement ends when the string is fully inserted, leaving the original characters up to `stop` untouched.

Other tools for substitution include the functions `sub` and `gsub`. The `sub` function searches a given string `x` for a smaller string `pattern` contained within. It then replaces the first instance with a new string, given as the `replacement` argument. The `gsub` function does the same thing, but it replaces *every* instance of `pattern`. Here's an example:

```
R> bar <- "How much wood could a woodchuck chuck"
R> sub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl chuck"
```

```
R> gsub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl hurl"
```

Note the `replacement` value doesn't need to have the same number of characters as the `pattern` being replaced.

The help files `?grep` and `?substr` (the former also documents `sub` and `gsub`) have more details, as well as noting a handful of other pattern-matching functions and techniques. You can disable case sensitivity, which is the default when searching for substrings with functions like `grep` and `gsub`.

Exercise 4.4

a. By using `cat` on a string with the appropriate escape sequences, re-create the following output:

```
"The quick brown fox
        jumped over
               the lazy dogs"
```

b. Suppose you've stored the values `num1 <- 4` and `num2 <- 0.75`. Write a line of R code that returns the following string:

```
[1] "The result of multiplying 4 by 0.75 is 3"
```

Make sure your code produces a string with the correct multiplication result for *any* two numbers stored as `num1` and `num2`.

c. On my local machine, I've specified the directory for my work on this book in R as `"/Users/tdavies/Documents/Book"`. Imagine it is your machine, and write a line of code that replaces *`tdavies`* in this string with your own name.

d. Using the appropriate R function, determine which entries of the character string vector given by

```
cutlery <- c("the","dish","ran","away","with","the","spoon")
```

contain the substring `"th"`.

e. In Section 4.2.4, you stored the following string:

```
bar <- "How much wood could a woodchuck chuck"
```

   i. Store a new string by gluing the following words onto `bar`: `"if a woodchuck could chuck wood"`.

   ii. In the result of (i), replace all instances of `wood` with `metal`.

f. Store the string `"Two 6-packs for $15.99"`, and then do the following:

   i. Use `substr` and a check for equality to confirm that the substring beginning with character 5 and ending with character 10 is `"6-pack"`.

   ii. Make it a better deal by using `substr` to change the price to $12.99.

4.3 Factors

In this section, you'll look at some simple functions related to creating, handling, and inspecting *factors*. Factors are R's most natural way of representing data values which take on only one of a finite number of distinct categories, rather than belonging to a continuum. Categorical data can play an important role in data science, and you'll look at factors again from a statistical perspective in Chapter 13.

4.3.1 Identifying Categories

To see how factors work, we'll start with a simple dataset. Suppose you find eight people and record their first name, sex, and birth month in Table 4-4.

Table 4-4: An Example Dataset of Eight Individuals

Person	Sex	Birth month
Liz	Female	April
Jolene	Female	January
Susan	Female	December
Boris	Male	September
Rochelle	Female	November
Tim	Male	July
Simon	Male	July
Amy	Female	June

There's really only one sensible way to represent the name of each person in R, which is as a vector of character strings:

```
R> firstname <- c("Liz","Jolene","Susan","Boris","Rochelle","Tim","Simon",
                  "Amy")
```

You have more flexibility when recording sex, however. Coding females as 0 and males as 1, a numeric option would be as follows:

```
R> sex.num <- c(0,0,0,1,0,1,1,0)
```

Character strings are also possible, and many prefer this because you don't need to remember the numeric code for each group, as shown here:

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

There is, however, a fundamental difference between an individual's name and their sex when stored as data. Where a person's name is a unique identifier that can take any one of an infinite number of possibilities, there are typically only two options for recording a person's sex. It's best to represent this kind of variable, wherein all possible values fall into a finite number of categories, as a factor in R.

You typically create factors from a numeric or a character vector. You can't fill matrices or multidimensional arrays using factor values; factors can take only the form of vectors. To create a factor, use the function `factor`, as in this example working with `sex.num` and `sex.char`:

```
R> sex.num.fac <- factor(x=sex.num)
R> sex.num.fac
[1] 0 0 0 1 0 1 1 0
Levels: 0 1
R> sex.char.fac <- factor(x=sex.char)
R> sex.char.fac
[1] female female female male   female male   male   female
Levels: female male
```

Here, you obtain factor versions of the two vectors storing the sex values.

At first glance, these objects don't look too different from the character and numeric vectors from which they were created. Indeed, factor objects work in much the same way as vectors, but with extra information attached (R's internal representation of factor objects is a bit different as well). Simple functions like `length` and `which` work the same way on factor objects as with vectors, for example.

The most important extra piece of information (or *attribute*; see Section 6.2.1) that a factor object contains is its *levels* (or *categories*), which define the possible values of the factor. R prints these levels at the bottom of each factor vector. To extract the levels as a vector of character strings, use the `levels` function:

```
R> levels(x=sex.num.fac)
[1] "0" "1"
R> levels(x=sex.char.fac)
[1] "female" "male"
```

You can also relabel a factor using `levels`. Here's an example:

```
R> levels(x=sex.num.fac) <- c("1","2")
R> sex.num.fac
[1] 1 1 1 2 1 2 2 1
Levels: 1 2
```

This relabels the females `1` and the males `2`.

Factors are subsetted in the same way as any other vector:

```
R> sex.char.fac[2:5]
[1] female female male   female
Levels: female male
R> sex.char.fac[c(1:3,5,8)]
[1] female female female female female
Levels: female male
```

After subsetting a factor object, the object continues to store *all* defined levels, even if the subsetted object no longer represents some levels.

If you want to subset from a factor using a logical flag vector, keep in mind that R recognizes the levels of a factor as character strings, even if the original data vector was numeric, so you need to use a string when requesting or testing for a particular level. For example, to identify all the men using the newly relabeled `sex.num.fac`, use this:

```
R> sex.num.fac=="2"
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

Since the elements in `firstname` and `sex` have corresponding positions, you can then use a logical vector like the previous one to obtain the names of all the men (this time using the `"male"`/`"female"` factor):

```
R> firstname[sex.char.fac=="male"]
[1] "Boris" "Tim"   "Simon"
```

Of course, this simple subsetting operation could have been achieved in much the same way with the raw numeric vector `sex.num` or the raw character vector `sex.char`. In the next section, you'll explore distinctive advantages to R representing categorical data as a factor.

4.3.2 Defining and Ordering Levels

The sex factor from the previous section represents the simplest kind of factor variable; there are only two possible levels with no ordering, in that one level is not intuitively considered "higher than" or "following" the other. Here, you'll look at factors with more than two levels, as well as defining and customizing levels in greater detail.

To motivate things, we'll focus on the month of birth (MOB) data for the eight individuals in Table 4-4, first storing the corresponding entries as a simple character vector:

```
R> mob <- c("Apr","Jan","Dec","Sep","Nov","Jul","Jul","Jun")
```

Like sex, it makes sense to treat MOB as a categorical variable. If we use `factor` in exactly the same way as before, we get something like this:

```
R> mob.fac.basic <- factor(x=mob)
R> mob.fac.basic
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Apr Dec Jan Jul Jun Nov Sep
```

There are two potential problems with this basic version of the MOB factor. First, not all possible categories are represented because the original `mob` character vector contains only seven unique months. Tabulating the data with the function `table`, we see two observations categorized as "July," with the remaining months represented only once:

```
R> table(mob.fac.basic)
mob.fac.basic
Apr Dec Jan Jul Jun Nov Sep
  1   1   1   2   1   1   1
```

Second, the levels of the factor aren't arranged in the "natural" calendar order of months; the default behavior in R is to sort alphabetically.

While these two issues may seem minor in isolation, they can have important consequences when wrangling data and using categorical variables in statistical analyses. You can deal with both problems to varying extents by customizing additional arguments to the `factor` function. The most important of these is the `levels` argument, which takes a character vector defining all the possible categories you want, in the order that you want. Redefine the MOB factor as follows:

```
R> ms <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov",
           "Dec")
R> mob.fac <- factor(x=mob,levels=ms)
R> mob.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Here, R defines the `mob.fac` vector in terms of the same eight individual entries (the `mob` object) as earlier. The new factor has 12 levels due to using `levels=ms` inside the call to `factor`, even though you haven't actually made observations for the levels `"Feb"`, `"Mar"`, `"May"`, `"Aug"`, or `"Oct"`. (Note that if your R console window is too narrow to print all the levels to the screen, you may see … , indicating there's more output. Widen your window and reprint the object to see the hidden levels.) In addition, R arranges the new category levels in the order you supplied them to the `levels` argument. Once more using the `table` function, you obtain the following:

```
R> table(mob.fac)
mob.fac
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  1   0   0   1   0   1   2   0   1   0   1   1
```

NOTE

You can impose a stricter, more formal ordering on a factor by setting the optional argument `ordered=TRUE` *when using the* `factor` *function (the default has this at* `FALSE`*, which is how I've used* `factor` *in the preceding code). Doing this, however, can have important consequences in the implementation of various statistical methods, such as regression and other types of modeling: It changes R's handling and interpretation of categorical variables. As such, it's used only in very specific circumstances and is generally safe to ignore. See the help file* `?factor` *for further details on these more complicated features of factors.*

The most direct way to re-order the levels of a given factor after the fact is to apply the `factor` function to the existing factor with the preferred order appearing in the `levels` argument. To illustrate this, re-create the `sex.char.fac` factor from Section 4.3.1 here:

```
R> sex.char <- c("female","female","female","male","female","male","male",
                 "female")
R> sex.char.fac <- factor(x=sex.char)
R> sex.char.fac
[1] female female female male   female male   male   female
Levels: female male
```

To arrange the categories the other way around, with males first, do the following:

```
R> sex.char.fac2 <- factor(x=sex.char.fac,levels=c("male","female"))
R> sex.char.fac2
[1] female female female male   female male   male   female
Levels: male female
```

The individual entries are entirely unchanged; only the order of the level definitions are modified.

The `relevel` function is useful if you just want to ensure a particular level comes first (we sometimes refer to this as a *reference category* or *reference level*). As the sex factor has only two levels to begin with, you could execute the following line to equivalently create `sex.char.fac2`:

```
R> relevel(x=sex.char.fac,ref="male")
[1] female female female male   female male   male   female
Levels: male female
```

Note that you supply the desired reference category as a character string to the `ref` argument, and you must take care to ensure its value matches exactly (including case sensitivity) one of the existing levels of the target factor.

You can see that `relevel` changes only the category that comes first by operating on a factor with more than two levels. Using `mob.fac` from earlier, consider the following:

```
R> relevel(x=mob.fac,ref="Dec")
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Dec Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
```

Here, you ask that December becomes the first category; all other levels remain in the same order as originally defined when `mob.fac` was created. (Note also that you haven't actually changed `mob.fac` or `sex.char.fac` as they're stored; to do that, you need to overwrite/reassign the objects with their releveled selves.) The reference level or "first" category of a factor affects the treatment of categorical variables in some situations (you'll see examples of this in Chapter 21's discussion of multiple regression models).

4.3.3 Dropping Categories

Remember that changes to the levels of a factor, both in terms of the available categories and in terms of their ordering, are far from just cosmetic. There's a big difference, for example, between a dataset with zero observations in some of the categories and the same dataset defined with fewer categories to begin with. That said, you may occasionally need to simplify a factor by removing categories that aren't represented by the observation set at hand. This typically occurs after you've subsetted a larger dataset and are interested only in populated categories.

Consider again the `mob.fac` object defined in the previous section. You created this specifically to include levels that weren't represented by the entries in the `mob` vector; that is, they were associated with "zero counts" when you tabulated the factor. The object is reprinted here:

```
R> mob.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Suppose you're given this factor but are interested only in analyzing the categories represented by the data. The `droplevels` function creates a new factor with the exact same data but strips away any categories with zero counts:

```
R> mob.fac2 <- droplevels(x=mob.fac)
R> mob.fac2
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan Apr Jun Jul Sep Nov Dec
```

Notice that the remaining categories of the new `mob.fac2` are precisely the same as those of the `mob.fac.basic` vector created at the very beginning of Section 4.3.2, but their order is different (retaining the order imposed by `mob.fac`).

4.3.4 Working with the Numeric Representation of Factors

Just like logical-valued vectors, all factors have an equivalent numeric (integer) form that essentially designates the category number to which a given observation belongs. Category numbers can be useful when using R for things like data visualization.

To see the integer form of any factor, use the `as.numeric` function directly on the object of interest. Return once more to the `sex.char.fac` factor as the preceding sections show:

```
R> sex.char.fac
[1] female female female male   female male   male   female
Levels: female male
```

Its numeric form is as follows:

```
R> as.numeric(sex.char.fac)
[1] 1 1 1 2 1 2 2 1
```

You're returned a vector of integers with the same number of elements as the original factor, identifying the category each entry belongs to. The output tells you that the first three individuals are in category 1, the fourth in category 2, the fifth in category 1, and so on. The numbering of the categories always follows the defined order of the factor levels; in this example, the numeric form agrees with the displayed order of the factor levels in `sex.char.fac`, in that females are identified as category 1, and males as 2.

If the defined order of the levels changes, so too do the numeric identifiers of each entry. Taking the `sex.char.fac2` object from Section 4.3.2 (in which the males are now defined as the first category), observe the consistency of the numeric form of the factor:

```
R> sex.char.fac2
[1] female female female male   female male   male   female
Levels: male female
R> as.numeric(sex.char.fac2)
[1] 2 2 2 1 2 1 1 2
```

Again, while the individual entries of `sex.num.fac2` are exactly the same, their numeric representation is now different purely by virtue of the re-ordered factor levels.

The `as.numeric` function is one of a collection of so-called "as-dot" functions we use to convert between different object and data types (known formally as *coercion*). We'll study its behavior in greater detail in Section 6.2.4.

4.3.5 Collapsing Factors

In some situations, you may be dealing with factors that have many levels and, for one reason or another, you wish to collapse the factor to have fewer levels. For example, you may have several categories with

very low counts and, for the purposes of your analysis, seek to simplify matters by collecting together these rarer levels into some overall "Other" category.

To illustrate this, we'll use the `mob.fac` object created in Section 4.3.2 on page 84, detailing the month of birth of eight individuals (Table 4-4). As a reminder, here is the form of the object and the tabulated counts:

```
R> mob.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
R> table(mob.fac)
mob.fac
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  1   0   0   1   0   1   2   0   1   0   1   1
```

Suppose we wish to collapse this object so that the unique categories are not the individual months, but instead are coded as quarters. That is, we wish to create a new object whereby all individuals with a birth month of January, February, or March are assigned the same category, Q1; those with a birth month of either April, May, or June are categorized as Q2, and so on. One way to do this would be to convert the factor to its numeric representation (as per Section 4.3.4), recognize the fact that numbers 1 through 3 would correspond to Q1, 4 through 6 would be Q2, and so on, and use this to construct a new factor from scratch with the `factor` function. This is a lot of work! Instead, we can use the existing factor and simply re-define the levels themselves. The definition and membership of the new, collapsed categories must be supplied in the form of a *named list*, which is a construct for collecting together multiple objects. You've not yet met lists; these will be scrutinized in the following chapter (Section 5.1).

For now, consider the following syntax, which first creates an identical copy of `mob.fac` (named `mob.fac.q`) and subsequently proceeds to manipulate the categories by changing the representation of the levels in a particular way:

```
R> mob.fac.q <- mob.fac
R> levels(x=mob.fac.q) <- list(Q1=c("Jan","Feb","Mar"),Q2=c("Apr","May","Jun"),
                               Q3=c("Jul","Aug","Sep"),Q4=c("Oct","Nov","Dec"))
```

Note that the key to this action involves overwriting (or assigning) the `levels` attribute of the target factor object with the desired definition. In a call to `list`, you collect the labels of the pre-existing factor that you want to collapse together as a vector of character strings, and assign each vector to a tag that will become the new label (the latter needn't be supplied in double quotes). In this example, you can see I intend to re-code my factor `mob.fac.q` to be made up of four unique levels, labeled `Q1`, `Q2`, `Q3`, and `Q4`. The new category `Q1` will be given to any of the original observations that correspond to January, February, or March, while `Q2` will represent any entries corresponding to April, May, or June, and so on. You must ensure the character string labels you're collecting together match the levels of the original factor exactly.

Inspecting and tabulating the newly re-coded object `mob.fac.q`, we see the following:

```
R> mob.fac.q
[1] Q2 Q1 Q4 Q3 Q4 Q3 Q3 Q2
Levels: Q1 Q2 Q3 Q4
R> table(mob.fac.q)
mob.fac.q
Q1 Q2 Q3 Q4
 1  2  3  2
```

You can compare the individual entries of `mob.fac.q` with those of the original `mob.fac` to ensure things correspond as expected. For instance, the first entry of `mob.fac.q` is `Q2`;this makes perfect sense because the first entry of `mob.fac` is `Apr`. Tabulation further confirms things have worked correctly; there are exactly one, two, three, and two individual months that fall in Q1, Q2, Q3, and Q4 of the calendar year, respectively.

4.3.6 Combining Factors

As you've seen, it's usually simple to combine multiple vectors of the same kind (whether numeric, logical, or character) using the `c` function. In R versions ≥ 4.1.0, you can do the same thing. As a quick toy example, consider the following two factor objects, each of which has four elements:

```
R> foo <- factor(x=c("A","B","A","C"))
R> foo
[1] A B A C
Levels: A B C
R> bar <- factor(x=c("D","E","E","E"))
R> bar
[1] D E E E
Levels: D E
```

Note that `foo` and `bar` are completely different objects; the former has three levels labeled `"A"`, `"B"` and `"C"`, and the latter has only two possible categories, `"D"` and `"E"`. Create a new factor by combining everything via `c`:

```
R> baz <- c(foo,bar)
R> baz
[1] A B A C D E E E
Levels: A B C D E
```

Here, the new factor has eight observations spanning five possible categories.

More realistically, you may simply want to add newly obtained observations to an existing object, where the new observations take on categories the existing object has already defined. Let's see this behavior on the data in Table 4-4; specifically, the MOB factor vector `mob.fac`, from Section 4.3.2. Suppose you now observe three more individuals with MOB values `"Oct"`, `"Feb"`, and `"Feb"`. You can quickly create something like

```
R> new.mob.values <- factor(x=c("Oct","Feb","Feb"))
R> new.mob.values
```

```
[1] Oct Feb Feb
Levels: Feb Oct
```

and append them to the existing observations of `mob.fac`:

```
R> mob.fac.ext <- c(mob.fac,new.mob.values)
R> mob.fac.ext
[1] Apr Jan Dec Sep Nov Jul Jul Jun Oct Feb Feb
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

The three new entries now appear at the end of the eight original values. Furthermore, the fact that the levels of `new.mob.values` (only "Feb" and "Oct") aren't exactly equivalent to the entire set of 12 possible levels of `mob.fac` doesn't matter; R recognizes that `Feb` and `Oct` already exist as category labels in `mob.fac`.

This behavior means you must take care with things that might be missed, such as case sensitivity. For example, if you accidentally misspell a given entry, R interprets this as a new category, as illustrated here:

```
R> bad.mob.values <- factor(x=c("Oct","Feb","FeB"))
R> c(mob.fac,bad.mob.values)
[1] Apr Jan Dec Sep Nov Jul Jul Jun Oct Feb FeB
Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec FeB
```

This is something you'd be keen to avoid in practice. While such a mistake is easily visible in a small example, in situations with hundreds or thousands of observations, attention to detail is key.

4.3.7 Creating Factors from Continuous Values

You often create factors from data originally measured on a continuum, for example the weight of a set of adults or the amount of a drug given to a patient. Sometimes you need to group (or *bin*) these types of observations into categories, like Small/Medium/Large or Low/High. In R, you can mold these kinds of data into discrete factor categories using the `cut` function. Consider the following numeric vector of length 10:

```
R> Y <- c(0.53,5.4,1.5,3.33,0.45,0.01,2,4.2,1.99,1.01)
```

Suppose you want to bin the data as follows: *Small* refers to observations in the interval [0, 2), *Medium* refers to [2, 4), and *Large* refers to [4, 6]. A square bracket refers to *inclusion* of its nearest value, and a parenthesis indicates *exclusion*, so an observation y falls in the Small interval if $0 \leq y < 2$, in Medium if $2 \leq y < 4$, or in Large if $4 \leq y \leq 6$. Use `cut` and supply your desired break intervals to the `breaks` argument:

```
R> br <- c(0,2,4,6)
R> cut(x=Y,breaks=br)
[1] (0,2] (4,6] (0,2] (2,4] (0,2] (0,2] (0,2] (4,6] (0,2] (0,2]
Levels: (0,2] (2,4] (4,6]
```

This gives you a factor, with each observation now assigned an interval. However, notice that the boundary intervals are back-to-front; you want the boundary levels on the left like [0,2), rather than the right as they appear by default, (0,2]. Fix this by setting the logical argument `right` to `FALSE`:

```
R> cut(x=Y,breaks=br,right=FALSE)
[1] [0,2) [4,6) [0,2) [2,4) [0,2) [0,2) [2,4) [4,6) [0,2) [0,2)
Levels: [0,2) [2,4) [4,6)
```

Now you've swapped which boundaries are inclusive and exclusive. This is important because it changes which categories the values fall into. Notice that the seventh observation has changed categories. But there's still a problem: The final interval currently *excludes* 6, and you want this maximum value to be *included* in the highest level. You can fix this with another logical argument, `include.lowest`, which you also use to include the highest value if `right` is `FALSE`, as the `?cut` help file indicates:

```
R> cut(x=Y,breaks=br,right=FALSE,include.lowest=TRUE)
[1] [0,2) [4,6] [0,2) [2,4) [0,2) [0,2) [2,4) [4,6] [0,2) [0,2)
Levels: [0,2) [2,4) [4,6]
```

The intervals are now defined how you want. Finally, you can add better labels to the categories by passing a character string vector to the

`labels` argument, rather than using the interval levels that R applies by default:

```
R> lab <- c("Small","Medium","Large")
R> cut(x=Y,breaks=br,right=FALSE,include.lowest=TRUE,labels=lab)
[1] Small  Large  Small  Medium Small  Small  Medium Large  Small  Small
Levels: Small Medium Large
```

The order of your supplied labels must match the order of the levels in the factor object.

Exercise 4.5

At the time of writing, the New Zealand Parliament consists of the political parties Labour, National, Greens, ACT, and Māori. There are several other parties in the political landscape that are not represented in parliament, labeled as Other. Suppose you asked 20 New Zealanders which party they identified most with and obtained the following data:

- There were 12 males and 8 females; the individuals numbered 1, 5–7, 12, and 14–16 were females.
- The individuals numbered 1, 4, 12, 15, 16, and 19 identified with National; no one identified with Māori; the individuals numbered 6 and 9 identified with Greens; 11 and 20 identified with ACT; individual 10 identified with Other; and the rest identified with Labour.

a. Use your knowledge of vectors (for example, subsetting and overwriting) to create two character vectors: `sex` with possible entries `"M"` (male) and `"F"` (female) and `party` with possible entries `"National"`, `"Labour"`, `"Greens"`, `"Maori"`, `"ACT"`, and `"Other"`. Make sure to place the entries in the correct positions as outlined earlier.

b. Create two different factors using `sex` and `party`; name them `sex.fac` and `party.fac`. You should explicitly specify the `levels` argument for the latter (because the sample doesn't represent all possible categories).

c. Identify the numeric representation of `party.fac`.

d. Use subsetting to do the following:

   i. Return the factor vector of chosen parties for only the male participants.

   ii. Return the factor vector of genders for those who chose National.

e. Another six people joined the survey, with the results `c("National","Maori","Maori","Labour","ACT","Labour")` for the preferred party and `c("M","M","F","F","F","M")` as their gender. Create new factors (named `party.fac2` and `sex.fac2`) that combine these results with the original factors from (b).

f. As a brief aside, reconsider the original `party.fac` factor created in (b). Tabulate the factor: Are any categories associated with "zero counts"? If so, create another version named `party.fac3` that strips away any unpopulated categories from `party.fac`. Confirm via tabulation that `party.fac3` doesn't contain zero-count levels.

Suppose you also asked all individuals to state how confident they were that National will win more seats in Parliament than Labour in the next election and to attach a subjective percentage to that confidence. You obtained the following results, corresponding to each of the 26 individuals now present in the sample: 75, 55, 20, 100, 10, 84, 50, 99, 33, 52, 35, 53, 25, 30, 100, 80, 56, 0, 64, 31, 10, 29, 40, 5, 18, 33.

g. From these values, create a factor with levels of confidence as follows: Low for percentages [0,30]; Moderate for

percentages (30,70]; and High for percentages (70,100].

h. From the factor created in (g), use `party.fac2` created in (e) to extract the levels corresponding to those individuals who originally said they identified with Labour. Do this also for National. Is there anything to note about the levels of these sub-groups?

4.4 Dates and Times

Sometimes, your analysis requires dealing with dates and times. This could be for something as simple as annotating a plot, or something more involved where calculations depend on specific dates. For the former it usually suffices to use plain character strings, but in the case of the latter, you need more specialized objects.

Dates (and date-times) are complex, and we'll look only at some simple features of base R here. As your needs become more sophisticated it's often beneficial to consider additional R packages designed specifically for dealing with date-times, such as `lubridate` (Grolemund & Wickham 2011).

4.4.1 Creating and Formatting Dates

Let's start simple. You can write a date down in many different ways and store it as a character string. For example, the following objects `d1`, `d2`, and `d3` all represent the same date:

```
R> d1 <- "2025-02-19"
R> d2 <- "2/19/25"
R> d3 <- "19Feb2025"
```

As character strings, you can't perform any operations with these objects beyond using them as names or labels. To instruct R to interpret them as proper calendar dates, use the `as.Date` function and, in so doing, ensure R knows precisely how you've formatted the string so it correctly

identifies the year, month, and day. The default is a "year-month-day" format; specifically:

YYYY-MM-DD or *YYYY/MM/DD*

In the strings, only `d1` is in one of these specific forms. We tell R to treat it as a proper date with the following call:

```
R> date1 <- as.Date(x=d1)
R> date1
[1] "2025-02-19"
```

At a glance, it seems nothing has changed; `date1` looks just like the character string `d1` from which it was created. But internally, the *class* of the object has changed, meaning you can ultimately do things with it that make sense in terms of calendar dates (you'll look at more technical details related to object classes in Section 6.2.2).

If the strings representing the dates are in a non-default format, you must use the optional `format` argument in the `as.Date` function. To this argument, supply another character string made up of *conversion specification* tags that tell `as.Date` how to interpret the date you've given it. There are a host of different conversion specifications; Table 4-5 summarizes a few of the most common and what they represent. Find a full list by inspecting the documentation file obtained by typing `?strptime` at the R prompt.

Table 4-5: Common Date-Specific Conversion Specifications for Date-Time Objects

Date marker	Role
`%Y`	Year as numeric *YYYY*
`%y`	Year as numeric *YY*
`%m`	Month as numeric *MM*

Date marker	Role
`%B`	Month as full text (as per `month.name`)
`%b`	Month as abbreviation (as per `month.abb`)
`%d`	Day as numeric *DD*

To see some of these conversion specifications in action, convert `d2` and `d3` to proper R dates as follows:

```
R> date2 <- as.Date(x=d2,format="%m/%d/%y")
R> date2
[1] "2025-02-19"
R> date3 <- as.Date(x=d3,format="%d%b%Y")
R> date3
[1] "2025-02-19"
```

In creating `date2`, you've told `as.Date` to expect `x` as a numeric month (`%m`), a forward slash `/`, the day (`%d`), another forward slash (`/`), and a two-digit year (`%y`). For `date3`, you tell `as.Date` to expect the date as day (`%d`), abbreviated month (`%b`), and a four-digit year (`%Y`). Looking at how R prints the resulting objects, `date2` and `date3` are now entirely equivalent to `date1`: they all represent February 19, 2025.

Note that R stores, as built-in constants, full and abbreviated versions of the calendar months (in the locale of the current system) as the objects `month.name` and `month.abb`, respectively. These are immediately accessible at the R prompt, and you can use them as a reference for the `%B` and `%b` conversion specifications:

```
R> month.name
[1] "January"   "February"  "March"     "April"     "May"       "June"
[7] "July"      "August"    "September" "October"   "November"  "December"
R> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

You can store multiple dates in vectors. Suppose the following strings are the birthdates of five individuals:

```
R> birthstrings <- c("Jul-4-1985","Mar-3-1954","Jul-8-1955","Oct-20-1988",
                     "Jul-26-2019")
```

You can convert them to dates all at once with the line

```
R> DOB <- as.Date(x=birthstrings,format="%b-%d-%Y")
R> DOB
[1] "1985-07-04" "1954-03-03" "1955-07-08" "1988-10-20" "2019-07-26"
```

and the usual rules regarding vector extraction/subsetting (cf. Section 2.3.3) apply:

```
R> DOB[3]
[1] "1955-07-08"
R> which(DOB>as.Date("1985-08-01"))
[1] 4 5
```

The first line provides the date at the third position in `DOB`, and the second line uses a relational operator (Section 4.1.2) in conjunction with the `which` function (Section 4.1.5) to establish that only the fourth and fifth entries of `DOB` are dates that come after (are greater than) August 1, 1985.

You can interrogate any date object for the components that comprise it using the same conversion specifications in Table 4-5, returning the results as character strings. To do this, use `format` as a function and stipulate the specific contents of the date you want to an argument of the same name. For example, using the `DOB` date vector, the line

```
R> format(x=DOB,format="%d")
[1] "04" "03" "08" "20" "26"
```

picks out just the days of the five specific dates in `DOB`. You can find strings of the four-digit years like so:

```
R> format(x=DOB,format="%Y")
[1] "1985" "1954" "1955" "1988" "2019"
```

You're also given freedom to request more complicated combinations. For example, the following line returns both the abbreviated month and two-digit year of the requested date values:

```
R> format(x=DOB,format="%b %y")
[1] "Jul 85" "Mar 54" "Jul 55" "Oct 88" "Jul 19"
```

Other useful functions for interrogating dates include `weekdays` and `quarters`, which also return character strings of the corresponding weekday or yearly quarter of their corresponding argument:

```
R> weekdays(x=DOB)
[1] "Thursday"  "Wednesday" "Friday"    "Thursday"  "Friday"
R> quarters(x=DOB)
[1] "Q3" "Q1" "Q3" "Q4" "Q3"
```

4.4.2 Performing Simple Operations with Dates

Now that you've learned how to create formal objects for dates, let's explore how they differ from basic character strings, using the `d1` and `date1` objects created in Section 4.4.1. Recall from Section 4.2.1 that attempting, say, an arithmetic operation on a character string rightly results in an error:

```
R> d1
[1] "2025-02-19"
R> d1 + 1
Error in d1 + 1 : non-numeric argument to binary operator
```

But look what happens if you instead try to "add 1" to the formal date object of `d1`, namely `date1`:

```
R> date1
[1] "2025-02-19"
R> date1 + 1
[1] "2025-02-20"
```

Here, the operation returns a new date in which the day has been incremented by one.

The preceding example hints to the underlying numeric representation of a date object. Consider the result of the `as.numeric` function directly on `date1`:

```
R> as.numeric(date1)
[1] 20138
```

The resulting number expresses the given date in terms of the number of days since January 1, 1970 (known as the "Unix epoch"). This is clear if you look at the numeric form of this particular date:

```
R> as.numeric(as.Date(x="1970-01-01"))
[1] 0
```

Dates before the Unix epoch have a corresponding negative number; for example, the following line shows Christmas Day in 1969 as being seven days prior to this point:

```
R> as.numeric(as.Date(x="1969-12-25"))
[1] -7
```

The utility of the date format in this sense now starts to become appArent. If your desired tasks depend upon measuring time from some fixed point, then you needn't explicitly concern yourself with the differing days of varying months or leap years; R handles all of that based on the dates of interest. For example, although `date1 + 1` is trivial, suppose you need to calculate the date exactly 139 days from February 19, 2025. You can find this as July 8 of the same year:

```
R> date1 + 139
[1] "2025-07-08"
```

In much the same fashion, it's also easy to see how many days separate two dates:

```
R> term <- as.Date(x="1947-07-30")
R> ramb <- as.Date(x="1946-07-06")
R> term - ramb
Time difference of 389 days
```

Vectorized operations also hold true. Return to the five hypothetical individuals with birthdates stored in the `DOB` object, created in Section 4.4.1. Suppose you need to find out precisely how old each of them was, in days, on February 19, 2025:

```
R> date1 - DOB
Time differences in days
[1] 14475 25921 25429 13271  2035
```

You can also use `as.Date` to "reverse" a numeric (day) representation of a date to the corresponding calendar date, provided you supply an appropriate reference date (against which R will compare the number) to the `origin` argument (this can be a character string in a default date form or an existing date object). For example, the line

```
R> as.Date(x=481,origin="1970-01-01")
[1] "1971-04-27"
```

tells you that the 481st day after the Unix epoch is April 27, 1971; similarly, the line

```
R> as.Date(x=481,origin=date1)
[1] "2026-06-15"
```

informs you that June 15, 2026 is the date that occurs 481 days after February 19, 2025 (`date1`). Note that at the time of writing there is no default for `origin` when using `as.Date` in this way, so you must specify it each time.

4.4.3 Incorporating Time

The principles from the previous section carry through to full date-time definitions, where you can additionally specify hours, minutes, seconds, and even fractions of seconds. Unsurprisingly, this comes with additional complexity. Definitions of and calculations involving such objects must respect time zones, which in turn can raise even more issues, such as handling daylight savings. For these reasons we'll look at a couple of simple examples here.

First, consider the following command, which, when you execute it at the R prompt, gives the present date and time as reported by your local machine (your result will be different to that displayed here!):

```
R> Sys.time()
[1] "2022-03-14 14:15:08 NZDT"
```

I wrote this sentence shortly after a quarter past two in the afternoon on New Zealand daylight saving time, March 14, 2022. The output hints at the standard form for full date-time specifications; following the date itself, the time is given as two-digit 24-hour, minute, and second components separated by colons, and there is clearly an allowance for time zone.

Given a character string provided in this way, the `strptime` function can create the corresponding date-time object. For example, you can create 34 minutes and 12 seconds past 7 PM on October 31, 2022 as follows:

```
R> dtime1 <- strptime(x="2022-10-31 19:34:12",format="%Y-%m-%d %H:%M:%S")
R> dtime1
[1] "2022-10-31 19:34:12 NZDT"
```

Here, note two things: First, you use `strptime` in the same way as `as.Date`, whereby the first argument is a string made up of the desired date-time components, and the second uses recognized conversion specifications to define the format (see Table 4-6).

Table 4-6: Common Time-Specific Conversion Specifications for Date-Time Objects

Time marker	Role
`%H`	Hours (in 2-digit, 24-hour form)
`%M`	Minutes (as 2-digit numeric)
`%S`	Seconds (as 2-digit numeric)

Second, the created object has been assigned a time zone; by default, R queries your system locale. In this case, it's correctly identified I'm in New Zealand, and the specified date falls during daylight saving time (NZDT).

You can dictate specific time zones using the optional `tz` argument in `strptime`. You can instruct R to store the same date-time as Greenwich Mean Time / UTC by supplying the string `"GMT"`:

```
R> dtime2 <- strptime(x="2022-10-31 19:34:12",format="%Y-%m-%d %H:%M:%S",
                      tz="GMT")
R> dtime2
[1] "2022-10-31 19:34:12 GMT"
```

By typing `OlsonNames()` at your R prompt, you're provided the collection of time zone values able to be supplied to `tz` in a command like `strptime`. Query what R identifies your present time zone to be by typing `Sys.timezone()` at the prompt.

As best it can (there are various subtleties that can impart discrepancies between systems), R automatically identifies daylight savings statuses based on the provided date. For example, using the `OlsonNames()` entry for US Pacific time, note that the same date-time yields a `PDT` suffix, denoting Pacific daylight saving:

```
R> dtime3 <- strptime(x="2022-10-31 19:34:12",format="%Y-%m-%d %H:%M:%S",
                      tz="US/Pacific")
R> dtime3
[1] "2022-10-31 19:34:12 PDT"
```

However, if you set a date-time outside the usual daylight saving part of the year using the same time zone, like Christmas Day, R identifies the resulting object as Pacific Standard Time:

```
R> dtime4 <- strptime(x="2022-12-25 08:45:00",format="%Y-%m-%d %H:%M:%S",
                      tz="US/Pacific")
R> dtime4
[1] "2022-12-25 08:45:00 PST"
```

As always, these seemingly minor technicalities aren't just cosmetic; arguably the most important consequences arise when using the date-time objects in calculations. Taking the difference between `dtime1` and `dtime2` (which are defined to be exactly the same date-time, just in different time zones), for example, tells you that in New Zealand daylight saving time is 13 hours *ahead* of GMT:

```
R> dtime1 - dtime2
Time difference of -13 hours
```

Despite the order of differencing, the result is given as a negative here, implying that the New Zealand Daylight Time version, `dtime1`, needs 13 hours *added* to it in order for there to be zero difference between the two zones.

NOTE

Remember, daylight saving times are introduced, removed, or changed often at the whim of governments. As such, it's quite possible that the exact results of the preceding examples may well change when you run them in the future!

Finally, as earlier, note there is an underlying numeric representation of a full date-time object. In Section 4.4.2 this was provided in days since the Unix epoch (January 1, 1970) for date-only values. For full date-times, the default use of `as.numeric` function provides the converted value in terms of the number of *seconds* since January 1, 1970, at 00:00:00 GMT:

```
R> as.numeric(dtime1)
[1] 1667198052
```

Exercise 4.6

Aside from the usual Christmas and New Year periods, New Zealand presently has seven statutory public holidays. In the years 2024 and 2025, their observed dates were as follows:

Holiday	2024	2025
Waitangi Day	Feb. 6	Feb. 6
Good Friday	Mar. 29	Apr. 18
Easter Monday	Apr. 1	Apr. 21
ANZAC Day	Apr. 25	Apr. 25
King's Birthday	Jun. 3	Jun. 2
Matariki	Jun. 28	Jun. 20
Labour Day	Oct. 28	Oct. 27

a. Use `as.Date` to create two date vectors: one for the 2024 holidays and one for the days in 2025.

b. Use your two date vectors from (a) to complete the following tasks.

    i. Find the weekdays corresponding to each of the holidays in both 2024 and 2025. By inspecting the output (or, more formally, using `which`), establish whether they're exactly the same for each holiday. Which holidays, if any, are on different days of the week between the two years?

    ii. Repeat the previous task, but for the *quarters* of the year in which each holiday falls.

    iii. Format the 2024 date vector to obtain just the months in abbreviated letter form, and the 2025 date vector to obtain the days in two-digit form.

iv. Exactly how many days separate Waitangi Day and King's Birthday in 2024?

v. Exactly how many days separate Matariki in 2024 with the same holiday in 2025?

vi. How many "sleeps" must an impatient child wait from Labour Day until Christmas Day (December 25) in 2025?

In Semester 1 of a given academic year, I am usually involved in lecturing STAT372, a course at the University of Otago on stochastic modeling, and the Semester 2 course STAT260, which deals with data visualization and R programming. In 2022, the last lecture of STAT372 commenced at 11 AM, Friday, May 27, and the first lecture of STAT260 commenced at 1 PM, Monday, July 11.

c. Create two separate date-time objects representing the times (accurate to the second) and dates of the last lecture of STAT372 and the first lecture of STAT260, being sure to set the correct time zone to New Zealand time (simply identified as `"NZ"` in the output of `OlsonNames()`).

d. Use your date-time objects from (c) to complete the following tasks:

   i. Determine the time, in seconds, that elapsed from the Unix epoch to the first lecture of STAT260.

   ii. Each lecture at the University of Otago is 50 minutes in duration. Noting that arithmetic operations on your date-time objects will be performed in *seconds*, use an appropriately scaled addition operation on your STAT372 object to create a new date-time object that represents the time and date of the *end* of the last lecture of STAT372. Inspect the resulting object to ensure it shows the correct ending time.

iii. Using your object from (ii), determine the duration of relative calm I may enjoy between the end of lecturing STAT372 and the beginning of STAT260.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`TRUE`, `FALSE`	Reserved logical values	Section 4.1.1, p. 60
`T`, `F`	Unreserved versions of logicals	Section 4.1.1, p. 60
`==`, `!=`, `>`, `<`, `>=`, `<=`	Relational operators	Section 4.1.2, p. 61
`any`	Check whether any entries are `TRUE`	Section 4.1.2, p. 63
`all`	Check whether all entries are `TRUE`	Section 4.1.2, p. 63
`&&`, `&`, `||`, `|`, `!`	Logical operators	Section 4.1.3, p. 65
`which`	Determine indexes of `TRUE`s	Section 4.1.5, p. 69
`" "`	Designate a character string	Section 4.2.1, p. 73
`nchar`	Get number of characters in a string	Section 4.2.1, p. 73
`cat`	Concatenate strings (no return)	Section 4.2.2, p. 74
`paste`	Paste strings (return a string)	Section 4.2.2, p. 74

Function/operator	Brief description	First occurrence
`paste0`	As per `paste`, with `sep=""` by default	Section 4.2.2, p. 76
`\`	Invoke escape sequence inside string	Section 4.2.3, p. 77
`grep`	String matching	Section 4.2.4, p. 78
`substr`	Subset a string	Section 4.2.4, p. 79
`sub`, `gsub`	String matching and replacement	Section 4.2.4, p. 79
`factor`	Create a factor vector	Section 4.3.1, p. 82
`levels`	Get levels of a factor	Section 4.3.1, p. 82
`table`	Tabulate/count category membership	Section 4.3.2, p. 84
`relevel`	Set first (reference) level of factor	Section 4.3.2, p. 85
`droplevels`	Strip away unused levels of factor	Section 4.3.3, p. 86
`cut`	Create factor from continuum	Section 4.3.7, p. 90
`as.Date`	Create date object	Section 4.4.1, p. 93
`%Y, %y, %m, %B, %b, %d`	Date conversion specifications	Section 4.4.1, p. 93
`month.name`, `month.abb`	Built-in month name objects	Section 4.4.1, p. 94
`format`	Extract components of a date	Section 4.4.1, p. 95

Function/operator	Brief description	First occurrence
`weekdays`	Identify weekday of a date	Section 4.4.1, p. 95
`quarters`	Identify quarter of a date	Section 4.4.1, p. 95
`as.numeric`	Numeric representation of factor	Section 4.4.2, p. 96
`Sys.time`	Report your local date-time	Section 4.4.3, p. 97
`strptime`	Create date-time object	Section 4.4.3, p. 98
`%H`, `%M`, `%S`	Time conversion specifications	Section 4.4.3, p. 98
`OlsonNames`	Olson time zone names	Section 4.4.3, p. 98

5
LISTS AND DATA FRAMES



Vectors, matrices, and arrays are efficient and convenient data storage structures in R, but they have a distinct limitation: They can store only one type of data. In this chapter, you'll explore two more data structures, lists and data frames, which can store multiple types of values at once.

5.1 Lists of Objects

The *list* is an incredibly useful data structure you can use to group together any mix of R structures and objects. A single list could contain a numeric matrix, a logical array, a single character string, and a factor object. You can even have a list as a component of another list. In this section, you'll create, modify, and access components of these flexible structures.

5.1.1 Creating Lists and Accessing Components

Creating a list is much like creating a vector. You supply the elements that you want to include to the `list` function, separated by commas. For

example:

```
R> foo <- list(matrix(data=1:4,nrow=2,ncol=2),c(T,F,T,T),"hello")
R> foo
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1]  TRUE FALSE  TRUE  TRUE

[[3]]
[1] "hello"
```

In the list `foo`, you store a 2×2 numeric matrix, a logical vector, and a character string. These print in the order you supplied them to `list`. Just as with vectors, you can use the `length` function to check the number of components in a list:

```
R> length(x=foo)
[1] 3
```

You can retrieve components from a list using indexes, which you enter in double square brackets:

```
R> foo[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> foo[[3]]
[1] "hello"
```

We call this action a *member reference*. When you retrieve a component this way, you can treat it like a stand-alone object in the workspace; nothing special needs to be done. For example:

```
R> foo[[1]] + 5.5
     [,1] [,2]
```

```
[1,]  6.5  8.5
[2,]  7.5  9.5
R> foo[[1]][1,2]
[1] 3
R> foo[[1]][2,]
[1] 2 4
R> cat(foo[[3]],"you!")
hello you!
```

To overwrite a member of `foo`, use the assignment operator:

```
R> foo[[3]]
[1] "hello"
R> foo[[3]] <- paste(foo[[3]],"you!")
R> foo
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1]  TRUE FALSE  TRUE  TRUE

[[3]]
[1] "hello you!"
```

Say you want to access the second and third components of `foo` and store them as one object. Your first instinct might be to try something like this:

```
R> foo[[c(2,3)]]
[1] TRUE
```

But R hasn't done what you wanted. Instead, it returned the third element of the second component. This is because R always interprets double square brackets on a list with respect to a single member. Fortunately, member referencing with double square brackets isn't the only way to access components of a list. You can also use single square

bracket notation, or *list slicing*, which lets you select multiple list items at once:

```
R> bar <- foo[c(2,3)]
R> bar
[[1]]
[1]  TRUE FALSE  TRUE  TRUE

[[2]]
[1] "hello you!"
```

The result `bar` is a list with the two components stored in the order you requested them.

5.1.2 Naming

As with vectors (see Section 2.3.1), you can *name* list members to make the objects therein easier to retrieve and work with when coding. Let's start by adding names to the list `foo` from earlier:

```
R> names(foo) <- c("mymatrix","mylogicals","mystring")
R> foo
$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$mylogicals
[1]  TRUE FALSE  TRUE  TRUE

$mystring
[1] "hello you!"
```

This changes how the object prints to the console. Where earlier it printed `[[1]]`, `[[2]]`, and `[[3]]` before each component, now it prints the names you specified: `$mymatrix`, `$mylogicals`, and `$mystring`. You can perform member referencing using these names and the dollar operator, rather than the double square brackets:

```
R> foo$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

This is the same as calling `foo[[1]]`. In fact, even when an object is named, you can still use the numeric index to obtain a member:

```
R> foo[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Subsetting named members works the same way:

```
R> all(foo$mymatrix[,2]==foo[[1]][,2])
[1] TRUE
```

This confirms (using the `all` function from Section 4.1.2) that these two ways of extracting the second column of the matrix in `foo` provide an identical result.

To name the components of a list as you're creating it, assign a label to each component in the `list` command. Using some components of `foo`, create a new, named list:

```
R> baz <- list(tom=c(foo[[2]],T,T,T,F),dick="g'day mate",harry=foo$mymatrix*2)
R> baz
$tom
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

$dick
[1] "g'day mate"

$harry
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

The object `baz` now contains the three named components `tom`, `dick`, and `harry`:

```
R> names(baz)
[1] "tom"   "dick"  "harry"
```

To rename these members, assign a character vector of length 3 to `names(baz)` the same way you did for `foo` earlier.

NOTE

When using the `names` *function, R always expects and returns the component names as character strings in double quotes. However, if you're specifying names when creating a list (inside the* `list` *function) or using names to extract members with the dollar operator, R understands the names without quotes (in other words, they are* not *given as strings).*

5.1.3 Nesting

As noted earlier, a member of a list can itself be a list. When nesting lists like this, it's important to keep track of the depth of any member for subsetting or extraction later.

You can add components to any existing list by using the dollar operator and a *new* name. Here's an example using `foo` and `baz` from earlier:

```
R> baz$bobby <- foo
R> baz
$tom
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

$dick
[1] "g'day mate"

$harry
     [,1] [,2]
[1,]    2    6
```

```
[2,]    4    8
$bobby
$bobby$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$bobby$mylogicals
[1]  TRUE FALSE  TRUE  TRUE

$bobby$mystring
[1] "hello you!"
```

You define a fourth component to the list `baz`, `bobby`. You assign the member `bobby` the entire list `foo`. After printing the new `baz`, there are three components in `bobby`. Naming and indexes are now both layered, and you can use either (or combine them) to retrieve members of the inner list:

```
R> baz$bobby$mylogicals[1:3]
[1]  TRUE FALSE  TRUE
R> baz[[4]][[2]][1:3]
[1]  TRUE FALSE  TRUE
R> baz[[4]]$mylogicals[1:3]
[1]  TRUE FALSE  TRUE
```

These instruct R to return the first three elements of the logical vector stored as the second component (`[[2]]`, also named `mylogicals`) of the list `bobby`, which is the fourth component of the list `baz`. As long as you're aware of what's returned at each layer of a subset, you can continue to subset as needed using names and numeric indexes. Consider the third line in this example. The first layer of the subset is `baz[[4]]`, which is a list with three components. The second layer of subsetting extracts the component `mylogicals` from that list by calling `baz[[4]]$mylogicals`. This component represents a vector of length 4, so the third layer of subsetting retrieves the first three elements of that vector with the line `baz[[4]]$mylogicals[1:3]`.

We often use lists to return output from various R functions, but they can quickly become rather large objects in terms of system resources to store. When you have only one type of data, it's best to stick to using basic vector, matrix, or array structures to record and store the observations.

Exercise 5.1

a. Create a list that contains, in this order: a sequence of 20 evenly spaced numbers between –4 and 4 (recall the `seq` command); a 3×3 matrix of the logical vector `c(F,T,T,T,F,T,T,F,F)` filled column-wise; a character vector with the strings `"don"` and `"quixote"`; and a factor containing the observations `c("LOW","MED","LOW","MED","MED","HIGH")`. Then, do the following:

   i. Extract row elements `2` and `1` of columns `2` and `3`, in that order, of the logical matrix.

   ii. Use `sub` (refer to Section 4.2.4) to overwrite `"quixote"` with `"Quixote"` and `"don"` with `"Don"` inside the list. Then, using this newly overwritten list member and the appropriate escape sequences (see Section 4.2.3), concatenate to the console screen the following statement:

```
"Windmills! ATTACK!"
    -\Don Quixote/-
```

   iii. Use logical flag subsetting to obtain all values from the sequence between –4 and 4 that are greater than 1.

   iv. Using `which`, determine which indexes in the factor vector are assigned the `"MED"` level.

b. Create a new list with the factor vector from (a) as a component named `"facs"`; the numeric vector `c(3,2.1,3.3,4,1.5,4.9)` as a component named `"nums"`; and a nested list comprising the first three members of the (a) list (use list slicing to obtain this), named `"oldlist"`. Then, do the following:

   i. Extract the elements of `"facs"` that correspond to elements of `"nums"` that are greater than or equal to 3.

   ii. Add a new member to the list named `"flags"`. This member should be a logical vector of length 6, obtained as a twofold repetition of the third column of the logical matrix in the `"oldlist"` component.

   iii. Use `"flags"` and the logical negation operator `!` to extract the entries of `"nums"` corresponding to `FALSE`.

   iv. Overwrite the character string vector component of `"oldlist"` with the single character string `"Don Quixote"`.

5.2 Data Frames

A *data frame* is R's most natural way of presenting a dataset with a collection of recorded observations for one or more variables. Like lists, data frames have no restriction on the data types of the variables; you can store numeric data, factor data, and so on. The R data frame is essentially a list with extra rules. The most important distinction is that in a data frame (unlike a list), the members must all be vectors of equal length.

The data frame is one of the most important and frequently used objects in R for statistical data analysis. In this section, you'll create data

frames and learn about their general characteristics.

5.2.1 Construction

To create a data frame from scratch, use the `data.frame` function. Supply your data, grouped by variable, as vectors of the same length, which is the same way you'd construct a named list. Consider the following example dataset:

```
R> mydata <- data.frame(person=c("Peter","Lois","Meg","Chris","Stewie"),
                        age=c(42,40,17,14,1),
                        sex=factor(c("M","F","F","M","M")))
R> mydata
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
5 Stewie   1   M
```

You construct a data frame with the first name, age in years, and sex of five individuals. The returned object clarifies why vectors passed to `data.frame` must be of equal length: Vectors of differing lengths don't make sense in this context. If you pass vectors of unequal length to `data.frame`, R attempts to recycle any shorter vectors to match the longest; this can increase your risk of making mistakes in data entry. Data frames print to the console in rows and columns; they look more like a matrix than a named list. This natural spreadsheet style makes it easy to read and manipulate datasets. We refer to each row in a data frame as a *record*, and each column as a *variable*.

A useful summary-style function for data frames is `str` (which means *structure*). This provides a quick look at a data frame by previewing its contents and the data types of the present variables:

```
R> str(mydata)
'data.frame':   5 obs. of  3 variables:
 $ person: chr  "Peter" "Lois" "Meg" "Chris" ...
```

```
 $ age   : num  42 40 17 14 1
 $ sex   : Factor w/ 2 levels "F","M": 2 1 1 2 2
```

Here, `str` correctly notes `mydata` has five records and three variables. It notes the names of each variable and the corresponding type (`chr` for character strings, `num` for numeric, and `Factor` for a factor, which also reveals the number of levels present), and prints out the first few observations of each. The `str` function is especially useful when dealing with large datasets that may also have many variables.

You can easily request specific pieces of basic descriptive information about a data frame. For example, you report the size of a data frame—the number of records and variables—just as with the dimensions of a matrix (see Section 3.1.1):

```
R> nrow(mydata)
[1] 5
R> ncol(mydata)
[1] 3
R> dim(mydata)
[1] 5 3
```

The `nrow` function retrieves the number of rows (records), `ncol` retrieves the number of columns (variables), and `dim` retrieves both.

For data frames with many records, it's helpful to take a quick look at the first few rows with the `head` command. In using `head`, you supply the data frame first and the desired number of rows second (the latter to the argument `n`, which defaults to 6). Here are the first three rows of `mydata`:

```
R> head(mydata,n=3)
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
```

Similarly, you can use the `tail` command to show the *last* `n` rows of a data frame:

```
R> tail(mydata,n=2)
  person age sex
4  Chris  14   M
5 Stewie   1   M
```

The variable names in a dataset are given by the column names of the corresponding R data frame object. Extract the character vector of these using the `colnames` function:

```
R> colnames(mydata)
[1] "person" "age"    "sex"
```

Though you can also define and extract row names, they're not usually as important as the column names. By default, R sequentially numbers rows of a data frame and assumes these numbers are the row names. However, they're still character strings:

```
R> rownames(mydata)
[1] "1" "2" "3" "4" "5"
```

If you want custom row names present in the creation of a data frame object, supply a vector of character strings equal to the number of records in the dataset to the optional `row.names` argument of `data.frame`.

Finally, the more general `dimnames` function returns a list of two members giving both the row and column names together:

```
R> dimnames(mydata)
[[1]]
[1] "1" "2" "3" "4" "5"

[[2]]
[1] "person" "age"    "sex"
```

With these functions you can also change both the variable (column) and record (row) names after your object has been created. See Section 6.2.1 for an example of this usage.

5.2.2 Basic Subsets

You can extract portions of data by specifying row and column index positions (much as with a matrix). Here's an example, continuing with the `mydata` object:

```
R> mydata[2,2]
[1] 40
```

This gives you the element at row 2, column 2—the age of Lois. Now extract the third, fourth, and fifth elements of the third column:

```
R> mydata[3:5,3]
[1] F M M
Levels: F M
```

This returns a factor vector with the sex of Meg, Chris, and Stewie. The following extracts the entire third and first columns (in that order):

```
R> mydata[,c(3,1)]
  sex person
1   M  Peter
2   F   Lois
3   F    Meg
4   M  Chris
5   M Stewie
```

This results in another data frame giving the sex and name of each person.

You can also use the names of the vectors you passed to `data.frame` to access variables even if you don't know their column index positions, which can be useful for large datasets. Use the same dollar operator as with member-referencing named lists:

```
R> mydata$age
[1] 42 40 17 14  1
```

You can subset this returned vector, too:

```
R> mydata$age[2]
[1] 40
```

This returns the same result as the earlier call of `mydata[2,2]`.

5.2.3 Column Insertions and Data Frame Combinations

Say you want to add data to an existing data frame. This could be a set of observations for a new variable (adding to the number of columns) or more records (adding to the number of rows). To do so, you can use some more functions you've already seen applied to matrices.

Recall the `rbind` and `cbind` functions from Section 3.1.1, which let you append rows and columns, respectively. These same functions extend data frames intuitively. For example, suppose you had another record to include in `mydata`: the age and sex of another individual, Brian. The first step is to create a new data frame that contains Brian's information:

```
R> newrecord <- data.frame(person="Brian",age=7,
                           sex=factor("M",levels=levels(mydata$sex)))
R> newrecord
  person age sex
1  Brian   7   M
```

To avoid confusion, make sure the variable names and the data types match the data frame you're adding to. Note that for a factor, the same rules apply as if you were extending a stand-alone factor object (refer to Section 4.3.6). In `newrecord`, you choose to explicitly define Brian's value for `sex` as a single value of a factor with the same levels as the existing factor in `mydata` (using the `levels` function). However, simply writing `sex=factor("M")` or even `sex="M"` when creating `newrecord` also works in the subsequent code.

Once that's done, call the following:

```
R> mydata <- rbind(mydata,newrecord)
R> mydata
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
```

```
5 Stewie   1   M
6  Brian   7   M
```

Using `rbind`, you combine `mydata` with the new record and overwrite the original `mydata` with the result.

Adding a variable to a data frame is also straightforward. Say you're given data on the classification of how funny these six individuals are, which you define as a "degree of funniness." The degree of funniness can take three possible values: `Low`, `Med` (medium), and `High`. Suppose Peter, Lois, and Stewie have a high degree of funniness, Chris and Brian have a medium degree, and Meg has a low degree. In R, you'd have a factor vector like this:

```
R> funny <- c("High","High","Low","Med","High","Med")
R> funny <- factor(x=funny,levels=c("Low","Med","High"))
R> funny
[1] High High Low  Med  High Med
Levels: Low Med High
```

The first line creates the basic character vector as `funny`, and the second line overwrites `funny` by turning it into a factor. The order of these elements must correspond to the records in your data frame.

Now use `cbind` to append this factor vector as a column to the existing `mydata`:

```
R> mydata <- cbind(mydata,funny)
R> mydata
  person age sex funny
1  Peter  42   M  High
2   Lois  40   F  High
3    Meg  17   F   Low
4  Chris  14   M   Med
5 Stewie   1   M  High
6  Brian   7   M   Med
```

The `rbind` and `cbind` functions aren't the only ways to extend a data frame. One useful alternative for adding a variable is to use the dollar operator, much like adding a new member to a named list, as in Section

5.1.3. Suppose you want to add another variable to `mydata` by including a column with the age of the individuals in months, not years, calling this new variable `age.mon`:

```
R> mydata$age.mon <- mydata$age*12
R> mydata
  person age sex funny age.mon
1  Peter  42   M  High     504
2   Lois  40   F  High     480
3    Meg  17   F   Low     204
4  Chris  14   M   Med     168
5 Stewie   1   M  High      12
6  Brian   7   M   Med      84
```

This creates a new `age.mon` column with the dollar operator and at the same time assigns it the vector of ages in years (already stored in `mydata` as `age`) multiplied by 12.

5.2.4 Logical Record Subsets

In Section 4.1.5, you used logical flag vectors to subset data structures. This is a particularly useful technique with data frames, where you'll often want to examine a subset of entries that meet certain criteria. For example, when working with data from a clinical drug trial, a researcher might want to examine the results for just male participants and compare them to the results for females. Or the researcher might want to look at the characteristics of individuals who responded most positively to the drug.

Let's continue to work with `mydata`. Say you want to examine all records corresponding to males. From Section 4.3.1, you know that the following line identifies the relevant positions in the `sex` factor vector:

```
R> mydata$sex=="M"
[1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE
```

This flags the male records. You can use this with the matrix-like syntax from Section 5.2.1 to get the male-only subset:

```
R> mydata[mydata$sex=="M",]
  person age sex funny age.mon
1  Peter  42   M  High     504
4  Chris  14   M   Med     168
5 Stewie   1   M  High      12
6  Brian   7   M   Med      84
```

This returns data for all variables for only the male participants. You can use the same behavior to choose which variables to return in the subset. For example, since you're selecting only males, you could omit `sex` from the result using a negative numeric index in the column dimension:

```
R> mydata[mydata$sex=="M",-3]
  person age funny age.mon
1  Peter  42  High     504
4  Chris  14   Med     168
5 Stewie   1  High      12
6  Brian   7   Med      84
```

If you don't have the column number or you want to have more control over the returned columns, use a character vector of variable names instead:

```
R> mydata[mydata$sex=="M",c("person","age","funny","age.mon")]
  person age funny age.mon
1  Peter  42  High     504
4  Chris  14   Med     168
5 Stewie   1  High      12
6  Brian   7   Med      84
```

The logical conditions you use to subset a data frame can be as simple or as complicated as necessary. The logical flag vector you place in the square brackets just has to match the number of records in the data frame. Now extract from `mydata` the full records for individuals who are more than 10 years old *or* have a high degree of funniness:

```
R> mydata[mydata$age>10|mydata$funny=="High",]
  person age sex funny age.mon
1  Peter  42   M  High     504
```

```
2   Lois  40   F  High     480
3    Meg  17   F   Low     204
4  Chris  14   M   Med     168
5 Stewie   1   M  High      12
```

Sometimes, asking for a subset will yield no records. In this case, R returns a data frame with zero rows:

```
R> mydata[mydata$age>45,]
[1] person  age     sex     funny   age.mon
<0 rows> (or 0-length row.names)
```

In this example, `mydata` returns no records because there are no individuals older than 45. To check whether a subset contains any records, use `nrow` on the result; if this is equal to zero, then no records satisfy the specified condition(s).

5.2.5 The Explicit subset Command

Lastly, it's worth noting the availability of an explicit R function for the kinds of subsetting operations you've seen so far: `subset`. You can use this function instead of the square-bracket operator to achieve the same results. However, the square-bracket operator is a more literal, direct approach to subsetting (in addition to being a bit more compact).

The `subset` function works on vectors, matrices and data frames; I'll illustrate it here briefly for the latter. Its first argument `x` is the object to be interrogated; the second argument, which shares the name of the function itself, is where you stipulate row conditions; and the third argument, `select`, is where you stipulate the desired columns to be returned.

Continuing with the `mydata` object, suppose you wish to obtain the `age.mon` and `sex` values, in that order, for those individuals with an age greater than 10 years *or* with a high degree of funniness. Using skills from Section 5.2.4, you do something like this:

```
R> mydata[mydata$age>10|mydata$funny=="High",c("age.mon","sex")]
  age.mon sex
```

```
1     504   M
2     480   F
3     204   F
4     168   M
5      12   M
```

Now do the same thing using the `subset` function:

```
R> subset(x=mydata,subset=age>10|funny=="High",select=c(age.mon,sex))
  age.mon sex
1     504   M
2     480   F
3     204   F
4     168   M
5      12   M
```

A benefit of the explicit `subset` function is that you don't need to preface the variables used in setting the row conditions (`age` and `funny` in the previous example) by the name of the data frame itself with the dollar operator (you can see `mydata$` being used with the square-bracket operator).

Exercise 5.2

a. Create and then store this data frame as `dframe` in your R workspace:

person	sex	funny
Stan	M	High
Francine	F	Med
Steve	M	Low
Roger	M	High

person	sex	funny
Hayley	F	Med
Klaus	M	Med

The variables `person`, `sex`, and `funny` should be identical in nature to the variables in the `mydata` object from Section 5.2. That is, `person` should be a character vector, `sex` should be a factor with levels `F` and `M`, and `funny` should be a factor with levels `Low`, `Med`, and `High`.

b. Identify the structure of `dframe`. In addition, get R to report the row and column names.

c. Stan and Francine are 41 years old, Steve is 15, Hayley is 21, and Klaus is 60. Roger is extremely old: 1,600 years. Append these data (in the appropriate order) as a new numeric column variable in `dframe` called `age`.

d. Use your knowledge of reordering the column variables based on column index positions to overwrite `dframe`, bringing it in line with `mydata`. That is, the first column should be `person`, the second column `age`, the third `sex`, and the fourth `funny`.

e. Turn your attention to `mydata` as it was left after you included the `age.mon` variable in Section 5.2.3, page 114. Create a new version of `mydata` called `mydata2` by deleting the `age.mon` column.

f. Now use `rbind` to combine `mydata2` with `dframe`, naming the resulting object `mydataframe`.

g. Write a single line of code that extracts from `mydataframe` just the names and ages of any records where the individual is female AND has a level of funniness equal to `Med` OR `High`.

h. Use your knowledge of handling character strings in R to extract all records from `mydataframe` that correspond to people whose names start with *S*.
Hint: Recall `substr` from Section 4.2.4 (note that you can apply `substr` to a vector of multiple character strings).

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`list`	Create a list	Section 5.1.1, p. 104
`[[ ]]`	Unnamed member reference	Section 5.1.1, p. 104
`[ ]`	List slicing (multiple members)	Section 5.1.1, p. 105
`names`	Obtain list member names	Section 5.1.2, p. 106
`$`	Get named member/variable	Section 5.1.2, p. 106
`data.frame`	Create a data frame	Section 5.2.1, p. 110
`str`	Summarize structure of data frame	Section 5.2.1, p. 110
`nrow`, `ncol`, `dim`	Dimensions/size of data frame	Section 5.2.1, p. 111
`head`, `tail`	Show first/last rows of data frame	Section 5.2.1, p. 111

Function/operator	Brief description	First occurrence
`colnames`	Variable names of data frame	Section 5.2.1, p. 111
`rownames`	Record names of data frame	Section 5.2.1, p. 111
`dimnames`	List of both variable and record names	Section 5.2.1, p. 112
`[ , ]`	Extract data frame row/columns	Section 5.2.2, p. 112
`subset`	Explicit subset command	Section 5.2.5, p. 117

6
SPECIAL VALUES, CLASSES, AND COERCION



You've learned about the unique properties and applications of numeric values, logicals, character strings, and factors. Now you'll look at some special values in R that aren't as well-defined. You'll see how they might come about and how to handle and test for them. Then you'll look at different data types in R and some general object class concepts.

6.1 Special Values

Many situations in R call for special values. For example, the software has unique terms it reserves for when a dataset has missing observations or when calculating a practically infinite number. You can use these special values to mark abnormal or missing values in vectors, arrays, or other data structures.

6.1.1 Infinity

In Section 2.1, I mentioned that R imposes limits on how extreme a number can be before the computer cannot reliably represent it. When

a number is too large for R to represent, it deems the value *infinite*. The mathematical concept of infinity (∞) does not correspond to a specific number; R simply has to define an extreme cutoff point. The precise cutoff value varies from system to system and is governed in part by the amount of memory R has access to. The special object `Inf`, which is case sensitive, represents this value. R only associates `Inf` with numeric vectors. Let's create some objects to test it out:

```
R> foo <- Inf
R> foo
[1] Inf
R> bar <- c(3401,Inf,3.1,-555,Inf,43)
R> bar
[1] 3401.0    Inf    3.1 -555.0    Inf   43.0
R> baz <- 90000^100
R> baz
[1] Inf
```

Here, you define an object `foo` that's a single instance of an infinite value. You also define a numeric vector, `bar`, with two infinite elements, and then raise 90, 000 to a power of 100 in `baz` to produce a result R deems infinite.

R represents negative infinity with `-Inf`:

```
R> qux <- c(-42,565,-Inf,-Inf,Inf,-45632.3)
R> qux
[1]    -42.0    565.0     -Inf     -Inf      Inf -45632.3
```

This creates a vector with two negative-infinite values and one positive-infinite value.

Though infinity doesn't represent a specific value, you can perform mathematical operations on infinite values in R to a certain extent. For example, multiplying `Inf` by any negative value results in `-Inf`:

```
R> Inf*-9
[1] -Inf
```

If you add to or multiply infinity, you also get infinity as a result:

```
R> Inf+1
[1] Inf
R> 4*-Inf
[1] -Inf
R> -45.2-Inf
[1] -Inf
R> Inf-45.2
[1] Inf
R> Inf+Inf
[1] Inf
R> Inf/23
[1] Inf
```

Zero and infinity go hand in hand when it comes to division. Dividing any (finite) numeric value by infinity, positive or negative, results in zero:

```
R> -59/Inf
[1] 0
R> -59/-Inf
[1] 0
```

Though it isn't mathematically defined, in R, dividing any nonzero value by zero results in infinity (positive or negative, depending on the sign of the numerator):

```
R> -59/0
[1] -Inf
R> 59/0
[1] Inf
R> Inf/0
[1] Inf
```

This is because evaluations like this can arise in code if the divisor is extremely close to zero, but not exactly zero. Again, like the need for `Inf` itself, R is limited to a certain level of precision for very tiny numbers.

NOTE

In general computing, we call these arithmetical issues with respect to `Inf` numerical overflow *and* numerical underflow. *The former term describes extreme values that exceed a system's ability to represent them, and the latter describes values with an unachievable decimal-place precision, often by way of values so close to zero as to be indistinguishable.*

You typically want to detect infinite values in a data structure as a first step. The functions `is.infinite` and `is.finite` take in a collection of values, typically a vector, and return for each element a logical value answering the question posed. Here's an example using `qux` from earlier:

```
R> qux
[1]    -42.0    565.0     -Inf     -Inf      Inf -45632.3
R> is.infinite(x=qux)
[1] FALSE FALSE  TRUE  TRUE  TRUE FALSE
R> is.finite(x=qux)
[1]  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

Note that these functions don't distinguish between positive or negative infinity, and the result of `is.finite` is always the opposite (the negation) of the result of `is.infinite`.

Finally, relational operators function as you might expect:

```
R> -Inf<Inf
[1] TRUE
R> Inf>Inf
[1] FALSE
R> qux==Inf
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
R> qux==-Inf
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```

Here, the first line confirms that R treats `-Inf` as less than `Inf`, and the second line shows that `Inf` is not greater than `Inf`. The third and fourth lines, again using `qux`, test for equality, which is a useful way to distinguish between positive and negative infinity.

6.1.2 NaN: Not a Number

In some situations, it's impossible to express the result of a calculation using a number, `Inf`, or `-Inf`. We label these difficult-to-quantify special values `NaN` in R, which stands for *Not a Number*.

As with infinite values, R associates `NaN` values with numeric observations. It's possible to define or include a `NaN` value directly, but we rarely encounter them in this way:

```
R> foo <- NaN
R> foo
[1] NaN
R> bar <- c(NaN,54.3,-2,NaN,90094.123,-Inf,55)
R> bar
[1]      NaN    54.30    -2.00      NaN 90094.12     -Inf    55.00
```

Typically, `NaN` is the unintended result of attempting a calculation that's impossible to perform with the specified values.

In Section 6.1.1, you saw that adding or subtracting a number from `Inf` or `-Inf` results in `Inf` or `-Inf`. However, attempting to cancel representations of infinity in any way results in `NaN`:

```
R> -Inf+Inf
[1] NaN
R> Inf/Inf
[1] NaN
```

Here, the first line won't result in zero because R can't interpret positive and negative infinity in that numeric sense, so you get `NaN` as a result. The same thing happens if you attempt to divide `Inf` by itself. In addition, although you saw earlier that dividing a nonzero value by zero results in positive or negative infinity, dividing *zero* by zero results in `NaN`:

```
R> 0/0
[1] NaN
```

Any mathematical operation involving `NaN` results in `NaN`:

```
R> NaN+1
[1] NaN
R> 2+6*(4-4)/0
[1] NaN
R> 3.5^(-Inf/Inf)
[1] NaN
```

In the first line, adding 1 to `NaN` is still `NaN`. In the second line, you obtain `NaN` from the `(4-4)/0`, which is `0/0`, so the result is also `NaN`. In the third line, `NaN` results from `-Inf/Inf`, so the result of the remaining calculation is again `NaN`. This gives you an idea of how `NaN` or infinite values might unintentionally crop up. If you have a function that passes various values to a fixed calculation and you don't take care to prevent, for example, `0/0` from occurring, then the code returns `NaN`.

Like with `Inf`, we use a special function (`is.nan`) to detect the presence of `NaN` values. Unlike infinite values, however, you can't use relational operators with `NaN`. Here's an example using `bar`, which was defined earlier on page 122:

```
R> bar
[1]      NaN    54.30    -2.00      NaN 90094.12     -Inf    55.00
R> is.nan(x=bar)
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
R> !is.nan(x=bar)
[1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
R> is.nan(x=bar)|is.infinite(x=bar)
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
R> bar[-(which(is.nan(x=bar)|is.infinite(x=bar)))]
[1]    54.30    -2.00 90094.12    55.00
```

Using the `is.nan` function on `bar` flags the two `NaN` positions as `TRUE`. In the second example, you use the negation operator `!` to flag the positions where the elements are NOT `NaN`. Using the element-wise OR, `|` (see Section 4.1.3), you then identify elements that are either `NaN` OR infinite. Finally, the last line uses `which` to convert these logical values into numeric index positions so that you can remove them with negative

indexes in square brackets (see Section 4.1.5 for a refresher on using `which`).

For more details on the functionality and behavior of `NaN` and `Inf`, enter `?NaN` or `?Inf` at the R prompt.

Exercise 6.1

a. Store the following vector:

```
foo <- c(13563,-14156,-14319,16981,12921,11979,9568,8833,
         -12968,8133)
```

Then use vector subsetting with the appropriate functions to do the following:

i. Obtain all elements of `foo` that, when raised to a power of 75, are NOT represented as infinite values.

ii. Obtain the set of elements of `foo` that exclude those resulting in negative infinity when raised to a power of 75.

b. Store the following 3×4 matrix as the object `bar`:

$$\begin{bmatrix} 77875.40 & 27551.45 & 23764.30 & -36478.88 \\ -35466.25 & -73333.85 & 36599.69 & -70585.69 \\ -39803.81 & 55976.34 & 76694.82 & 47032.00 \end{bmatrix}$$

Now do the following:

i. Identify the coordinate-specific indexes of the entries of `bar` that are `NaN` when you raise `bar` to a power of 65 and divide by infinity.

Hint: Remember the `arr.ind` argument to the `which` function.

ii. Return the values in `bar` that are NOT `NaN` when raising `bar` to a power of 67 and adding infinity to the result. Confirm this is identical to identifying those values in `bar` that, when raised to a power of 67, are not equal to negative infinity.

iii. Identify those values in `bar` that are either negative infinity OR finite when you raise `bar` to a power of 67.

6.1.3 NA: Not Available

In statistical analyses, datasets often contain missing values. For example, someone filling out a questionnaire may not respond to a particular item, or a researcher may incorrectly record some observations from an experiment. Identifying and handling missing values is important so that you can still use the rest of the data. R provides a standard special term to represent missing values, `NA`, which reads as *Not Available*.

`NA` entries are not the same as `NaN` entries. Whereas we use `NaN` only with respect to numeric operations, missing values can occur for any type of observation. As such, `NA`s can exist in both numeric and non-numeric settings. Here's an example:

```
R> foo <- c("character","a",NA,"with","string",NA)
R> foo
[1] "character" "a"         NA          "with"      "string"    NA
R> bar <- factor(c("blue",NA,NA,"blue","green","blue",NA,"red","red",NA,
                 "green"))
R> bar
 [1] blue  <NA>  <NA>  blue  green blue  <NA>  red   red   <NA>  green
Levels: blue green red
R> baz <- matrix(c(1:3,NA,5,6,NA,8,NA),nrow=3,ncol=3)
```

```
R> baz
     [,1] [,2] [,3]
[1,]    1   NA   NA
[2,]    2    5    8
[3,]    3    6   NA
```

The object `foo` is a character vector with entries 3 and 6 missing; `bar` is a factor of length 11 with elements 2, 3, 7, and 10 missing; and `baz` is a numeric matrix with row 1, columns 2 and 3, and row 3, column 3, elements missing. In the factor, the `NA`s print as `<NA>` to differentiate between bona fide levels of the factor and the missing observations.

Like the other special values so far, you can identify `NA` elements using the function `is.na`. This is often useful for removing or replacing `NA` values. Consider the following numeric vector:

```
R> qux <- c(NA,5.89,Inf,NA,9.43,-2.35,NaN,2.10,-8.53,-7.58,NA,-4.58,2.01,NaN)
R> qux
 [1]    NA  5.89   Inf    NA  9.43 -2.35   NaN  2.10 -8.53 -7.58    NA -4.58
[13]  2.01   NaN
```

This vector has a total of 14 entries, including `NA`, `NaN`, and `Inf`. Let's identify `NA` elements:

```
R> is.na(x=qux)
 [1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
[13] FALSE  TRUE
```

As you can see, `is.na` flags the corresponding `NA` entries in `qux` as `TRUE`. But note that it also flags elements 7 and 14, which are `NaN`, not `NA`. Strictly speaking, `NA` and `NaN` are different entities, but numerically they're practically the same, since you can do very little with either value. Using `is.na` labels both as `TRUE`, allowing the user to remove or recode both at the same time.

To identify `NA` and `NaN` entries separately, use `is.nan` in conjunction with logical operators like so:

```
R> which(x=is.nan(x=qux))
[1]  7 14
```

This identifies the index positions whose elements are specifically NaN. To identify only NA entries, try the following:

```
R> which(x=(is.na(x=qux)&!is.nan(x=qux)))
[1]  1  4 11
```

This identifies the element indexes for only the NA entries (by checking for entries where is.na is TRUE AND where is.nan is NOT TRUE).

After locating the offending elements, you could use negative indexes in square brackets to remove them, though R offers a more direct option. The function na.omit takes a structure and deletes all NAs from it; na.omit also applies to NaNs if the elements are numeric:

```
R> quux <- na.omit(object=qux)
R> quux
[1]  5.89   Inf  9.43 -2.35  2.10 -8.53 -7.58 -4.58  2.01
attr(,"na.action")
[1]  1  4  7 11 14
attr(,"class")
[1] "omit"
```

Note that the structure passed to na.omit is given as the argument object and that some additional output displays in printing the returned object. These extra details inform the user that elements in the original vector were removed (in this case, the element positions provided in the attribute na.action). We'll discuss attributes further in Section 6.2.1.

Similar to NaN, arithmetic calculations with NA result in NA. Using relational operators with either NaN or NA also results in NA:

```
R> 3+2.1*NA-4
[1] NA
R> 3*c(1,2,NA,NA,NaN,6)
[1]   3   6  NA  NA NaN  18
R> NA>76
[1] NA
R> 76>NaN
[1] NA
```

You can find more details on the usage and finer technicalities of NA values by entering ?NA at the R prompt.

6.1.4 NULL

We often use the null value, NULL, to explicitly define an empty entity, which is quite different from specifying a missing entity with NA. An instance of NA clearly denotes an existing position that you can access or overwrite if necessary—not so for NULL. You can see an indication of this by comparing the assignment of NA with that of a NULL:

```
R> foo <- NULL
R> foo
NULL
R> bar <- NA
R> bar
[1] NA
```

Note that bar, the NA object, prints with an index position [1]. This suggests you have a vector with a single element. In contrast, you explicitly instructed foo to be empty with NULL. Printing this object doesn't provide a position index because there's no position to access.

This interpretation of NULL also applies to vectors that have other well-defined items. Consider the following:

```
R> c(2,4,NA,8)
[1]  2  4 NA  8
R> c(2,4,NULL,8)
[1] 2 4 8
```

The first line creates a vector of length 4, with the third position coded as NA. The second line creates a similar vector using NULL instead of NA. The result is a vector with a length of only 3. That's because NULL cannot take up a position in the vector. As such, it makes no sense to assign NULL to multiple positions in a vector (or any other structure). Here's an example:

```
R> c(NA,NA,NA)
[1] NA NA NA
R> c(NULL,NULL,NULL)
NULL
```

R interprets the first line as "three possible slots with unrecorded observations." The second line simply provides "emptiness three times," which R interprets as one single, unsubsettable, empty object.

At this point, you might wonder why there's even a need for `NULL`. If something is empty and doesn't exist, why define it in the first place? The answer lies in the need to be able to explicitly state or check whether a certain object has been defined. This occurs often when calling functions in R. For example, when a function contains optional arguments, the function has to internally check which of those arguments have been supplied and which are missing or empty. The `NULL` value is a useful and flexible tool that the author of a function can use to facilitate such checks. The following example foreshadows this, and you'll see more complete examples in Chapter 11.

You use the `is.null` function to check whether something is explicitly `NULL`. Suppose you have a function with an optional argument `opt.arg` that, if supplied, should be a character vector of length 3. Say a user calls this function with the following:

```
R> opt.arg <- c("string1","string2","string3")
```

Now if you were to check whether the argument was supplied using `is.na`, you might call this:

```
R> is.na(x=opt.arg)
[1] FALSE FALSE FALSE
```

The position-holding nature of `NA` values means that this check is element-wise and returns an answer for each value in `opt.arg`. This is problematic because you want only a single answer: Is `opt.arg` empty or is it supplied? This is when `NULL` comes to the party:

```
R> is.null(x=opt.arg)
[1] FALSE
```

As `opt.arg` is not empty, the function can proceed as necessary. If the argument *is* empty, using `NULL` over `NA` for the check is better for these purposes:

```
R> opt.arg <- c(NA,NA,NA)
R> is.na(x=opt.arg)
[1] TRUE TRUE TRUE
R> opt.arg <- c(NULL,NULL,NULL)
R> is.null(x=opt.arg)
[1] TRUE
```

As noted earlier, filling a vector with `NULL` isn't usual practice; it's done here just for illustration. But usage of `NULL` is far from specific to this particular example. You'll use it throughout both ready-to-use and user-contributed functionality in R.

The empty `NULL` has an interesting effect if you include it in arithmetic or relational comparisons:

```
R> NULL+53
numeric(0)
R> 53<=NULL
logical(0)
```

Rather than `NULL` as you might expect, the result is an "empty" vector of a type determined by the nature of the attempted operation. `NULL` typically dominates any arithmetic, even if it includes other special values:

```
R> NaN-NULL+NA/Inf
numeric(0)
```

`NULL` also occurs naturally when examining lists and data frames. For an example, first define a new list `foo`:

```
R> foo <- list(member1=c(33,1,5.2,7),member2="NA or NULL?")
R> foo
```

```
$member1
[1] 33.0  1.0  5.2  7.0

$member2
[1] "NA or NULL?"
```

Look at what happens when you try to access a member in `foo` by the name of `member3`, which isn't included in this list:

```
R> foo$member3
NULL
```

The result of `NULL` signals that `member3` in `foo` doesn't exist, or in R terms, is empty. Therefore, you can fill it with whatever you want:

```
R> foo$member3 <- NA
R> foo
$member1
[1] 33.0  1.0  5.2  7.0

$member2
[1] "NA or NULL?"

$member3
[1] NA
```

The same principle applies when querying a data frame for a nonexistent column or variable using the dollar operator (as in Section 5.2.3).

For more technical details on how R handles `NULL` and `is.null`, access the help file with `?NULL`.

Exercise 6.2

a. Consider the following line of code:

```
foo <- c(4.3,2.2,NULL,2.4,NaN,3.3,3.1,NULL,3.4,NA)
```

Decide which of the following statements are correct. Then use R to confirm.

i. The length of `foo` is 8.

ii. Calling `which(x=is.na(x=foo))` doesn't result in `4` and `8`.

iii. Checking `is.null(x=foo)` provides you with the locations of the two `NULL` values that are present.

iv. Executing `is.na(x=foo[8])+4/NULL` doesn't result in `NA`.

b. Create and store a list containing a single member: the vector `c(7,7,NA,3,NA,1,1,5,NA)`. Then, do the following:

i. Use the `names` function (see Section 5.1.2) to name this single member `"alpha"`.

ii. Confirm that the list doesn't have a member with the name `"beta"` using the appropriate "is-dot" function.

iii. Add a new member called `beta` to the list, which is the vector obtained by identifying the index positions of `alpha` that are `NA`.

6.2 Types, Classes, and Coercion

By now, you've studied many of the fundamental features of R for representing, storing, and handling data. In this section, you'll examine how to formally distinguish between different kinds of values and structures and look at some simple examples of conversion from one type to another.

6.2.1 Attributes

Each R object you create comes with additional information about the nature of the object itself. This additional information is referred to as the object's *attributes*. You've come across attributes already on several occasions. In Section 2.3.1, you created name attributes for individual entries of a vector and you identified the dimensions attribute of a matrix using `dim` in Section 3.1.1. In Section 4.3.1, you used `levels` to examine the levels attribute of a factor. You also noted that `names` can get the member names of a list in Section 5.1.2, and you saw the use of `rownames`, `colnames`, and `dimnames` to access the names attributes associated with the records and variables of a data frame in Section 5.2.1. In this chapter, you saw that an attribute annotates the result of applying `na.omit` to a vector in Section 6.1.3.

In general, attributes are either *explicit* or *implicit*. Explicit attributes are immediately visible to the user, while R determines implicit attributes internally. You can print explicit attributes for a given object with the `attributes` function, which takes any object and returns a named list. Consider, for example, the following 3×3 matrix:

```
R> foo <- matrix(data=1:9,nrow=3,ncol=3)
R> foo
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

R> attributes(foo)
$dim
[1] 3 3
```

Here, calling `attributes` returns a list with one member: `dim`. Knowing about member referencing with list objects (see Section 5.1.2), you can retrieve the contents of `dim` with `attributes(foo)$dim`, but if you know the name of an attribute, you can also use `attr` like so:

```
R> attr(x=foo,which="dim")
[1] 3 3
```

This function takes the object in as `x` and the name of the attribute as a character string to `which`. To make things even more convenient, the most common attributes have their own functions (usually named after the attribute) to access the corresponding value. For the dimensions of a matrix, you've already seen the function `dim`:

```
R> dim(foo)
[1] 3 3
```

These attribute-specific functions also allow access to implicit attributes, which, while still controllable by the user, are set automatically by the software as a matter of necessity. The `names` and `levels` functions seen in previous chapters are also attribute-specific functions.

Explicit attributes are often optional; if unspecified, they are `NULL`. For example, when building a matrix with the `matrix` function, use the optional argument `dimnames` to annotate the rows and columns with names. Pass `dimnames` a list made up of two members, both character vectors of the appropriate lengths, with the first giving row names and the second giving column names. Let's define the matrix `bar` as follows:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
                 c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Because the dimension names are attributes, the `dimnames` appear when you call `attributes(bar)`:

```
R> attributes(bar)
$dim
[1] 3 3

$dimnames
```

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

$dimnames[[2]]
[1] "D" "E" "F"
```

Note that `dimnames` is itself a list, nested inside the larger attributes list. This is just like the corresponding attribute of a data frame object from Section 5.2.1. To extract the values of this attribute, you can use either list member referencing, `attr` as shown earlier, or the attribute-specific function:

```
R> dimnames(bar)
[[1]]
[1] "A" "B" "C"

[[2]]
[1] "D" "E" "F"
```

You can modify some attributes after creating an object (as discussed in Section 5.1.2, where you renamed members of a list). Here, to make `foo` match `bar` exactly, give `foo` some `dimnames` by assigning them to the attribute-specific function:

```
R> dimnames(foo) <- list(c("A","B","C"),c("D","E","F"))
R> foo
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

While we've been using matrices in the discussion here, you can treat optional attributes for other objects in R the same way. Attributes aren't restricted to built-in R objects, either. You can define custom objects you build yourself with their own attributes and attribute-specific functions. Just remember that the role of an attribute is typically to provide descriptive data about an object, or you could end up unnecessarily overcomplicating your object structures.

6.2.2 Object Class

An object's *class* is one of the most useful attributes for describing an entity in R. Every object you create is identified, either implicitly or explicitly, with at least one class. R provides support for *object-oriented* programming, meaning it stores entities as objects of defined classes, which have methods that act upon them. In such a language, we formally refer to class identity as *inheritance*.

NOTE

This section focuses on the most common classing structure in R, S3. *There is another structure,* S4, *which is essentially a more formal set of rules for the identification and treatment of different objects. For most practical intents, understanding and using S3 is more than sufficient. You can find further details in R's online documentation.*

The class of an object is explicit in situations with user-defined object structures or an object such as a factor or data frame where other attributes play an important part in the handling of the object itself. For example, level labels of a factor or variable names in a data frame are modifiable attributes that play a primary role in accessing the observations of each object. Elementary R objects such as vectors, matrices, and arrays, on the other hand, are implicitly classed, which means R doesn't identify the class with the `attributes` function. Whether implicit or explicit, you can always retrieve the class of a given object using the attribute-specific function `class`.

Stand-Alone Vectors

Let's demonstrate the `class` function. Create the following simple vectors to use as examples:

```
R> num.vec1 <- 1:4
R> num.vec1
[1] 1 2 3 4
```

```
R> num.vec2 <- seq(from=1,to=4,length=6)
R> num.vec2
[1] 1.0 1.6 2.2 2.8 3.4 4.0
R> char.vec <- c("a","few","strings","here")
R> char.vec
[1] "a"       "few"     "strings" "here"
R> logic.vec <- c(T,F,F,F,T,F,T,T)
R> logic.vec
[1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE
R> fac.vec <- factor(c("Blue","Blue","Green","Red","Green","Yellow"))
R> fac.vec
[1] Blue   Blue   Green  Red    Green  Yellow
Levels: Blue Green Red Yellow
```

You can pass any object to `class`, and it returns a character vector as output. Here's what happens when you use it on the five example vectors:

```
R> class(num.vec1)
[1] "integer"
R> class(num.vec2)
[1] "numeric"
R> class(char.vec)
[1] "character"
R> class(logic.vec)
[1] "logical"
R> class(fac.vec)
[1] "factor"
```

The output from using `class` on the character vector, the logical vector, and the factor vector matches the kind of data that's been stored. The output from the numeric vectors is a little more intricate, however. Until now, we've referred to any object with an arithmetically valid set of numbers as "numeric." If all the numbers stored in a vector are whole, then R identifies the vector as `"integer"`. R identifies numbers with decimal places (*floating-point* numbers) as `"numeric"`. This distinction is necessary because some tasks may strictly require integers, not floating-point numbers. Colloquially, I'll continue to refer to both types

as "numeric," and in fact, the `is.numeric` function returns `TRUE` for both integer and floating-point structures, as you'll see in Section 6.2.3.

Other Data Structures

As mentioned earlier, R's classes are essentially designed to facilitate object-oriented programming. As such, `class` usually reports on the nature of the data *structure*, rather than the type of data stored; it returns the data type only when used on stand-alone vectors. Let's try it on these matrices:

```
R> num.mat1 <- matrix(data=num.vec1,nrow=2,ncol=2)
R> num.mat1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> num.mat2 <- matrix(data=num.vec2,nrow=2,ncol=3)
R> num.mat2
     [,1] [,2] [,3]
[1,]  1.0  2.2  3.4
[2,]  1.6  2.8  4.0
R> char.mat <- matrix(data=char.vec,nrow=2,ncol=2)
R> char.mat
     [,1]  [,2]
[1,] "a"   "strings"
[2,] "few" "here"
R> logic.mat <- matrix(data=logic.vec,nrow=4,ncol=2)
R> logic.mat
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,] FALSE FALSE
[3,] FALSE  TRUE
[4,] FALSE  TRUE
```

(Recall from Section 4.3.1 that R uses factors only in vector form, so we ignore `fac.vec` for these constructions.) Now examine the output of `class`:

```
R> class(num.mat1)
[1] "matrix" "array"
```

```
R> class(num.mat2)
[1] "matrix" "array"
R> class(char.mat)
[1] "matrix" "array"
R> class(logic.mat)
[1] "matrix" "array"
```

Regardless of the data type, `class` reports the structure of the object itself; they are all arrays (of 2D form, hence the additional qualification of `"matrix"`). The same behavior occurs for other object structures, like higher-dimensional arrays, lists, and data frames.

Multiple Classes

Objects are permitted to inherit multiple classes, usually to distinguish a variant on a standard form of an object. We saw this in the preceding example, with a matrix being a special case of the more general array. Another example is a factor vector with a formally imposed ordering structure (a less-common technical variation to regular factors, as noted briefly in Section 4.3.2), which inherits the usual `factor` class and also contains the additional `ordered` class:

```
R> ordfac.vec <- factor(x=c("Small","Large","Large","Regular","Small"),
                        levels=c("Small","Regular","Large"),
                        ordered=TRUE)
R> ordfac.vec
[1] Small   Large   Large   Regular Small
Levels: Small < Regular < Large
R> class(ordfac.vec)
[1] "ordered" "factor"
```

Earlier, you identified `fac.vec` as `"factor"` only, but the class attribute of the `ordfac.vec` object has two components. It's still identified as `"factor"`, but it also includes `"ordered"`, which identifies it as this particular variant of the `"factor"` class. Here, think of `"ordered"` as a *subclass* of `"factor"`. In other words, it's a special case that inherits from, and therefore fundamentally behaves like, a `"factor"`. For further technical details on R

subclasses, see Chapter 9 of *The Art of R Programming* by Matloff (2011).

NOTE

We're focusing on the `class` *function here because it's directly relevant to the object-oriented programming style exercised in this text, especially in Part II. There are other functions that show some of the complexities of R's classing rules. For example, the function* `typeof` *reports the type of data contained within an object, not just for vectors but also for matrices and arrays. Note, however, that the terminology in the output of* `typeof` *doesn't always match the output of* `class`*. See the help file* `?typeof` *for details on the values it returns.*

Use the `inherits` function to check whether a given R object identifies with any given class or subclass. With a selection of objects defined in this section, consider the following:

```
R> inherits(num.vec2,what="numeric")
[1] TRUE
R> inherits(logic.vec,what="logical")
[1] TRUE
R> inherits(logic.vec,what="numeric")
[1] FALSE
R> inherits(num.mat1,what="ordered")
[1] FALSE
R> inherits(ordfac.vec,what="ordered")
[1] TRUE
```

Here, you confirm (supplying the class name to check the object against to the `what` argument) that `num.vec2` identifies as class `"numeric"`, `logic.vec` is of class `"logical"` and does not identify with class `"numeric"`, and `num.mat1` doesn't inherit from the `"ordered"` class but `ordfac.vec` does.

To summarize, an object's class is first and foremost a descriptor of the data object, though for simple vectors, there's a quirk whereby the `class` function reports the type of data stored. If the vector entries are

exclusively whole numbers, then R classes the vector as `"integer"`, whereas `"numeric"` labels a vector with floating-point numbers.

6.2.3 Is-Dot Object-Checking Functions

Identifying the class of an object is essential for functions that operate on stored objects, especially those that behave differently depending on the class of the object. To check whether the object is a specific class or data type, use R's "is-dot" functions (for example, `is.numeric`). These functions return a `TRUE` or `FALSE` logical value as appropriate.

Is-dot functions exist for almost any sensible check you can think of. For example, consider once more the `num.vec1` vector from Section 6.2.2 and the following six checks:

```
R> num.vec1 <- 1:4
R> num.vec1
[1] 1 2 3 4
R> is.integer(num.vec1)
[1] TRUE
R> is.numeric(num.vec1)
[1] TRUE
R> is.matrix(num.vec1)
[1] FALSE
R> is.data.frame(num.vec1)
[1] FALSE
R> is.vector(num.vec1)
[1] TRUE
R> is.logical(num.vec1)
[1] FALSE
```

The first, second, and sixth is-dot functions check the kind of data the object stores, while the others check the structure of the object itself. The results are what we expect: `num.vec1` *is* an "integer" (and *is* "numeric"), and it *is* a "vector." It's not a matrix or a data frame, nor is it logical.

It's worth noting that these checks use more general categories than the formal classes identified with `class`. Recall that `num.vec1` was identified

solely as `"integer"` in Section 6.2.2, but using `is.numeric` here still returns `TRUE`. Similarly, for a data frame, an object of class `"data.frame"` returns `TRUE` for `is.data.frame` *and* `is.list` because a data frame is intuitively generalized to a list.

There's a difference between these object is-dot functions and functions such as `is.na` from Section 6.1. Think of the functions that check for special values like `NA` as a check for equality; they exist because it's not legal syntax to write something like `foo==NA`. Those functions from Section 6.1 thus operate in R's element-wise fashion, whereas the object is-dot functions inspect the object *itself*, returning only a single logical value.

6.2.4 As-Dot Coercion Functions

You've seen different ways to modify an object after it's been created—by accessing and overwriting elements, for example. But what about the structure of the object itself and the type of data contained within?

Coercion is the process of converting from one object or data type to another. Like other features you've met so far, R performs coercion either implicitly or explicitly. Implicit coercion occurs automatically when elements need to be converted to another type for an operation to complete. In fact, you've come across this behavior in Section 4.1.4, when you made use of the numeric representation of logical values (remember to think of logical values as integers, with 1 for `TRUE` and 0 for `FALSE`). Implicit coercion of logical values to their numeric counterparts occurs in lines of code like this:

```
R> 1:4+c(T,F,F,T)
[1] 2 2 3 5
```

In this operation, R recognizes that you're attempting an arithmetic calculation with `+`, so it expects numeric quantities. Since the logical vector isn't in this form, the software internally coerces it to 1s and 0s before completing the task.

Another common example of implicit coercion is when using `paste` and `cat` to glue together character strings, as in Section 4.2.2. R automatically coerces non-character entries to strings before the concatenation takes place. For example:

```
R> foo <- 34
R> bar <- T
R> paste("Definitely foo: ",foo,"; definitely bar: ",bar,".",sep="")
[1] "Definitely foo: 34; definitely bar: TRUE."
```

Here, R implicitly coerces the integer `34` and the logical `T` (fully represented as `TRUE`) to characters, since the output of `paste` is required be a string.

In other situations, coercion won't happen automatically and must be carried out by the user. You can achieve this explicit coercion with the "as-dot" functions (for example, `as.numeric`). Like is-dot functions, as-dot functions exist for most typical R data types and object classes. You can use explicit coercion in the previous two examples as follows:

```
R> as.numeric(c(T,F,F,T))
[1] 1 0 0 1
R> 1:4+as.numeric(c(T,F,F,T))
[1] 2 2 3 5
R> foo <- 34
R> foo.ch <- as.character(foo)
R> foo.ch
[1] "34"
R> bar <- T
R> bar.ch <- as.character(bar)
R> bar.ch
[1] "TRUE"
R> paste("Definitely foo: ",foo.ch,"; definitely bar: ",bar.ch,".",sep="")
[1] "Definitely foo: 34; definitely bar: TRUE."
```

Coercions are possible in most cases that "make sense." For example, it's easy to see why R is able to read something like this:

```
R> as.numeric("32.4")
[1] 32.4
```

However, the following conversion makes no sense:

```
R> as.numeric("g'day mate")
[1] NA
Warning message:
NAs introduced by coercion
```

Since there's no logical way to translate "g'day mate" into numbers, the entry returns as `NA` (in this case, R also issues a warning message). This means that in certain cases, multiple coercions are necessary to attain the final result. Suppose, for example, you want to coerce the character vector `c("1","0","1","0","0")` to a logical-valued vector. Direct character to logical coercion is not possible because even if all the character strings contained numbers, there's no guarantee in general that they would all be 1s and 0s, as shown here:

```
R> as.logical(c("1","0","1","0","0"))
[1] NA NA NA NA NA
```

However, you can convert character string numbers to a numeric data type, and you can easily coerce 1s and 0s to logicals. So, you can perform the coercion in those two steps as follows:

```
R> as.logical(as.numeric(c("1","0","1","0","0")))
[1]  TRUE FALSE  TRUE FALSE FALSE
```

Not all data-type coercion is entirely straightforward. For example, factors are trickier because R treats the levels as integers. In other words, regardless of how the levels of a given factor are actually labeled, the software refers to them internally as level 1, level 2, and so on. This is clear if you try to coerce a factor to a numeric data type. You saw this in Section 4.3.4, but here's a quick refresher:

```
R> baz <- factor(x=c("male","male","female","male"))
R> baz
```

```
[1] male   male   female male
Levels: female male
R> as.numeric(baz)
[1] 2 2 1 2
```

R assigns the numeric representation of the factor in the stored order of the factor labels (alphabetic by default). Here, level 1 therefore refers to `female`, and level 2 refers to `male`. This example is simple enough, though it's important to be aware of the behavior since coercion from factors with category labels that themselves are essentially numeric can cause confusion:

```
R> qux <- factor(x=c(2,2,3,5))
R> qux
[1] 2 2 3 5
Levels: 2 3 5
R> as.numeric(qux)
[1] 1 1 2 3
```

The numeric representation of the factor `qux` is `c(1,1,2,3)`. This highlights again that R treats the levels of `qux` as level 1 (even though it has a label of `2`), level 2 (which has a label of `3`), and level 3 (which has a label of `5`).

Coercion between object classes and structures can also be useful. For example, you might need to store the contents of a matrix as a single vector:

```
R> foo <- matrix(data=1:4,nrow=2,ncol=2)
R> foo
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> as.vector(foo)
[1] 1 2 3 4
```

Note that `as.vector` coerces the matrix by "stacking" the columns into a single vector. The same column-wise deconstruction occurs for higher-dimensional arrays, in order of layer or block:

```
R> bar <- array(data=c(8,1,9,5,5,1,3,4,3,9,8,8),dim=c(2,3,2))
R> bar
, , 1

     [,1] [,2] [,3]
[1,]    8    9    5
[2,]    1    5    1

, , 2

     [,1] [,2] [,3]
[1,]    3    3    8
[2,]    4    9    8

R> as.matrix(bar)
      [,1]
 [1,]    8
 [2,]    1
 [3,]    9
 [4,]    5
 [5,]    5
 [6,]    1
 [7,]    3
 [8,]    4
 [9,]    3
[10,]    9
[11,]    8
[12,]    8

R> as.vector(bar)
 [1] 8 1 9 5 5 1 3 4 3 9 8 8
```

You can see that `as.matrix` stores the array as a 12×1 matrix, and `as.vector` stores it as a single vector.

Similar commonsense rules for data types apply to coercion when working with object structures. For example, coercing the following list `baz` to a data frame produces an error:

```
R> baz <- list(var1=foo,var2=c(T,F,T),var3=factor(x=c(2,3,4,4,2)))
R> baz
$var1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
$var2
[1]  TRUE FALSE  TRUE

$var3
[1] 2 3 4 4 2
Levels: 2 3 4

R> as.data.frame(baz)
Error in (function (..., row.names = NULL, check.rows = FALSE,
 check.names = TRUE,  :
  arguments imply differing number of rows: 2, 3, 5
```

The error occurs because the variables don't have matching lengths. But there's no problem with performing the same kind of coercion on the following list, `qux`, which has equal-length members:

```
R> qux <- list(var1=c(3,4,5,1),var2=c(T,F,T,T),var3=factor(x=c(4,4,2,1)))
R> qux
$var1
[1] 3 4 5 1

$var2
[1]  TRUE FALSE  TRUE  TRUE
$var3
[1] 4 4 2 1
Levels: 1 2 4

R> as.data.frame(qux)
  var1  var2 var3
1    3  TRUE    4
2    4 FALSE    4
3    5  TRUE    2
4    1  TRUE    1
```

This stores the variables as a dataset in a column-wise fashion, in the order that your list supplies them as members.

This discussion on object classes, data types, and coercion isn't exhaustive, but it serves as a useful introduction to how R deals with issues surrounding the formal identification, description, and handling of the objects you create, which are issues present for most high-level languages. Once you're more familiar with R, the help files (such as the one you can access by entering `?as` at the prompt) provide further details about object handling in the software.

Exercise 6.3

a. Identify the class of the following objects. In each case, also decide whether the class is explicitly or implicitly defined.

   i. `foo <- array(data=1:36,dim=c(3,3,4))`

   ii. `bar <- as.vector(foo)`

   iii. `baz <- as.character(bar)`

   iv. `qux <- as.factor(baz)`

   v. `quux <- bar+c(-0.1,0.1)`

b. For each object defined in (a), find the sum of the result of calling `is.numeric` and `is.integer` on it separately. For example, `is.numeric(foo)+is.integer(foo)` computes the sum for (i). Turn the collection of five results into a factor with levels `0`, `1`, and `2`, identified by the results themselves. Coerce this factor to a numeric vector.

c. Turn this matrix

```
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
```

```
[2,]    3    6    9   12
[3,]    4    7   10   13
```

into the following:

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

Hint: Remember the matrix transpose operation from Section 3.1.3.

d. Store this matrix:

$$\begin{bmatrix} 34 & 0 & 1 \\ 23 & 1 & 2 \\ 33 & 1 & 1 \\ 42 & 0 & 1 \\ 41 & 0 & 2 \end{bmatrix}$$

Then, do the following:

i. Coerce the matrix to a data frame, and use `str` (refer back to Section 5.2.1) to examine the structural details of the resulting object.

ii. As a data frame, coerce and overwrite the second column to be logical-valued.

iii. As a data frame, coerce and overwrite the third column to be factor-valued.

iv. Recheck the structural details of the object, and compare them to the output obtained earlier.

Important Code in This Chapter

Function/operator	Brief description	First occurrence

Function/operator	Brief description	First occurrence
`Inf`, `-Inf`	Value for ±infinity	Section 6.1.1, p. 120
`is.infinite`	Element-wise check for `Inf`	Section 6.1.1, p. 121
`is.finite`	Element-wise check for finiteness	Section 6.1.1, p. 121
`NaN`	Value for invalid numerics	Section 6.1.2, p. 122
`is.nan`	Element-wise check for `NaN`	Section 6.1.2, p. 123
`NA`	Value for missing observation	Section 6.1.3, p. 125
`is.na`	Element-wise check for `NA` OR `NaN`	Section 6.1.3, p. 125
`na.omit`	Delete all `NA`s and `NaN`s	Section 6.1.3, p. 126
`NULL`	Value for “empty”	Section 6.1.4, p. 127
`is.null`	Check for `NULL`	Section 6.1.4, p. 128
`attributes`	List explicit attributes	Section 6.2.1, p. 130
`attr`	Obtain specific attribute	Section 6.2.1, p. 131
`dimnames`	Get array dimension names	Section 6.2.1, p. 132
`class`	Get object class (S3)	Section 6.2.2, p. 133
`inherits`	Test/check object class (S3)	Section 6.2.2, p. 136

Function/operator	Brief description	First occurrence
`is._`	Object-checking functions	Section 6.2.3, p. 136
`as._`	Object-coercion functions	Section 6.2.4, p. 138

7
BASIC PLOTTING



One particularly popular feature of R is its set of incredibly flexible plotting tools for data and model visualization. Mastering R's graphical functionality requires practice, but the fundamental concepts are straightforward. In this chapter, I'll cover the basic mechanics of plotting via an overview of the `plot` function, alongside some useful options for controlling the appearance of the final graphic. Then I'll cover the basics of `ggplot2`, a powerful library for visualizing data in R. You'll learn more about creating different types of statistical plots in Chapter 14 and get into more advanced plotting techniques in Part V.

7.1 Using plot with Coordinate Vectors

To visualize generating plots in R, treat your screen like it's a blank, two-dimensional canvas. You can plot points and lines using x-and y-coordinates. On paper, we usually represent these coordinates with points we write as a pair: (*x* value, *y* value). The R function `plot`, on the other hand, takes in two vectors—one vector of *x* locations and one vector of *y* locations—and opens a *graphics device* where it displays the result. If a graphics device is already open, R's default behavior is to

refresh the device, overwriting the current contents with the new plot. If you're using the RStudio editor, a graphics device is embedded as one of the four panes (see Appendix B).

For example, say you want to plot the points (1.1, 2), (2, 2.2), (3.5, –1.3), (3.9, 0), and (4.2, 0.2). In `plot`, you must provide the vector of the *x* locations first and the *y* locations second. Let's define these as `foo` and `bar`, respectively:

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> plot(foo,bar)
```

Figure 7-1 shows the resulting base R graphics device with the plot. (If you're using the RStudio editor, you'll see the plot appear in one of the four panes.) I'll use this simple dataset as a working example throughout this section.

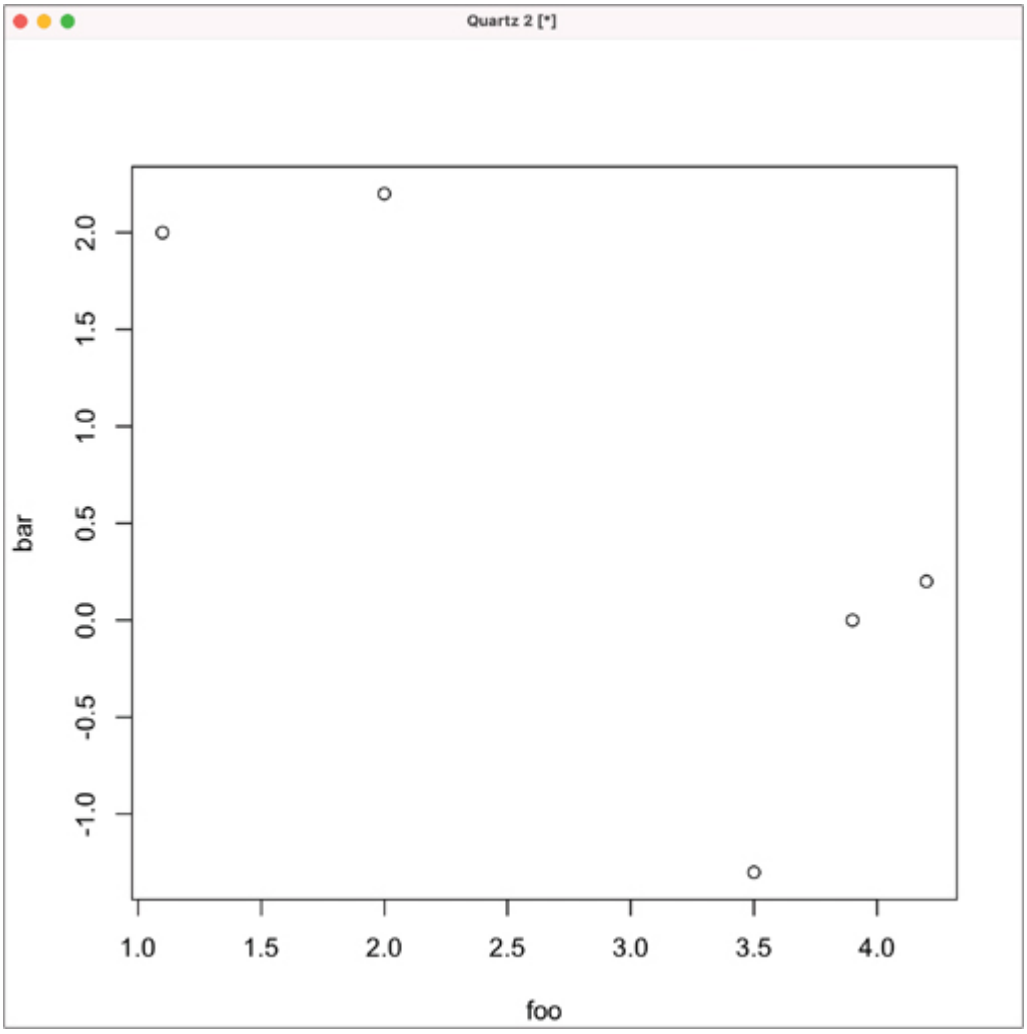


Figure 7-1: The five plotted points using R's default behavior

The *x* and *y* locations don't necessarily need to be specified as separate vectors. You can also supply coordinates in the form of a matrix, with the *x* values in the first column and the *y* values in the second column, or as a list. For example, setting up a matrix of the five points, the following code exactly reproduces Figure 7-1 (note the window pane itself will look slightly different depending on your operating system and editor):

```
R> baz <- cbind(foo,bar)
R> baz
     foo  bar
[1,] 1.1  2.0
```

```
[2,] 2.0  2.2
[3,] 3.5 -1.3
[4,] 3.9  0.0
[5,] 4.2  0.2
R> plot(baz)
```

The `plot` function is one of R's versatile *generic* functions. It works differently for different objects and allows users to define their own methods for handling objects (including user-defined object classes). Technically, R internally identifies the version of the `plot` command that you've just used as `plot.default`. The help file `?plot.default` provides additional details on this coordinate-based style of plotting.

7.2 Visual Controls and Graphical Parameters

You can supply a wide range of arguments to the `plot` function (or other plotting functions, such as those in Section 7.3) to control various visual features of the result. These arguments, most of which are referred to as *graphical parameters*, invoke simple visual enhancements, like coloring the points or controlling line types. Others are used for plot annotation, such as adding axis titles and labels. More technical arguments can control aspects of the graphics device and the coordinate systems underpinning the plot. The following is a list of some of the most common graphical controls; I'll briefly discuss each of these in turn in the following sections:

`type` Tells R how to plot the supplied coordinates (for example, as stand-alone points or joined by lines or both dots and lines)

`main`, `xlab`, `ylab` Provides options to include plot title, the horizontal axis label, and the vertical axis label, respectively

`col` Chooses color (or colors) to use for plotting points and lines

`pch` (point character) Selects which character to use for plotting individual points

`cex` **(character expansion)** Controls the size of plotted point characters

`lty` **(line type)** Specifies the type of line to use to connect the points (for example, solid, dotted, or dashed)

`lwd` **(line width)** Controls the thickness of plotted lines

`xlim`, `ylim` **(x-limits, y-limits)** Provides limits for the horizontal and vertical ranges (respectively) of the plotting region

`asp` **(aspect ratio)** Locks the per-unit size of the y-axis coordinates relative to the x-axis coordinates in terms of some multiplicative value

For further detail on each of these arguments, you can inspect the help files for the relevant lower-level functions that are typically called internally when a plot is created. The arguments `type`, `main`, `xlab`, and `ylab` are found under the `plot.default` or `title` functions (execute `?plot.default` or `?title` at the prompt to access the documentation); `col`, `pch`, `cex`, `lty`, and `lwd` are graphical parameters attributed to the global `par` function (execute `?par` at the prompt); and `xlim`, `ylim`, and `asp` can be found under the `plot.window` function (execute `?plot.window` at the prompt). In particular, you'll encounter the `par` and `title` functions in Chapter 23, where you'll explicitly use them to exert control over more advanced features of plotting in R.

7.2.1 Automatic Plot Types

By default, the `plot` function plots individual points, as Figure 7-1 shows. Other plot types have a different appearance. To control the plot type, specify a single character-valued option for the argument `type`.

For example, in many cases it makes sense to show lines connecting each coordinate, such as when plotting time series data. For this, specify plot type `"l"`. Using `foo` and `bar` from Section 7.1, the following code

```
R> plot(foo,bar,type="l")
```

produces the plot in the left panel of Figure 7-2.

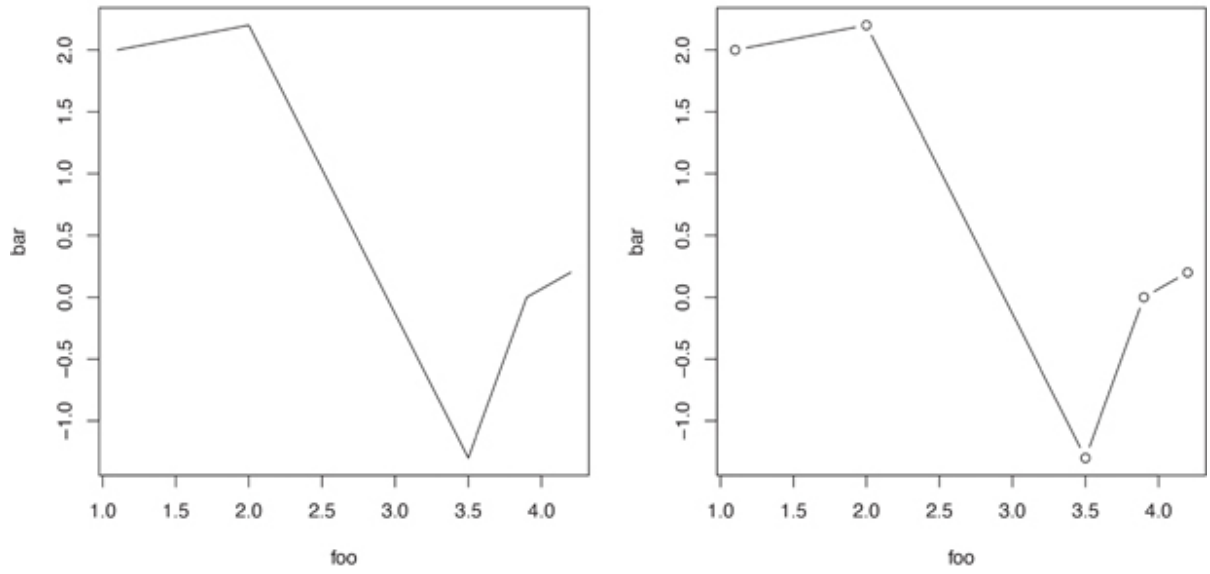


Figure 7-2: A line plot produced using five adjoined coordinates, setting `type="l"` *(left) or* `type="b"` *(right)*

You can interpret the default value for `type`, `"p"`, as "points only." This was used for the plot in Figure 7-1. In this last example, on the other hand, you set `type="l"` (meaning "lines only"). Other options include `"b"` for both points *and* lines (see the right panel of Figure 7-2) and `"o"` for overplotting the points with lines (this eliminates the gaps between points and lines visible for `type="b"`). The option `type="n"` results in no points or lines plotted, creating an empty plot, which can be useful for complicated plots that must be constructed in steps.

7.2.2 Title and Axis Labels

By default, a basic plot doesn't have a main title, and its axes are labeled with the names of the vectors being plotted. To make the plotted data easier to interpret, add a main title and more descriptive axis labels by supplying text as character strings to `main` for a title, `xlab` for the x-axis label, and `ylab` for the y-axis label. These strings may include escape sequences (see Section 4.2.3). The following code

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="x axis label",
        ylab="location y")
R> plot(foo,bar,type="b",main="My lovely plot\ntitle on two lines",xlab="",
        ylab="")
```

produces the plots in Figure 7-3.

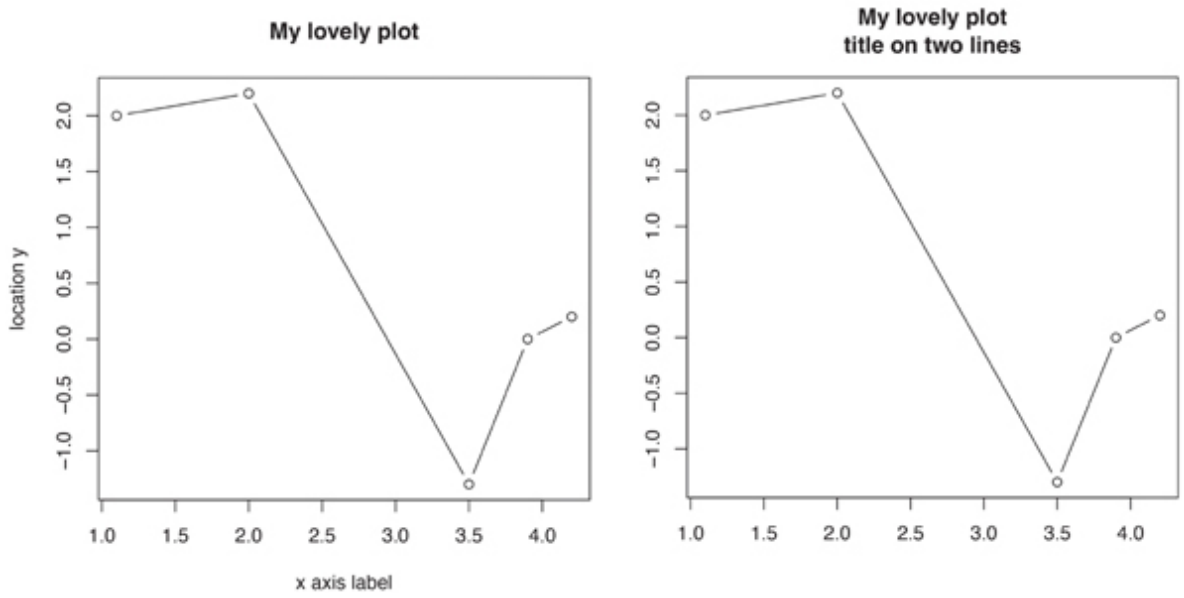


Figure 7-3: Two examples of plots with axis labels and titles

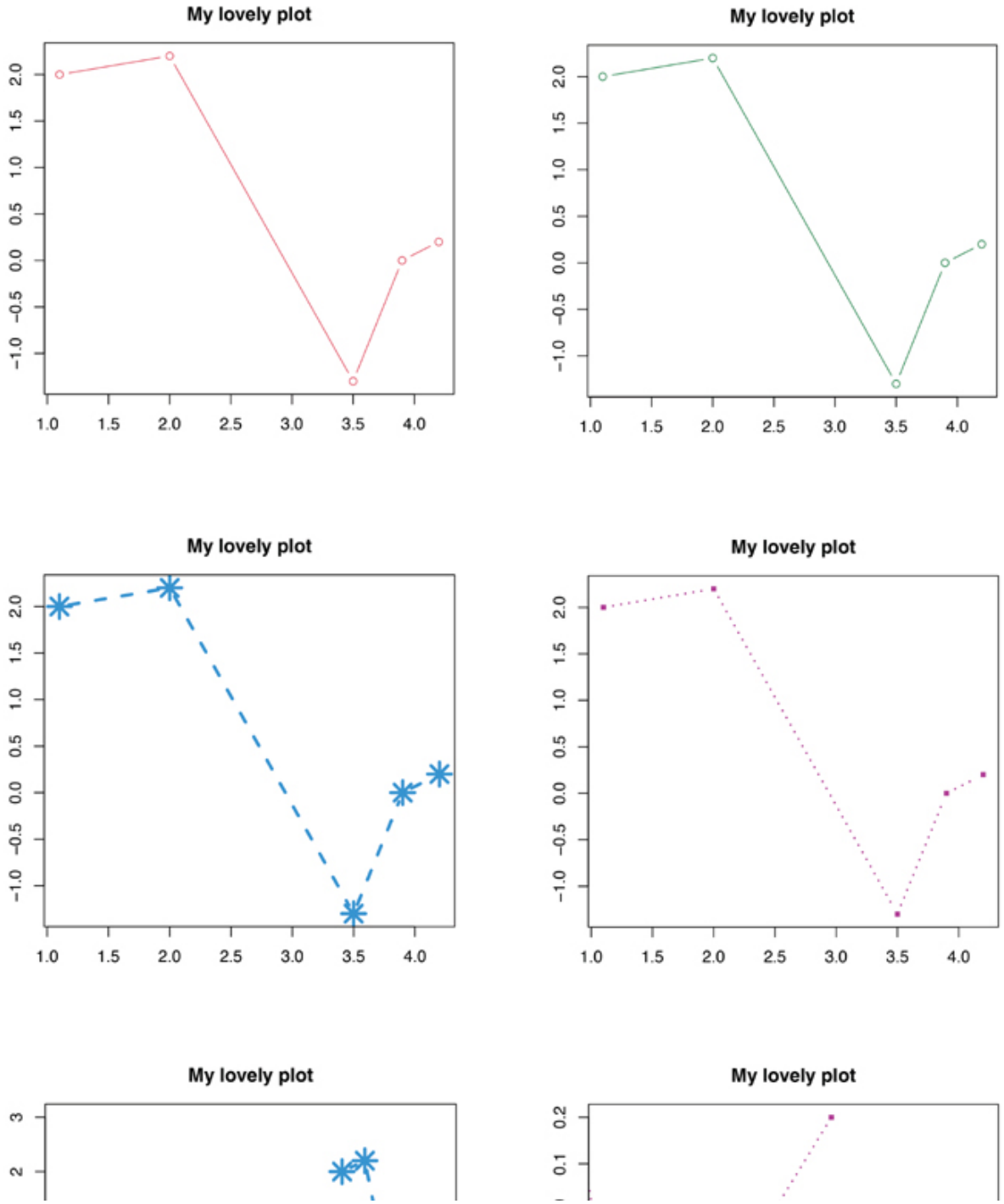
In the second plot, note how the new line escape sequence splits the title into two lines. In that plot, `xlab` and `ylab` are also set to the empty string `""` to prevent R from labeling the axes with the names of the *x* and *y* vectors.

7.2.3 Color

Adding color to a graph is far from just an aesthetic consideration. Color can make data much clearer—for example, by distinguishing factor levels or emphasizing important numeric limits. You can set colors with the `col` parameter in a number of ways. The simplest options are to use an integer selector or a character string. R recognizes a number of uniquely named colors, which you can see by entering `colors()` at the prompt. The default color is integer `1`, which corresponds to the character string `"black"`. The following code

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col=2)
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col="seagreen4")
```

creates two examples of colored graphs, as the top row of Figure 7-4 shows. (The other four examples in this figure are created in subsequent sections.)



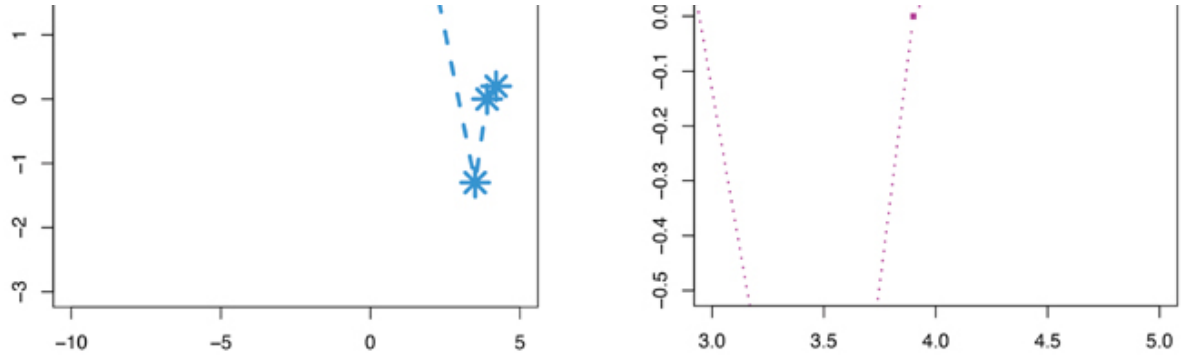


Figure 7-4: Experimenting with basic R plotting. Top row: Two examples of colored plots with `col=2` (left) and `col="seagreen4"` (right). Middle row: Two further examples making use of `pch`, `lty`, `cex`, and `lwd`. Bottom row: Setting plotting region limits `xlim=c(-10,5)`, `ylim=c(-3,3)` (left) and `xlim=c(3,5)`, `ylim=c(-0.5,0.2)` (right).

There are eight possible integer values (as shown in the leftmost plot in Figure 7-5) and over 650 character strings to specify color. But you aren't limited to these options, as you can also specify colors by using RGB (red, green, and blue) levels and by creating your own palettes. I'll talk more about these options in Chapter 25.

7.2.4 Line and Point Appearances

Use `pch` to alter the appearance of the plotted points and `lty` to alter the lines. The `pch` parameter controls the character you use to plot individual data points. You can specify a single character to use for each point, or you can specify a value between `1` and `25` (inclusive). The middle plot of Figure 7-5 shows the symbols corresponding to each integer. The `lty` parameter, which affects the type of line drawn, can take the values `1` through `6`. The rightmost plot of Figure 7-5 shows these options.

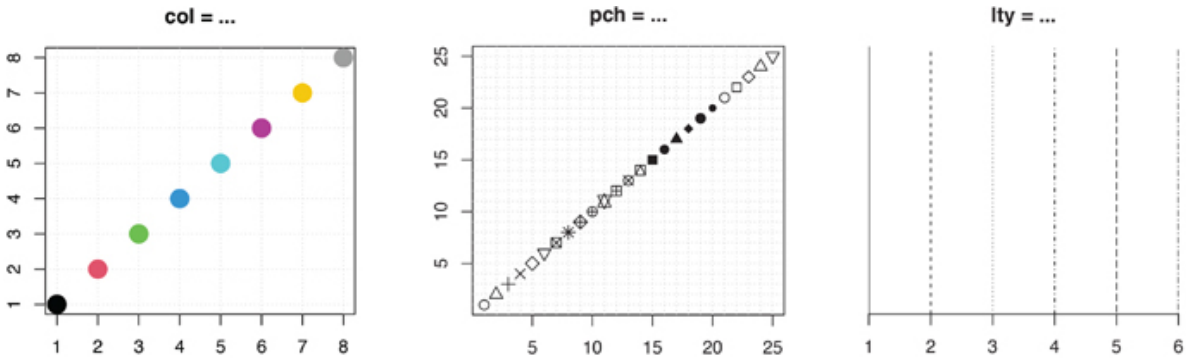


Figure 7-5: Some reference plots giving the results of possible integer options of `col` (left), `pch` (middle), and `lty` (right)

You can also control the size of plotted points using `cex` and the thickness of lines using `lwd`. The default size and thickness for both is `1`. To request half-size points, for example, specify `cex=0.5`; to specify double-thick lines, use `lwd=2`.

The following two lines

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=4,pch=8,lty=2,cex=2.3,lwd=3.3)
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=6,pch=15,lty=3,cex=0.7,lwd=2)
```

produce the two plots in the middle row of Figure 7-4, showing off `pch`, `lty`, `cex`, and `lwd`.

7.2.5 Custom Region Limits

As you can see in the plots of `foo` and `bar`, by default R sets the range of each axis by using the range of the supplied *x* and *y* values (plus a small constant to pad a little area around the outermost points). But you might need more space than this to, for example, annotate individual points, add a legend, or plot additional points that fall outside the original ranges (as you'll see in Section 7.3). You can set custom plotting area limits using `xlim` and `ylim`. Both parameters require a numeric vector of length 2, provided as `c(lower,upper)`.

Consider the plots in the bottom row of Figure 7-4, created with the following two commands:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=4,pch=8,lty=2,cex=2.3,lwd=3.3,xlim=c(-10,5),ylim=c(-3,3))
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=6,pch=15,lty=3,cex=0.7,lwd=2,xlim=c(3,5),ylim=c(-0.5,0.2))
```

These plots are exactly the same as the two in the middle row, except for one important difference. In the bottom-left plot of Figure 7-4, the x-and y-axes are set to be much wider than the observed data, and the

plot on the right restricts the plotting window to display only a portion of the data.

7.2.6 Aspect Ratio

By default, coordinate-based plots in R do not enforce the requirement that one unit in the x-axis direction must be represented by the same visual distance on the plot as one unit in the y-axis direction. In general, this is necessary and desirable behavior: If a single unit in both x-and y-directions were forced to take up the same planar space, then you wouldn't be able to produce a legible plot of two variables measured on different scales or those with very different levels of relative variability.

By extension, R permits the appearance of the plot itself to be sensitive to the specific rectangular shape of the plotting region, which the user can change interactively by resizing the plot device window with their mouse. Consider the two plots in Figure 7-6, one of which shows the five hypothetical points in `foo` and `bar` in a graphics window sized to be tall and thin, while the other shows the exact same points in a window sized to be short and wide. Single units in the *x*- and *y*-directions are not required to span the same planar distance and as such, axis scales are stretched or squashed to accommodate the size of the window.

To reproduce the appearances in Figure 7-6, use your mouse to resize the plot window after calling the following line:

```
R> plot(foo,bar,pch=19)
```

Given this behavior, the choice on the final size/shape of the plotting region simply comes down to whatever the user deems to result in the clearest, most legible plot. There are, however, certain situations where the relative spatial positioning of plotted points—their overall appearance—is directly relevant to interpretation. This usually crops up when the points themselves have some physical interpretation, such as in creating plots of spatially indexed observations (like maps of the locations of some geographical phenomenon). This demands control

over how *x*- and *y*-units are treated, which is determined by a given plot's *aspect ratio*.

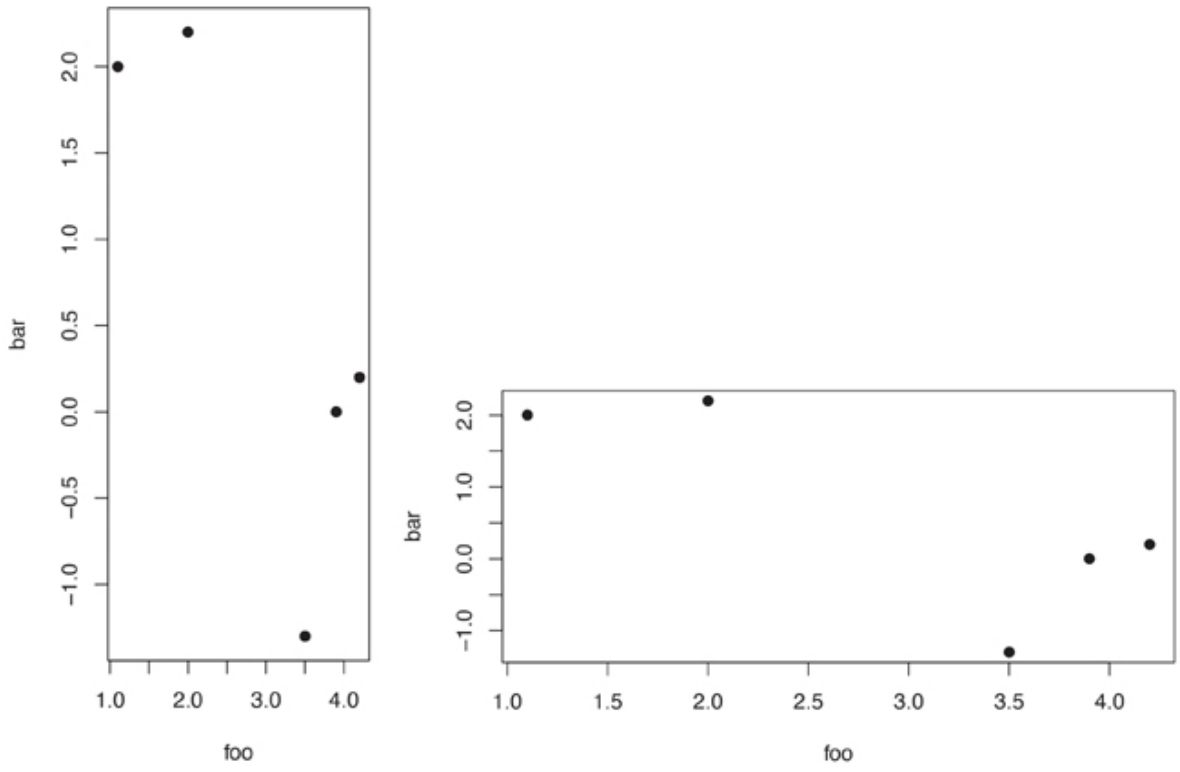


Figure 7-6: Contrasting the appearance of a default, coordinate-based plot of five hypothetical points in differently sized/shaped plotting windows

In base R plots, the aspect ratio is handled via the `asp` argument. Its default value, `NA`, results in the aforementioned behavior where the scales are left "unlocked." The `asp` argument can alternatively take a positive number to be interpreted as the "per-unit value" that a unit on the y-axis possesses, relative to a single unit on the x-axis. Setting `asp=1` will enforce a 1:1 ratio, locking the axis scales to ensure the size of a single vertical unit on the y-axis is always the same, in terms of visible distance, as a single horizontal unit on the x-axis. The final appearance of the plotted points will therefore always respect this chosen aspect ratio, regardless of the size of the actual plotting window.

To demonstrate, execute the following line

```
R> plot(foo,bar,pch=19,asp=1)
```

and resize the plot window in the same manner as in Figure 7-6. Figure 7-7 shows the results.

Notice that resizing the plot window now does not impact the overall appearance of the points.

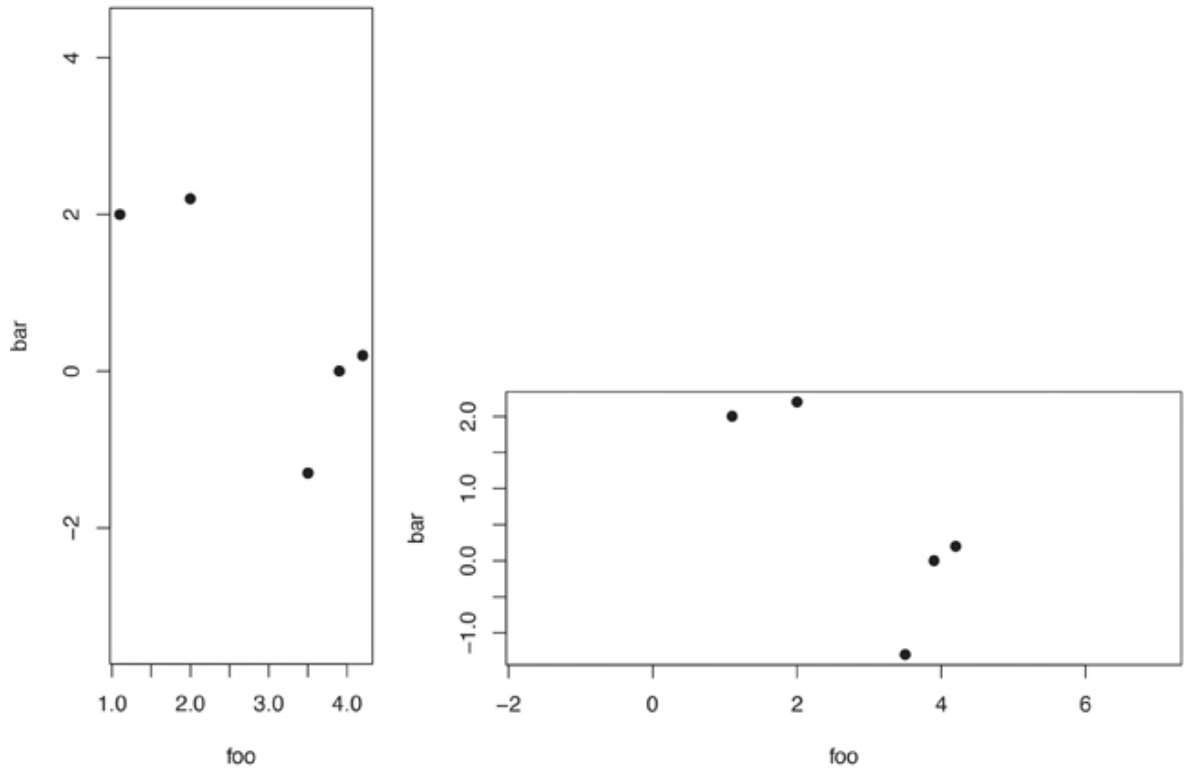


Figure 7-7: An illustration of locking the aspect ratio in a 1:1 fashion. The appearance of five hypothetical observations is now insensitive to the chosen rectangular shape of the plotting region.

Situations in which you might want to set an aspect ratio to be some value other than 1 include dealing with appropriate display of distances in various geographic projections (for example, where coordinates have been recorded in terms of latitude and longitude). This topic is explored a little further in Chapter 14 and is also encountered tangentially in a handful of examples throughout Part V.

7.3 Adding Points, Lines, and Text to an Existing Plot

By default, each call to `plot` refreshes the active graphics device, creating a new plotting region. But this isn't always desired; to build more complicated plots, it's easiest to start with an empty plotting region and progressively add any required points, lines, text, and legends to this canvas. Here are some useful, ready-to-use functions in R that add to a plot without refreshing or clearing the window:

`points`	Adds points
`lines`, **`abline`**, **`segments`**	Adds lines
`text`	Writes text
`arrows`	Adds arrows
`legend`	Adds a legend

The syntax for calling and setting parameters for these functions is the same as `plot`. The best way to see how these work is through an extended example, which I'll base on some hypothetical data made up of 20 (*x*, *y*) locations:

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
          -12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
```

Using these data, you'll build up the plot in Figure 7-8 (note that you may need to manually enlarge your graphics device and replot to ensure the legend doesn't overlap other features of the image). In plotting, "keep it clear and simple" is a generally accepted rule. Figure 7-8 is an exception for the sake of demonstrating the R commands used.

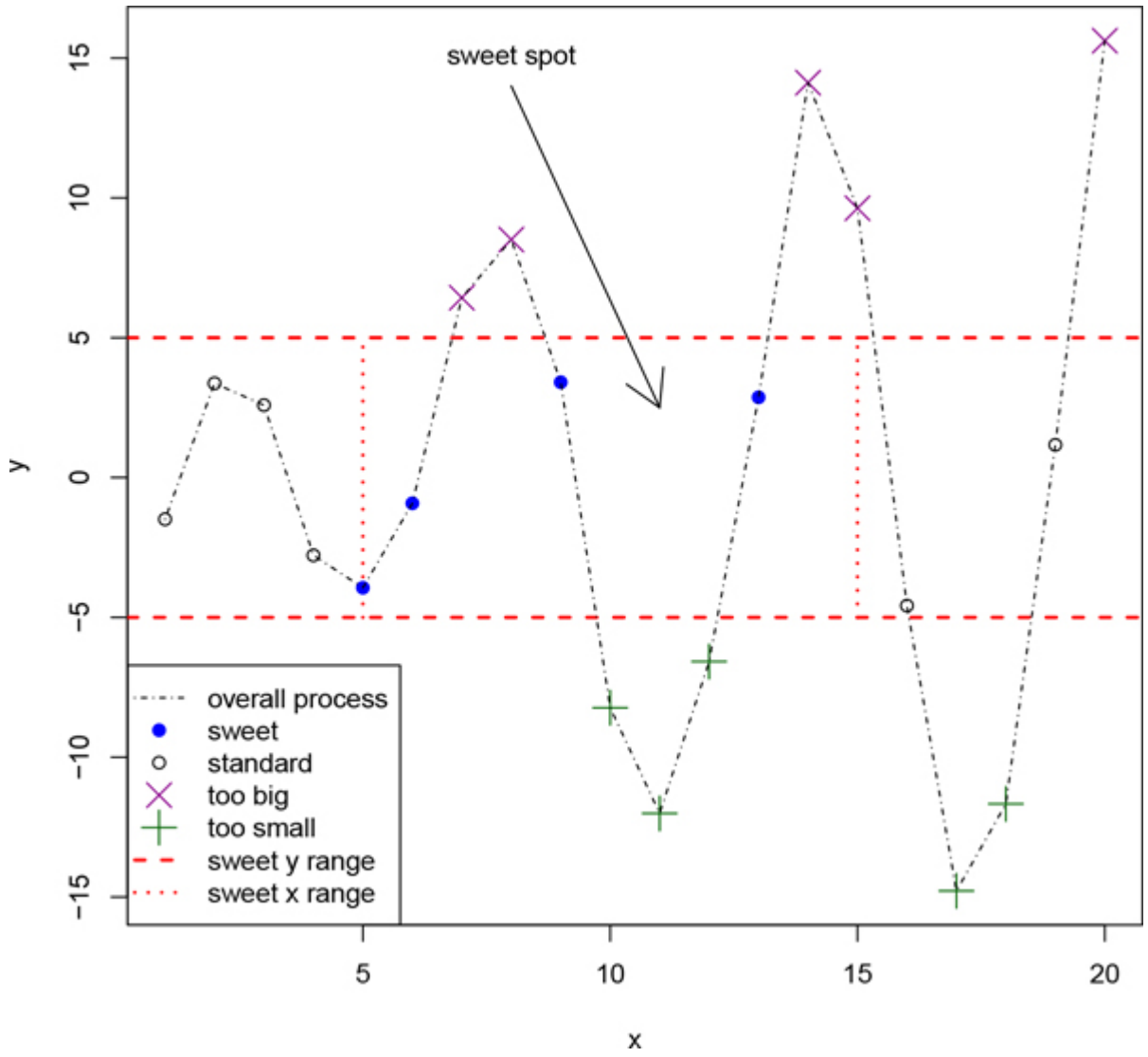
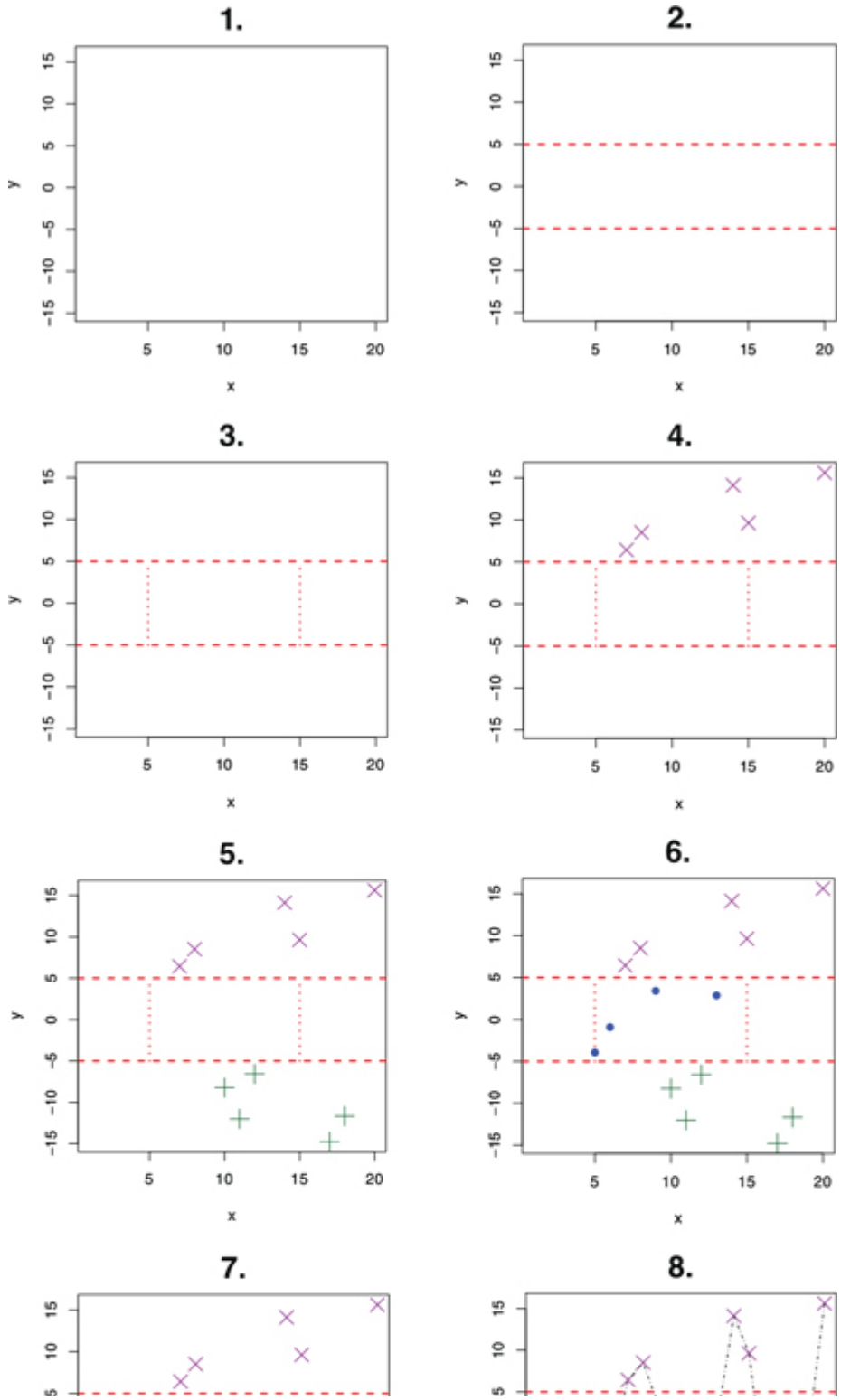


Figure 7-8: An elaborate final plot of some hypothetical data

In Figure 7-8, the data points display differently according to their *x* and *y* locations, depending on their relation to the "sweet spot" pointed out in the figure. Points with a *y* value greater than 5 are marked with a purple ×, and points with a *y* value less than –5 are marked with a green +. Points between these two *y* values but still outside of the sweet spot are marked with a °. Finally, points in the sweet spot (with *x* between 5 and 15 *and* with *y* between –5 and 5) are marked as a blue •. Red horizontal and vertical lines delineate the sweet spot, which is labeled with an arrow, and there's also a legend.

I used 10 lines of code to build this plot in its entirety (plus an additional line to add the legend). Figure 7-9 shows the plot, as it looks at each step.



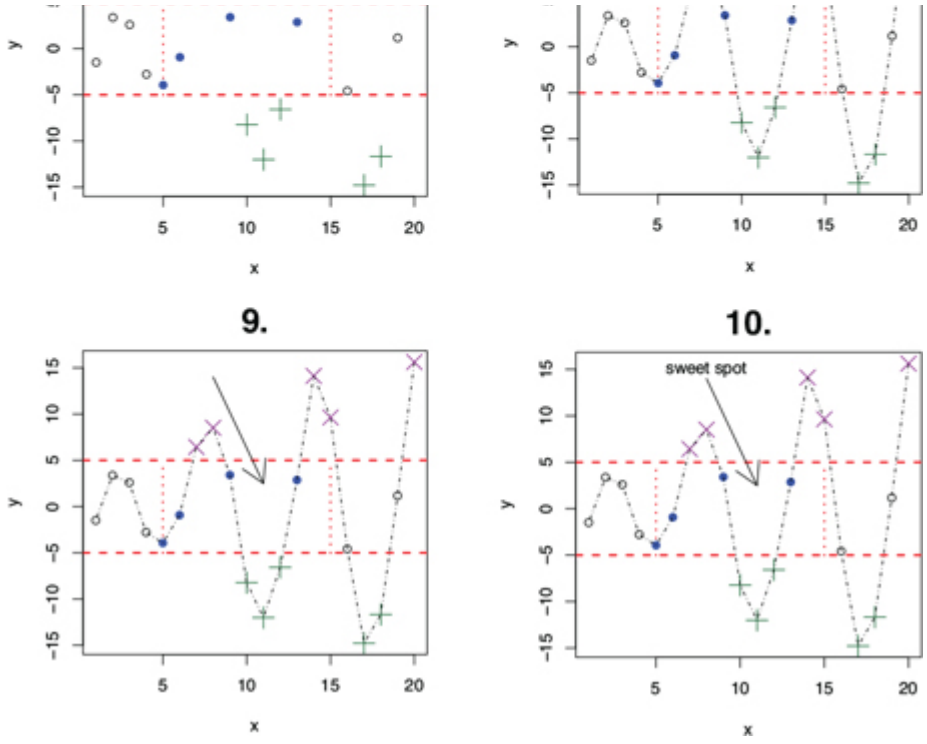


Figure 7-9: Building the final plot given in Figure 7-8. The plots 1 through 10 correspond to the itemized lines of code in the text.

The lines of code I used to build this plot are as follows:

1. First, create the empty plotting region where you can add points and draw lines. This first line tells R to plot the data in `x` and `y`, though the option `type` is set to `"n"`. This opens or refreshes the graphics device and sets the axes to the appropriate lengths with labels and axes, but it doesn't plot any points or lines (see Section 7.2):

```
R> plot(x,y,type="n",main="")
```

2. The `abline` function is a simple way to add straight lines spanning a plot. Specify the line (or lines) with *slope* and *intercept* values (see the later discussions on regression in Chapter 20). You can also simply add horizontal or vertical lines. This line of code adds two separate horizontal lines, one at $y = -5$ and the other at $y = 5$, using `h=c(-5,5)`. The three parameters (see Section 7.2) make these

two lines red, dashed, and double-thick. Writing `v=c(-5,5)` would instead draw vertical lines at $x = -5$ and $x = 5$:

```
R> abline(h=c(-5,5),col="red",lty=2,lwd=2)
```

3. The third line of code adds shorter vertical lines between the horizontal ones drawn in step 2 to form a box. Use `segments` for this, not `abline`, since you don't want these lines to span the entire plotting region. The `segments` command takes a "from" coordinate (given as `x0` and `y0`) and a "to" coordinate (`x1` and `y1`) and draws the corresponding line. The vector-oriented behavior of R matches up the two sets of "from" and "to" coordinates. Both lines are red and dotted and have double-thickness. Here, I supply vectors of length equal to 2 to `x0`, `y0`, `x1` and `y1`; the first segment uses the first entry in each of these arguments, and the second segment corresponds to the second:

```
R> segments(x0=c(5,15),y0=c(-5,-5),x1=c(5,15),y1=c(5,5),col="red",
           lty=3,lwd=2)
```

4. Use `points` to begin adding specific coordinates from `x` and `y` to the plot. Just like `plot`, `points` takes two vectors of equal lengths with x and y values. In this case, you want points plotted differently according to their location, so you use logical vector subsetting (see Section 4.1.5) to identify and extract elements of `x` and `y` where the y value is greater than or equal to 5. These (and only these) points are added as purple × symbols and are enlarged by a factor of 2 with `cex`:

```
R> points(x[y>=5],y[y>=5],pch=4,col="darkmagenta",cex=2)
```

5. The fifth line of code is much like the fourth; this time it extracts the coordinates where y values are less than or equal to –5. Use a + point character and set the color to dark green:

```
R> points(x[y<=-5],y[y<=-5],pch=3,col="darkgreen",cex=2)
```

6. The sixth step adds the blue "sweet spot" points, which identify with `(x>=5&x<=15)&(y>-5&y<5)`. This slightly more complicated set of conditions extracts the points whose *x* location lies between 5 and 15 (inclusive) AND whose *y* location lies between –5 and 5 (exclusive). Note that this line uses the "short" form of the logical operator `&` throughout since you want element-wise comparisons here (see Section 4.1.3):

```
R> points(x[(x>=5&x<=15)&(y>-5&y<5)],y[(x>=5&x<=15)&(y>-5&y<5)],
          pch=19,col="blue")
```

7. This next command identifies the remaining points in the dataset (with an *x* value that's either less than 5 OR greater than 15 AND a *y* value between –5 and 5). Since you don't specify any graphical parameters, these points plot with the default black ∘:

```
R> points(x[(x<5|x>15)&(y>-5&y<5)],y[(x<5|x>15)&(y>-5&y<5)])
```

8. Use `lines` to draw lines connecting the coordinates in `x` and `y`. Here, you also set `lty` to `4`, which draws a dash-dot-dash style line:

```
R> lines(x,y,lty=4)
```

9. The ninth line of code adds the arrow pointing to the sweet spot. Use the function `arrows` just like `segments`, where you provide a "from" coordinate (`x0`, `y0`) and a "to" coordinate (`x1`, `y1`). By default, the head of the arrow is located at the "to" coordinate, though you can alter this (and other options, such as the angle and length of the head) using optional arguments as the help file `?arrows` describes:

```
R> arrows(x0=8,y0=14,x1=11,y1=2.5)
```

10. The tenth line prints a label on the plot at the top of the arrow. As per the default behavior of `text`, the string supplied as `labels` is *centered* on the coordinates provided with the arguments `x` and `y`:

```
R> text(x=8,y=15,labels="sweet spot")
```

As a finishing touch, add the legend with the `legend` function, which gives you the final product from Figure 7-8:

```
R> legend("bottomleft",
          legend=c("overall process","sweet","standard",
                   "too big","too small","sweet y range","sweet x range"),
          pch=c(NA,19,1,4,3,NA,NA),lty=c(4,NA,NA,NA,NA,2,3),
          col=c("black","blue","black","darkmagenta","darkgreen","red","red"),
          lwd=c(1,NA,NA,NA,NA,2,2),pt.cex=c(NA,1,1,2,2,NA,NA))
```

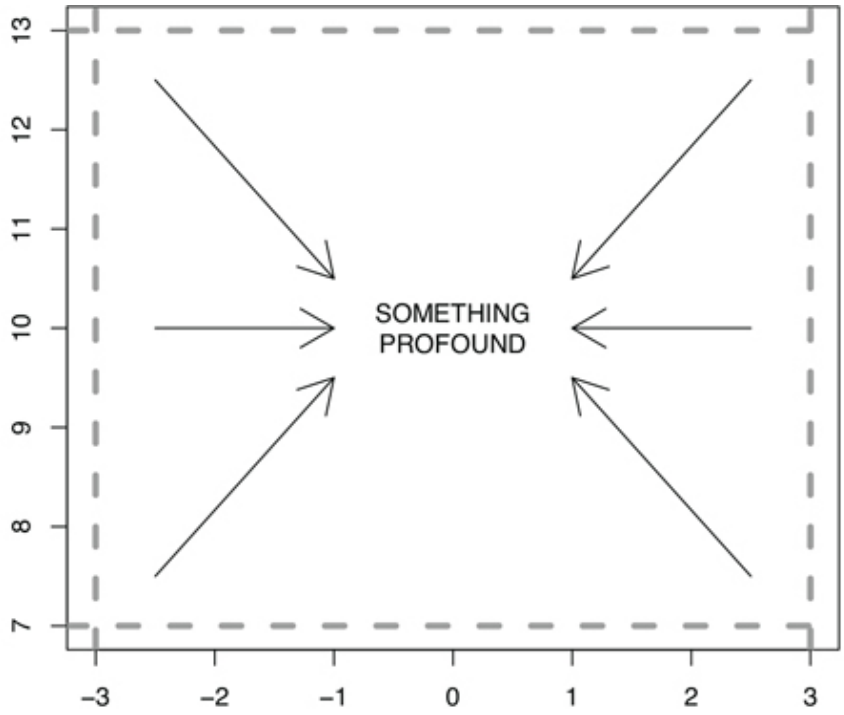
The first argument sets where the legend should be placed. There are various ways to do this (including setting exact x-and y-coordinates), but it often suffices to pick a corner using one of the four following character strings: `"topleft"`, `"topright"`, `"bottomleft"`, or `"bottomright"`. Next, you supply the labels as a vector of character strings to the `legend` argument. Then you need to supply the remaining argument values in vectors of the same length so that the right elements match up with each label.

For example, for the first label (`"overall process"`), you want a line of type 4 with default thickness and color. So, in the first positions of the remaining argument vectors, set `pch=NA`, `lty=4`, `col="black"`, `lwd=1`, and `pt.cex=NA` (all of these are default values, except for `lty`). Here, `pt.cex` simply refers to the `cex` parameter when calling `points` (using just `cex` in `legend` expands the text used, not the points).

Note that you have to fill in some elements in these vectors with `NA` when you don't want to set the corresponding graphical parameter. This preserves the equal lengths of the vectors supplied so R can track which parameter values correspond to each particular reference. As you work through this book, you'll see plenty more examples using `legend`.

Exercise 7.1

a. As closely as you can, re-create the following plot:



b. With the following data, create a plot of weight on the x-axis and height on the y-axis. Use different point characters or colors to distinguish between males and females and provide a matching legend. Label the axes and give the plot a title.

Weight (kg)	Height (cm)	Sex
55	161	Female
85	185	Male
75	174	Male
42	154	Female
93	188	Male
63	178	Male

Weight (kg)	Height (cm)	Sex
58	170	Female
75	167	Male
89	181	Male
67	178	Female

7.4 The ggplot2 Package

This chapter so far has shown off R's built-in graphical tools (which we often call *base R graphics* or *traditional R graphics*). Now let's look at another important suite of graphical tools: `ggplot2`, a prominent contributed package originally developed by Hadley Wickham (Wickham 2009).

Available on CRAN like any other contributed package, `ggplot2` offers unique alternatives to the standard plotting procedures in R. The *gg* stands for *grammar of graphics*, which is a particular approach to graphical production described by Wilkinson (2005). In following this approach, `ggplot2` standardizes the production of different plot and graph types and streamlines some of the more fiddly aspects of adding to existing plots (such as including a legend). It's based on the concept of defining and manipulating *layers*. For now, let's learn the elementary behavior of `ggplot2` using the same hypothetical data examples from Sections 7.1–7.3. You'll get familiar with the basic plotting declaration `ggplot` and how it differs from the generic `plot` function used earlier. I'll return to the topic of `ggplot2` in Chapter 14, and you'll investigate even more advanced abilities in Chapter 24.

7.4.1 Plotting a Toy Example

First, install the `ggplot2` package by downloading it manually or entering `install.packages("ggplot2")` at the prompt (see Section A.2.3). Then, load the package with the following:

```
R> library("ggplot2")
```

Next, let's go back to the five data points originally stored in Section 7.1as `foo` and `bar`, and store these together as a data frame object:

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> fb <- data.frame(foo,bar)
R> fb
  foo  bar
1 1.1  2.0
2 2.0  2.2
3 3.5 -1.3
4 3.9  0.0
5 4.2  0.2
```

We approach `ggplot2` graphics in a particular way according to the philosophy of the grammar of graphics. We must first provide the `ggplot` function with the data to be visualized and declare any overarching features of the desired plot. Then, we add layers to this defined scheme to actually display the data. Consider `ggplot2`'s version of Figure 7-1, the left image of Figure 7-10, which you obtain with the following code:

```
R> ggplot(data=fb,mapping=aes(x=foo,y=bar)) + geom_point()
```

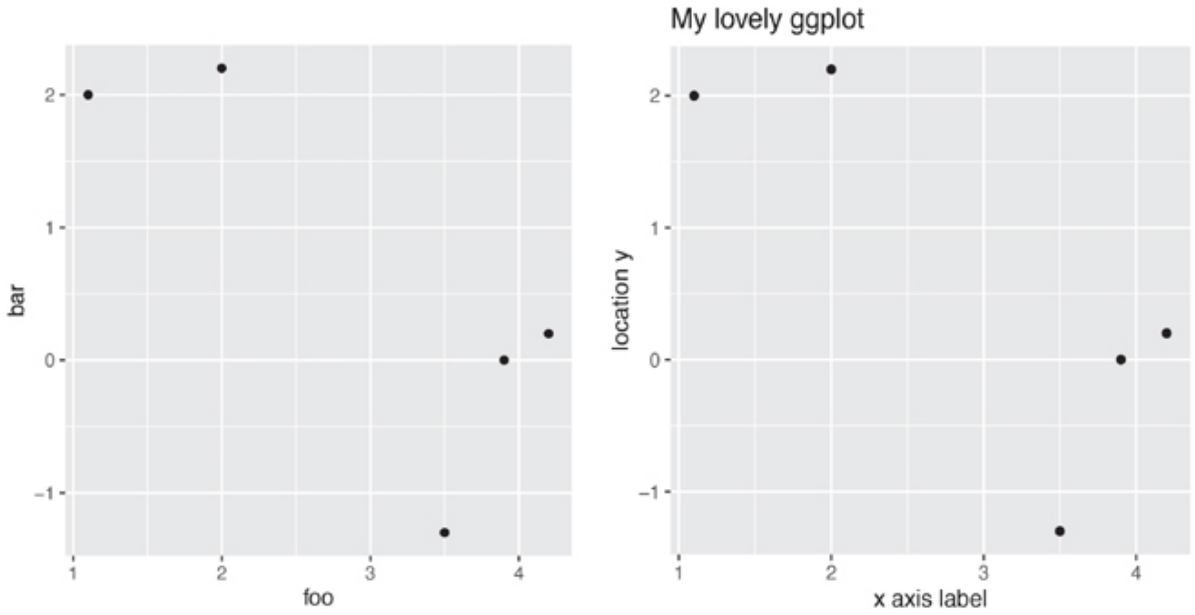


Figure 7-10: Five plotted points using `ggplot2`'s default behavior for the `qplot` function (left) and with title and axis labels added (right)

After the `data` argument being supplied the data of interest, you use the `mapping` argument to define key properties of the plot. The expression `aes(x=foo,y=bar)` tells `ggplot` that we want to map the `foo` variable of the supplied data frame to the x-axis and the `bar` variable to the y-axis. We call this an *aesthetic mapping*, and it's always performed using the package's `aes` function.

Then, we add (`+`) a layer using a geometric modifier (geom). In the `ggplot2` package, *geoms* are functions prefaced by `geom_`; the call to `geom_point()` instructs the data to be displayed, following the aforementioned aesthetic map, as points. If you simply execute the `ggplot` function call in the preceding line (that is, you don't add the point geom), the plot opens with the correct axis limits and labels (on the left of Figure 7-10), but shows no data points.

You add a title and axis labels by adding special layers to the plot. By executing the next line, you obtain the plot on the right of Figure 7-10:

```
R> ggplot(data=fb,mapping=aes(x=foo,y=bar)) + geom_point() +
       ggtitle("My lovely ggplot") + xlab("x axis label") + ylab("location y")
```

Here, the `xlab` and `ylab` functions intentionally have names that match the typical axis label arguments in base R plots, as in Section 7.2. The main title is supplied via `ggtitle` (you can also use the `labs` function for additional customization).

Beyond the obvious syntactical differences, there's a fundamental difference between how `ggplot2` and base R graphics deal with plots. Constructing plots using the built-in, base R graphics tools is essentially a live, step-by-step process. This was particularly noticeable in Section 7.3, where you treated the graphics device as an active canvas where you added points, lines, and other features one by one. By contrast, `ggplot2` plots are stored as objects, which means they have an underlying, static representation until you *change* the object—what you essentially visualize with `ggplot` and its geoms is the `print`ed object at any given time. To highlight this, enter the following code:

```
R> baz <- plot(foo,bar)
R> baz
NULL
R> qux <- ggplot(data=fb,mapping=aes(x=foo,y=bar)) + geom_point()
R> qux
```

The first assignment uses the built-in `plot` function. When you run that line of code, the plot in Figure 7-1 pops up instantly. Since nothing is actually stored in the workspace, printing the supposed object `baz` yields the empty `NULL` value. On the other hand, it makes sense to store the `ggplot` content (stored as the object `qux` here). This time, when you perform the assignment, *no* plot automatically displays. The graphic, which matches Figure 7-10, displays only when you enter `qux` at the prompt, which invokes the `print` method for that object. This may seem like a minor point, but the fact that you can "pre-store" a plot this way before displaying it opens up new ways to modify or enhance plots before displaying them (as you'll see in a moment), and it can be a distinct advantage over base R graphics.

7.4.2 Setting Appearance Constants with Geoms

For a toy example like the one in the preceding section, you might reasonably consider the `ggplot2` code to be overly complicated, especially compared to the base R code. But when it comes to real-world data visualization, the layer- and object-based construction of these plots can be quite beneficial, as you'll see in later examples in this book.

For now, let's continue learning some `ggplot2` syntax using the same toy example data. As you've just seen, to add and customize points and lines in a `ggplot2` graphic, you alter the object itself. Say you want to connect the five points in the plot of `foo` and `bar` with a line, as you did in Section 7.1. To achieve this, you merely add another layer to the plot from earlier using the appropriate line geom:

```
R> ggplot(data=fb,mapping=aes(x=foo,y=bar)) + geom_point() + geom_line()
```

The resulting plot is on the left of Figure 7-11.

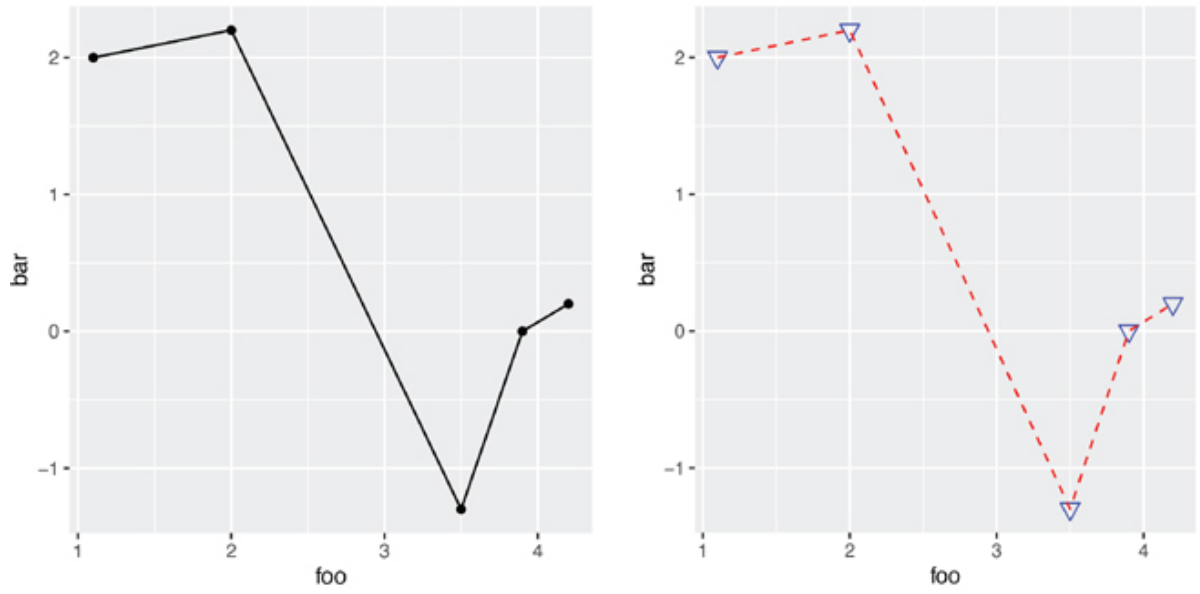


Figure 7-11: Two simple plots that use geometric modifiers to alter the appearance of a `ggplot` object. Left: Adding points and lines using default settings. Right: Using the geoms to affect point character, size, and color, as well as line type and color.

Here, you haven't supplied any arguments to either of the geom functions, which means they'll operate on the same data and mappings

as originally supplied to `ggplot`. To change the appearance of these features, specify optional arguments like so:

```
R> ggplot(data=fb,mapping=aes(x=foo,y=bar)) +
          geom_point(size=3,shape=6,color="blue") +
          geom_line(color="red",linetype=2)
```

Note that some of `ggplot2`'s argument names used here for things such as point characters and size (`shape` and `size`) are different from the base R graphics arguments (`pch` and `cex`). But `ggplot2` is actually compatible with many of the common graphical parameters used in R's standard `plot` function, so you can use those arguments here too if you prefer. For instance, setting `cex=3` and `pch=6` in `geom_point` in this example results in the same image.

The object-oriented nature of `ggplot2` graphics means tweaking a plot or experimenting with different visual features no longer requires you to rerun every plotting command each time you change something. This is facilitated by geoms. Say you like the line type on the right side of Figure 7-11 but want a different point character. To experiment, first store the graphical object with the lines only, and then use `geom_point` with that object to try different point styles:

```
R> myggplot <- ggplot(data=fb,mapping=aes(x=foo,y=bar)) +
                    geom_line(color="red",linetype=2)
R> myggplot + geom_point(size=3,shape=3,color="blue")
R> myggplot + geom_point(size=3,shape=7,color="blue")
```

The first line stores the "line-only" plot in `myqplot`, and the next lines manipulate the `myggplot` object with different point layers. The second and third lines produce the graphics on the left and right of Figure 7-12, respectively.

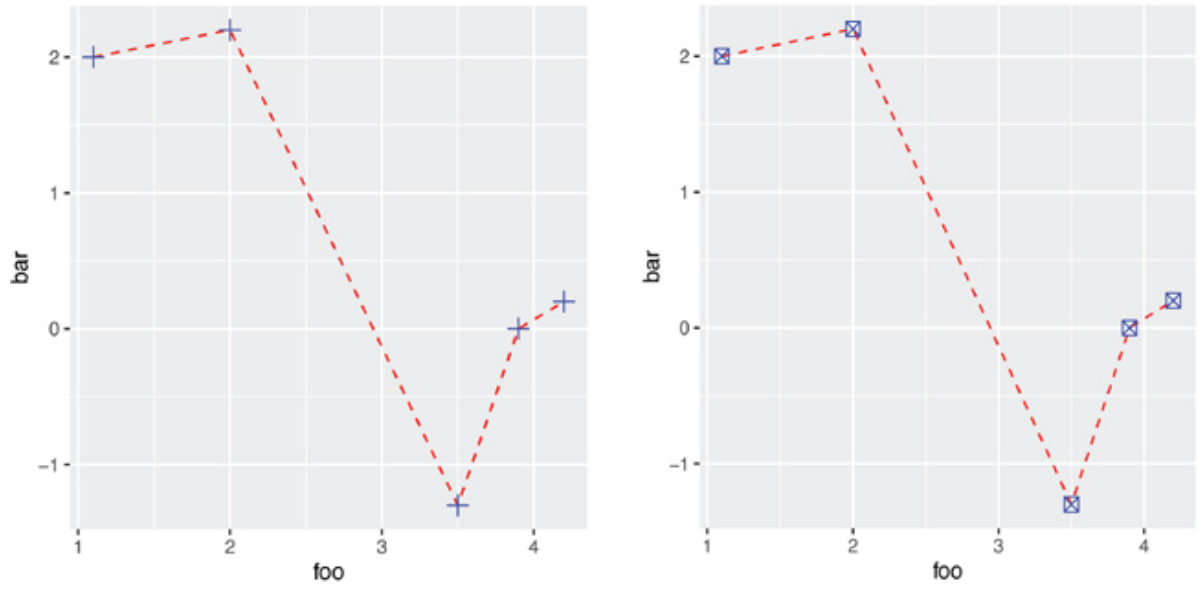


Figure 7-12: Using the object-oriented nature of `ggplot2` graphics to experiment with different point characters on an existing object

You can call a number of geometric modifiers using a function name beginning with `geom_` in `ggplot2`. To obtain a list, ensure the package is loaded and enter `??"geom_"` as a help search at the prompt.

7.4.3 Aesthetic Mapping with Geoms

Geoms and `ggplot2` also provide efficient ways to apply different styles to different subsets of your data. If you split a dataset into categories using a factor object and declare its aesthetic mapping, `ggplot2` automatically applies particular styles to different categories. This obviates much of the effort that goes into isolating subsets of data and plotting them separately (as you did in Section 7.3).

As usual, we can best illustrate this with an example. Let's return to the 20 observations you manually plotted, step-by-step, to produce the purposefully elaborate plot in Figure 7-8 and form a data frame to use:

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
          -12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
R> xydata <- data.frame(x,y)
```

In Section 7.3, you defined several categories that classified each observation as either "standard," "sweet," "too big," or "too small" based on their `x` and `y` values. Using those same classification rules, let's explicitly define a factor to correspond to `x` and `y`, and add this to our data frame:

```
R> ptype <- rep("too_big",length(x=x))
R> ptype[y<=-5] <- "too_small"
R> ptype[(x>=5&x<=15)&(y>-5&y<5)] <- "sweet"
R> ptype[(x<5|x>15)&(y>-5&y<5)] <- "standard"
R> ptype <- factor(x=ptype)
R> ptype
 [1] standard  standard  standard  standard  sweet     sweet     too_big
 [8] too_big   sweet     too_small too_small too_small sweet     too_big
[15] too_big   standard  too_small too_small standard  too_big
Levels: standard sweet too_big too_small
R> xydata$ptype <- ptype
```

Now you have a factor with 20 values identified by four levels; pretend the `xydata` object is a real observed dataset, as shown here:

```
R> str(xydata)
'data.frame':   20 obs. of  3 variables:
 $ x    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ y    : num  -1.49 3.37 2.59 -2.78 -3.94 -0.92 6.43 8.51 3.41 -8.23 ...
 $ ptype: Factor w/ 4 levels "standard","sweet",..: 1 1 1 1 2 2 3 3 2 4 ...
```

Use the `ptype` variable to tell `ggplot` how to map your aesthetics according to these categories:

```
R> gg <- ggplot(data=xydata,mapping=aes(x=x,y=y,color=ptype,shape=ptype))
```

Here, you set both the `color` and `shape` aesthetics to correspond to `ptype`. Executing the following line

```
R> gg + geom_point()
```

results in the plot on the left of Figure 7-13.

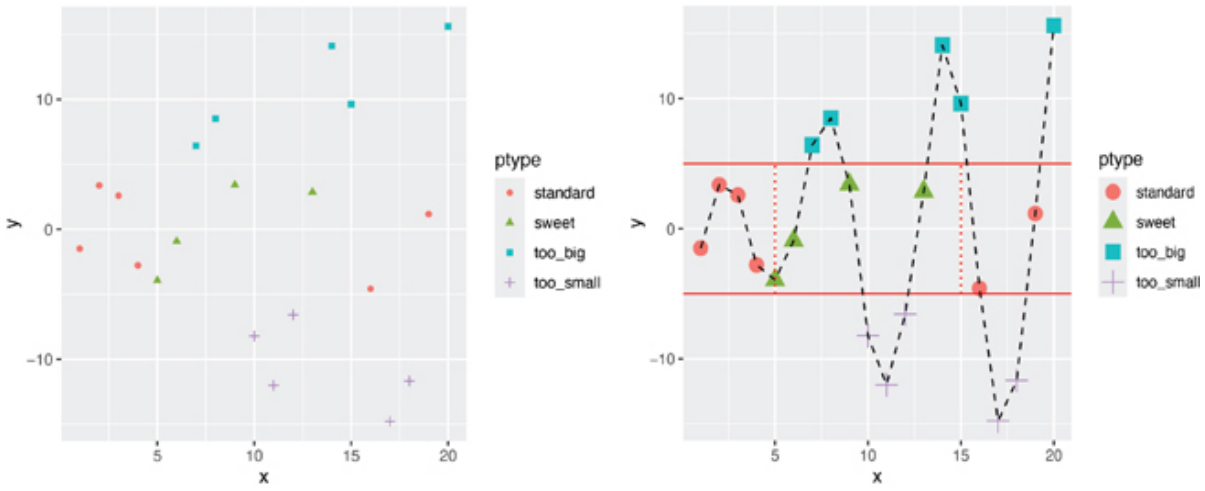


Figure 7-13: A demonstration of aesthetic mapping using `ggplot`. Left: A plot of 20 hypothetical observations in the `xydata` data frame, which maps point character and color to a categorical variable. Right: Augmenting the left plot using various geoms to override the default mappings and add further customization.

Let's replot these data using the same `ggplot` object along with a suite of geom modifications in order to get something more like Figure 7-8. Executing the following produces the plot on the right of Figure 7-13:

```
R> gg + geom_point(size=4) + geom_hline(yintercept=c(-5,5),color="red") +
      geom_line(mapping=aes(group=1),color="black",lty=2) +
      geom_segment(mapping=aes(x=5,y=-5,xend=5,yend=5),color="red",lty=3) +
      geom_segment(mapping=aes(x=15,y=-5,xend=15,yend=5),color="red",lty=3)
```

In the first addition, you supply `geom_point(size=4)` to increase the size of all the points on the graph. The use of `geom_hline` then draws two horizontal rules at specified `yintercept` locations on the vertical axis. In the geoms that follow, you add a line connecting all the points, plus horizontal and vertical lines to mark out the "sweet spot." For these last three additions to the `gg` object, you have to use `aes` to set alternate aesthetic mappings specific to those geoms. Let's look closer at what's going on.

Think of the mapping defined in the initial call to `ggplot` as applying *globally*, and any geom-specific modifications to that mapping as *local* changes. Since you used `ptype` for aesthetic mapping in the initial call to `ggplot`, by default all other geoms map to each category in the same way

unless you override that default mapping with `aes`. For example, when you call `geom_line` to connect all the points, if you stick with the default mapping instead of including `mapping=aes(group=1)`, this geom draws lines connecting points within each category. You'd see four separate dashed lines: one connecting all "standard" points, another connecting all "sweet" points, and so on. But that's not what you want here; you want a line that connects all of the points, from left to right. So, you tell `geom_line` to treat all the observations as one group—effectively ignoring `ptype` for just this feature of the plot—by entering `aes(group=1)`. Thereafter, note the use of `geom_segment` to draw the two vertical dotted line segments. This function operates much like the `segments` function in base R; here, you must redefine the mapping based on a "from" coordinate (arguments `x` and `y`) and a "to" coordinate (`xend` and `yend`) to achieve the desired effect.

Plotting in R, from base graphics to contributed packages like `ggplot2`, behaves much like we might expect given what we've seen of the language thus far. The element-wise matching in specifying both coordinates and design features allows you to create intricate plots with a handful of straight-forward and intuitive functions. Once you display a plot, you can save it as a file by selecting the graphics device and choosing File ▸ Save (or, if using the RStudio editor, by clicking the Export button above the plotting pane). However, you can also use R code to write plots to a file directly, as you'll see in Section 8.3.

The graphical capabilities explored in this section are merely the tip of the iceberg. Plotting in R is somewhat of an art form and comes with a learning curve; even the most experienced programmers frequently open help files when plotting to remind themselves of various features, particularly for functions with many optional arguments (like `legend`). You'll continue to develop and deploy your plotting skills from this point onward, focusing specifically on common data plots in Chapter 14 and on more advanced customization in Chapters 23 through 26.

Exercise 7.2

In Exercise 7.1 (b), you used base R graphics to plot some weight and height data, distinguishing males and females using different points or colors. Repeat this task using `ggplot2`.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`plot`	Create/display base R plot	Section 7.1, p. 146
`type`	Set plot type	Section 7.2.1, p. 148
`main`, `xlab`, `ylab`	Set axis labels	Section 7.2.2, p. 149
`col`	Set point/line color	Section 7.2.3, p. 149
`pch`, `cex`	Set point type/size	Section 7.2.4, p. 151
`lty`, `lwd`	Set line type/width	Section 7.2.4, p. 151
`xlim`, `ylim`	Set plot region limits	Section 7.2.5, p. 152
`asp`	Control *y*/*x* aspect ratio	Section 7.2.6, p. 153
`abline`	Add vertical/horizontal line	Section 7.3, p. 157

Function/operator	Brief description	First occurrence
`segments`	Add specific line segments	Section 7.3, p. 157
`points`	Add points	Section 7.3, p. 157
`lines`	Add lines following coordinates	Section 7.3, p. 158
`arrows`	Add arrows	Section 7.3, p. 158
`text`	Add text	Section 7.3, p. 158
`legend`	Add/control legend	Section 7.3, p. 158
`ggplot`	Initiate a `ggplot2` graphic	Section 7.4.1, p. 161
`mapping`, `aes`	Set aesthetic mappings (`ggplot2`)	Section 7.4.1, p. 161
`geom_point`	Add points (`ggplot2`)	Section 7.4.1, p. 161
`ggtitle`	Add title (`ggplot2`)	Section 7.4.1, p. 162
`xlab`, `ylab`	Axis labels (`ggplot2`)	Section 7.4.1, p. 162
`geom_line`	Add lines (`ggplot2`)	Section 7.4.2, p. 163
`size`, `shape`, `color`	Set geom constants (`ggplot2`)	Section 7.4.2, p. 163

Function/operator	Brief description	First occurrence
`linetype`	Set line type (`ggplot2`)	Section 7.4.2, p. 163
`geom_hline`	Add horizontal lines (`ggplot2`)	Section 7.4.3, p. 166
`yintercept`	Horizontal line location (`ggplot2`)	Section 7.4.3, p. 166
`geom_segment`	Add line segments (`ggplot2`)	Section 7.4.3, p. 166

8
READING AND WRITING FILES



Now that we've established the fundamentals of R syntax and behavior, I'll cover one more essential aspect of working with R: loading and saving data in an active workspace by reading and writing files. To work with a large dataset, you need to read in the data from an external file, whether it's stored as plaintext, appears in a spreadsheet file, or is found at a specific URL online. R provides standard functions to import these datasets, usually resulting in a data frame object in your workspace. You can also "export" data frames and other objects from your R session, writing them out as new files on your computer, and you can save any plots you create as image files using the command line. In this chapter, I'll review useful features for these tasks.

8.1 R-Ready Datasets

First, let's take a brief look at some of the datasets that are built into the software or are part of user-contributed packages. These datasets are useful samples to practice with and to experiment with functionality. We'll make frequent use of them in Parts III through V.

Run the following line at the prompt to open a window that lists available ready-to-use datasets along with a one-line description:

```
R> data(package=.packages(all.available=TRUE))
```

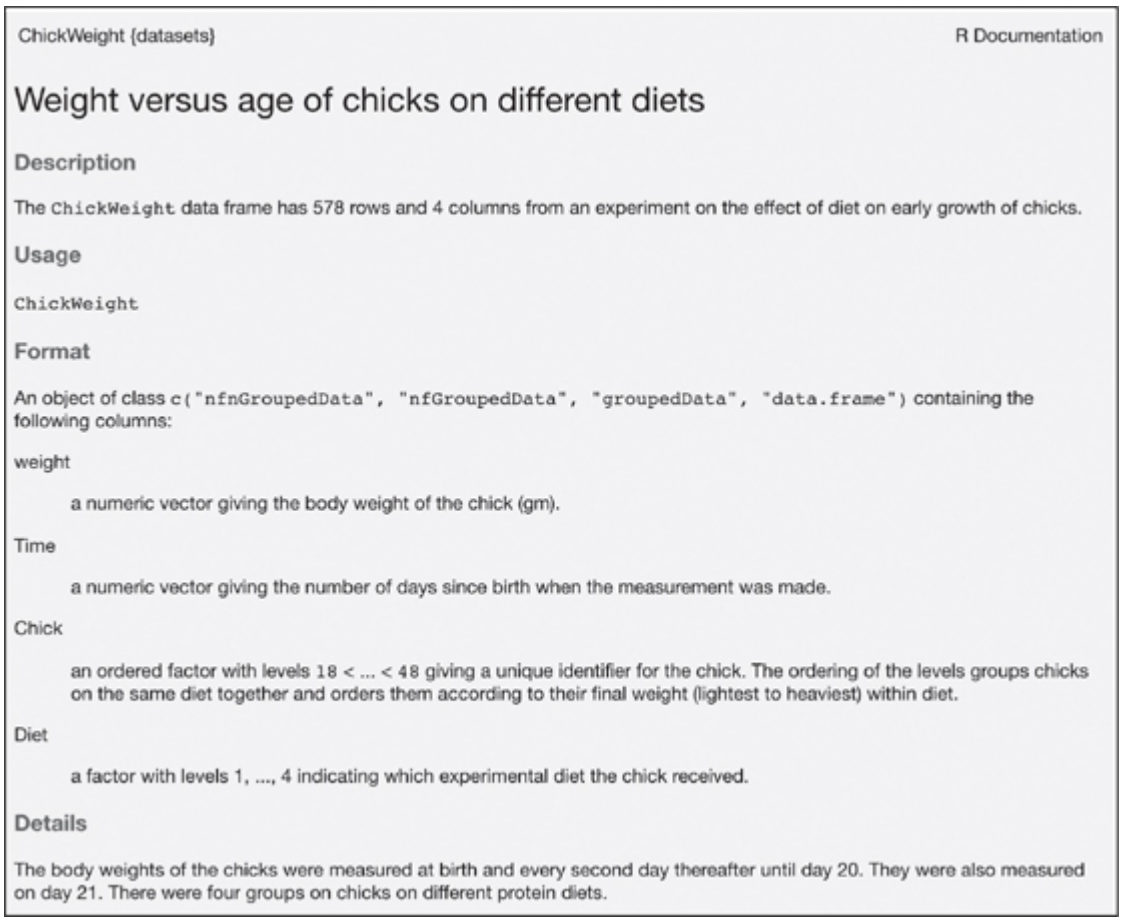
These datasets are organized in alphabetical order by name and grouped by package. (The exact list that appears depends on which contributed packages you've previously installed; see Section A.2.)

8.1.1 Built-In Datasets

There are a number of datasets contained within the built-in, automatically loaded package `datasets`. To see a summary of the datasets contained in this package, modify the previous line of code specifying the desired package name as a character string:

```
R> data(package="datasets")
```

R-ready datasets all have a corresponding help file that contains important details about the data and how they're organized. For example, to see the help file for the built-in dataset `ChickWeight`, enter `?ChickWeight` at the prompt. This results in Figure 8-1.



ChickWeight {datasets} R Documentation

Weight versus age of chicks on different diets

Description

The `ChickWeight` data frame has 578 rows and 4 columns from an experiment on the effect of diet on early growth of chicks.

Usage

```
ChickWeight
```

Format

An object of class `c("nfnGroupedData", "nfGroupedData", "groupedData", "data.frame")` containing the following columns:

weight

a numeric vector giving the body weight of the chick (gm).

Time

a numeric vector giving the number of days since birth when the measurement was made.

Chick

an ordered factor with levels `18 < ... < 48` giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

Diet

a factor with levels 1, ..., 4 indicating which experimental diet the chick received.

Details

The body weights of the chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. There were four groups on chicks on different protein diets.

Figure 8-1: An excerpt of the help file for the `ChickWeight` dataset

This file explains the variables and their values; it notes that the data are stored in a data frame with 578 rows and 4 columns. The objects in `datasets` are automatically loaded, so you can access `ChickWeight` by directly entering its name at the prompt. Let's look at the first 15 records:

```
R> ChickWeight[1:15,]
   Weight Time Chick Diet
1      42    0     1    1
2      51    2     1    1
3      59    4     1    1
4      64    6     1    1
5      76    8     1    1
6      93   10     1    1
7     106   12     1    1
8     125   14     1    1
```

```
9     149   16     1    1
10    171   18     1    1
11    199   20     1    1
12    205   21     1    1
13     40    0     2    1
14     49    2     2    1
15     58    4     2    1
```

You can treat this dataset like any other data frame you've created in R; note the use of `[1:15,]` to access the desired rows from such an object, as Section 5.2.1 detailed.

8.1.2 Contributed Datasets

There are many more R-ready datasets that come as part of contributed packages. To access them, first install and load the relevant package. Consider the dataset `ice.river`, which is in the contributed package `tseries` by Trapletti & Hornik (2024). First, install the package by running the line `install.packages("tseries")` at the prompt. To access the components of the package, load it using `library`:

```
R> library("tseries")
```

You'll see some output written to the console as the package loads. Once loaded, you can enter `library(help="tseries")` to see the list of functions and datasets in this package. The `ice.river` object represents one dataset. Enter `?ice.river` to find out details about the data contained therein. This help file describes `ice.river` as a "time series object" consisting of river flow, precipitation, and temperature measurements—data initially reported in Tong et al. (1985). To access this object itself, you must explicitly load it using the `data` function; thereafter, you can work with `ice.river` in your workspace. Here are the first five records:

```
R> data(ice.river)
R> ice.river[1:5,]
     flow.vat flow.jok prec temp
[1,]    16.10     30.2  8.1  0.9
[2,]    19.20     29.0  4.4  1.6
```

```
[3,]    14.50     28.4  7.0  0.1
[4,]    11.00     27.8  0.0  0.6
[5,]    13.60     27.8  0.0  2.0
```

The availability and convenience of these R-ready datasets make it easy to test code, and I'll use them in subsequent chapters for demonstrations. To analyze your own data, however, you'll often have to import them from some external file.

8.2 Reading in External Data Files

R has a variety of functions for reading characters from stored files and making sense of them. Let's look at how to read *table-format* files, the standard format for datasets and hence the easiest for R to read and import.

8.2.1 The Table Format

Table-format files are essentially plaintext files with three key features that fully define how R should read the data.

Header If a *header* is present, it's always the first line of the file. You can use this optional feature to provide names for each column of the table (in other words, the variable names of the dataset). When importing a file into R, you need to tell the software if a header is present so that it knows whether the first line contains the variable names or merely the values of the first record of the dataset.

Delimiter The all-important *delimiter* is a character you use to separate the entries in each line. This tells R when a specific entry begins and ends (in other words, its exact position in the table). You cannot use the delimiter character for anything else in the file.

Missing value You use this unique character string exclusively to denote a missing value. When reading the file, R turns these entries into the form it recognizes: `NA`.

Typically, these files have a *.txt* (highlighting the plaintext style) or *.csv* (for *comma-separated values*) extension.

Let's try an example, using a variation on the data frame `mydata` as defined at the end of Section 5.2.3. Figure 8-2 shows an appropriate table-format file, *mydatafile.txt*, which has the data from that data frame with a few randomly selected entries now marked with an asterisk. You can find this data file on the book's website at *https://www.nostarch.com/book-of-r-2nd-edition*, or you can create it yourself from Figure 8-2 using a plaintext editor.

From inspecting the content of the text file, we can see the first line is indeed a header giving the variable names. The values themselves are delimited with a single space. Assume those values that have been replaced by an asterisk (*) are missing values. Also, note that each new record must start on a new line.

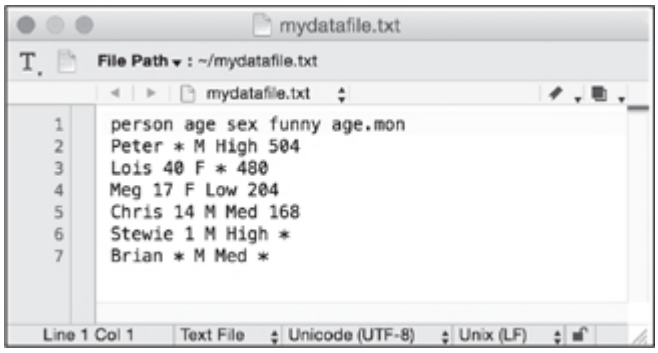


Figure 8-2: A plaintext table-format file

Suppose you're handed this plaintext file for data analysis in R. The ready-to-use command `read.table` imports table-format files and produces a data frame object within your workspace as follows:

```
R> mydatafile <- read.table(file="+++&&&/Users/tdavies/mydatafile.txt&&&+++",
                            header=TRUE,sep=" ",na.strings="*")
R> mydatafile
  person age sex funny age.mon
1  Peter  NA   M  High     504
2   Lois  40   F  <NA>     480
3    Meg  17   F   Low     204
```

```
4  Chris  14   M   Med     168
5 Stewie   1   M  High      NA
6  Brian  NA   M   Med      NA
```

In a call to `read.table`, `file` takes a character string with the filename and folder location (using forward slashes), `header` is a logical value telling R whether `file` ought to treat the first line as the variable names (`TRUE` in this case), `sep` takes a character string providing the delimiter (a single space, `" "`, in this case), and `na.strings` requests the characters used to denote missing values (here they are represented by `"*"`).

If you don't want to type the entire folder location each time, first set your working directory via `setwd` (Section 1.3.4) and then simply use the filename and its extension as the character string supplied to the `file` argument. However, both approaches require you to know exactly where your file is located when you're working at the R prompt. Fortunately, R possesses some useful additional tools should you forget your file's precise location. You can use `list.files` to view textual output of the contents of any folder. The following example betrays the messiness of my local user directory:

```
R> list.files("/Users/tdavies")
 [1] "bands-SCHIST1L200.txt" "Brass"                 "Desktop"
 [4] "Documents"             "DOS Games"             "Downloads"
 [7] "Dropbox"               "Exercise2-20Data.txt"  "Google Drive"
[10] "iCloud"                "Library"               "log.txt"
[13] "Movies"                "Music"                 "mydatafile.txt"
[16] "OneDrive"              "peritonitis.sav"       "peritonitis.txt"
[19] "Personal9414"          "Pictures"              "Public"
[22] "Research"              "Rintro.tex"            "Rprofile.txt"
[25] "Rstartup.R"            "spreadsheetfile.csv"   "spreadsheetfile.xlsx"
[28] "TakeHome_template.tex" "WISE-P2L"              "WISE-P2S.txt"
[31] "WISE-SCHIST1L200.txt"
```

One important feature to note here is that it can be difficult to distinguish between files and folders. Files will typically have an extension, and folders won't; however, `WISE-P2L` is a file that happens to have no extension and looks no different from any of the listed folders.

You can also find files interactively from R. The `file.choose` command opens your filesystem viewer directly from the R prompt, just as any other program does when you want to open something. You can navigate to the folder of interest, and after you select your file (see Figure 8-3), only a character string is returned:

```
R> file.choose()
[1] "/Users/tdavies/mydatafile.txt"
```

NOTE

The precise appearance of the returned string will vary according to your operating system. While the output here comes from my Mac, on a PC running Windows you'll see the filepath begin with the drive letter (which you'll need when specifying the paths themselves), and folder separators are often presented with `\\` *as opposed to* `/` *(either form may be used).*

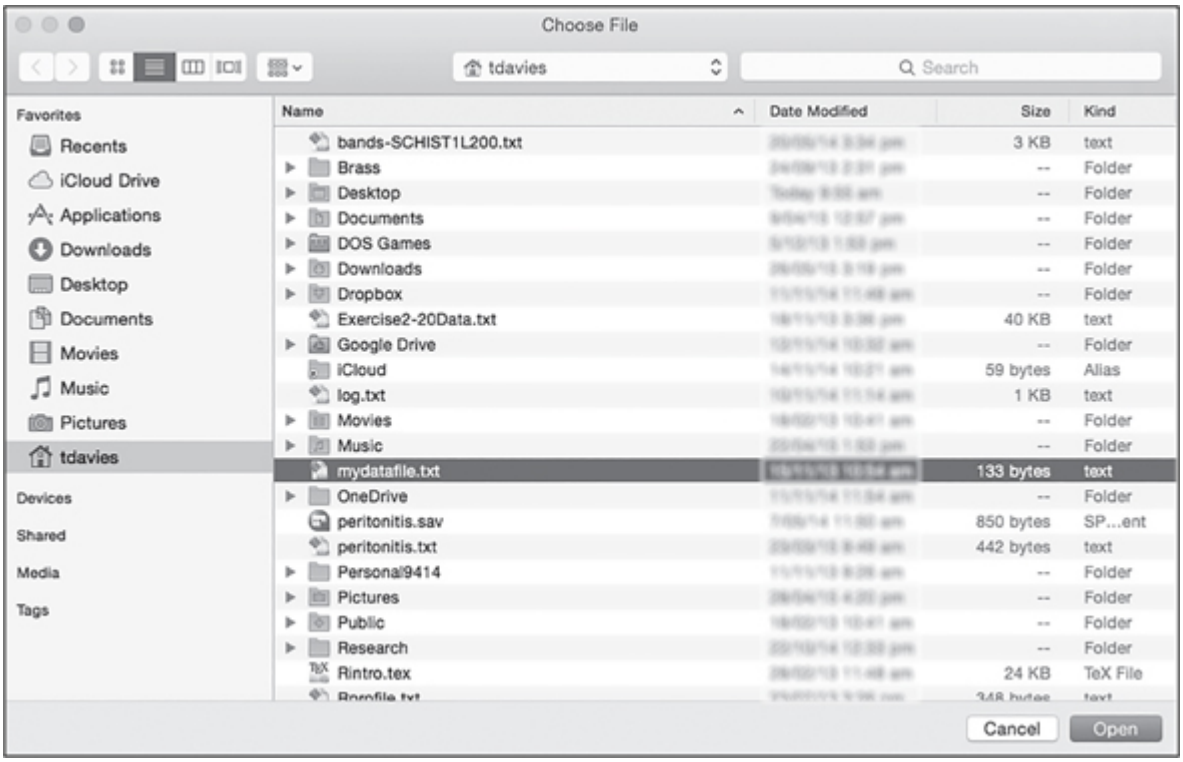


Figure 8-3: My local file navigator opened as the result of a call to `file.choose`*. When the file of interest opens, the R command returns the full filepath to that file as a character*

string.

This command is particularly useful, as it returns the character string of the directory in precisely the format a command such as `read.table` requires. So, calling the following line and selecting *mydatafile.txt*, as in Figure 8-3, produces an identical result to the explicit use of the filepath in `file`, shown earlier:

```
R> mydatafile <- read.table(file=file.choose(),header=TRUE,sep=" ",
                            na.strings="*")
```

If your file successfully loads, you'll return to the R prompt without receiving any error messages.

When importing a table-format dataset into R to use as a data frame object, keep in mind the difference between character string observations and factor observations. No factor attribute information is stored in the plaintext file, and `read.table` keeps non-numeric values as basic character strings by default. If instead you want any text-based columns to automatically convert into formal R factors, set the optional argument `stringsAsFactors` to `TRUE` in the call to `read.table` (I'll provide an example of this in a moment). I chose not to do so in the earlier examples when reading in *mydatafile.txt*, and the consequence of this is evidenced if we inspect the output from a call to `str` on the resulting object:

```
R> str(mydatafile)
'data.frame':   6 obs. of  5 variables:
 $ person : chr  "Peter" "Lois" "Meg" "Chris" ...
 $ age    : int  NA 40 17 14 1 NA
 $ sex    : chr  "M" "F" "F" "M" ...
 $ funny  : chr  "High" NA "Low" "Med" ...
 $ age.mon: int  504 480 204 168 NA NA
```

As it doesn't make much sense to formally declare the `person` column (the first name of each individual) as a categorical variable, leaving that variable as basic character strings is fine. However, we'd likely still want to use `sex` and `funny` as factors, and so in this instance we ought to convert

those columns to formally identify as such. You can then overwrite `sex` and `funny` with factor versions of themselves if you want them as that data type:

```
R> mydatafile$sex <- factor(x=mydatafile$sex)
R> mydatafile$funny <- factor(x=mydatafile$funny,levels=c("Low","Med","High"))
```

Here, we simply use `factor` to turn `mydatafile$sex` into factor format and don't impart any more control over the final product in terms of the category labels and ordering. In overwriting `mydatafile$funny`, however, we make use of the `levels` argument to ensure the categories are ordered from `Low` to `High` (recall from Section 4.3.2 that had we not done so, the categories would be arranged alphabetically). If we now look at the output of `str`,

```
R> str(mydatafile)
'data.frame':   6 obs. of  5 variables:
 $ person : chr  "Peter" "Lois" "Meg" "Chris" ...
 $ age    : int  NA 40 17 14 1 NA
 $ sex    : Factor w/ 2 levels "F","M": 2 1 1 2 2 2
 $ funny  : Factor w/ 3 levels "Low","Med","High": 3 NA 1 2 3 2
 $ age.mon: int  504 480 204 168 NA NA
```

the object type associated with `sex` and `funny` has been changed as instructed.

8.2.2 Spreadsheet Workbooks

Next, let's examine some ubiquitous spreadsheet software file formats. The standard file format for Microsoft Excel is *.xlsx* (or the older *.xls*). In general, these files are not directly compatible with R. There are functions in some contributed R packages that attempt to bridge this gap—for example, `gdata` by Warnes et al. (2024) or `XLConnect` by Mirai Solutions GmbH (2024)—but in most situations it's much easier to first save the spreadsheet file as a table format, such as *.csv* (which stands for *comma-separated values*). Consider the hypothetical data from Exercise 7.1 (b), which has been stored in an Excel file, *spreadsheetfile.xlsx*. Figure

8-4 shows a screenshot; as with *mydatafile.txt*, you can find this file on the book's website at *https://www.nostarch.com/book-of-r-2nd-edition*.

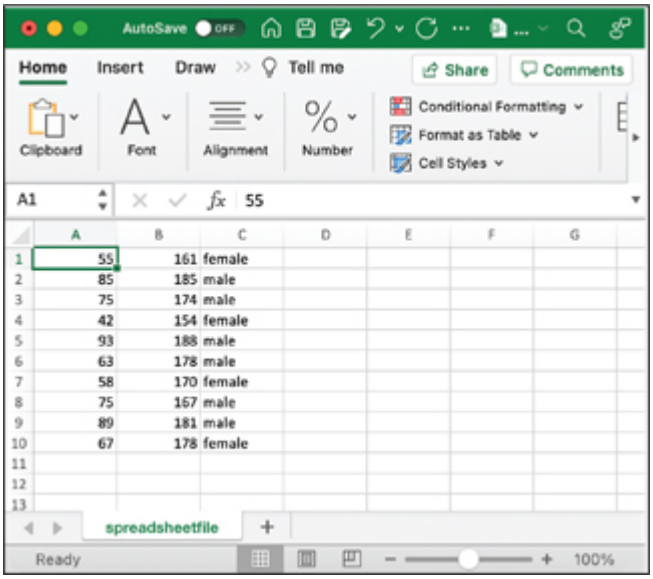


Figure 8-4: A spreadsheet file of the data from Exercise 7.1 shown in Microsoft Excel

To read this spreadsheet with R, first convert it to a table format. In Excel, File ▸ Save As . . . provides a wealth of options. Save the spreadsheet as a comma-separated file, *spreadsheetfile.csv* (this is also supplied on the book's website if you need it). R has a shortcut version of `read.table` for these files, `read.csv`. On my machine, I read in the newly created *spreadsheetfile.csv* with the following line:

```
R> spread <- read.csv(file="+++&&&/Users/tdavies/spreadsheetfile.csv&&&+++",
                      header=FALSE,stringsAsFactors=TRUE)

R> spread
   V1  V2     V3
1  55 161 female
2  85 185   male
3  75 174   male
4  42 154 female
5  93 188   male
6  63 178   male
7  58 170 female
```

```
8  75 167   male
9  89 181   male
10 67 178 female

R> str(spread)
'data.frame':   10 obs. of  3 variables:
 $ V1: int  55 85 75 42 93 63 58 75 89 67
 $ V2: int  161 185 174 154 188 178 170 167 181 178
 $ V3: Factor w/ 2 levels "female","male": 1 2 2 1 2 2 1 2 2 1
```

Here, the `file` argument again specifies the desired file, which has no header. At the time of writing, `read.csv` has `header=TRUE` by default (as opposed to the `header=FALSE` default for `read.table`). So, I explicitly set `header=FALSE` in this call to `read.csv`; notice that R provides "dummy" variable names V1, V2, and so on when no header is present (recall from Sections 5.2.1 and 6.2.1 that you can use the `colnames` function to set these column names within R if desired). In addition, I set `stringsAsFactors=TRUE` because I'd like the only present non-numeric column to be automatically coerced to a factor. There are no missing values, so I don't bother specifying `na.strings` (if there were missing values, this argument is simply used in the same way as shown earlier). By definition, *.csv* files are comma-delimited, which `read.csv` correctly implements by default, so you don't need the `sep` argument.

Reading tabular data into R is fairly straightforward, but you need to be aware of how the data file is headed and delimited and how missing entries are identified. When dealing with spreadsheet files, be aware of the file type (for example, it's difficult to tell whether a given spreadsheet is *.xlsx* or *.csv* by just looking at it in Excel). The simple table format is a natural and common way for datasets to be stored, but if you need to read in a file with a more complicated structure than those seen here, R and its contributed packages make available some more sophisticated functions. See, for example, the documentation for the `scan` and `readLines` functions, which provide advanced control over how to parse a file. You can also find documentation on `read.table` and `read.csv` by accessing `?read.table` from the prompt.

8.2.3 Web-Based Files

With an internet connection, R can read in files from a website with the `read.table` command. All the same rules concerning headers, delimiters, and missing values remain in place; you just have to specify the URL address of the file instead of a local folder location.

As an example, you'll use the archived online repository of datasets made available by the *Journal of Statistics Education* through the American Statistical Association at *https://jse.amstat.org/jse_data_archive.htm*.

One of the many available data files is a table-format dataset containing data on the characteristics of 308 diamonds. This comes from an analysis by Singfat Chu (2001) based on an advertisement in a Singaporean newspaper. The dataset is available at *https://nostarch.com/download/diamonds.zip*, and Figure 8-5 shows the result of clicking the link.

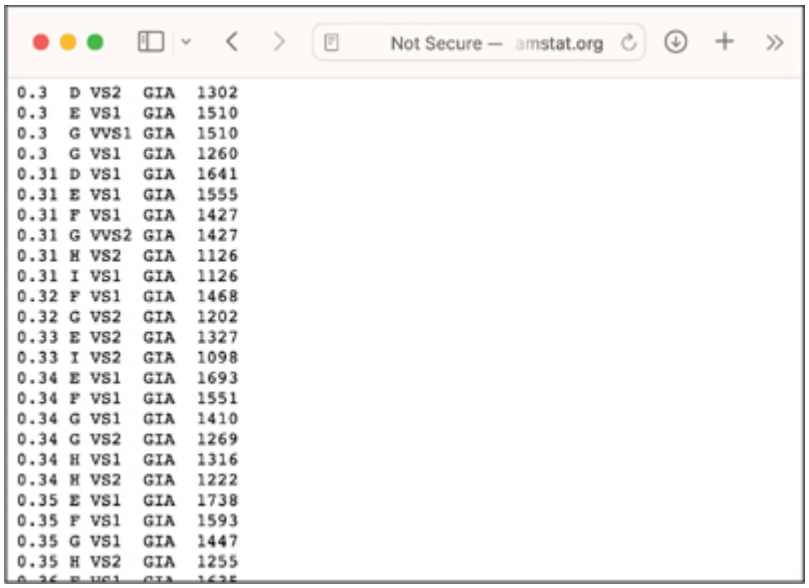


Figure 8-5: A table-format data file found online

For details on what's recorded in this table, see the documentation file (*http://jse.amstat.org/v9n2/4C.txt*) and the accompanying article by Chu (2001). Note that of the five columns, the first and fifth are numeric, and the others would be well represented by factors. The

delimiter is blank whitespace, there's no header, and there are no missing values (so you don't have to specify a value used to represent them).

With this in mind, you can create a data frame directly from the R prompt by using the full URL as a character string and supplying that to a call to `read.table`:

```
R> dia.url <- "http://jse.amstat.org/v9n2/4Cdata.txt"
R> diamonds <- read.table(dia.url,stringsAsFactors=TRUE)
```

Because there's no header in the table, you can leave the default `header` value `FALSE`. The default value for `sep` is `""`, a general indicator for varying whitespace (not to be confused with `" "`, meaning an explicit space character), which is also exactly what this table uses. However, remember that the default value for `stringsAsFactors` is `FALSE` and, just like in Section 8.2.2, we choose to set this to `TRUE` for automatic coercion of any non-numeric columns into factor format.

Following the import, you can supply names (based on the information about the dataset found in the in the *4C.txt* documentation) to each column as follows:

```
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
R> diamonds[1:5,]
  Carat Color Clarity Cert Price
1  0.30     D     VS2  GIA  1302
2  0.30     E     VS1  GIA  1510
3  0.30     G    VVS1  GIA  1510
4  0.30     G     VS1  GIA  1260
5  0.31     D     VS1  GIA  1641

R> str(diamonds)
'data.frame':   308 obs. of  5 variables:
 $ Carat  : num  0.3 0.3 0.3 0.3 0.31 0.31 0.31 0.31 0.31 0.31 ...
 $ Color  : Factor w/ 6 levels "D","E","F","G",..: 1 2 4 4 1 2 3 4 5 6 ...
 $ Clarity: Factor w/ 5 levels "IF","VS1","VS2",..: 3 2 4 2 2 2 2 5 3 2 ...
 $ Cert   : Factor w/ 3 levels "GIA","HRD","IGI": 1 1 1 1 1 1 1 1 1 1 ...
 $ Price  : int  1302 1510 1510 1260 1641 1555 1427 1427 1126 1126 ...
```

Viewing the first five records shows that the data frame displays as you intended, and a call to `str` confirms the `Clarity` and `Cert` columns are, as intended, represented as factors.

8.2.4 Other File Formats

File formats besides *.txt* or *.csv* files can be read into R, such as the data file format *.dat*. You can also import these files using `read.table`, though they may contain extra information at the top that must be skipped using the optional `skip` argument. (The `skip` argument asks for the number of lines at the top of the file that should be ignored before R begins the import.)

As mentioned in Section 8.2.2, there are also contributed packages that can cope with other statistical software files. The R package `foreign` (R Core Team 2025), available from CRAN, provides support for reading data files used by statistical programs such as Stata, SAS, Minitab, and SPSS.

Other contributed packages on CRAN can help R handle files from various database management systems (DBMSs). For example, the `RODBC` package (Ripley & Lapsley 2024) lets you query Microsoft Access databases and return the results as a data frame object. Other interfaces include the packages `RMySQL` (Ooms et al. 2024) and `RJDBC` (Urbanek 2022).

8.3 Writing Out Data Files and Plots

Writing out new files from data frame objects with R is as easy as reading in files. R's vector-oriented behavior is a fast and convenient way to recode datasets, which is perfect if you need to restructure some data and write them back out to a file.

8.3.1 Datasets

Use the `write.table` function to write table-format files to your computer. You supply a data frame object as `x`, and this function writes its contents to a new file with a specified name, delimiter, and missing value string.

For example, the following line takes the `mydatafile` object as it was left at the end of Section 8.2.1 and writes it out to disk as a new text file on my local machine:

```
R> write.table(x=mydatafile,file="+++&&&/Users/tdavies/somenewfile.txt&&&+++",
                sep="@",na="??",quote=FALSE,row.names=FALSE)
```

After specifying the data frame object to write out, provide `file` with the folder location, ending in the filename you want for your new data file. This command creates a new table-format file, *somenewfile.txt*, in the specified folder location, delimited by `@` and with missing values denoted with `??` (because you're actually creating a new file, the `file.choose` command doesn't tend to be used here). The variable (column) names within `mydatafile` automatically write to the file as a header. The optional logical argument `quote` determines whether to encapsulate each non-numeric entry in double quotes (if you explicitly need them in your file for, say, formatting requirements of other software); request no quotes by setting the argument to `FALSE`. Another optional logical argument, `row.names`, asks whether to include the row names of `mydatafile` (in this example, this would be the default numbers `1` to `6`), which you also omit with `FALSE`. Figure 8-6 shows the resulting file, which you can open in a text editor.

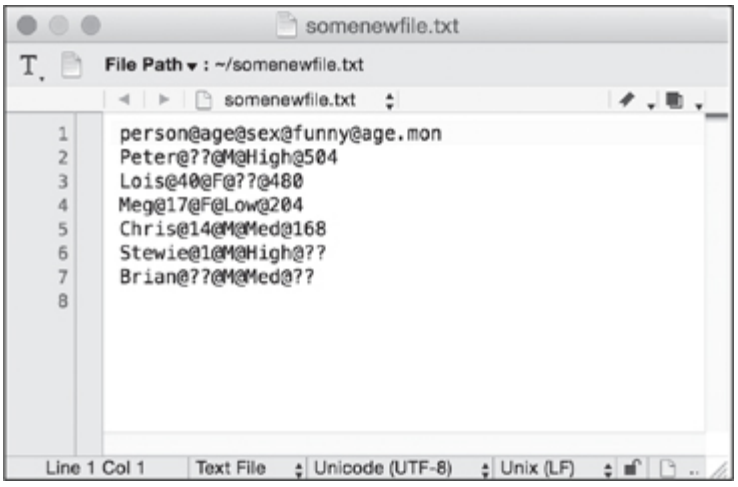


Figure 8-6: The contents of somenewfile.txt

Like `read.csv`, `write.csv` is a shortcut version of the `write.table` function designed specifically for writing your data frame out as a *.csv* file.

8.3.2 Plots and Graphics Files

You can also write plots directly to a file. In Chapter 7, you created and displayed plots in an active graphics device. This graphics device needn't be a screen window; it can be a specified file. Instead of displaying the plot immediately on the screen, you can have R follow these steps: Open a "file" graphics device, run any plotting commands to create the final plot, and close the device. R supports direct writing to *.jpeg*, *.bmp*, *.png*, and *.tiff* files using functions of the same names. For example, the following code (remember to re-specify the */Users/tdavies* part of the filepath to a valid location on your own machine) uses these three steps to create a *.jpeg* file:

```
R>
jpeg(filename="+++&&&/Users/tdavies/myjpegplot.jpeg&&&+++",width=600,height=600)
R> plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
        main="a saved .jpeg plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
R> dev.off()
null device
          1
```

The file-targeted graphics device quietly opens upon the call to `jpeg`, where I provide the intended name of the file and its folder location as `filename`. By default, the dimensions of the device are set to 480×480 pixels, but here I change them to 600×600. You could also set these dimensions by supplying other units (inches, centimeters, or millimeters) to `width` and `height` and by specifying the unit with an optional `units` argument.

Once the device command is given, you execute any R plotting commands you need in order to create the image; this example plots some points and then includes additional points with a second command. The final graphical result silently writes to the specified file

just as it would have been displayed on the screen. When you finish plotting, you must explicitly close the file device with a call to `dev.off()`, which prints information on the remaining active device (here, you can loosely interpret "null device" as "nothing is left open"). If `dev.off()` *isn't* called, then R continues to output any subsequent plotting commands to the file, possibly overwriting what you have there. The left plot in Figure 8-7 shows the resulting file created in this example.

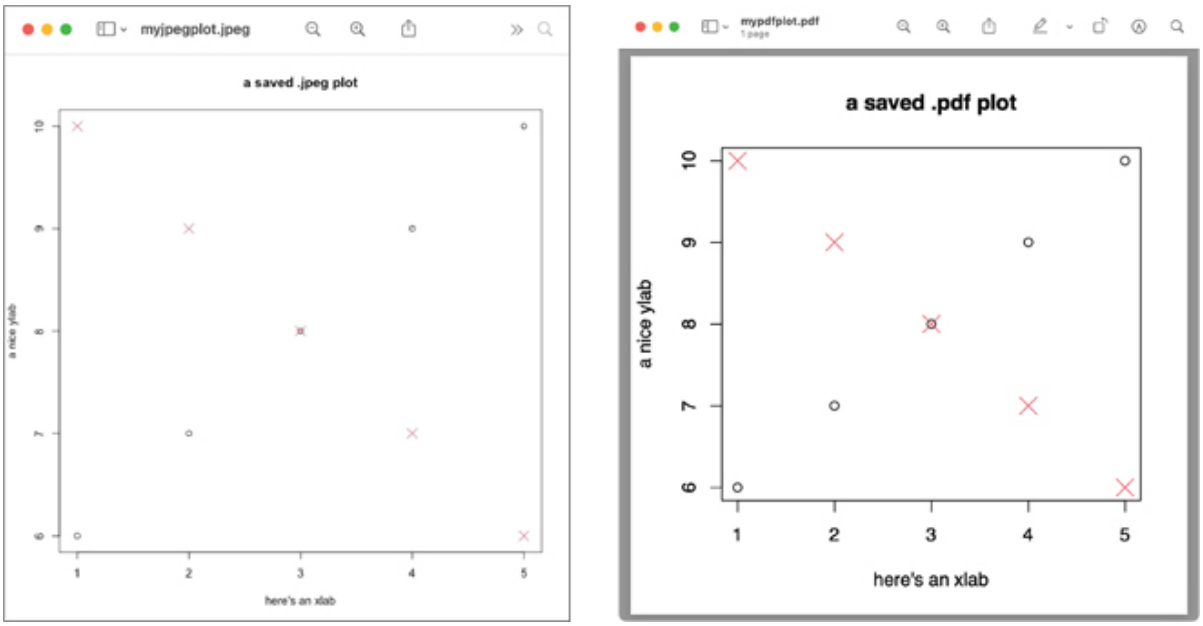


Figure 8-7: R plots that have been written directly to disk: a .jpeg *version (left) and a* .pdf *version (right) of the same plotting commands*

You can also store R plots as other file types, such as PDFs (using the `pdf` function) and EPS files (using the `postscript` function). Though some argument names and default values are different for these functions, they follow the same basic premise. You specify a folder location, a filename, and width and height dimensions; enter your plotting commands; and then close the device with `dev.off()`. The right panel of Figure 8-7 shows the *.pdf* file created with the following code:

```
R> pdf(file="+++&&&/Users/tdavies/mypdfplot.pdf&&&+++",width=5,height=5)
R> plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
        main="a saved .pdf plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
```

```
R> dev.off()
null device
          1
```

Here, you use the same plotting commands as before, and there are just a few minor differences in the code. The argument for the file is `file` (as opposed to `filename`), and the units for `width` and `height` default to inches in `pdf`. The difference of appearance between the two images in Figure 8-7, most notably the relative sizes of the points and text, results primarily from these differences in width and height.

This same process—opening a file-targeted graphics device and closing it off with `dev.off()` after plotting—also works for `ggplot2` graphics. True to style, however, the `ggplot2` package provides a convenient alternative. You can use the `ggsave` function to write the most recently plotted `ggplot2` graphic to file, performing the device open/close action in one line. For example, the following code creates and displays a `ggplot2` object from a simple dataset:

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> fb <- data.frame(foo,bar)
R> ggplot(fb,mapping=aes(x=foo,y=bar)) +
          geom_point(size=3,shape=8,color="darkgreen") +
          geom_line(color="orange",linetype=4)
```

To save this plot to a file on my computer, I use the following line:

```
R> ggsave(filename="+++&&&/Users/tdavies/mypngggplot.png&&&+++")
Saving 7 x 7 in image
```

This writes the image to a *.png* file in the specified `filename` directory. (Note that dimensions are reported if you don't explicitly set them using `width` and `height`; these vary depending on how you size the window in which the plot is currently displayed.) Figure 8-8 shows the resulting image file for this example.

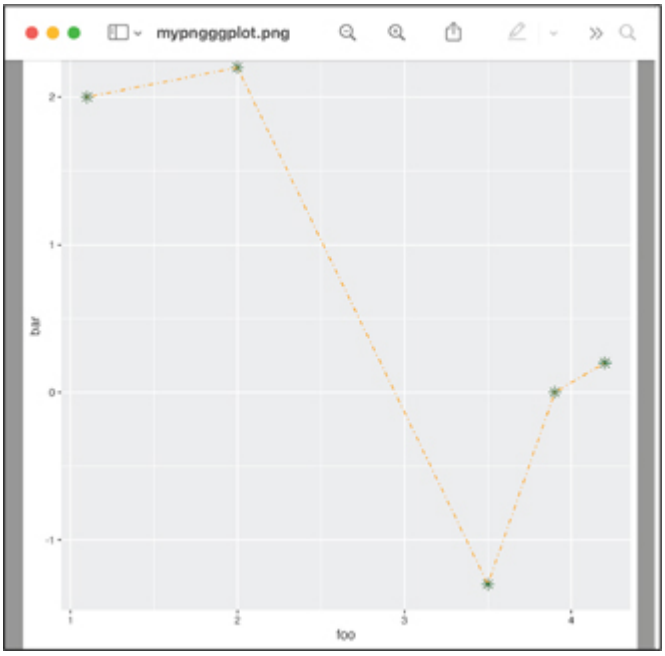


Figure 8-8: The .png *file created using* `ggplot2`*'s* `ggsave` *command*

Beyond just being concise, `ggsave` is convenient in a few other ways. For one, you can use the same command to create a variety of image file types; the type is determined by the extension you supply in the `filename` argument. Also, `ggsave` has a range of optional arguments if you want to control the size of the image and the quality or scaling of the graphic.

For more details on saving images from base R graphics, see the `?jpeg`, `?pdf`, and `?postscript` help files. You can consult `?ggsave` for more on saving images with `ggplot2`.

Exercise 8.1

a. R's built-in `datasets` library contains the `quakes` data frame. Make sure you can access this object and view the corresponding help file to get an idea of what this data represents. Then, do the following:

i. Select only those records that correspond to a magnitude (`mag`) of greater than or equal to `5` and write them to a table-format file, *q5.txt*, in an existing folder on your machine. Use a delimiting character of `!` and don't include any row names.

ii. Read the file back into your R workspace, naming the object `q5.dframe`.

b. The contributed package `car` (technically, one of its dependency packages, `carData`) has a data frame called `Duncan`, which provides historical data on perceived job prestige in 1950. Install the `car` package, load it, and access the `Duncan` dataset and its help file. Then, do the following:

i. Using base R tools, write code that plots `education` on the x-axis and `income` on the y-axis, with both x- and y-axis limits fixed to be [0, 100]. Provide appropriate axis labels. For jobs with a `prestige` value of less than or equal to `80`, use a black ° as the point character. For jobs with `prestige` greater than 80, use a blue •.

ii. Add a legend explaining the difference between the two types of points.

iii. Once your plot looks good, execute code to save it as a 500×500-pixel *.png* image.

c. In Section 7.4.3, you created a `ggplot2` graphic of 20 observations displayed as the bottom image of Figure 7-13 on page 166. Use `ggsave` to save a copy of this plot as a *.tiff* file.

8.4 Ad Hoc Object Read/Write Operations

For the typical R user, the most common input/output operations probably revolve around datasets and plots. But if you need to read or write other kinds of R objects, such as lists, arrays, or even entire R workspaces/environments, you can do that too.

Ensure the `ggplot2` library is loaded and then run the following chunk of code. This creates some objects we'll use to investigate the read/write options throughout this section:

```
R> x <- 8
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> fb <- data.frame(foo,bar)
R> lookatme <- ggplot(fb,mapping=aes(x=foo,y=bar)) +
               geom_point(size=3,shape=8,color="darkgreen") +
               geom_line(color="orange",linetype=4)
R> somelist <- list(aa=c(5,2,45),
                    bb=matrix(data=c(T,T,F,F,F,F,T,F,T),nrow=3,ncol=3),
                    cc=factor(c(1,1,2,3,1,1,3),labels=c("A","B","C")))
```

Here, we've stored the single digit 8 as the object `x`, replicated code from Section 8.3.2 to produce the plot in Figure 8-8 stored as the object `lookatme`, and stored a list with three different members as the object `somelist`. Note that, as per details in Section 7.4.1, it's perfectly valid to store a `ggplot2` graphic as an R object; nothing displays until said object prints by typing the object name at the R prompt.

By executing `ls`, we see that the previous code results in having six different objects present in our R workspace:

```
R> ls()
[1] "bar"      "fb"       "foo"      "lookatme" "somelist" "x"
```

If you've not started in a clean/empty R session, you'll also see the names of any other objects you've previously created in this output.

8.4.1 save and load

The `save` and `load` commands provide perhaps the most general functionality when saving R objects. With these commands, which formally save the contents of R *environments* (see Section 9.1.1) and deal with a *.Rdata* file extension, you may write/read both single and multiple objects to/from disk in one go.

Suppose we want to save the objects `lookatme` and `somelist` to enjoy at a later date. On my local system, the line

```
R> save(somelist,lookatme,file="+++&&&/Users/tdavies/mySavedObjects.Rdata&&&+++")
```

writes these objects to the *mySavedObjects.Rdata* file in my home drive. Simply supply one or more objects, separated by commas, as the first argument(s), and then provide a filepath as a character string (remember your filepath will differ from mine, reflecting where you choose to place the new file).

How do we know this worked? First, let's clear the current R session, deleting all visible objects as we did in Section 2.2 with `rm`, and confirm our "clean slate" using `ls`:

```
R> rm(list=ls())
R> ls()
character(0)
```

We can now load in the previously saved objects with `load`, and behold:

```
R> load(file="+++&&&/Users/tdavies/mySavedObjects.Rdata&&&+++")
R> ls()
[1] "lookatme" "somelist"
```

The two loaded objects are exactly as they were originally created, with the same object names. Any valid operations on lists, for instance (see Section 5.1), are applicable to `somelist`. If you enter `lookatme` at the R prompt (remember, the `ggplot2` library must be loaded in whatever session you've loaded `lookatme` into), the plot will appear.

NOTE

To save every *object in a given R session (effectively saving your entire workspace to, for example, pick up where you left off at some later time) use the* `save.image` *function, which requires you supply only a* `file` *argument. Indeed, this is essentially what R by default asks, via a dialog box, if you'd like done when you quit the software (see Section 1.4.1).*

While clearly very flexible in terms of saving R objects, the `save`/`load` commands have a potential drawback in retention of their original names. This means that any `load`ed objects that possess the same name as any objects already present in the workspace doing the loading will silently overwrite those existing items, which could be problematic.

8.4.2 saveRDS and readRDS

The `saveRDS` and `readRDS` commands, which work with a *.rds* file extension, are alternatives to `save` and `load`. Technically speaking, these functions *serialize* the target object, translating the content into a format that can be efficiently stored as a file on your computer's hard drive. While `saveRDS` and `readRDS` operate only with respect to a single R object, we can still use lists to combine common individual objects together as one if desired. This makes `saveRDS` and `readRDS` ideal when, for instance, you want to share a given object with colleagues.

Imagine you want to share the `somelist` object from Section 8.4 with a collaborator. To illustrate this, ensure you have `somelist` in your current workspace and execute the following line (remember to replace my filepath with one relevant to your local system):

```
R>
saveRDS(object=somelist,file="+++&&&/Users/tdavies/mySerializedObject.rds&&&+++")
```

Just like a vanilla `save`, we supply `saveRDS` with the object we wish to store on disk (to the `object` argument) and provide the desired filepath and filename (the latter with an *.rds* extension).

So far, so good. Now pretend you've returned to it on a new day, in a new R session, and want to use this list for some task. You read the

object into R thusly:

```
R> somenewname <-
readRDS(file="+++&&&/Users/tdavies/mySerializedObject.rds&&&+++")
```

Like `load`, you need to supply `readRDS` with the path and name of wherever you've placed the *.rds* file. Unlike `load`, `readRDS` isn't used in isolation; you need to assign the result of `readRDS` to a locally named variable. In this example, `somenewname` is a list with members (and member names) entirely identical to `somelist` (you can compare the two if you've run the previous line in the same session).

We can see it doesn't matter at all that we had originally named the content of interest as `somelist`; you're able to read in the *.rds* file and name the overall thing whatever you want. This prevents the potential for naming conflicts mentioned at the end of Section 8.4.1. As such, `saveRDS`/`readRDS` are typically preferred over `save`/`load`, whether you're working with others who might want the object of interest or storing objects for your own use.

8.4.3 dput and dget

Similar in role to `saveRDS` and `readRDS`, you use the `dput` and `dget` commands for single R objects. The user also has freedom in object naming when reading a file in. The main difference is how the object itself is stored on your hard drive. Rather than serialization, `dput`/`dget` deal with human-readable plaintext representations of the object (technically, we call this encoding the American Standard Code for Information Interchange, or ASCII).

Let's remain with the `somelist` object; re-run the line that creates it (found at the beginning of Section 8.4, page 184) if necessary. The following line saves the list object to my own drive via `dput`:

```
R> dput(x=somelist,file="+++&&&/Users/tdavies/myASCIIObject.txt&&&+++")
```

Note that while the *.txt* file extension isn't strictly necessary, it allows your operating system to automatically associate it with any plaintext

editing software.

To read the file back into R as the list object it represents, just point `dget` to the appropriate location:

```
R> somenewname2 <- dget(file="+++&&&/Users/tdavies/myASCIIObject.txt&&&+++")
```

As earlier, you can confirm the content of `somenewname2` is identical to the original `somelist` (and identical to `somenewname` from Section 8.4.2 when working with the serialized version).

Figure 8-9 shows what the stored file looks like when opened in a third-party plaintext application.

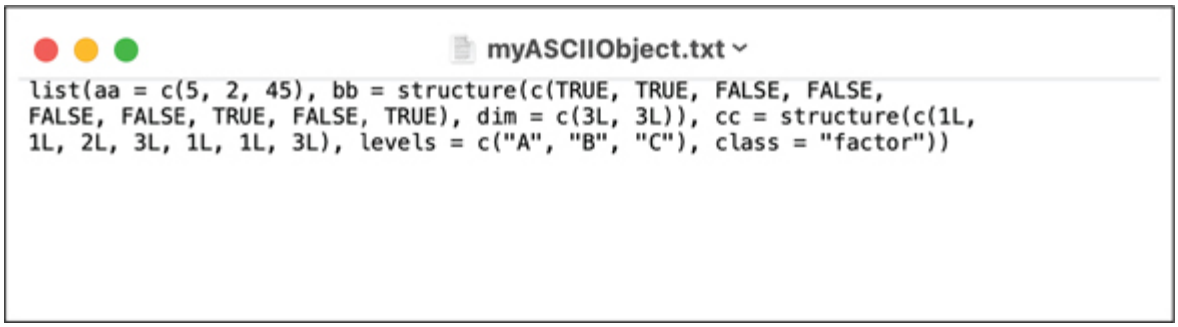


Figure 8-9: Viewing myASCIIObject.txt*, created by calling* `dput` *on* `somelist`*, in my default plaintext editor*

While the content doesn't look exactly like the R syntax with which you might be familiar (the specific structure and attributes of any content must be correctly defined), it is at the very least human-readable; we can see that the file describes a list with three distinct members of varying type. If you try and force a text editor to open *mySerializedObject.rds*, the content won't be human-readable.

8.4.4 Suitability

While there are no hard-and-fast rules regarding how to save ad hoc R objects, consider the following points when deciding what might be the most suitable in any given situation:

- Use `save`/`load` (*.Rdata* extensions) to save one or more R objects "in place"—that is, either as a workspace or a subset thereof. This is

convenient if you want to have access to frequently used objects quickly and with the same name. They are sometimes less ideal for sharing objects with others given the potential for naming conflicts when loading an *.Rdata* file into an existing R session.

- `saveRDS`/`readRDS` (*.rds* extensions) reliably serialize a target object for efficient storage. The user gains freedom in naming the object that's read in. The user can save only single objects, but the commands will work on lists if several objects need to be collected together.
- Use `dput`/`dget` (*.txt* extensions) like `saveRDS`/`readRDS`, but note these save/read the object in plaintext ASCII form as opposed to using serialization. The benefit is that the resulting file is human-readable. The drawback is that `dput`/`dget` are slightly less reliable for saving very complex objects, and they are less efficient in terms of file size compared to using serialization.

With these considerations in mind, many (this author included) prefer serialization via the `saveRDS`/`readRDS` commands for general-purpose use.

Exercise 8.2

a. In Section 3.2.1 (page 52) you created a simple 3×4×2 array named `AR`. Create this object once more, and do the following:

   i. Save the `AR` object to disk (somewhere convenient on your computer) twice: once by serialization, another using ASCII encoding.

   ii. Use the appropriate commands to read the two stored versions of the object back into your R

workspace; be sure to name them differently to one another.

iii. Check that all three objects—the original `AR` and the two read-in versions—are identical.

b. In Exercise 8.1 (page 183) you worked with the built-in `quakes` data, as well as the `Duncan` data from the contributed package `car`. Create a list, `exer`, that contains the two datasets `quakes` and `Duncan`. Then, do the following:

i. Write the list object directly to disk using ASCII encoding, calling it *Exercise8-2.txt*. Briefly inspect the contents of the file in a text editor.

ii. Read *Exercise8-2.txt* back into your workspace; call the resulting object `list.of.dataframes`. Check that `list.of.dataframes` contains the two data frame objects.

For the next tasks, you need to download the resources for this book, available at *https://nostarch.com/book-of-r-2nd-edition*. Unzip the archive, and take note of the filepath to its contents on your local machine.

c. One of the available resource files is *mysteryobject.rds*.

i. Use the appropriate command to read *mysteryobject.rds* into your R session (giving it any name of your choosing). What is the mystery object?

ii. Suppose you felt this object was important, and you wanted to retain it with the name you've chosen for future reference. Use the appropriate command to save the object to disk as an *.Rdata* file.

d. Begin with a clear R workspace by starting a new R session or by running `rm(list=ls())` at the prompt (do the latter only if you're happy deleting all objects that are present in your current session). Note that another of the available resource files you've downloaded is *objects.Rdata*.

   i. Assign the number 37 to an object named `x`.

   ii. Use the appropriate command to load *objects.Rdata* into your session.

   iii. Has your object `x` changed?

   iv. Use `ls()` to determine how many other objects are now in your workspace. Type their names into the R console to find out what they represent.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`data`	Load contributed dataset	Section 8.1.2, p. 171
`read.table`	Import table-format data file	Section 8.2.1, p. 173
`list.files`	Print specific folder contents	Section 8.2.1, p. 173
`file.choose`	Interactive file selection	Section 8.2.1, p. 174
`read.csv`	Import comma-delimited file	Section 8.2.2, p. 176
`write.table`	Write table-format file to disk	Section 8.3.1, p. 180

Function/operator	Brief description	First occurrence
`jpeg`, `bmp`, `png`, `tiff`	Write image/plot file to disk	Section 8.3.2, p. 180
`dev.off`	Close file graphics device	Section 8.3.2, p. 181
`pdf`, `postscript`	Write image/plot file to disk	Section 8.3.2, p. 182
`ggsave`	Write `ggplot2` plot file to disk	Section 8.3.2, p. 182
`save`, `load`	Save/load environment objects (*.Rdata*)	Section 8.4.1, p. 185
`saveRDS`, `readRDS`	Save/read a serialized R object (*.rds*)	Section 8.4.2, p. 186
`dput`, `dget`	Save/read an ASCII R object	Section 8.4.3, p. 187

PART II
PROGRAMMING

9
CALLING FUNCTIONS



Understanding how functions are called and interpreted in an R session will prove useful when writing your own functions in R. In this chapter, you'll first look at how R compartmentalizes variable names. You'll see R's rules for naming arguments and objects, and how R searches for arguments and other variables when a function is called. Then you'll look at some alternative ways to specify arguments when calling a function.

9.1 Scoping

We begin with R's *scoping rules*, which determine how the language compartmentalizes objects and retrieves them in a given session. This framework also defines the situations in which duplicate object names can exist at once. For example, you've used the argument `data` when calling `matrix` (Section 3.1), but `data` is also the name of a ready-to-use function that loads datasets from contributed packages (Section 8.1.2). In this section, you'll gain a basic understanding of how R behaves internally in these circumstances, which will help you later on when

programming and executing your own functions and those of other packages.

9.1.1 Environments

R enforces scoping rules with virtual *environments*. Think of environments as separate compartments that store data structures and functions. They allow R to distinguish between identical names that are associated with different scopes and therefore stored in different environments. Environments are dynamic entities: New environments can be created, and existing environments can be manipulated or removed.

NOTE

Technically speaking, environments don't actually contain *items. Rather, they have* pointers *to the locations of those items in the computer's memory. But using the "compartment" metaphor and thinking of objects "belonging to" these compartments is useful to get a general sense of how environments work.*

There are different kinds of environments with various roles. Some important ones include: *global* environments, *package* environments and *namespaces*, and *local* or *execution* environments.

Global Environments

The *global environment* is the compartment set aside for user-defined objects. Every object you've created or overwritten so far resides in the global environment of your current R session. In Section 1.4.1, I mentioned that a call to `ls()` lists all the objects, variables, and user-defined functions in the active workspace; more precisely, `ls()` prints the names of everything in the current global environment.

Starting with a new R workspace, the following code creates two objects and confirms their existence in the global environment:

```
R> foo <- 4+5
R> bar <- "stringtastic"
R> ls()
[1] "bar" "foo"
```

But what about all the ready-to-use objects and functions? Why aren't those printed alongside `foo` and `bar` as members of this environment? In fact, those objects and functions belong to package-specific environments.

Package Environments and Namespaces

The *package environment* refers to the items made available to the user by each package in R. The package *namespace* environment dictates how the functions and objects in the package must import any variables it needs from other package. Think of the package environment as the external interface to the user, and the package namespace as the internal interface to other R objects.

As per an earlier comment, you may informally think of all the ready-to-use functions and objects you're working with in this book as "belonging to" specific package environments. The same is true for the functions and objects of any contributed packages you've explicitly loaded with a call to `library`. You can use `ls` to list the items in a package environment as follows:

```
R> ls("package:graphics")
 [1] "abline"          "arrows"          "assocplot"       "axis"
 [5] "Axis"            "axis.Date"       "axis.POSIXct"    "axTicks"
 [9] "barplot"         "barplot.default" "box"             "boxplot"
[13] "boxplot.default" "boxplot.matrix"  "bxp"             "cdplot"
[17] "clip"            "close.screen"    "co.intervals"    "contour"
[21] "contour.default" "coplot"          "curve"           "dotchart"
[25] "erase.screen"    "filled.contour"  "fourfoldplot"    "frame"
[29] "grconvertX"      "grconvertY"      "grid"            "hist"
[33] "hist.default"    "identify"        "image"           "image.default"
[37] "layout"          "layout.show"     "lcm"             "legend"
[41] "lines"           "lines.default"   "locator"         "matlines"
```

```
[45] "matplot"         "matpoints"       "mosaicplot"      "mtext"
[49] "pairs"           "pairs.default"   "panel.smooth"    "par"
[53] "persp"           "pie"             "plot"            "plot.default"
[57] "plot.design"     "plot.function"   "plot.new"        "plot.window"
[61] "plot.xy"         "points"          "points.default"  "polygon"
[65] "polypath"        "rasterImage"     "rect"            "rug"
[69] "screen"          "segments"        "smoothScatter"   "spineplot"
[73] "split.screen"    "stars"           "stem"            "strheight"
[77] "stripchart"      "strwidth"        "sunflowerplot"   "symbols"
[81] "text"            "text.default"    "title"           "xinch"
[85] "xspline"         "xyinch"          "yinch"
```

Here, we see all of the user-visible objects contained in the built-in `graphics` package environment. Note that this list includes functions from Chapter 7, such as `arrows`, `plot`, and `segments`.

Local/Execution Environments

Calling a function in R creates a new environment, the *local* (or *execution*) environment. This environment contains all the objects and variables created in and visible to the function in a specific instance of that function running, including any arguments you've supplied to it upon execution. This feature allows the presence of argument names that are identical to other object names accessible in a given workspace.

For example, say you call `matrix` and pass in the argument `data`, as follows:

```
R> youthspeak <- matrix(data=c("OMG","LOL","WTF","YOLO"),nrow=2,ncol=2)
R> youthspeak
     [,1]  [,2]
[1,] "OMG" "WTF"
[2,] "LOL" "YOLO"
```

Calling this function creates an execution environment containing the `data` vector. When you execute the function, it looks for `data` in this local environment. That means R isn't confused by other objects or functions named `data` in other environments (such as the `data` function automatically loaded from the `utils` package environment). If R doesn't

find a required item in the local environment, only then does it begin to widen its search for that item (I'll discuss this feature further in Section 9.1.2). Once the function completes, this local environment is automatically removed, or "garbage collected." The same comments apply to the `nrow` and `ncol` arguments.

9.1.2 Search Path

To access data structures and functions from environments other than the immediate global environment, R follows a *search path*. The search path lays out all the environments that a given R session has available to it.

The search path is basically a list of the environments that R searches when an object is requested. If the object isn't found in one environment, R proceeds to the next one. You can view R's search path at any time using `search()`:

```
R> search()
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

From the command prompt, this path always begins at the global user environment (`.GlobalEnv`) and ends after the `base` package environment (`package:base`). These belong to a hierarchy: Imagine an arrow pointing from left to right between each pair of environments. For my current session, if I request a certain object at the R prompt, the program inspects `.GlobalEnv` ▸ `tools:RGUI` ▸ `package:stats` ▸ `...` ▸ `package:base` in turn, stopping the search when it has found and retrieved the desired object. This is known as *lexical scoping*. Note that, depending on your operating system and whether you're using the built-in GUI or some third-party editor, `tools:RGUI` might be different or not be included in your search path.

If R doesn't find what it's looking for by following the environments in the search path, it reaches the *empty* environment. While this

environment isn't listed explicitly in the output from `search()`, it's always the final destination after `package:base`. The empty environment marks the end of the search path.

For example, calling the following causes a number of things to happen internally:

```
R> baz <- seq(from=0,to=3,length.out=5)
R> baz
[1] 0.00 0.75 1.50 2.25 3.00
```

R first searches the global environment for a `seq` function, and when this isn't found, it goes on to search in the enclosing environment, which is the next level up in the search path (according to the left-to-right arrows mentioned earlier). It doesn't find it there, so R keeps going through the path to the next environment, searching the packages that have been loaded (automatically or otherwise) until it finds what it's looking for. In this example, R locates `seq` in the built-in `base` package environment. Then it executes the `seq` function (creating a temporary local environment) and assigns the results to a new object, `baz`, which resides in the global environment. In the subsequent call to print `baz`, R begins by searching the global environment and immediately finds the requested object.

You can look up the enclosing environment of any function using `environment`, as follows:

```
R> environment(seq)
<environment: namespace:base>
R> environment(arrows)
<environment: namespace:graphics>
```

Here, I've identified the package namespace of `base` as the "owner" of the `seq` function and the `graphics` package as the "owner" of the `arrows` function.

Each environment has a *parent* to direct the order of the search path under lexical scoping rules. For example, examining the earlier output from the call `search()`, the parent of `package:stats` is `package:graphics`. The

specific parent-child structure is dynamic in the sense that the search path changes when loading additional libraries or `attach`ing data frames. When you load a contributed package with a call to `library`, this inserts the desired package into the search path. For example, in Exercise 8.1 (page 183), you installed the contributed package `car`. After loading this package, your search path includes its contents:

```
R> library("car")
Loading required package: carData
R> search()
 [1] ".GlobalEnv"        "package:car"       "package:carData"
 [4] "tools:RGUI"        "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"   "Autoloads"         "package:base"
```

Note that the `car` package environment in the path is inserted directly after the global environment. This is where each subsequently loaded package will be placed (followed by any additional packages it depends on for its own functionality—note the automatic presence of the supporting package `carData`).

As noted earlier, R stops searching once it's exhausted the entire search path and reached the empty environment. If you request a function or object that you haven't defined, that doesn't exist, or that is perhaps in a contributed package that you've forgotten to load (this is quite common), then an error is thrown. These "cannot find" errors are recognizable for both functions and other objects:

```
R> neither.here()
Error in neither.here() : could not find function "neither.here"
R> nor.there
Error: object 'nor.there' not found
```

In summary, environments help compartmentalize the huge amount of functionality available to R. This becomes particularly important when there are functions with the same name in different packages in the search path. At that point, *masking*, discussed in Section 12.3, comes into play. As you get more comfortable with R and want more precise

control over how it operates, it's worth investigating in more detail how R handles environments. For more technical details on these topics, Chapter 7 of Wickham (2019) and Chapter 15 of Peng (2022) are particularly good. Gupta (2012) also provides an informative online article on the subject.

9.1.3 Reserved and Protected Names

A few key terms are strictly forbidden from being used as object names in R. These *reserved* names, or *keywords*, are necessary to protect fundamental operations and data types frequently used in the language.

The following identifiers are reserved:

- `if` and `else`
- `for`, `while`, and `in`
- `function`
- `repeat`, `break`, and `next`
- `TRUE` and `FALSE`
- `Inf` and `-Inf`
- `NA`, `NaN`, and `NULL`

I haven't yet covered some of the terms on this list. These items represent the core tools for programming in the R language, and you'll begin to explore them in the following chapter. The last three bullet points include the familiar logical values (Section 4.1) and special terms we use to represent things like infinity and missing entries (Section 6.1).

If you try to assign a new value to any reserved terms, an error occurs:

```
R> NaN <- 5
Error in NaN <- 5 : invalid (do_set) left-hand side to assignment
```

Because R is case sensitive, it's possible to assign values to any casevariant of the reserved names, but this can be confusing and is

inadvisable:

```
R> False <- "confusing"
R> nan <- "this is"
R> cat(nan,False)
this is confusing
```

Also be wary of assigning values to `T` and `F`, the abbreviations of `TRUE` and `FALSE`. The full identifiers `TRUE` and `FALSE` are reserved, but the abbreviated versions are not:

```
R> T <- 42
R> F <- TRUE
R> F&&TRUE
[1] TRUE
```

Assigning values to `T` and `F` this way will affect any subsequent code that intends to use `T` and `F` to refer to `TRUE` and `FALSE`. The second assignment (`F <-TRUE`) is perfectly legal to R's interpreter, but it's confusing given the normal usage of `F` as an abbreviation: The line `F&&TRUE` now represents a `TRUE&&TRUE` comparison! It's best to simply avoid these types of assignments.

If you've been following along with the examples in your R console, it's prudent at this point to clear the global environment (thereby deleting the objects `False`, `nan`, `T`, and `F` from your workspace). Use the trick of executing `rm(list=ls())` at the prompt to do so (see Section 2.2), which also removes all other present objects. Alternatively, remove them selectively like so:

```
R> rm(list=c("T","F","nan","False"))
```

Exercise 9.1

a. Identify the first 20 items in the built-in and automatically loaded `methods` package. How many items are there in total?

b. Determine the environment that owns each of the following functions:

   i. `read.table`

   ii. `data`

   iii. `matrix`

   iv. `jpeg`

c. Use `ls` and a character string equality test (see Section 4.2.1) to confirm the function `smoothScatter` is part of the `graphics` package.

9.2 Argument Matching

Another set of rules that determines how R interprets function calls has to do with *argument matching*. Argument matching conditions allow you to provide arguments to functions either with abbreviated names or without names at all.

9.2.1 Exact

So far, you've mostly been using *exact* matching of arguments, where you write out each argument tag in full. This is the most exhaustive way to call a function, but it's helpful when getting started with R.

Other benefits of exact matching include the following:

- Exact matching is less prone to misspecification of arguments than other matching styles.
- The order in which arguments are supplied doesn't matter.
- Exact matching is useful when a function has many possible arguments but you want to specify only a few.

The main drawbacks of exact matching are as follows:

- It can be cumbersome for relatively simple operations.
- Exact matching requires the user to remember or look up the full, case-sensitive tags.

As an example, in Section 6.2.1, you used exact matching to execute the following:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
                c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

This creates a 3×3 matrix object `bar` with a `dimnames` attribute for the rows and columns. Since the argument tags are fully specified, the order of the arguments doesn't matter. You could switch around the arguments, and the function still has all the information it requires:

```
R> bar <- matrix(nrow=3,dimnames=list(c("A","B","C"),c("D","E","F")),ncol=3,
                data=1:9)
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

This behaves the same as the previous function call and provides an identical result. For the sake of consistency, you usually won't switch around arguments each time you call a function, but this example shows a benefit of exact matching: you don't have to worry about the order of any optional arguments or about skipping them.

9.2.2 Partial

Partial matching lets you identify arguments with an abbreviated tag. This can shorten your code and it still lets you provide arguments in any order.

Let's call `matrix` in a way that takes advantage of partial matching:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,dat=1:9)
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Notice I've shortened the `nrow`, `dimnames`, and `ncol` argument tags to the first two letters and shortened the `data` argument to the first three. For partial matching, there's no set number of letters you have to provide, as long as each argument is still uniquely identifiable for the function being called. Partial matching has the following benefits:

- It requires less code than exact matching.
- Argument tags are still visible (which limits the possibility of misspecification).
- The order of supplied arguments still doesn't matter.

But partial matching also has some limitations. For one, it gets trickier if there are multiple arguments whose tags start with the same letters. Here's an example:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,d=1:9)
Error in matrix(nr = 3, di = list(c("A", "B", "C"), c("D", "E", "F")),  :
  argument 4 matches multiple formal arguments
```

An error has occurred. The fourth argument tag is designated as `d`, which is meant to stand for `data`. This is illegal because another argument, `dimnames`, also starts with `d`. Even though `dimnames` is specified separately as `di` earlier in the same line, the call isn't valid.

Drawbacks of partial matching include the following:

- The user must be aware of other potential arguments that can be matched by the shortened tag (even if they aren't specified in the call or have a default value assigned).
- Each tag must have a unique identification, which can be difficult to remember.
- Readability of the code by someone else is slightly more difficult, as any valid abbreviations are entirely the choice of the original author and may not look familiar to collaborators.

9.2.3 Positional

The most compact mode of function calling in R is *positional matching*. This is when you supply arguments without tags, and R interprets them based solely on their order.

We usually use positional matching for relatively simple functions with only a few arguments, or functions that are very familiar to the user. For this type of matching, you *must* be aware of the precise positions of each argument. You can find that information in the "Usage" section of the function's help file or print it to the console with the `args` function. Here's an example:

```
R> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
NULL
```

This shows the defined order of arguments of the `matrix` function, as well as the default value for each argument. To construct the matrix `bar` with positional matching, execute the following:

```
R> bar <- matrix(1:9,3,3,F,list(c("A","B","C"),c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

The benefits of positional matching are as follows:

- Shorter, cleaner code, particularly for routine tasks.
- No need to remember specific argument tags.

In creating the matrix `bar` using exact and partial matching, notice you didn't need to supply anything for the `byrow` argument, which, by default, is set to `FALSE`. However, with positional matching in this particular example you do (it's given in the fourth position in the most recent call to `matrix` as the abbreviated `F`) because R relies on position alone to interpret the function call. If you leave out the `F`, you get an error, as follows:

```
R> bar <- matrix(1:9,3,3,list(c("A","B","C"),c("D","E","F")))
Error in matrix(1:9, 3, 3, list(c("A", "B", "C"), c("D", "E", "F"))) :
  invalid 'byrow' argument
```

Here, R tries to assign the fourth argument (the list you intended for `dimnames`) as the value for the logical `byrow` argument. This brings us to the drawbacks of positional matching:

- You must look up and exactly match the defined order of arguments.
- Someone reading your code who isn't familiar with the functions where positional matching has been used will need to look these up to understand how the functions themselves are being used.

9.2.4 Mixed

Since each matching style has pros and cons, we can mix these three styles in a single function call.

For instance, you can avoid the type of error shown in the previous example like so:

```
R> bar <- matrix(1:9,3,3,dim=list(c("A","B","C"),c("D","E","F")))
R> bar
  D E F
```

```
A 1 4 7
B 2 5 8
C 3 6 9
```

Here, I've used positional matching for the first three arguments. At the same time, I've used partial matching to explicitly tell R that the list is meant as a `dimnames` value, not for `byrow`.

9.2.5 Ellipses

Many functions, including `c`, `data.frame`, and `list`, exhibit *variadic* behavior. They can accept any number of arguments, and it's up to the user to decide how many arguments to provide. When you call a function like `data.frame`, for example, you can specify any number of members (that is, column variables) as arguments.

In R, this behavior is signaled by the *dot-dot-dot* designation (…), also called the *ellipsis*. You can see whether an ellipsis is used in a function on its help page or with `args`. In `data.frame`, the first argument position is represented by an ellipsis:

```
R> args(data.frame)
function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
          fix.empty.names = TRUE, stringsAsFactors = FALSE)
NULL
```

When you call a function and supply an argument that can't be matched with one of the function's defined argument tags, this would normally produce an error. But if the function is defined with an ellipsis, any arguments that aren't matched to other argument tags are matched to the ellipsis.

Functions that employ ellipses generally fall into two groups. The first group includes functions like `c`, `data.frame`, and `list`, where the ellipsis represents the "main ingredients" of the function. That is, the primary objective of the function is to use contents of the ellipsis in the resulting object or output. The second group consists of functions where the ellipsis acts as a *supplementary* or *potential* repository of

optional arguments. This is common when the function of interest calls other *sub-functions* that themselves require additional arguments depending upon the originally supplied items. Rather than explicitly copy all the arguments desired by the sub-function into the argument list of the "parent" function, the parent function can instead be defined including an ellipsis that is subsequently provided to the sub-function.

Here's an example of the ellipsis used for supplementary arguments with the generic `plot` function:

```
R> args(plot)
function (x, y, ...)
NULL
```

Examining the arguments shows that the optional arguments such as point size (argument tag `cex`) or line type (argument tag `lty`), if supplied, are matched to the ellipsis. These optional arguments are then passed in to the function to be used by various methods that tweak graphical parameters.

Ellipses are a convenient programming tool for writing variadic functions or functions where an unknown number of arguments may be supplied. This will become clearer when you start writing your own functions in Chapter 11. However, when writing functions like this, it's important to properly document the intended use of … so potential users of the function know exactly which arguments can be passed to it and what those arguments are subsequently used for in execution.

Exercise 9.2

a. Use positional matching with `seq` (see Section 2.3.2) to create a sequence of values between –4 and 4 that progresses in steps of 0.2.

b. In each of the following lines of code, identify which style of argument matching is being used: exact, partial, positional, or mixed. If mixed, identify which arguments are specified in each style.

   i. `array(8:1,dim=c(2,2,2))`

   ii. `rep(1:2,3)`

   iii. `seq(from=10,to=8,length=5)`

   iv. `sort(decreasing=T,x=c(2,1,1,2,0.3,3,1.3))`

   v. `which(matrix(c(T,F,T,T),2,2))`

   vi. `which(matrix(c(T,F,T,T),2,2),a=T)`

c. Suppose you explicitly ran the plotting function `plot.default` and supplied values to arguments tagged `type`, `pch`, `xlab`, `ylab`, `lwd`, `lty`, and `col`. Use the function documentation to determine which of these arguments fall under the umbrella of the ellipsis.

9.3 Pipes

A relatively recent addition to the base R language is the *pipe* operator, denoted by `|>`. Available in versions 4.1.0+, the pipe operation permits the "chaining together" of several function calls whereby the result of one call is fed directly, or *piped*, into the next. Proponents of piping often say that code is more readable in a piped form.

9.3.1 Syntax and Examples

To see the basic usage of the pipe operator, let's start with a simple example. Another of R's built-in, ready-to-use datasets is `mtcars`, containing a number of descriptive details related to performance aspects of 32 classic automobiles (type `?mtcars` at the prompt to bring up the documentation and learn about the variables present).

Recall the `head` command (Section 5.2.1), an easy way to view the first `n` rows (defaulting to 6) of any data frame object. Traditional syntax would see you typing something like this:

```
R> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

For the piped version of this, we make the following call, where we start with the `mtcars` object and pipe it into `head`:

```
R> mtcars |> head()
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Notice that the `head` function, which requires at least one argument (the data frame upon which it is to operate), appears on the right-hand side of the most recent expression with *no* arguments. This makes sense only in a piped setting and highlights the most important thing to remember when piping: from left to right, the object being piped automatically fills the first argument position of the destination/target function.

To further highlight this behavior, suppose we wish to use a non-default value of `n` in a call to `head` in order to view the first three rows. The `n` argument takes the second position in `head`, and so using positional matching (Section 9.2.3) with traditional R syntax results in this:

```
R> head(mtcars,3)
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

So far, so good. With a pipe, however, take note of the following form:

```
R> mtcars |> head(3)
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

Here, we see the expression `head(3)`; the first argument has been accounted for via the pipe, and additional arguments to `head` are supplied starting from that which is defined second.

We usually use pipes to chain multiple commands or operations together. You'll often see them used in wrangling or summarizing data frame objects. As an example, recall the `colSums` function, briefly noted in Section 12.2.3 as the fastest way to take the sum of each column of a numeric matrix or data frame object. The related `colMeans` function finds the column-wise means. Suppose you now wished to find the mean horsepower (`hp`) and miles per gallon (`mpg`), in that order, for cars with an automatic transmission (identified through the `am` variable with value `0`). This is achievable without pipes using the explicit `subset` function (refer to Section 5.2.5) like so:

```
R> colMeans(subset(mtcars,am==0,c(hp,mpg)))
       hp       mpg
160.26316  17.14737
```

Note that many people would also use the square-bracket notation; a line like `colMeans(mtcars[mtcars$am==0,c("hp","mpg")])` would do the same thing.

With pipes, the previous code looks like this:

```
R> mtcars |> subset(am==0,c(hp,mpg)) |> colMeans()
       hp       mpg
160.26316  17.14737
```

The aforementioned behavior of piped results passing to the first argument of the follow-up functions is also clear here.

NOTE

The base R pipe operator `|>` *was motivated by an earlier implementation of the pipe in the contributed* `magrittr` *package (Bache & Wickham 2022), which takes the form* `%>%`*. For general use,* `|>` *has largely superseded* `%>%`*, but you'll still see people using the latter or come across older snippets of code featuring the* `magrittr` *pipe. The two versions of pipe mostly work in the same way, though there remain a few technical differences. For a good starting point on these topics, see Wickham et al. (2023).*

9.3.2 Should You Pipe?

The main difference between use of traditional R syntax and piped commands is that we interpret the former in an "inside-out" fashion (with the first operations taking place at the innermost position of function executions, and proceed outwards), with the latter operating in a "left-to-right" manner wherein the first operations take place on the left and proceed right.

Using pipes in your code is a personal choice and depends in part on what you typically use R for. If, for instance, you tend to use R for only subsetting, rearranging, or otherwise wrangling datasets, you may find the pipe operator useful if you prefer the way it allows you to write or present code. On the other hand, if you spend most of your time creating your own custom functions for bespoke tasks or for fitting complicated statistical models, piping might not offer as much value. The content of this book will stick to the traditional "non-piped" way of calling functions (though you'll see pipes briefly reappear in Section 24.4).

Either way, it's worth noting that anything you can achieve with the use of pipe operators you can also achieve without them, and there is no performance advantage one way or another. As you gain more experience with the R language, you'll settle on a coding style that best suits the types of applications you work on and what you find most readable.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`ls`	Inspect environment objects	Section 9.1.1, p. 195
`search`	Current search path	Section 9.1.2, p. 196
`environment`	Function environment properties	Section 9.1.2, p. 197
`args`	Show function arguments	Section 9.2.3, p. 202
`|>`	Pipe operator	Section 9.3.1, p. 205

10
CONDITIONS AND LOOPS



To write more sophisticated programs with R, you'll need to control the flow and order of execution in your code. One fundamental way to do this is to make the execution of certain sections of code dependent on some *condition*, always provided as a single logical value (`TRUE` or `FALSE`). Another basic control mechanism is the *loop*, which repeats a block of code a certain number of times. In this chapter, we'll explore these core programming techniques using `if...else` statements, `for` and `while` loops, and other control structures.

10.1 if Statements

The `if` statement is the key to controlling exactly which operations are carried out in a given chunk of code. An `if` statement runs a block of code only if a certain condition is met. In other words, these constructs allow a program to respond differently depending on whether a condition is `TRUE` or `FALSE`.

10.1.1 Stand-Alone Statement

Let's start with the stand-alone `if` statement, which looks something like this in R:

```
if(condition){
 do any code in here
}
```

Place the *`condition`* in parentheses after the `if` keyword. If *`condition`* evaluates to `TRUE`, the code in the braces (`{}`) will be executed. If *`condition`* isn't satisfied (that is, the result at evaluation is `FALSE`), the code in the braces is skipped, and R does nothing (or continues on to execute any code after the closing brace).

As a simple example, create the following two objects at the R prompt:

```
R> a <- 3
R> mynumber <- 4
```

Now, in the R editor, write the code chunk:

```
if(a<=mynumber){
    a <- a^2
}
```

What will the value of `a` be after executing this code? It depends on the condition defining the `if` statement, as well as what's actually specified in the braced area. In this case, evaluating the condition `a<=mynumber` results in `TRUE` since 3 is less than or equal to 4. That means the code inside the braces is executed, which sets `a` to `a^2`, equalling 9.

Highlight the entire chunk of code in the editor and send it to the console for execution using the normal keystroke shortcut for your system and R editor (refer to Section 1.3.2 for typical scenarios). Once it's executed, you'll see something like this:

```
R> if(a<=mynumber){
+   a <- a^2
+ }
```

The `if` statement is copied to the console, and each separate line after the first is prefaced by a `+`, which indicates that R is expecting more input before it begins execution. More generally, when a left brace is entered at the console prompt, the R interpreter won't actually start running any code until that section is closed with a right brace. To avoid redundancy, in future examples I won't show this repetition of code as it is shown in the console.

Now that the code has been run, look at the object `a`, shown here:

```
R> a
[1] 9
```

Unsurprisingly, it agrees with what we determined would happen, given the condition itself and the operation inside the braced area.

Suppose you now executed the same `if` statement again, right away. Will `a` be squared once more, giving 81? Nope! Since `a` is now stored as 9 and `mynumber` is still 4, the condition `a<=mynumber` will be `FALSE`, and the code in the braces will not be executed; `a` will remain at 9.

NOTE

You can change the `+` symbol by assigning a different character to the `continue` component of R's `options` command, in the way you saw how to reset the prompt in Section 1.3.1.

The `if` statement offers great flexibility; you can place any kind of code in the braced area, including more `if` statements (see the upcoming discussion of nesting in Section 10.1.4), enabling your program to make a sequence of decisions.

To illustrate a more complicated `if` statement, consider the following two new objects:

```
R> myvec <- c(2.73,5.40,2.15,5.29,1.36,2.16,1.41,6.97,7.99,9.52)
R> myvec
 [1] 2.73 5.40 2.15 5.29 1.36 2.16 1.41 6.97 7.99 9.52
```

```
R> mymat <- matrix(c(2,0,1,2,3,0,3,0,1,1),5,2)
R> mymat
     [,1] [,2]
[1,]    2    0
[2,]    0    3
[3,]    1    0
[4,]    2    1
[5,]    3    1
```

Use these two objects in the code chunk given here:

```
if(any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6){
    cat("Condition satisfied --\n")
    new.myvec <- myvec
    new.myvec[seq(1,9,2)] <- NA
    mylist <- list(aa=new.myvec,bb=mymat+0.5)
    cat("-- a list with",length(mylist),"members now exists.")
}
```

Sending this to the console produces the following output:

```
Condition satisfied --
-- a list with 2 members now exists.
```

Indeed, an object, `mylist`, has been created that you can examine:

```
R> mylist
$aa
[1]   NA 5.40   NA 5.29   NA 2.16   NA 6.97   NA 9.52

$bb
     [,1] [,2]
[1,]  2.5  0.5
[2,]  0.5  3.5
[3,]  1.5  0.5
[4,]  2.5  1.5
[5,]  3.5  1.5
```

In this example, the condition consisted of two parts separated by an OR statement using `||`, which produces a single logical result. Let's walk

through it:

1. The first part of the condition looks at `myvec`, takes `1` away from each element, and checks whether any of the results are greater than 9. If you run this part on its own, it yields `FALSE`:

```
R> myvec-1
[1] 1.73 4.40 1.15 4.29 0.36 1.16 0.41 5.97 6.99 8.52
R> (myvec-1)>9
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
R> any((myvec-1)>9)
[1] FALSE
```

2. The second part of the condition uses positional matching in a call to `matrix` to construct a two-row, five-column, column-filled matrix using entries of the original `myvec`. Then, it checks the number in the second row of the first column of that result to see whether it's less than or equal to 6, which it is:

```
R> matrix(myvec,2,5)
     [,1] [,2] [,3] [,4] [,5]
[1,] 2.73 2.15 1.36 1.41 7.99
[2,] 5.40 5.29 2.16 6.97 9.52
R> matrix(myvec,2,5)[2,1]
[1] 5.4
R> matrix(myvec,2,5)[2,1]<=6
[1] TRUE
```

This means the overall condition being checked by the `if` statement works out to be `FALSE||TRUE`, which evaluates as `TRUE`:

```
R> any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6
[1] TRUE
```

As a result, the code inside the braces is accessed and executed. First, it prints the `"Condition satisfied"` string and copies `myvec` to `new.myvec`. Using `seq`, it then accesses the odd-numbered indexes of `new.myvec` and overwrites them with `NA`. Next, it creates `mylist`. In this list, `new.myvec` is

stored in a member named `aa`, and then it takes the original `mymat`, increases all its elements by 0.5, and stores the result in `bb`. Lastly, it prints the length of the resulting list.

Note that `if` statements don't have to match the exact style I'm using here. For example, you may prefer to open the left brace on a new line after the condition, or use a different amount of indentation.

10.1.2 else Statements

The `if` statement executes a chunk of code if and only if a defined condition is `TRUE`. If you want something different to happen when the condition is `FALSE`, you can add an `else` declaration, which looks like this:

```
if(condition){
    do any code in here if condition is TRUE
} else {
    do any code in here if condition is FALSE
}
```

You set the condition, then in the first set of braces you place the code to run if the condition is `TRUE`. After this, you declare `else` followed by a new set of braces where you can place code to run if the condition is `FALSE`.

Let's return to the first example in Section 10.1.1, once more storing these values at the console prompt:

```
R> a <- 3
R> mynumber <- 4
```

In the editor, create a new version of the earlier `if` statement as follows:

```
if(a<=mynumber){
    cat("Condition was",a<=mynumber)
    a <- a^2
} else {
    cat("Condition was",a<=mynumber)
    a <- a-3.5
}
```

Here, you again square `a` if the condition `a<=mynumber` is `TRUE`, but if `FALSE`, `a` is overwritten by the result of itself minus 3.5. You also print text to the console stating whether the condition was met. Since `a` and `mynumber` have been reset to their original values, the first run of the `if` loop computes `a` as 9, just as earlier, outputting the following:

```
Condition was TRUE
R> a
[1] 9
```

Immediately highlight and execute the whole `if...else` statement again (without resetting `a` and `mynumber`). This time around, `a<=mynumber` will evaluate to `FALSE` and the code corresponding the `else` declaration will be run:

```
Condition was FALSE
R> a
[1] 5.5
```

10.1.3 Element-Wise Checks with ifelse

An `if` statement can check the condition of only a single logical value. If you pass in, for example, a vector of logicals for the condition, the `if` statement will terminate with an error:

```
R> if(c(FALSE,TRUE,FALSE,TRUE,TRUE)){}
Error in if (c(FALSE, TRUE, FALSE, TRUE, TRUE)) { :
  the condition has length > 1
```

However, the shortcut function `ifelse` can perform this kind of vector-oriented check for simple tasks. To demonstrate how it works, consider the objects `x` and `y` defined as follows:

```
R> x <- 5
R> y <- -5:5
R> y
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

Now, suppose you want to produce the result of `x/y` but with any instance of `Inf` (that is, any instance where `x` is divided by zero) replaced with `NA`. In other words, for each element in `y`, you want to check whether `y` is zero. If so, the code should output `NA`, and if not, it should output the result of `x/y`.

As you've just seen, a simple `if` statement won't work here. Since it accepts only a single logical value, it can't run through the entire logical vector produced by `y==0`:

```
R> y==0
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Instead, you can use the element-wise `ifelse` function:

```
R> result <- ifelse(test=y==0,yes=NA,no=x/y)
R> result
[1] -1.000000 -1.250000 -1.666667 -2.500000 -5.000000       NA  5.000000  2.500000
[9]  1.666667  1.250000  1.000000
```

Using exact matching, this command creates the desired `result` vector in one line. You must specify three arguments: `test` takes a logical-valued data structure, `yes` provides the element to return if the condition is satisfied, and `no` gives the element to return if the condition is `FALSE`. As noted in the function documentation (which you can access with `?ifelse`), the returned structure will be of "the same length and attributes as `test`."

Exercise 10.1

a. Create the following two vectors:

```
vec1 <- c(2,1,1,3,2,1,0)
vec2 <- c(3,8,2,2,0,0,0)
```

Without executing them, determine which of the following `if` statements would result in the string being printed to the console. Then confirm your answers in R.

i. `if((vec1[1]+vec2[2])==10){ cat("Print me!") }`

ii. `if(vec1[1]>=2&&vec2[1]>=2){ cat("Print me!") }`

iii. `if(all((vec2-vec1)[c(2,6)]<7)){ cat("Print me!") }`

iv. `if(!is.na(vec2[3])){ cat("Print me!") }`

b. With `vec1` and `vec2` from (a), use `ifelse` to multiply the two corresponding elements of the two vectors together *if* their sum is greater than 3. Otherwise, the code should simply sum the two elements.

c. In the editor, write a full `if...else` statement that takes a square character matrix (a matrix with an equal number of rows and columns containing character strings) and checks *if* any of the character strings on the diagonal (top left to bottom right) begin with the letter *g*, lowercase *or* uppercase. If satisfied, these specific entries should be overwritten with the string `"HERE"`. Otherwise, the entire matrix should be replaced with an identity matrix of the same dimensions. Then, try your code on the following matrices, checking the result each time:

i.
```
mymat <- matrix(as.character(1:16),4,4)
```

ii.
```
mymat <- matrix(c("DANDELION","Hyacinthus","Gerbera",
                  "MARIGOLD","geranium","ligularia",
                  "Pachysandra","SNAPDRAGON",
                  "GLADIOLUS"),3,3)
```

iii.
```
mymat <-
matrix(c("GREAT","exercises","right","here"),
               2,2,byrow=T)
```

> **Hint**: This requires some thought; you will find the functions `diag` from Section 3.1.2; `any` from Section 4.1.2; and `substr` from Section 4.2.4 useful.

10.1.4 Nested and Stacked Statements

You can place an `if` statement within the outcome of another `if` statement. By *nesting* or *stacking* several statements, you can weave intricate paths of decision-making by checking a number of conditions at various stages during execution.

We'll look at nesting first. In the editor, modify the `mynumber` example once more as follows:

```
if(a<=mynumber){
    cat("First condition was TRUE\n")
    a <- a^2
    if(mynumber>3){
        cat("Second condition was TRUE")
        b <- seq(1,a,length=mynumber)
    } else {
        cat("Second condition was FALSE")
        b <- a*mynumber
    }
} else {
    cat("First condition was FALSE\n")
    a <- a-3.5
    if(mynumber>=4){
        cat("Second condition was TRUE")
        b <- a^(3-mynumber)
    } else {
        cat("Second condition was FALSE")
        b <- rep(a+mynumber,times=3)
    }
}
```

Here, you see the same initial decision being made as earlier. The value `a` is squared if it's less than or equal to `mynumber`; if not, it has 3.5

subtracted from it. But now there's another `if` statement *nested* within each braced area. If the first condition is satisfied and `a` is squared, you then check whether `mynumber` is greater than 3. If `TRUE`, `b` is assigned `seq(1,a,length=mynumber)`. If `FALSE`, `b` is assigned `a*mynumber`.

If the first condition fails and you subtract 3.5 from `a`, then you check another sub-condition to see whether `mynumber` is greater than or equal to 4. If it is, then `b` becomes `a^(3-mynumber)`. If it isn't, `b` becomes `rep(a+mynumber, times=3)`. Note that I've indented the code within each subsequent braced area to make it easier to see which lines are relevant to each possible decision. While that doesn't affect execution, it makes things much more readable and is a good habit to develop.

Now reset `a <- 3` and `mynumber <- 4` either directly in the console or from the editor. When you run the example and inspect the values of the resulting objects, you'll get the following output:

```
First condition was TRUE
Second condition was TRUE
R> a
[1] 9
R> b
[1] 1.000000 3.666667 6.333333 9.000000
```

The result indicates exactly which code was invoked: The first condition and second condition were both `TRUE`. Trying another run of the same code, after first setting

```
R> a <- 6
R> mynumber <- 4
```

you see this output:

```
First condition was FALSE
Second condition was TRUE
R> a
[1] 2.5
R> b
[1] 0.4
```

This time the first condition fails, but the sub-condition checked inside the `else` statement is `TRUE`.

Alternatively, you could accomplish the same thing by sequentially *stacking* `if` statements and using a combination of logical expressions in each condition. In the following example, you check for the same four situations, but this time you stack `if` statements by placing a new `if` declaration immediately following an `else` declaration:

```
if(a<=mynumber && mynumber>3){
    cat("Same as 'first condition TRUE and second TRUE'")
    a <- a^2
    b <- seq(1,a,length=mynumber)
} else if(a<=mynumber && mynumber<=3){
    cat("Same as 'first condition TRUE and second FALSE'")
    a <- a^2
    b <- a*mynumber
} else if(mynumber>=4){
    cat("Same as 'first condition FALSE and second TRUE'")
    a <- a-3.5
    b <- a^(3-mynumber)
} else {
    cat("Same as 'first condition FALSE and second FALSE'")
    a <- a-3.5
    b <- rep(a+mynumber,times=3)
}
```

Just as before, only one of the four braced areas will end up being executed. Comparing this to the nested version, the first two braced areas correspond to what was originally the first condition (`a<=mynumber`) being satisfied, but this time you use `&&` to check two expressions at once. If neither of those two situations is met, this means the first condition is false, so in the third statement, you just have to check whether `mynumber>=4`. For the final `else` statement, you don't need to check any conditions because that statement will be executed only if all the previous conditions were not met.

If you again reset `a` and `mynumber` to 3 and 4, respectively, and execute the stacked statements shown earlier, you get the following result:

```
Same as 'first condition TRUE and second TRUE'
R> a
[1] 9
R> b
[1] 1.000000 3.666667 6.333333 9.000000
```

This produces the same values for `a` and `b` as earlier. If you execute the code again using the second set of initial values (`a` as 6 and `mynumber` as 4), you get the following:

```
Same as 'first condition FALSE and second TRUE'
R> a
[1] 2.5
R> b
[1] 0.4
```

This again matches the results of using the nested version of the code.

10.1.5 The switch Function

Say you need to choose which code to run based on the value of a single object (a common scenario). One option is to use a series of `if` statements, where you compare the object with various possible values to produce a logical value for each condition. Here's an example:

```
if(mystring=="Homer"){
    foo <- 12
} else if(mystring=="Marge"){
    foo <- 34
} else if(mystring=="Bart"){
    foo <- 56
} else if(mystring=="Lisa"){
    foo <- 78
} else if(mystring=="Maggie"){
    foo <- 90
```

```
} else {
    foo <- NA
}
```

The goal of this code is simply to assign a numeric value to an object `foo`, where the exact number depends on the value of `mystring`. The `mystring` object can take one of the five possibilities shown, or if `mystring` doesn't match any of these, `foo` is assigned `NA`.

This code works just fine as it is. For example, setting

```
R> mystring <- "Lisa"
```

and executing the chunk, you'll see this:

```
R> foo
[1] 78
```

Setting the following

```
R> mystring <- "Peter"
```

and executing the chunk again, you'll see this:

```
R> foo
[1] NA
```

This setup using `if...else` statements is quite cumbersome for such a basic operation, though. R can handle this type of multiple-choice decision in a far more compact form via the `switch` function. For example, you could rewrite the stacked `if` statements as a much shorter `switch` statement as follows:

```
R> mystring <- "Lisa"
R> foo <- switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,Maggie=90,NA)
R> foo
[1] 78
```

and

```
R> mystring <- "Peter"
R> foo <- switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,Maggie=90,NA)
R> foo
[1] NA
```

The first argument, `EXPR`, is the object of interest and can be either a numeric or a character string. The remaining arguments provide the values or operations to carry out based on the value of `EXPR`. If `EXPR` is a string, these argument tags must *exactly* match the possible results of `EXPR`. Here, the `switch` statement evaluates to 12 if `mystring` is `"Homer"`, 34 if `mystring` is `"Marge"`, and so on. The final, untagged value, `NA`, indicates the result if `mystring` doesn't match any of the preceding items.

The integer version of `switch` works in a slightly different way. Instead of using tags, the outcome is determined purely with positional matching. Consider the following example:

```
R> mynum <- 3
R> foo <- switch(mynum,12,34,56,78,NA)
R> foo
[1] 56
```

Here, you provide an integer `mynum` as the first argument, and it's positionally matched to `EXPR`. The example code then shows five untagged arguments: `12` to `NA`. The `switch` function simply returns the value in the specific position requested by `mynum`. Since `mynum` is 3, the statement assigns 56 to `foo`. Had `mynum` been 1, 2, 4, or 5, `foo` would've been assigned 12, 34, 78, or `NA`, respectively. Any other value of `mynum` (less than 1 or greater than 5) returns `NULL`:

```
R> mynum <- 0
R> foo <- switch(mynum,12,34,56,78,NA)
R> foo
NULL
```

In these types of situations, the `switch` function behaves the same as a set of stacked `if` statements, so it can serve as a convenient shortcut. However, if you need to examine multiple conditions at once or to

execute a more complicated set of operations based on this decision, you'll need to use the explicit `if` and `else` control structures.

Exercise 10.2

a. Write an explicit stacked set of `if` statements that does the same thing as the integer version of the `switch` function illustrated earlier. Test it with `mynum <- 3` and `mynum <- 0`, as in the text.

b. Suppose you are tasked with computing the precise dosage amounts of a certain drug in a collection of hypothetical scientific experiments. These amounts depend upon some predetermined set of "dosage thresholds" (`lowdose`, `meddose`, and `highdose`), as well as a predetermined dose level factor vector named `doselevel`. Look at the items (i) through (iv) after the following instructions to see the intended form of these objects. Write a set of nested `if` statements that produce a new numeric vector called `dosage`, according to the following rules:

   - First, *if* there are any instances of `"High"` in `doselevel`, perform the following operations:
     1. Check *if* `lowdose` is greater than or equal to 10. If so, overwrite `lowdose` with 10; *otherwise*, overwrite `lowdose` by itself divided by 2.
     2. Check *if* `meddose` is greater than or equal to 26. If so, overwrite `meddose` by 26.
     3. Check *if* `highdose` is less than 60. If so, overwrite `highdose` with 60; *otherwise*, overwrite `highdose` by itself multiplied by 1.5.

4. Create a vector named `dosage` with the value of `lowdose` repeated (`rep`; see Section 2.3.2) to match the `length` of `doselevel`.
5. Overwrite the elements in `dosage` corresponding to the index positions of instances of `"Med"` in `doselevel` by `meddose`.
6. Overwrite the elements in `dosage` corresponding to the index positions of instances of `"High"` in `doselevel` by `highdose`.

- *Otherwise* (in other words, if there are no instances of `"High"` in `doselevel`), perform the following operations:
    1. Create a new version of `doselevel`, a factor vector with levels `"Low"` and `"Med"` only, and label these with `"Small"` and `"Large"`, respectively (refer to Section 4.3 for details or see `?factor`).
    2. Check to see *if* `lowdose` is less than 15 AND `meddose` is less than 35. If so, overwrite `lowdose` by itself multiplied by 2 and overwrite `meddose` by itself plus `highdose`.
    3. Create a vector named `dosage`, which is the value of `lowdose` repeated (`rep`) to match the `length` of `doselevel`.
    4. Overwrite the elements in `dosage` corresponding to the index positions of instances of `"Large"` in `doselevel` by `meddose`.

Now confirm the following:

i. Given

```
lowdose <- 12.5
meddose <- 25.3
highdose <- 58.1
```

```
doselevel <- factor(c("Low","High","High","High","Low",
                      "Med","Med"),levels=c("Low",
                      "Med","High"))
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1] 10.0 60.0 60.0 60.0 10.0 25.3 25.3
```

ii. Using the same `lowdose`, `meddose`, and `highdose` thresholds as in (i), given

```
doselevel <- factor(c("Low","Low","Low","Med","Low",
                      "Med","Med"),levels=c("Low",
                      "Med","High"))
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1] 25.0 25.0 25.0 83.4 25.0 83.4 83.4
```

Also, `doselevel` has been overwritten as follows:

```
R> doselevel
[1] Small Small Small Large Small Large Large
Levels: Small Large
```

iii. Given

```
lowdose <- 9
meddose <- 49
highdose <- 61
doselevel <- factor(c("Low","Med","Med"),
                    levels=c("Low","Med","High"))
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1]  9 49 49
```

Also, `doselevel` has been overwritten as follows:

```
R> doselevel
[1] Small Large Large
Levels: Small Large
```

iv. Using the same `lowdose`, `meddose`, and `highdose` thresholds as (iii), as well as the same `doselevel` as (i), the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1]  4.5 91.5 91.5 91.5  4.5 26.0 26.0
```

c. Assume the object `mynum` will only ever be a single integer between 0 and 9. Use a `switch` statement inside a call to `ifelse` (the latter using the test condition `mynum>0`) to produce an overall command that takes in `mynum` and returns a matching character string for all possible values 0, 1, . . . , 9. Supplied with `3`, for example, it should return `"three"`; supplied with `0`, it should return `"zero"`.

10.2 for Loops

A *loop* is a core programming mechanism that repeats a specified section of code, typically while incrementing an index or counter. There are two main styles of looping: the `for` loop and the `while` loop. We'll start with the former, which is designed to repeat a given chunk of code as it works its way through a pre-specified vector, element by element.

10.2.1 Definition and Basics

The R `for` loop takes the following general form:

```
for(loopindex in loopvector){
   do any code in here
}
```

Here, the *loopindex* is a placeholder that represents an element in the *loopvector*; it starts off as the first element in the vector and moves to the next element with each loop repetition. When the `for` loop begins, it runs the code in the braced area, replacing any occurrence of the *loopindex* with the first element of the *loopvector*. When the loop reaches the closing brace, the *loopindex* increments, taking on the second element of the *loopvector*, and the braced area repeats. This continues until the loop reaches the final element of the *loopvector*, at which point the braced code executes for the final time, and the loop exits. That these placeholders must be specified means that you must know, or be able to easily calculate, the number of times the loop should repeat in order to use `for` loops.

Here's a simple example written in the editor:

```
for(myitem in 5:7){
    cat("--BRACED AREA BEGINS--\n")
    cat("the current item is",myitem,"\n")
    cat("--BRACED AREA ENDS--\n\n")
}
```

This loop prints the current value of the *loopindex* (which I've named `myitem` here) as it increments from 5 to 7. Here's the output after sending to the console:

```
--BRACED AREA BEGINS--
the current item is 5
--BRACED AREA ENDS--

--BRACED AREA BEGINS--
the current item is 6
--BRACED AREA ENDS--

--BRACED AREA BEGINS--
```

```
the current item is 7
--BRACED AREA ENDS--
```

You can use loops to manipulate objects that exist outside the loop. Consider the following example:

```
R> counter <- 0
R> for(myitem in 5:7){
+   counter <- counter+1
+   cat("The item in run",counter,"is",myitem,"\n")
+ }
The item in run 1 is 5
The item in run 2 is 6
The item in run 3 is 7
```

Here, I've initially defined an object, `counter`, and set it to zero in the workspace. Then, inside the loop, `counter` is overwritten by itself plus 1. Each time the loop repeats, `counter` increases, and the current value prints to the console.

Let's work through a more complicated example, this time based on real data. Recall that in Section 8.1.1 you briefly looked at the built-in `ChickWeight` data frame, a set of observations giving the weights of chicks recorded over time for chicks on four different diets. You can remind yourself of the structure of the data by using `str` (cf. Section 5.2.1) or by viewing the help file accessed by typing `?ChickWeight` at the prompt. Here's what the first six rows look like:

```
R> head(ChickWeight)
  Weight Time Chick Diet
1     42    0     1    1
2     51    2     1    1
3     59    4     1    1
4     64    6     1    1
5     76    8     1    1
6     93   10     1    1
```

Suppose you're interested in working out the average maximum weight of all the chicks. This is not immediately accessible from the

data object because each chick has multiple rows in the data frame (corresponding to the weight measurements taken at different points in time). While the help file for the data says that the chicks were measured up to day 21, not all chicks were able to be followed for that length of time (meaning their maximum recorded weights occurred at earlier times). This is important because it means we can't just extract the rows corresponding to `ChickWeight$Time==21` (doing so means we'd inadvertently omit some of the chicks from the calculation).

As such, I'll illustrate how you might use a `for` loop to help us compute the desired value. We'll do this by cycling through each unique chick identifier in turn, finding the maximum weight of that chick, and adding this to a running total. The running total will then be divided by the overall number of chicks, thus giving us an average.

First, we need the set of unique chick identifiers. Chick IDs are stored as a factor as the `Chick` column, and so we can use the `levels` function seen in Section 4.3.1 and apply it to the `Chick` column:

```
R> chick.ids <- levels(ChickWeight$Chick)
R> chick.ids
 [1] "18" "16" "15" "13" "9"  "20" "10" "8"  "17" "19" "4"  "6"  "11" "3"
[15] "1"  "12" "2"  "5"  "14" "7"  "24" "30" "22" "23" "27" "28" "26" "25"
[29] "29" "21" "33" "37" "36" "31" "39" "38" "32" "40" "34" "35" "44" "45"
[43] "43" "41" "47" "49" "46" "50" "42" "48"
R> length(chick.ids)
[1] 50
```

This stores the individual IDs (that is, the factor labels) as `chick.ids`; we can see there are 50 chicks in total. Making use of this object, consider the following code written in the R script editor:

```
chick.total <- 0
for(i in 1:length(chick.ids)){
    chicksub <- ChickWeight[ChickWeight$Chick==chick.ids[i],]
    maxweight <- max(chicksub$weight)
    chick.total <- chick.total + maxweight
}
chick.avg <- chick.total/length(chick.ids)
```

We first set up `chick.total` and initialize it to zero. Then the `for` loop begins, looping via the index position `i`. The first action in the loop is to create `chicksub`, a subset of the entire `ChickWeight` data made up of just those rows corresponding to the `i`th chick. Then, `maxweight` is computed as the maximum value of the `weight` column of these subsetted records. The third line overwrites `chick.total` by itself with `maxweight` added. When the loop increments, `chicksub` and `maxweight` are overwritten to correspond to the next entry in `chick.ids`, and `chick.total` thus maintains a running total of the maximum weight. Once the loop completes, we divide the final value of `chick.total` by the number of chicks, thereby giving us the desired average maximum weight. After sending the loop to the R console for evaluation, we see the following result of 206.66 grams:

```
R> chick.avg
[1] 206.66
```

10.2.2 Via Index or Value

The previous examples utilize slightly different approaches to the nature of the indexing performed by the `for` loops. There is a difference between using the *`loopindex`* to directly represent elements in the *`loopvector`* and using it to represent *indexes* of a vector. The following two loops use these two different approaches to `print` double each number in `myvec`:

```
R> myvec <- c(0.4,1.1,0.34,0.55)
R> for(i in myvec){
+   print(2*i)
+ }
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
R> for(i in 1:length(myvec)){
+   print(2*myvec[i])
+ }
```

```
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
```

The first loop uses the *loopindex* `i` to directly represent the elements in `myvec`, printing the value of each element multiplied by 2. In the second loop, on the other hand, you use `i` to represent integers in the sequence `1:length(myvec)`. These integers form all the possible index positions of `myvec`, and you use these indexes to extract `myvec`'s elements (once again multiplying each element by 2 and printing the result). Though it takes a slightly longer form, using vector index positions provides more flexibility in terms of how you can use the *loopindex*. This will become clearer when your needs demand more complicated `for` loops, such as in the next example.

Suppose you want to write some code that will inspect any list object and gather information about any matrix objects stored as members in the list. Consider the following list:

```
R> foo <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),cc=matrix(c(T,T,F,T,F,F),3,2),
               dd="string here",ee=matrix(c("red","green","blue","yellow")))
R> foo
$aa
[1] 3.4 1.0

$bb
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$cc
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,]  TRUE FALSE
[3,] FALSE FALSE
```

```
$dd
[1] "string here"

$ee
     [,1]
[1,] "red"
[2,] "green"
[3,] "blue"
[4,] "yellow"
```

Here, you've created `foo`, which contains three matrices of varying dimensions and data types. You'll write a `for` loop that goes through each member of a list like this one and checks whether the member is a matrix. If it is, the loop will retrieve the number of rows and columns and the data type of the matrix.

Before you write the `for` loop, you should create some vectors that will store information about the list members: `name` copies the list member names; `is.mat` will indicate whether each member is a matrix; `nc` and `nr` will store the number of rows and columns for each matrix; and `data.type` will store the data type of each matrix:

```
R> name <- names(foo)
R> name
[1] "aa" "bb" "cc" "dd" "ee"
R> is.mat <- rep(NA,length(foo))
R> is.mat
[1] NA NA NA NA NA
R> nr <- is.mat
R> nc <- is.mat
R> data.type <- is.mat
```

The `is.mat`, `nr`, `nc`, and `data.type` objects are all initialized as vectors of length `length(foo)` filled with `NA`s. Their entries will be updated position by position as the `for` loop is executed. Enter the following code into the editor:

```
for(i in 1:length(foo)){
    member <- foo[[i]]
```

```
    if(is.matrix(member)){
        is.mat[i] <- "Yes"
        nr[i] <- nrow(member)
        nc[i] <- ncol(member)
        data.type[i] <- class(as.vector(member))
    } else {
        is.mat[i] <- "No"
    }
}
bar <- data.frame(name,is.mat,nr,nc,data.type)
```

Initially, the *loopindex* `i` is set up to increment through the index positions of `foo` (the sequence `1:length(foo)`). In the braced code, the first command is to write the member of `foo` at position `i` to an object `member`. Next, you can check whether that member is a matrix using `is.matrix` (refer to Section 6.2.3). If `TRUE`, you do the following: Set the `i`th position of `is.mat` vector to `"Yes"`; set the `i`th element of `nr` and `nc` to the number of rows and number of columns of `member`, respectively; and set the `i`th element of `data.type` to the result of `class(as.vector(member))`. This final command first coerces the matrix into a vector with `as.vector` and then uses the `class` function (covered in Section 6.2.2) to find the data type of the elements therein. If `member` isn't a matrix and the `if` condition fails, the corresponding entry in `is.mat` is set to `"No"`, and the entries in the other vectors aren't changed (so they remain `NA`).

After the loop is complete, a data frame `bar` is created from the vectors. After executing the code, `bar` looks like this:

```
R> bar
  name is.mat nr nc data.type
1   aa     No NA NA      <NA>
2   bb    Yes  2  2   integer
3   cc    Yes  3  2   logical
4   dd     No NA NA      <NA>
5   ee    Yes  4  1 character
```

As you can see, this matches the nature of the matrices present in the list `foo`.

10.2.3 Nests

You can also nest `for` loops, just like `if` statements. When a `for` loop is nested in another `for` loop, the inner loop executes in full before the outer loop *`loopindex`* increments, at which point the inner loop executes all over again. Create the following objects in your R console:

```
R> loopvec1 <- 5:7
R> loopvec1
[1] 5 6 7
R> loopvec2 <- 9:6
R> loopvec2
[1] 9 8 7 6
R> foo <- matrix(NA,length(loopvec1),length(loopvec2))
R> foo
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

The following execution fills `foo` with the result of multiplying each integer in `loopvec1` by each integer in `loopvec2`:

```
R> for(i in 1:length(loopvec1)){
+   for(j in 1:length(loopvec2)){
+       foo[i,j] <- loopvec1[i]*loopvec2[j]
+   }
+ }
R> foo
     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   54   48   42   36
[3,]   63   56   49   42
```

Note that nested loops require a unique *`loopindex`* for each use of `for`. In this case, the *`loopindex`* is `i` for the outer loop and `j` for the inner loop. When the code is executed, `i` is first assigned `1`, the inner loop begins, and then `j` is also assigned `1`. The only command in the inner loop is to take the product of the `i`th element of `loopvec1` and the `j`th element of

`loopvec2` and assign it to row `i`, column `j` of `foo`. The inner loop repeats until `j` reaches `length(loopvec2)` and fills the first row of `foo`; then `i` increments, and the inner loop starts up again. The entire procedure is complete after `i` reaches `length(loopvec1)`, the inner loop is run one final time, and the matrix is filled.

You can even define inner *`loopvector`*s to match the current value of the *`loopindex`* of the outer loop. Using `loopvec1` and `loopvec2` from earlier, here's an example:

```
R> foo <- matrix(NA,length(loopvec1),length(loopvec2))
R> foo
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
R> for(i in 1:length(loopvec1)){
+   for(j in 1:i){
+       foo[i,j] <- loopvec1[i]+loopvec2[j]
+   }
+ }
R> foo
     [,1] [,2] [,3] [,4]
[1,]   14   NA   NA   NA
[2,]   15   14   NA   NA
[3,]   16   15   14   NA
```

In this code the `i`th row, `j`th column element of `foo` is filled with the sum of `loopvec1[i]` and `loopvec2[j]`. However, the inner loop values for `j` are now decided based on the "current" value of `i`. For example, when `i` is `1`, the inner *`loopvector`* is `1:1`, so the inner loop executes only once before returning to the outer loop. With `i` as `2`, the inner *`loopvector`* is then `1:2`, and so on. This makes it so each row of `foo` is only partially filled. Extra care must be taken when programming loops this way. Here, for example, the values for `j` depend on the length of `loopvec1`, so an error will occur if `length(loopvec1)` is greater than `length(loopvec2)`.

Any number of `for` loops can be nested, but the computational expense can become a problem if nested loops are used unwisely. Loops in general add some computational cost, so to produce more efficient code in R, you should always ask "Can I do this in a vector-oriented fashion?" Only when the individual operations are not possible or straightforward to achieve en masse should you explore an iterative, looped approach. You can find some relevant and valuable comments on R loops and associated best-practice coding in the "R Help Desk" article by Ligges & Fox (2008).

Exercise 10.3

a. In the interests of efficient coding, rewrite the nested loop example from this section, where the matrix `foo` was filled with the multiples of the elements of `loopvec1` and `loopvec2`, using only a single `for` loop (remember, arithmetic can be done easily in a vector-based fashion).

b. In Section 10.1.5, you used the command

```
switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,
       Maggie=90,NA)
```

to return a number based on the supplied value of a single character string. This line won't work if `mystring` is a vector of multiple character strings. Write some code that will take a character vector and return a corresponding vector of the appropriate numeric values. Test it on the following vector:

```
c("Peter","Homer","Lois","Stewie","Maggie","Bart")
```

c. In Section 10.2.1, you studied an example that used a `for` loop to calculate the average maximum weight (`chick.avg`) of

chicks using the built-in `ChickWeight` data frame. This example used an indexer `i`, which cycled through a *`loopvector`* made up to the *index positions* of the unique chick identifiers `chick.ids`. Use details from Section 10.2.1 to make small tweaks to this loop so that it calculates `chick.avg` with an indexer that *directly represents* the entries of `chick.ids`, not the index positions. Confirm you get the same result as in the example.

d. Suppose you have a list named `mylist` that can contain other lists as members, but assume those "member lists" or "sublists" cannot themselves contain lists (that is, the nesting can only go one deep). Write nested `for` loops that can search any possible `mylist` defined in this way and count how many matrices are present amongst all members.
Hint 1: In the outermost loop, use a conditional check to see whether a given member is itself a list. If it is, commence an inner loop that searches through the members of the sub-list looking for matrix objects.
Hint 2: Initialize a counter to zero before starting your nested loops; this counter should increment each time a matrix is found, regardless of whether it is a straightforward member of `mylist` or a member of a sub-list of `mylist`.

   Then confirm the following:

   i. The answer is 4 if you have the following:

```
mylist <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),
               cc=matrix(c(T,T,F,T,F,F),3,2),
               dd="string here",ee=list(c("hello",
               "you"),matrix(c("hello","there"))),
               ff=matrix(c("red","green","blue",
                           "yellow")))
```

   ii. The answer is 0 if:

```
mylist <- list("tricked you",as.vector(matrix(1:6,3,
                                              2)))
```

iii. The answer is 2 if:

```
mylist <- list(list(1,2,3),list(c(3,2),2),
               list(c(1,2),matrix(c(1,2))),
               rbind(1:10,100:91))
```

10.3 while Loops

In contrast to `for` loops, you typically deploy a `while` loop in situations where you cannot easily determine how many times the desired operations need to be run. Instead, we build the loop based solely on a logical-valued condition.

10.3.1 Definition and Basics

A `while` loop runs and repeats while a specified condition returns `TRUE`, and takes the following general form:

```
while(loopcondition){
   do any code in here
}
```

The loop proceeds as follows. Upon execution, the *`loopcondition`* is evaluated. If the condition is found to be `TRUE`, the braced area code is executed line by line as usual until complete, at which point the *`loopcondition`*is checked again. The loop terminates only when the condition evaluates to `FALSE`, and it does so immediately; the braced code is *not* run one last time.

This means the operations carried out in the braced area must somehow cause the loop to exit, either by affecting the *`loopcondition`* in some way or by declaring `break`, which you'll see a little later. If not, the loop will keep repeating forever, creating an *infinite loop*, which will freeze the console (and, depending on the operations specified inside

the braced area, R can crash because of memory constraints). If that occurs, you can terminate the loop in the R user interface by pressing ESC.

As a simple example of a `while` loop, consider the following code:

```
myval <- 5
while(myval<10){
    myval <- myval+1
    cat("\n'myval' is now",myval,"\n")
    cat("'mycondition' is now",myval<10,"\n")
}
```

Here, you set a new object `myval` to `5`. Then you start a `while` loop with the condition `myval<10`. Since this is `TRUE` to begin with, you enter the braced area. Inside the loop you increment `myval` by 1, print its current value, and print the logical value of the condition `myval<5`. The loop continues until the condition `myval<10` is `FALSE` at the next evaluation. Execute the code chunk, and you'll see the following:

```
'myval' is now 6
'mycondition' is now TRUE

'myval' is now 7
'mycondition' is now TRUE

'myval' is now 8
'mycondition' is now TRUE

'myval' is now 9
'mycondition' is now TRUE

'myval' is now 10
'mycondition' is now FALSE
```

As expected, the loop repeats until `myval` is set to `10`, at which point `myval<10` returns `FALSE`, causing the loop to exit because the initial condition is no longer `TRUE`.

10.3.2 Condition Object Modification

In more complicated settings, it's often useful to set the *loopcondition* to be a separate object so that you can modify it as necessary within the braced area. For the next example, you'll use a `while` loop to iterate through an integer vector and create an identity matrix (see Section 3.1.3) with the dimension matching the current integer. This loop should stop when it reaches a number in the vector that's greater than 5 or when it reaches the end of the integer vector.

In the editor, define some initial objects, followed by the loop itself:

```
mylist <- list()
counter <- 1
mynumbers <- c(4,5,1,2,6,2,4,6,6,2)
mycondition <- mynumbers[counter]<=5
while(mycondition){
    mylist[[counter]] <- diag(mynumbers[counter])
    counter <- counter+1
    if(counter<=length(mynumbers)){
        mycondition <- mynumbers[counter]<=5
    } else {
        mycondition <- FALSE
    }
}
```

The first object, `mylist`, will store any matrices that the loop creates. You'll use the vector `mynumbers` to provide the matrix sizes, and you'll use `counter` and `mycondition` to control the loop.

The *loopcondition*, `mycondition`, is initially set to `TRUE` since the first element of `mynumbers` is less than or equal to 5. Inside the loop, the first line uses double square brackets and the value of `counter` to dynamically create a new entry at that position in `mylist` (you did this earlier with named lists in Section 5.1.3). This entry is assigned an identity matrix whose size matches the corresponding element of `mynumbers`. Next, the `counter` is incremented, and now you have to update `mycondition`. Here, you want to check whether `mynumbers[counter]<=5`, but you also need to check whether you've reached the end of the integer vector (otherwise,

trying to retrieve an index position outside the range of `mynumbers` will lead to `mycondition` evaluating as `NA`, in turn causing `while` to throw an error). Use an `if` statement to first check the condition `counter<=length(mynumbers)`. If `TRUE`, then set `mycondition` to the outcome of `mynumbers[counter]<=5`. If not, this means you've reached the end of `mynumbers`, so make sure the loop exits by setting `mycondition <- FALSE`.

Execute the loop with those predefined objects, and it will produce the `mylist` object shown here:

```
R> mylist
[[1]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1

[[3]]
     [,1]
[1,]    1

[[4]]
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

As expected, you have a list with four members—identity matrices of size 4×4, 5×5, 1×1, and 2×2—matching the first four elements of `mynumbers`. The loop stopped executing when it reached the fifth element of `mynumbers` (`6`) since that's greater than 5.

Let's finish with another real-data example, revisiting the `ChickWeight` data frame we first experimented with in Section 10.2.1 (visit the help file or refer back to this section for an explanation of the data). Assume we're now interested in just those chicks on Diet 1. The help file for the dataset notes that the chicks are sorted in order of final weight (lightest to heaviest) within each of the four diets. Suppose we want to proceed through the chicks in this order, and stop when the average maximum weight is greater than 100 grams. A `while` loop is sensible here because we don't know in advance how many of the chicks will need to be included. First, we'll extract those unique chick IDs that correspond to just Diet 1:

```
R> chick.diet1 <- ChickWeight[ChickWeight$Diet=="1",]
R> chick.ids <- levels(droplevels(chick.diet1$Chick))
R> chick.ids
 [1] "18" "16" "15" "13" "9"  "20" "10" "8"  "17" "19" "4"  "6"  "11" "3"
[15] "1"  "12" "2"  "5"  "14" "7"
R> length(chick.ids)
[1] 20
```

Note we need to use the `droplevels` function (see Section 4.3.3) when extracting the unique chick IDs for Diet 1. Although `chick.diet1` is already a subset containing records of just the chicks on Diet 1, remember that a factor (which the `Chick` column is) will retain all levels as originally defined, even if there are zero records for some. Executing `droplevels` strips these zerocount levels away from the factor; a subsequent call to `levels` thus reveals only those that remain for Diet 1.

To achieve our objective, we'll progress through these 20 chick IDs in turn, calculating the average maximum weight after including each chick (that is, one additional chick at a time). Each time, we'll compute the average and halt when this average becomes greater than 100. Consider the following code in the R editor:

```
chick.total <- 0
chick.avg <- 0
chick.index <- 0
```

```
while(chick.avg<=100){
    chick.index <- chick.index + 1
    chicksub <- chick.diet1[chick.diet1$Chick==chick.ids[chick.index],]
    maxweight <- max(chicksub$weight)
    chick.total <- chick.total + maxweight
    chick.avg <- chick.total/chick.index
}
```

We define three objects—`chick.total`, `chick.avg`, and `chick.index`—setting them all to zero. The former two objects will store the running total maximum weight and average maximum weight. The latter will maintain the index value of the "current" chick as the loop progresses. The loop then starts, hinging on the condition `chick.avg<=100`. This is `TRUE` to begin with, since we initialized `chick.avg` to be zero. Entering the loop, the first operation increments the value stored in `chick.index` to 1. We then create `chicksub`, giving us just the values corresponding to the first chick in the `chick.diet1` data frame. The maximum weight of this chick is found and added to the running total. Using this running total, the "current" value of `chick.avg` is updated by dividing `chick.total` by `chick.index` (the latter representing how many chicks have contributed to the running total thus far). The condition is then checked again, and provided it remains `TRUE`, the operations repeat after `chick.index` increments to 2, and so on.

After the loop is sent to the R console and completes execution, we can inspect the final value stored as `chick.avg` (that is, the specific value that caused the *`loopcondition`* to fail):

```
R> chick.avg
[1] 103.4
```

By looking at `chick.index`, we can determine how far along the set of 20 `chick.ids` the loop got, and use our knowledge of vector subsetting from Section 2.3.3 to report the contributing chick IDs:

```
R> chick.index
[1] 10
```

```
R> chick.ids[1:chick.index]
[1] "18" "16" "15" "13" "9"  "20" "10" "8"  "17" "19"
```

This reveals that the average maximum chick weight grew to over 100 grams following inclusion of the 10th chick (with ID 19).

Exercise 10.4

a. The first example of Section 10.3.2 provided code for storing identity matrices in a list based on a certain condition. For this code, think about what the resulting `mylist` would look like for each of the following possible `mynumbers` vectors, without executing anything:

   i. `mynumbers <- c(2,2,2,2,5,2)`

   ii. `mynumbers <- 2:20`

   iii. `mynumbers <- c(10,1,10,1,2)`

   Then confirm your answers in R (note you'll have to reset the initial values of `mylist`, `counter`, and `mycondition` each time, just as in the text).

b. The *factorial* of a non-negative integer x, expressed as $x!$, yields x multiplied by the product of all integers less than x, down to 1. Formally, it is written like this:

$$\text{“}x \text{ factorial”} = x! = x \times (x - 1) \times (x - 2) \times \ldots \times 1$$

   Note there is the special case of *zero factorial*, which is always 1. That is:

$$0! = 1$$

For example, to work out 3 factorial, compute the following:

$$3 \times 2 \times 1 = 6$$

To work out 7 factorial, compute the following:

$$7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5,040$$

Write a `while` loop that computes and stores as a new object the factorial of any non-negative integer `mynum` by decrementing `mynum` by `1` at each repetition of the braced code. You can do this by defining a *`loopcondition`* that will be satisfied while `mynum` remains greater than 1.

Using your loop, confirm the following:

i. The result of using `mynum <- 5` is `120`

ii. Using `mynum <- 12` yields `479001600`

iii. Having `mynum <- 0` correctly returns `1`

c. Consider the following code, where the operations in the braced area of the `while` loop have been omitted:

```
mystring <- "R fever"
index <- 1
ecount <- 0
result <- mystring
while(ecount<2 && index<=nchar(mystring)){
    # several omitted operations #
}
result
```

Your task is to complete the code in the braced area so that it inspects `mystring` character by character until it reaches the second instance of the letter *e* or the end of the string, whichever comes first. The `result` object should be the entire character string if there is no second *e* or the

character string made up of all the characters up to, but not including, the second *e* if there is one. For example, `mystring <- "R fever"` should provide `result` as `"R fev"`. You must achieve this by following these operations in the braces:

1. Use `substr` (Section 4.2.4) to extract the single character of `mystring` at position `index`.
2. Use a check for equality to determine whether this single-character string is either `"e"` OR `"E"`. If so, increase `ecount` by 1.
3. Next, perform a separate check to see whether `ecount` is equal to 2. If so, use `substr` to set `result` equal to the characters between `1` and `index-1` inclusive.
4. Increment `index` by 1.

Test your code to confirm the previous `result` for `mystring <- "R fever"`. Furthermore, confirm the following:

- Using `mystring <- "beautiful"` provides `result` as `"beautiful"`
- Using `mystring <- "ECCENTRIC"` provides `result` as `"ECC"`
- Using `mystring <- "ElAbOrAte"` provides `result` as `"ElAbOrAt"`
- Using `mystring <- "eeeeek!"` provides `result` as `"e"`

10.4 Implicit Looping

Think of loops constructed using `for` and `while` as explicit, in the sense that you, the coder, directly defines the looping instructions for the R interpreter to follow. In some situations, especially for relatively routine loops (such as executing some function on each member of a list), you can avoid the need to specify some of the details associated with explicit

looping by using the `apply` suite of functions, which provide "loop-like" behavior in a short-hand, *implicit* fashion.

10.4.1 The apply Function

The `apply` function is the most basic form of implicit looping; it takes a function and applies it to a particular *margin* of an array (such as the rows or the columns).

For a simple illustrative example, we'll use the following matrix:

```
R> foo <- matrix(1:12,4,3)
R> foo
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Say you want to find the sum of each row. Calling the following just results in the grand total of all elements, which is not what you want:

```
R> sum(foo)
[1] 78
```

Instead, you could use a `for` loop like this one:

```
R> row.totals <- rep(NA,times=nrow(foo))
R> for(i in 1:nrow(foo)){
+   row.totals[i] <- sum(foo[i,])
+ }
R> row.totals
[1] 15 18 21 24
```

This cycles through each row and stores the sum in `row.totals`, but you can use `apply` to get the same result in a more compact form.

To call `apply`, you have to specify at least three arguments. The first argument, `X`, is the object you want to cycle through. The next argument, `MARGIN`, takes an integer that flags which margin of `X` to operate

on. Finally, FUN provides the function you want to perform on each margin. With the following call, you get the same result as the earlier for loop:

```
R> row.totals2 <- apply(X=foo,MARGIN=1,FUN=sum)
R> row.totals2
[1] 15 18 21 24
```

The MARGIN index follows the positional order of the dimension for matrices and arrays: 1 always refers to rows, 2 to columns, 3 to layers, 4 to blocks, and so on (see Chapter 3). To instruct R to sum each column of foo instead, simply change the MARGIN argument to 2:

```
R> apply(X=foo,MARGIN=2,FUN=sum)
[1] 10 26 42
```

NOTE

It is sub-optimal to take row or column sums using either an explicit or an implicit loop; this was done here for the sake of illustration. Instead, you should use the built-in R functions rowSums *and* colSums*, which are much faster.*

The operations supplied to FUN should be appropriate for the MARGIN selected. So, if you select rows or columns with MARGIN=1 or MARGIN=2, make sure the FUN function is appropriate for vectors. Or if you have a three-dimensional array and use apply with MARGIN=3, be sure to set FUN to a function appropriate for matrices. For a higher-dimensional example like that, let's use the following example array:

```
R> bar <- array(1:18,dim=c(3,3,2))
R> bar
, , 1

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

```
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

Then, make the following call:

```
R> apply(bar,3,FUN=diag)
     [,1] [,2]
[1,]    1   10
[2,]    5   14
[3,]    9   18
```

This extracts the diagonal elements of each of the matrix layers of `bar`. Each call to `diag` on a matrix returns a vector of length 3, and these vectors are returned and joined together as columns of a new matrix (here, you can see the first column of the result contains the three values along the diagonal of the first layer of `bar`, and the second column corresponds to those same entries in the second layer). The `FUN` argument can also be any appropriate user-defined function; you'll see specific examples of this in Chapter 11.

10.4.2 Some apply Variants

There are different flavors of the basic `apply` function, used for different specific purposes.

tapply

The `tapply` function performs operations on subsets of the object of interest, where those subsets are defined in terms of one or more factor vectors. As an example, let's return to the code from Section 8.2.3, which reads in a web-based data file on diamond pricing, sets

appropriate variable names of the data frame, and displays the first five records:

```
R> dia.url <- "http://jse.amstat.org/v9n2/4Cdata.txt"
R> diamonds <- read.table(dia.url)
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
R> diamonds[1:5,]
  Carat Color Clarity Cert Price
1  0.30     D     VS2  GIA  1302
2  0.30     E     VS1  GIA  1510
3  0.30     G    VVS1  GIA  1510
4  0.30     G     VS1  GIA  1260
5  0.31     D     VS1  GIA  1641
```

To add up the total value of the diamonds present for the full dataset but separated according to `Color`, you can use `tapply` like this:

```
R> tapply(diamonds$Price,INDEX=diamonds$Color,FUN=sum)
     D      E      F      G      H      I
113598 242349 392485 287702 302866 207001
```

This sums the relevant elements of the target vector `diamonds$Price`. The corresponding factor `diamonds$Color` is passed to `INDEX`, and the function of interest is specified with `FUN=sum` exactly as earlier.

lapply

Another particularly useful alternative is `lapply`, which operates member by member on a list. In Section 10.2, recall you wrote a `for` loop to inspect matrices in the following list:

```
R> baz <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),cc=matrix(c(T,T,F,T,F,F),3,2),
               dd="string here",ee=matrix(c("red","green","blue","yellow")))
```

Using `lapply`, you can check for matrices in the list with a single short line of code:

```
R> lapply(baz,FUN=is.matrix)
$aa
```

```
[1] FALSE

$bb
[1] TRUE

$cc
[1] TRUE

$dd
[1] FALSE

$ee
[1] TRUE
```

Note that no margin or index information is required for `lapply`; R knows to apply `FUN` to each member of the specified list. The returned value is itself a list.

sapply

Another variant, `sapply`, returns the same results as `lapply` but in the form of an array:

```
R> sapply(baz,FUN=is.matrix)
   aa    bb    cc    dd    ee
FALSE  TRUE  TRUE FALSE  TRUE
```

Here, the result is provided as a vector because each individual result (from `is.matrix`) is just a single element. In this example, `baz` also has a `names` attribute that is copied to the corresponding entries of the returned object (providing a *named* vector; see Section 2.3.1).

Other Variants

Other variants of `apply` include `vapply`, which is similar to `sapply` albeit with some relatively subtle differences, and `mapply`, which can operate on multiple vectors or lists at once. To learn more about `mapply`, see the `?mapply` help file; `vapply` and `sapply` are both covered in the `?lapply` help file.

10.4.3 When to Use Implicit Loops

All of R's `apply`-style functions allow for additional arguments to be passed to `FUN`; most of them do this via an ellipsis (Section 9.2.5). For example, take another look at the matrix `foo`:

```
R> apply(foo,1,sort,decreasing=TRUE)
     [,1] [,2] [,3] [,4]
[1,]    9   10   11   12
[2,]    5    6    7    8
[3,]    1    2    3    4
```

Here, you've applied `sort` to each row of the matrix and supplied the additional argument `decreasing=TRUE` to sort the rows from largest to smallest.

Some programmers prefer using the suite of `apply` functions wherever possible to improve the compactness and neatness of their code. However, note that these functions generally don't offer substantial improvement in terms of computational speed or efficiency over an explicit loop (this is particularly the case with modern versions of R), and they can even be slower. Plus, when you're first learning the R language, explicit loops can be easier to read and follow since the operations are laid out clearly line by line.

Exercise 10.5

a. Using `foo` from Section 10.4.1, continue on from the most recent example in the text and write an implicit loop that calculates the product (recall the function `prod` from Section 2.3.4) of all the column elements of the matrix returned by the call to `apply(foo,1,sort,decreasing=TRUE)`.

b. Convert the following `for` loop, which makes use of the matrix transpose function `t` (see Section 3.1.3) to an

implicit loop that does exactly the same thing:

```
matlist <- list(matrix(c(T,F,T,T),2,2),
                matrix(c("a","c","b","z","p","q"),3,2),
                matrix(1:8,2,4))
matlist
for(i in 1:length(matlist)){
    matlist[[i]] <- t(matlist[[i]])
}
matlist
```

c. In R, store the following 4×4×2×3 array:

```
qux <- array(96:1,dim=c(4,4,2,3))
```

This is a four-dimensional array consisting of three blocks, with each block being an array made up of two layers of 4×4 matrices.

Now do the following:

i. Write an implicit loop that obtains the diagonal elements of all second-layer matrices only to produce the following matrix:

```
     [,1] [,2] [,3]
[1,]   80   48   16
[2,]   75   43   11
[3,]   70   38    6
[4,]   65   33    1
```

ii. Write an implicit loop that returns the `dim`ensions of each of the three matrices formed by accessing the fourth column of every matrix in `qux`, regardless of layer or block, wrapped by another implicit loop that finds the row sums of that returned structure, resulting in the following vector:

```
[1] 12  6
```

d. Recall the example in Section 10.2.1, where a `for` loop was used to calculate the average maximum weight of chicks from the built-in `ChickWeight` dataset; this value was found to be 206.66 grams. Find this same value without using an explicit loop by following these steps:

   i. Use `tapply` with `FUN` set to `max` to obtain the maximum weight for each chick (using the chick ID as the "grouping" variable).

   ii. `sum` the values found in step (i).

   iii. Divide the value from step (ii) by the overall number of chicks. Ensure the result matches the one found in the earlier example.

10.5 Other Control Flow Mechanisms

To round off this chapter, you'll look at three more control flow mechanisms: `break`, `next`, and `repeat`. These mechanisms are often used in conjunction with the loops and `if` statements you've seen already.

10.5.1 break

Normally a `for` loop will exit only when the *`loopindex`* exhausts the *`loopvector`*, and a `while` loop exits only when the *`loopcondition`* evaluates to `FALSE`. But you can also preemptively terminate either a `for` or a `while` loop by declaring `break`.

For example, say you have a number, `foo`, that you want to divide by each element in a numeric vector `bar`:

```
R> foo <- 5
R> bar <- c(2,3,1.1,4,0,4.1,3)
```

Furthermore, say you want to divide `foo` by `bar` element by element but want to halt execution if one of the results evaluates to `Inf` (which results

when dividing by zero; see Section 6.1.1). To do this, you can check each iteration with the `is.finite` function (also seen in Section 6.1.1), and you can issue a `break` command to terminate the loop if it returns `FALSE`:

```
R> loop1.result <- rep(NA,length(bar))
R> loop1.result
[1] NA NA NA NA NA NA NA
R> for(i in 1:length(bar)){
+   temp <- foo/bar[i]
+   if(is.finite(temp)){
+       loop1.result[i] <- temp
+   } else {
+       break
+   }
+ }
R> loop1.result
[1] 2.500000 1.666667 4.545455 1.250000       NA       NA       NA
```

Here, you can see the loop divides the numbers normally until it reaches the fifth element of `bar` and divides by zero, resulting in `temp` becoming `Inf`. Upon the resulting conditional check, we thereby encounter `break` in the `else` part of the code and the loop ends immediately. The remaining entries of the storage vector `loop1.result` stay as they were originally set: `NA`s.

Invoking `break` in `for` or `while` can be considered a fairly drastic move. Often, a programmer will include it only as a safety catch that's meant to highlight or prevent unintended calculations. For more routine operations, it's best to use another method. For instance, you could easily replicate the example loop as a `while` loop (Section 10.3) or the vector-oriented `ifelse` function (Section 10.1.3), rather than relying on a `break`.

10.5.2 next

Instead of `break`ing and completely ending a `for` or `while` loop, you can use `next` to simply advance to the next iteration on demand and continue

execution. Consider the following, where using `next` avoids the aforementioned division by zero:

```
R> loop2.result <- rep(NA,length(bar))
R> loop2.result
[1] NA NA NA NA NA NA NA
R> for(i in 1:length(bar)){
+   if(bar[i]==0){
+       next
+   }
+   loop2.result[i] <- foo/bar[i]
+ }
R> loop2.result
[1] 2.500000 1.666667 4.545455 1.250000       NA 1.219512 1.666667
```

First, the loop checks to see whether the `i`th element of `bar` is zero. If it is, `next` is declared, and as a result, R ignores any subsequent lines of code in the braced area of the loop and returns to the top, automatically advancing to the next value of the *`loopindex`*. In the current example, the loop skips the fifth entry of `bar` (leaving the original `NA` value for that place) and continues through the rest of `bar`.

Note that if you use either `break` or `next` deep in a heavily nested loop, the command applies only to the loop in which it was called. In other words, only the specific loop using the declaration will exit or advance to the next iteration, and any others will continue as normal. For an example, let's return to the nested `for` loops from Section 10.2.3 that you used to fill a matrix with multiples of two vectors. This time you'll use `next` in the inner loop to skip certain values:

```
R> loopvec1 <- 5:7
R> loopvec1
[1] 5 6 7
R> loopvec2 <- 9:6
R> loopvec2
[1] 9 8 7 6
R> baz <- matrix(NA,length(loopvec1),length(loopvec2))
R> baz
```

```
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
R> for(i in 1:length(loopvec1)){
+   for(j in 1:length(loopvec2)){
+       temp <- loopvec1[i]*loopvec2[j]
+       if(temp>=54){
+           next
+       }
+       baz[i,j] <- temp
+   }
+ }
R> baz
     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   NA   48   42   36
[3,]   NA   NA   49   42
```

Here, we instruct the inner loop to skip to the `next` iteration if the product of the current elements is greater than or equal to 54. Note the effect applies *only to that innermost loop*—that is, only the `j` *`loopindex`* is preemptively incremented, while `i` is left untouched.

10.5.3 repeat

A lesser-known option for repeating a set of operations is the `repeat` statement. The general definition is simple:

```
repeat{
    do any code in here
}
```

Notice that a `repeat` statement doesn't include any kind of *`loopindex`* or *`loopcondition`*. To stop repeating the code inside the braces in an infinite loop, you must therefore make sure the operations will at some point cause the loop to reach a `break` declaration.

To see `repeat` in action, you'll use it to calculate the famous mathematical series: the *Fibonacci sequence*. The Fibonacci sequence is an infinite series of integers 0, 1, 1, 2, 3, 5, 8, 13, . . . where each term in the series is determined by the sum of the two previous terms. Formally, if F_n represents the nth Fibonacci number, then you have

$$F_n = F_{n-1} + F_{n-2}; \qquad n = 2, 3, 4, 5, \ldots$$

where:

$$F_0 = 0 \text{ and } F_1 = 1$$

The following `repeat` statement computes and prints (from F_2) the Fibonacci sequence, ending when it reaches a term greater than 150:

```
R> fib.a <- 0
R> fib.b <- 1
R> repeat{
+   temp <- fib.a+fib.b
+   fib.a <- fib.b
+   fib.b <- temp
+   cat(fib.b,", ",sep="")
+   if(fib.b>150){
+       cat("BREAK NOW...\n")
+       break
+   }
+ }
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

First, the sequence is initialized by storing the first two terms, 0 and 1, as `fib.a` and `fib.b`, respectively. Then, the `repeat` statement is entered, and it uses `fib.a` and `fib.b` to compute the next term in the sequence, stored as `temp`. Next, `fib.a` is overwritten to be `fib.b`, and `fib.b` is overwritten to be `temp` so that the two variables move forward through the series. That is, `fib.b` becomes the newly calculated Fibonacci number, and `fib.a` becomes the second-to-last number in the series so

far. Use of `cat` then prints the new value of `fib.b` to the console. Finally, a check is made to see whether the latest term is greater than 150; if it is, `break` is declared.

When you run the code, the braced area is repeated over and over until `fib.b` reaches the first number that is greater than 150, namely, 89 + 144 = 233. Once that happens, the `if` statement condition evaluates as `TRUE`, and R runs into `break`, terminating the loop.

The `repeat` statement is not as commonly used as the standard `while` or `for` loops, but it's useful if you don't want to be bound by specifying the *`loopindex`* and *`loopvector`* of a `for` loop or the *`loopcondition`* of a `while` loop. However, with `repeat`, you have to take a bit more care to prevent infinite loops by thinking about where a `break` declaration ought to be placed.

NOTE

For those already familiar with coding, `repeat` *can be used as a version of the "dowhile" mechanism explicitly found in some other programming languages. This concept is a slight tweak to the behavior the usual* `while` *loop covered in this chapter, used when the specified code needs to be executed once before the first evaluation of the* `loopcondition`*.*

Exercise 10.6

a. Using the same objects from Section 10.5.1,

```
foo <- 5
bar <- c(2,3,1.1,4,0,4.1,3)
```

do the following:

i. Write a `while` loop—*without* using `break` or `next`—that reaches exactly the same result as the `break`

example in Section 10.5.1. That is, produce the same vector as `loop1.result` in the text. To do this, define an external *`loopcondition`* object initialized to `TRUE` and an external counter variable initialized to 1. Inside the loop, you can use the counter variable as an indexer to evaluate each division and manually increment the counter, the *`loopcondition`* set based on the result of a call to `is.finite`.

ii. Obtain the same result as `loop2.result`, the example concerning `next`, using an `ifelse` function instead of a loop. To do this, you should base the `test` argument on a call to `is.finite`, evaluating the vectorized division of `foo` by `bar` and responding appropriately.

b. To demonstrate `while` loops in Section 10.3.2, you used the vector

```
mynumbers <- c(4,5,1,2,6,2,4,6,6,2)
```

to progressively fill `mylist` with identity matrices whose dimensions matched the values in `mynumbers`. The loop was instructed to stop when it reached the end of the numeric vector or a number that was greater than 5.

i. Write a `for` loop using a `break` declaration that does the same thing.

ii. Write a `repeat` statement that does the same thing. Here, you'll need to define an external counter that's manually incremented. To use `break`, you'll need to not only check each entry of `mynumbers` against the value 5 but also check whether the value of your counter has reached the end of the length of `mynumbers`.

c. Suppose you have two lists, `matlist1` and `matlist2`, both filled with numeric matrices as their members. Assume that all members have finite, non-missing values, but *do not* assume that the dimensions of the matrices are the same throughout. Write a nested pair of `for` loops that aim to create a result list, `reslist`, of all possible *matrix products* (refer to Section 3.1.3) of the members of the two lists according to the following guidelines:

- The `matlist1` object should be indexed/searched in the outer loop, and the `matlist2` object should be indexed/searched in the inner loop.
- You're interested only in the possible matrix products of the members of `matlist1` with the members of `matlist2` in that order.
- If a particular multiple isn't possible (that is, if the `ncol` of a member of `matlist1` doesn't match the `nrow` of a member of `matlist2`), then you should skip that multiplication, store the string `"not possible"` at the relevant position in `reslist`, and proceed directly to the `next` matrix multiplication.
- You can define a `counter` that is incremented at each comparison (inside the inner loop) to keep track of the current position of `reslist`.

Note, therefore, that the `length` of `reslist` will be equal to `length(matlist1)*length(matlist2)`. Now confirm the following results:

i. If you have

```
matlist1 <- list(matrix(1:4,2,2),matrix(1:4),
                 matrix(1:8,4,2))
matlist2 <- matlist1
```

then all members of `reslist` should be `"not possible"`, apart from members `[[1]]` and `[[7]]`.

ii. If you have

```
matlist1 <- list(matrix(1:4,2,2),matrix(2:5,2,2),
                 matrix(1:16,8,2))
matlist2 <- list(matrix(1:8,2,4),matrix(10:7,2,2),
                 matrix(9:2,4,2))
```

then the only `"not possible"` members of `reslist` should be `[[3]]`, `[[6]]`, and `[[9]]`.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`if( ){ }`	Conditional check	Section 10.1.1, p. 210
`if( ){ } else { }`	Check and alternative	Section 10.1.2, p. 213
`ifelse`	Element-wise `if...else` check	Section 10.1.3, p. 214
`switch`	Multiple `if` choices	Section 10.1.5, p. 219
`for( ){ }`	Iterative loop	Section 10.2.1, p. 223
`while( ){ }`	Conditional loop	Section 10.3.1, p. 232
`apply`	Implicit loop by margin	Section 10.4.1, p. 239

Function/operator	Brief description	First occurrence
`tapply`	Implicit loop by factor	Section 10.4.2, p. 240
`lapply`	Implicit loop by member	Section 10.4.2, p. 241
`sapply`	As `lapply`, array returned	Section 10.4.2, p. 241
`break`	Exit explicit loop	Section 10.5.1, p. 244
`next`	Skip to next loop iteration	Section 10.5.2, p. 245
`repeat{ }`	Repeat code until `break`	Section 10.5.3, p. 246

11
WRITING FUNCTIONS



Defining a function allows you to repeat execution of custom code without endlessly copying and pasting. It also allows other users to easily carry out the same computations on their own data or objects. In this chapter, you'll learn about writing your own R functions. You'll learn how to define and use arguments, return output from a function, and specialize your functions in other ways.

11.1 The function Command

To define a function, use the `function` command and assign the results to an object name. Once you've done this, you can call the function using that object name just like any other built-in or contributed function in the workspace. This section will walk you through the basics of function creation and discuss some associated issues, such as returning objects and specifying arguments.

11.1.1 Function Creation

A function definition always follows this general format:

```
functionname <- function(arg1,arg2,arg3,...){
   do any code in here when called
   return(returnobject)
}
```

The *functionname* placeholder can be any valid R object name, which is what you'll ultimately use to call the function. Assign the result of a call to `function`, which must itself be followed by parentheses encapsulating any arguments you want the function to have, to *functionname*. The schematic here includes three argument placeholders plus an ellipsis. Of course, the number of arguments, their tags, and whether to include an ellipsis all depend on the particular function you're defining. If the function doesn't require any arguments, simply use empty parentheses: `()`. If you do include arguments in this definition, note that they are not objects in the workspace and they do not have any type or class attributes associated with them; they are merely a declaration of argument names that will be used by *functionname*. This differs from some other programming languages that require formally defining the type of value of each argument alongside their tags. As such, it's important to have clear notes or documentation to say what the function expects these arguments to be, particularly if your function is intended for others to use.

Calling the function runs the code in the braced area (we typically refer to this as the *function body* or *body code*). It can include `if` statements, loops, and other function calls and should treat any defined arguments as available objects in the function's execution environment. When encountering an internal function call during execution, R follows the search rules discussed in Chapter 9.

Often, the function body will include one or more calls to the `return` command. When R encounters a `return` statement during execution, the function exits, returning control to the user at the command prompt. This mechanism allows you to pass results from operations in the function back to the user. This output is denoted in the aforementioned schematic by *returnobject*, which is typically assigned an object created or calculated earlier in the function body. If there is no `return` statement,

the function will attempt to return the object created by the last executed expression (I'll discuss this subtlety more in Section 11.1.2).

It's time for an example. Let's turn the Fibonacci sequence generator from Section 10.5.3 into a function in the editor:

```
myfib <- function(){
    fib.a <- 0
    fib.b <- 1
    cat(fib.a,", ",fib.b,", ",sep="")
    repeat{
        temp <- fib.a+fib.b
        fib.a <- fib.b
        fib.b <- temp
        cat(fib.b,", ",sep="")
        if(fib.b>150){
            cat("BREAK NOW...")
            break
        }
    }
}
```

I've named the function `myfib`, and it doesn't use or require any arguments. The body code is identical to the example in Section 10.5.3, except I've added the third line, `cat(fib.a,", ",fib.b,", ",sep="")`, to ensure the first two terms, 0 and 1, also print to the screen.

Before you can call `myfib` from the console, you have to send the function definition there. Highlight the code in the editor and press CTRL-R (CTRL-ENTER in RStudio) in Windows or ⌘-RETURN in macOS:

```
R> myfib <- function(){
+   fib.a <- 0
+   fib.b <- 1
+   cat(fib.a,", ",fib.b,", ",sep="")
+   repeat{
+       temp <- fib.a+fib.b
+       fib.a <- fib.b
```

```
+       fib.b <- temp
+       cat(fib.b,", ",sep="")
+       if(fib.b>150){
+           cat("BREAK NOW...")
+           break
+       }
+   }
+ }
```

This imports the function into the workspace (if you enter `ls()` at the command prompt, `"myfib"` now appears in the list of present objects). This step is required anytime you create or modify a function and want to use it from the command prompt.

Now you can call the function from the console:

```
R> myfib()
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

It computes and prints the Fibonacci sequence until it reaches a term greater than 150, just as instructed.

Adding Arguments

Rather than printing a fixed set of terms, let's add an argument to control how many Fibonacci numbers are printed. Consider the following new function, `myfib2`, with this modification:

```
myfib2 <- function(thresh){
    fib.a <- 0
    fib.b <- 1
    cat(fib.a,", ",fib.b,", ",sep="")
    repeat{
        temp <- fib.a+fib.b
        fib.a <- fib.b
        fib.b <- temp
        cat(fib.b,", ",sep="")
        if(fib.b>thresh){
            cat("BREAK NOW...")
            break
```

```
        }
    }
}
```

This version takes a single argument, `thresh`. In the body code, `thresh` acts as a threshold determining when to end the `repeat` procedure, halt printing, and complete the function; once a value of `fib.b` that is greater than `thresh` is calculated, the `repeat` statement will exit after encountering the call to `break`. Therefore, the output printed to the console will be the Fibonacci sequence up to and including the first `fib.b` value bigger than `thresh`. This means that `thresh` must be supplied as a single numeric value; supplying a character string, for example, would make no sense.

After importing the definition of `myfib2` into the console, note the same results as given by the original `myfib` when you set `thresh=150`:

```
R> myfib2(thresh=150)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

But now you can print the sequence to any limit you want (this time using positional matching to specify the argument):

```
R> myfib2(1000000)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
832040, 1346269, BREAK NOW...
```

Returning Results

To use the results of a function in future operations (rather than just printing output to the console), you need to return content to the user. Continuing with the current example, here's a Fibonacci function that stores the sequence in a vector and returns it:

```
myfib3 <- function(thresh){
    fibseq <- c(0,1)
    counter <- 2
    repeat{
        fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
```

```
        counter <- counter+1
        if(fibseq[counter]>thresh){
            break
        }
    }
    return(fibseq)
}
```

In `myfib3`, you first create the vector `fibseq` and assign it the first two terms of the sequence. This vector will ultimately become the *`returnobject`*(going by the schematic introduced at the beginning of this section). You also create a `counter` initialized to `2` to keep track of the current position in `fibseq`. Then the function enters a `repeat` statement, which overwrites `fibseq` with `c(fibseq,fibseq[counter-1]+fibseq[counter])`. That expression constructs a new `fibseq` by appending the sum of the most recent two terms to the contents of what is already stored in `fibseq`. For example, with `counter` starting at `2`, the first run of this line will sum `fibseq[1]` and `fibseq[2]`, appending the result as a third entry onto the original `fibseq`.

Next, `counter` is incremented, and the condition is checked. If the most recent value of `fibseq[counter]` is not greater than `thresh`, the loop repeats. If it *is* greater, the loop breaks, and you reach the final line of `myfib3`. Calling `return` ends the function and passes out the specified *`returnobject`* (in this case, the final contents of `fibseq`).

After importing `myfib3`, consider the following executions:

```
R> myfib3(150)
[1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233
R> foo <- myfib3(10000)
R> foo
 [1]     0     1     1     2     3     5     8    13    21    34    55    89
[13]   144   233   377   610   987  1597  2584  4181  6765 10946
R> bar <- foo[1:7]
R> bar
[1] 0 1 1 2 3 5 8
```

Here, the first line calls `myfib3` with `thresh` assigned `150`. The output is still printed to the screen, but this isn't the result of the `cat` command as it was earlier; it is the *`returnobject`* vector. You can assign this *`returnobject`* to a new variable in your workspace, done in the second line of code using the name `foo`, thereby making `foo` just another R object in the global environment that you can manipulate. For example, in the third line you use `foo` to create `bar` with a simple vector subset, selecting the first seven terms of the Fibonacci sequence (that is, F_0 through F_6 as per the definition on page 246). This wouldn't have been possible with either `myfib` or `myfib2` because these previous versions of the function didn't `return` a vector.

11.1.2 The return Statement

If there's no `return` statement inside a function, the function will end when the last line of the function body has been run, at which point it will return the most recently assigned or created object in the function. If nothing is created, such as in `myfib` and `myfib2` from earlier, the function returns `NULL`. To demonstrate this point, enter the following two dummy functions in the editor:

```
dummy1 <- function(){
    aa <- 2.5
    bb <- "string me along"
    cc <- "string 'em up"
    dd <- 4:8
}

dummy2 <- function(){
    aa <- 2.5
    bb <- "string me along"
    cc <- "string 'em up"
    dd <- 4:8
    return(dd)
}
```

The first function, `dummy1`, simply creates four different objects in its own execution environment (not the user's global environment) and doesn't explicitly return anything. On the other hand, `dummy2` creates the same objects and explicitly returns the last one, `dd`. If you import and run the two functions, both provide the same return object:

```
R> foo <- dummy1()
R> foo
[1] 4 5 6 7 8
R> bar <- dummy2()
R> bar
[1] 4 5 6 7 8
```

A function will end as soon as it evaluates a `return` command, without executing any remaining code in the function body. To emphasize this, consider one more version of the dummy function:

```
dummy3 <- function(){
    aa <- 2.5
    bb <- "string me along"
    return(aa)
    cc <- "string 'em up"
    dd <- 4:8
    return(bb)
}
```

Here, `dummy3` has two calls to `return`: one in the middle and one at the end. But when you import and execute the function, it returns only one thing:

```
R> baz <- dummy3()
R> baz
[1] 2.5
```

Executing `dummy3` returns only the object `aa` because only the first instance of `return` is executed and the function exits immediately at that point. Given the current definition of `dummy3`, there is no possible way for

the last three lines (the assignment of `cc` and `dd` and the `return` of `bb`) to ever be run.

Using `return` adds another function call to your code, so technically, it introduces a little extra computational expense. Because of this, some argue that you should avoid `return` statements unless absolutely necessary. But the additional computational cost of the call to `return` is small enough to be negligible for most purposes. Plus, `return` statements can improve the readability of code, making it easier to see where the author of a function intends it to complete and precisely what is intended to be supplied as output. I'll use `return` throughout the remainder of this work.

Exercise 11.1

a. Write another Fibonacci sequence function, naming it `myfib4`. This function should provide an option to perform either the operations available in `myfib2`, where the sequence is simply printed to the console, or the operations in `myfib3`, where a vector of the sequence is formally returned. Your function should take two arguments: The first, `thresh`, should define the limit of the sequence (just as in `myfib2` or `myfib3`); the second, `printme`, should be a logical value. If `TRUE`, then `myfib4` should just print; if `FALSE`, then `myfib4` should return a vector. You may write `myfib4` either explicitly using the body code of `myfib2` and `myfib3` or, alternatively, by using the existing definitions of `myfib2` and `myfib3` as helper functions. Confirm the correct results arise from the following calls:

   - `myfib4(thresh=150,printme=TRUE)`
   - `myfib4(1000000,T)`

- `myfib4(150,FALSE)`
- `myfib4(1000000,printme=F)`

b. In Exercise 10.4 on page 236, you were tasked with writing a `while` loop to perform integer factorial calculations.

   i. Using your factorial `while` loop (if you didn't do so earlier, use the version provided with the solutions for Exercise 10.4), write your own R function, `myfac`, to compute the factorial of an integer argument `int` (you may assume `int` will always be supplied as a non-negative integer). Perform a quick test of the function by computing 5 factorial, which is 120; 12 factorial, which is 479,001,600; and 0 factorial, which is 1.

   ii. Write another version of your factorial function, naming it `myfac2`. This time, you may still assume `int` will be supplied as an integer but not that it will be non-negative. If negative, the function should return `NaN`. Test `myfac2` on the same three values as previously, but also try using `int=-6`.

11.2 Arguments

Arguments are an essential part of R functions. In this section, you'll consider how R evaluates arguments. You'll see how to write functions that have default argument values, make functions handle missing argument values, and pass extra arguments into an internal function call with ellipses.

11.2.1 Considering Lazy Evaluation

An important concept related to handling arguments in many high-level programming languages is *lazy evaluation*. Generally, this refers to the

fact that expressions are evaluated only when they are needed. This applies to arguments in the sense that they are accessed and used only at the point they appear in the function body.

Let's see exactly how R functions recognize and use arguments during execution. As a working example we'll build on throughout this chapter, you'll write a function to search through a specified list for matrix objects and attempt to post-multiply each with another matrix specified as a second argument (refer back to the relevant part of Section 3.1.3, found on page 48, for details on matrix multiplication). The function will store and return the result in a new list. If no matrices are in the supplied list or if no appropriate matrices (given the dimensions of the multiplying matrix) are present, the function should return a character string informing the user of these facts. You can assume that any matrices present in the specified list are numeric.

We'll name our first version of this function `multiples1`:

```
multiples1 <- function(x,mat,str1,str2){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        return(str2)
    } else {
```

```
        return(result)
    }
}
```

This function takes four arguments. The target list to search is intended to be supplied to `x`; the post-multiplying matrix is supplied to `mat`; and two other arguments, `str1` and `str2`, will take character strings to return if `x` has no suitable members.

Inside the function body, the `sapply` implicit looping function applies the function `is.matrix` to each member of the list argument `x`, creating a vector called `matrix.flags`. The result is a logical vector of equal length to `x`, with `TRUE` elements where the corresponding member of `x` is in fact a matrix. If there are no matrices in `x`, the function hits a `return` statement, which exits the function and outputs the argument `str1`.

If the function did not exit at that point, this means matrices are present in `x`. The next step is to retrieve the matrix member indexes by applying `which` to `matrix.flags`. A `counter` is initialized to `0` to keep track of how many successful matrix multiplications are carried out, and an empty list (`result`) is created to store any results.

Next, you enter a `for` loop. For each member of `indexes`, the loop stores the matrix member at that position as `temp` and checks to see whether it's possible to perform post-multiplication of `temp` by the argument `mat` (to perform the operation, `ncol(temp)` must equal `nrow(mat)`). If the matrices are compatible, `counter` increments and this position of `result` fills with the relevant calculation. If `FALSE`, nothing is done. The indexer, `i`, then takes on the next value of `indexes` and repeats until completion.

The final procedure in `multiples1` checks whether the `for` loop actually found any compatible matrix products. If no compatibility existed, the braced `if` statement code inside the `for` loop would never have been executed, and the `counter` would remain set to zero. So, if `counter` is still equal to zero upon completion of the loop, the function simply returns the `str2` argument. Otherwise, if compatible matrices were found,

appropriate results will have been computed, and `multiples1` returns the `result` list, which would have at least one member.

It's time to import and then test the function. You'll use the following three list objects:

```
R> foo <- list(matrix(1:4,2,2),"not a matrix",
               "definitely not a matrix",matrix(1:8,2,4),matrix(1:8,4,2))
R> bar <- list(1:4,"not a matrix",c(F,T,T,T),"??")
R> baz <- list(1:4,"not a matrix",c(F,T,T,T),"??",matrix(1:8,2,4))
```

You'll set the argument `mat` to the 2×2 identity matrix (post-multiplying any appropriate matrix by this simply returns the original matrix), and you'll pass in appropriate string messages for `str1` and `str2`. Here's how the function works on `foo`:

```
R> multiples1(x=foo,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

The function returns `result` with the two compatible matrices of `foo` (members `[[1]]` and `[[5]]`). Now let's try it on `bar` using the same arguments:

```
R> multiples1(x=bar,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[1] "no matrices in 'x'"
```

This time, the value of `str1` is returned. The initial check identified that there are no matrices in the list supplied to `x`, so the function exits before beginning its `for` loop. Finally, let's try `baz`:

```
R> multiples1(x=baz,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[1] "matrices in 'x' but none of appropriate dimensions given 'mat'"
```

Here, the value of `str2` is returned. Though there is a matrix in `baz` and the `for` loop in the body of `multiples1` has been executed, the matrix is not compatible for post-multiplication by `mat`.

Notice that the string arguments `str1` and `str2` are used only when the argument `x` does not contain a matrix with the appropriate dimensions. When you applied `multiples1` to `x=foo`, for example, there was no need to use `str1` or `str2`. R evaluates the defined expressions lazily, dictating that argument values are sought or evaluated only at the moment they are required during execution. In this function, `str1` and `str2` are required only when the input list doesn't have suitable matrices so, although it's not good practice, you could lazily ignore providing values for these arguments when `x=foo` like so:

```
R> multiples1(x=foo,mat=diag(2))
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

This returns the same results as before with no problem. Attempting this with `bar`, on the other hand, doesn't work:

```
R> multiples1(x=bar,mat=diag(2))
Error in multiples1(x = bar, mat = diag(2)) :
  argument "str1" is missing, with no default
```

Here, we are quite rightly chastised by R because it requires the value for `str1`. It informs us that the value is missing and there is no prior default assigned.

11.2.2 Setting Defaults

The previous example shows one case where it's useful to set default values for certain arguments. Default argument values are also sensible in many other situations, such as when the function has a large number of arguments or when arguments have natural values that are used more often than not. Let's write a new version of the `multiples1` function from Section 11.2.1, `multiples2`, which now includes default values for `str1` and `str2`:

```
multiples2 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        return(str2)
```

```
    } else {
        return(result)
    }
}
```

Here, you give `str1` a default value of `"no valid matrices"` by assigning the string value in the formal definition of the arguments. You also set a default for `str2` by assigning `str1` to it. If you import and execute this function on the three lists `foo`, `bar`, and `baz` from Section 11.2.1, you no longer need to explicitly provide values for those arguments:

```
R> multiples2(foo,mat=diag(2))
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

R> multiples2(bar,mat=diag(2))
[1] "no valid matrices"
R> multiples2(baz,mat=diag(2))
[1] "no valid matrices"
```

In other words, you can now call the function without being required to supply every argument in full because if any user-unspecified arguments *are* needed, their defaults will be used. If you don't want to use the default arguments in a specific call, you can still specify different values for those arguments when calling the function and those values will overwrite the defaults.

11.2.3 Checking for Missing Arguments

The `missing` function checks the arguments of a function to see if all required arguments have been supplied. It takes an argument tag and returns a single logical value of `TRUE` if the specified argument isn't found. You can use `missing` to avoid the error you saw in an earlier call to `multiples1` using `x=bar` (when `str1` ended up being needed but hadn't been supplied).

In some situations, the `missing` function can be particularly useful in the body code. Consider another modification to the example function:

```
multiples3 <- function(x,mat,str1,str2){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        if(missing(str1)){
            return("'str1' was missing, so this is the message")
        } else {
            return(str1)
        }
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        if(missing(str2)){
            return("'str2' was missing, so this is the message")
        } else {
            return(str2)
        }
    } else {
```

```
        return(result)
    }
}
```

The only differences between this version and `multiples1` are in the first and last `if` statements. The first `if` statement checks whether there are no matrices in `x`, in which case it returns a string message. In `multiples1`, that message was always `str1`, but now you use *another* `if` statement with `missing(str1)` to see whether the `str1` argument actually has a value first. If not, the function returns another character string saying that `str1` was `missing`. A similar alternative is defined for `str2`. After importing the function, here are the results of `multiples3` using `foo`, `bar`, and `baz`:

```
R> multiples3(foo,diag(2))
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

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

R> multiples3(bar,diag(2))
[1] "'str1' was missing, so this is the message"
R> multiples3(baz,diag(2))
[1] "'str2' was missing, so this is the message"
```

Using `missing` this way permits arguments to be left unsupplied in a given function call. It's primarily used when it's difficult to choose a default value for a certain argument, yet the function still needs to be able to sensibly deal with situations where that argument is needed but not provided by the user. In the current example, it makes more sense to

define defaults for `str1` and `str2`, as you did for `multiples2`, and avoid the extra code that implementation of `missing` requires.

11.2.4 Dealing with Ellipses

In Section 9.2.5, I introduced the ellipsis, also referred to as the "dot-dot-dot" notation. The ellipsis allows you to pass in extra arguments without having to first define them in the argument list, and these arguments can then be passed to another function call within the code body. When included in a function definition, the ellipsis is often (but not always) placed in the last position because it represents a variable number of arguments.

Building on the `myfib3` function from Section 11.1.1, let's use the ellipsis to write a function that can plot the specified Fibonacci numbers:

```
myfibplot <- function(thresh,plotit=TRUE,...){
    fibseq <- c(0,1)
    counter <- 2
    repeat{
        fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
        counter <- counter+1
        if(fibseq[counter]>thresh){
            break
        }

    }

    if(plotit){
        plot(0:(length(fibseq)-1),fibseq,...)
    } else {
        return(fibseq)
    }
}
```

In this function, an `if` statement checks to see whether the `plotit` argument is `TRUE` (which is the default value). If so, then you call `plot`, passing in `0:(length(fibseq)-1)` for the x-axis coordinates (recall as per the

definitions on page 246 that we number the terms starting from zero) and the Fibonacci numbers themselves for the y-axis. After these coordinates, you also pass the ellipsis directly into `plot`. In this case, the ellipsis represents any additional arguments a user might pass in to control the execution of `plot`.

After importing `myfibplot` and executing the following line, the plot in Figure 11-1 pops up in a graphics device:

```
R> myfibplot(150)
```

Here, you used positional matching to assign `150` to `thresh`, leaving the default value for the `plotit` argument. The ellipsis is empty in this call.

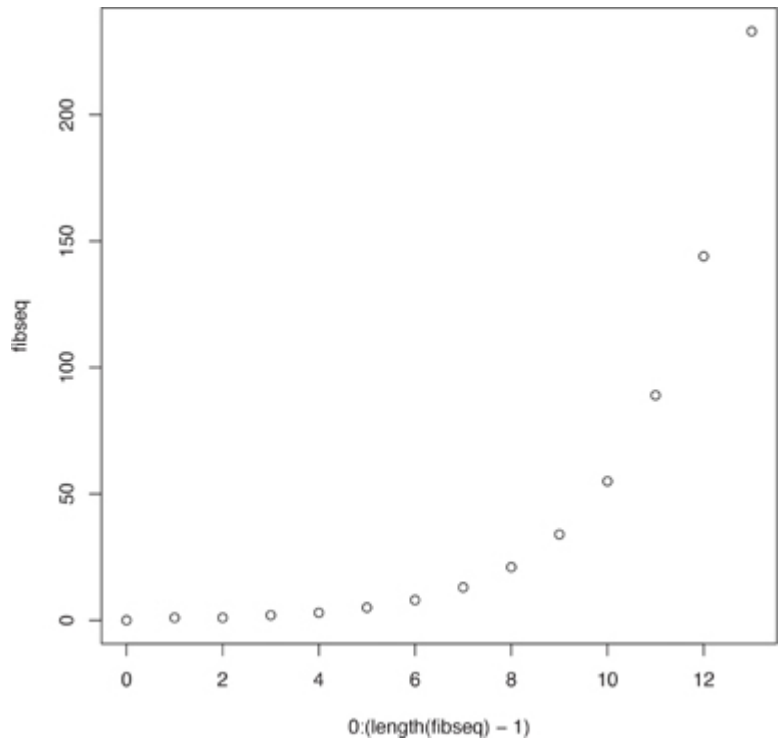


Figure 11-1: The default plot produced by a call to `myfibplot`, with `thresh=150`

Since you didn't specify otherwise, R simply follows the default behavior of `plot`. You can spruce things up by specifying more plotting options. The following line produces the plot in Figure 11-2:

```
R> myfibplot(150,type="b",pch=4,lty=2,main="Terms of the Fibonacci sequence",
             ylab="Fibonacci number",xlab="Term (n)")
```

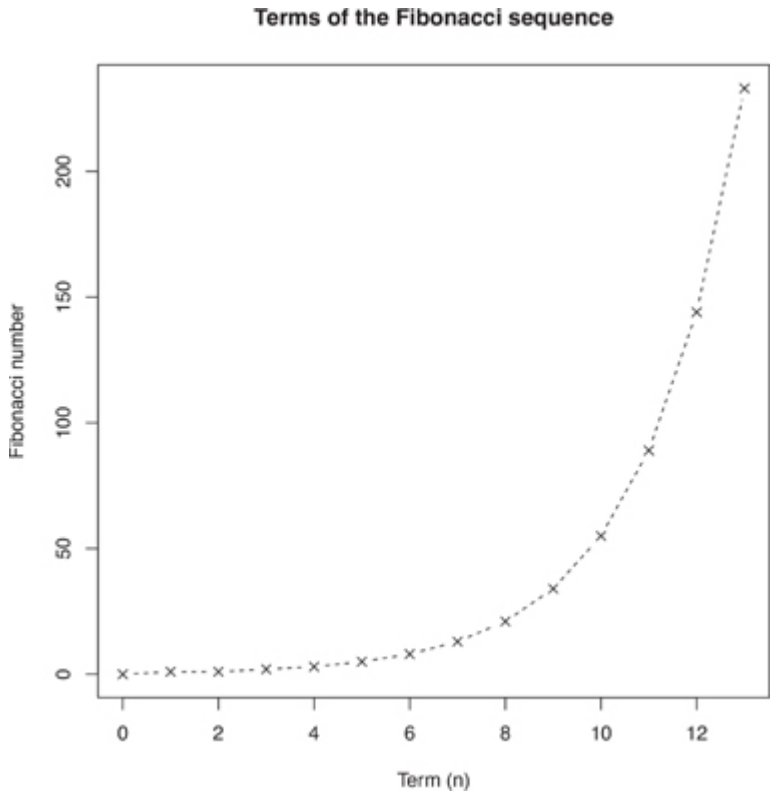


Figure 11-2: A plot produced by a call to `myfibplot` *with graphical parameters passed in using the ellipsis*

Here, the ellipsis allows you to pass arguments to `plot` through the call to `myfibplot`, even though the particular graphical parameters aren't explicitly defined arguments of `myfibplot`.

Ellipses can be convenient, but they require care. The ambiguous … can represent any number of mysterious arguments. Good function documentation is key to indicate the intended usage.

If you want to unpack the arguments passed in through an ellipsis, you can use the `list` function to convert those arguments into a list. Here's an example:

```
unpackme <- function(...){
    x <- list(...)
    cat("Here is ... in its entirety as a list:\n")
```

```
    print(x)
    cat("\nThe names of ... are:",names(x),"\n\n")
    cat("\nThe classes of ... are:",sapply(x,function(y){class(y)[1]}))
}
```

This toy function converts an ellipsis to a list with `x <- list(...)`. This subsequently allows the object `x` to be treated the same way as any other list. In this case, you can summarize the object by providing its `names` and `class` attributes. Notice the use of an anonymous function (Section 11.3.2) in the application of `sapply` (Section 10.4.2) to ensure only one—the first—class identifier of each member of `x` is obtained (this will affect things like matrices; see Section 6.2.2). Here's a sample run:

```
R> unpackme(aa=matrix(1:4,2,2),bb=TRUE,cc=c("two","strings"),
            dd=factor(c(1,1,2,1)))
Here is ... in its entirety as a list:
$aa
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$bb
[1] TRUE

$cc
[1] "two"     "strings"

$dd
[1] 1 1 2 1
Levels: 1 2


The names of ... are: aa bb cc dd


The classes of ... are: matrix logical character factor
```

Four tagged arguments—`aa`, `bb`, `cc`, and `dd`—are provided as the contents of the ellipsis, and they are explicitly identified within `unpackme` by using the simple `list(...)` operation. This construction can be useful for identifying or extracting specific arguments supplied through … in a given call.

Exercise 11.2

a. Accruing annual compound interest is a common financial benefit for investors. Given a principal investment amount P, an interest rate per annum i (expressed as a percentage), and a frequency of interest paid per year t, the final amount F after y years is given as follows:

$$F = P\left(1 + \frac{i}{100t}\right)^{ty}$$

Write a function that can compute F as per the following notes:

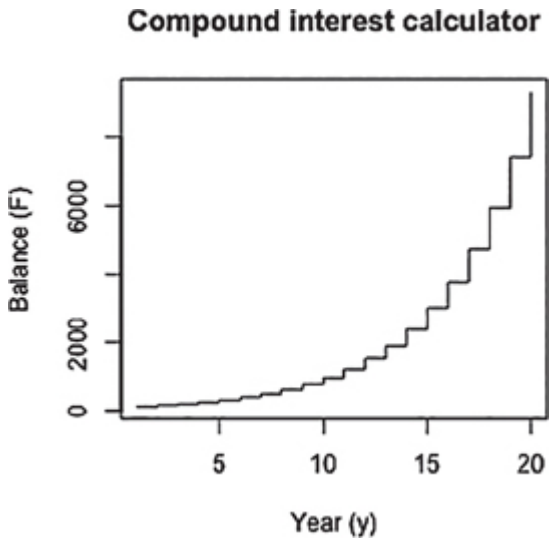
- Arguments must be present for P, i, t, and y. The argument for t should have a default value of 12.
- Another argument giving a logical value that determines whether to `plot` the amount F at each integer time should be included. For example, if `plotit=TRUE` (the default) and y is 5 years, the plot should show the amount F at y = 1, 2, 3, 4, 5.
- If this function is plotted, the plot should always be a step-plot, so `plot` should always be called with `type="s"`.
- If `plotit=FALSE`, the final amount F should be returned as a numeric vector corresponding to the same integer times, as shown earlier.

- An ellipsis should also be included for the sole reason of passing additional arguments to the call to `plot` if the user desires.

Now, using your function, do the following:

i. Work out the final amount after a 10-year investment of a principal of $5000, at an interest rate of 4.4 percent per annum compounded monthly.

ii. Re-create the following step-plot, which shows the result of $100 invested at 22.9 percent per annum, compounded monthly, for 20 years:



iii. Perform another calculation based on the same parameters as in (ii), but this time, assume the interest is compounded annually. Return and store the results as a numeric vector. Then, use `lines` to add a second step-line, corresponding to this annually accrued amount, to the plot created previously (the argument `type="s"` will work in a call to `lines`). Use a different color or line type and make use of the `legend` function to distinguish between the two lines.

b. We typically express a *quadratic equation* in the variable x in the following form:

$$k_1x^2 + k_2x + k_3 = 0$$

Here, k_1, k_2, and k_3 are constants. Given values for these constants, you can attempt to find up to two *real roots*—values of x that satisfy the equation. Write a function that takes k_1, k_2, and k_3 as arguments and finds and returns any solutions (as a numeric vector) in such a situation. This is achieved as follows:

- Evaluate $k_2^2 - 4k_1k_3$. If this is negative, there are no solutions, and the function should simply print an appropriate message to the console.
- If $k_2^2 - 4k_1k_3$ is 0, then there is one solution, computed by $-k_2 / 2k_1$.
- If $k_2^2 - 4k_1k_3$ is positive, then there are two solutions, given by $\left(-k_2 - \left(k_2^2 - 4k_1k_3\right)^{0.5}\right)/2k_1$ and $\left(-k_2 + \left(k_2^2 - 4k_1k_3\right)^{0.5}\right)/2k_1$.
- No default values are needed for the three arguments, but the function should check to see whether any are missing. If so, an appropriate character string message should be returned to the user, informing the user that the calculations are not possible.

Now test your function.

i. Confirm the following:

- $2x^2 - x - 5$ has roots –1.350781 and 1.850781.

- $x^2 + x + 1$ has no real roots.

ii. Attempt to find solutions to the following quadratic equations:

- $1.3x^2 - 8x - 3.13$
- $2.25x^2 - 3x + 1$
- $1.4x^2 - 2.2x - 5.1$
- $-5x^2 + 10.11x - 9.9$

iii. Test your programmed response in the function if one of the arguments is missing.

11.3 Specialized Functions

In this section, you'll study three kinds of specialized user-defined functions. First, you'll look at helper functions, which are designed to be called multiple times by another function (and they can even be defined inside the body of a parent function). Next, you'll learn about anonymous functions, functions without a name that can be directly defined as an argument to another function call. Finally, you'll look at recursive functions, which call themselves during execution.

11.3.1 Helper Functions

It is common for R functions to call other functions from within their body. A *helper function* is a general term we use to describe functions written and used specifically to facilitate the computations carried out by another function. They're a good way to improve the readability of complicated functions.

You can define a helper function either internally (within a function definition itself) or externally (within another environment, typically the global environment). In this section, you'll see an example of each.

Externally Defined

Building on the `multiples2` function from Section 11.2.2, here's a new version that splits the functionality over two separate functions, one of which is an externally defined helper function:

```
multiples_helper_ext <- function(x,matrix.flags,mat){
    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }
    return(list(result,counter))
}

multiples4 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    helper.call <- multiples_helper_ext(x,matrix.flags,mat)
    result <- helper.call[[1]]
    counter <- helper.call[[2]]

if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

If you import and execute this code on the sample lists `foo`, `bar`, and `baz` from Section 11.2.1 in the same way you did for `multiples1`, the behavior is exactly the same as this earlier version of the function. All you've done here is move the matrix-checking loop to an external function. The `multiples4` function now calls a helper function named `multiples_helper_ext`. Once the code in `multiples4` makes sure that there are matrices in the list `x` to be checked, it calls `multiples_helper_ext` to execute the required loop. Here, this helper function is defined externally and exists in the global environment for any other function to call, making that particular code easier to reuse if, say, you have other functions that need to run the same operations.

Internally Defined

If the helper function is intended to be used for only one particular parent function, it may make more sense to define the helper function internally, that is, within the execution environment of the function that needs it. The fifth version of the matrix multiplication function does just that:

```
multiples5 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    multiples_helper_int <- function(x,matrix.flags,mat){
        indexes <- which(matrix.flags)
        counter <- 0
        result <- list()
        for(i in indexes){
            temp <- x[[i]]
            if(ncol(temp)==nrow(mat)){
                counter <- counter+1
                result[[counter]] <- temp%*%mat
            }
```

```
        }
        return(list(result,counter))
    }

    helper.call <- multiples_helper_int(x,matrix.flags,mat)
    result <- helper.call[[1]]
    counter <- helper.call[[2]]

    if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

Now the helper function `multiples_helper_int` is defined within `multiples5`. That means it's visible only within the execution environment of `multiples5`, as opposed to residing in the global environment like `multiples_helper_ext`. It makes sense to internally define a helper function when (a) it's used only by a single parent function, and (b) it's called multiple times within the parent function. (Of course, `multiples5` satisfies only (a), and it's provided here just for the sake of illustration.)

11.3.2 Anonymous Functions

Often, you may need a function that performs a simple, one-line task. For example, when you use `apply`, you'll typically just want to pass in a short, simple function as an argument. That's where *anonymous* (or *disposable*) functions come in; they allow you to define a function intended for use in a single instance without explicitly creating a new object in your global environment.

Say you have a numeric matrix whose columns you want to repeat twice and then sort from smallest to largest:

```
R> foo <- matrix(c(2,3,3,4,2,4,7,3,3,6,7,2),3,4)
R> foo
     [,1] [,2] [,3] [,4]
```

```
[1,]    2    4    7    6
[2,]    3    2    3    7
[3,]    3    4    3    2
```

This is a perfect task for `apply`, which can apply a function to each column of the matrix. This function simply has to take in a vector, repeat it, and sort the result. Rather than define that short function separately, you can define an anonymous function and pass it directly to the `FUN` argument of `apply`:

```
R> apply(foo,MARGIN=2,FUN=function(x){sort(rep(x,2))})
     [,1] [,2] [,3] [,4]
[1,]    2    2    3    2
[2,]    2    2    3    2
[3,]    3    4    3    6
[4,]    3    4    3    6
[5,]    3    4    7    7
[6,]    3    4    7    7
```

The function is defined using `function` in the standard format, directly in the call to `apply`. It's defined with a single argument tag `x` which, due to the context in which we're using it, must be treated as a vector (because `FUN` is going to be applied to each column—a vector—of `foo` in turn). This anonymous function is defined, utilized, and then immediately "forgotten" once `apply` is complete. It is anonymous because it doesn't have a name, and it's disposable in the sense that it exists only for the duration of its use in the execution of `apply`. Using the `function` command this way is a shortcut more than anything else; plus, it avoids the unnecessary creation and storage of a function object in the global environment.

11.3.3 Recursive Functions

Recursion is when a function calls itself. This technique isn't commonly used in statistical analyses, but it pays to be aware of it. This section briefly illustrates what it means for a function to call itself.

Suppose you want to write a function that takes a single positive integer argument `n` and returns the corresponding *n*th term of the Fibonacci sequence (where $n = 0$ and $n = 1$ correspond to the initial two terms 0 and 1, respectively); refer back to Section 10.5.3, page 246. Earlier, you built up the Fibonacci sequence in an *iterative* fashion by using a loop. In a recursive function, instead of using a loop to repeat an operation, the function calls itself multiple times. Consider the following:

```
myfibrec <- function(n){
    if(n==0||n==1){
        return(n)
    } else {
        return(myfibrec(n-1)+myfibrec(n-2))
    }
}
```

The recursive `myfibrec` checks a single `if` statement that defines a *stopping condition*. If either `0` or `1` is supplied to the function, this means the user is requesting F_n, where $n = 0$ or $n = 1$. Since $F_0 = 0$ and $F_1 = 1$ as per the definition of the Fibonacci sequence, we can simply return `n` itself in these two specific cases. In general, however, for a positive value of `n` the function is tasked with returning the sum of `myfibrec(n-1)` and `myfibrec(n-2)`. That means if you call `myfibrec` with `n` greater than `1`, the function generates two more calls to `myfibrec`, using `n-1` and `n-2`. The recursion continues until it reaches a call for the first or second term, triggering the stopping condition, `if(n==0||n==1)`, which returns the corresponding value. Here's a sample call that retrieves the fourth Fibonacci number:

```
R> myfibrec(4)
[1] 3
```

Figure 11-3 shows the structure of this recursive call.

Note that an accessible stopping rule is critical to any recursive function. Without one, recursion will continue indefinitely. For example, the current definition of `myfibrec` works as long as the user

supplies a positive integer for the argument `n`. But in this case, if `n` is negative or is supplied as a decimal number (that is, not a strict integer), the stopping rule condition will never be satisfied, and technically the function will be permitted to recur indefinitely (though R has some automated safeguards to help prevent this and should return an error message rather than getting stuck in an infinite loop).

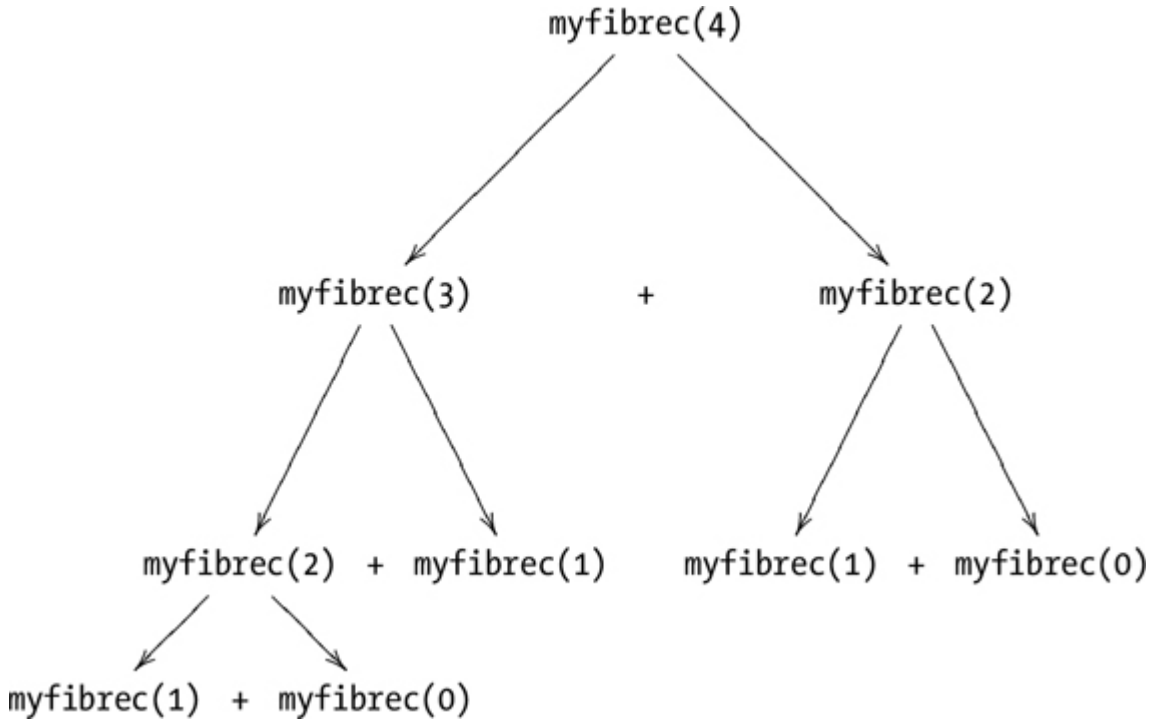


Figure 11-3: A visualization of the recursive calls made to `myfibrec` with `n=4`

Recursion is a powerful approach, especially when you don't know in advance how many times a function needs be called to complete a task. For many sort and search algorithms, recursion provides the speediest and most efficient solution. But in simpler cases, such as the Fibonacci example here, the recursive approach often requires more computational expense than an iterative looping approach. For beginners, I recommend sticking with explicit loops unless recursion is strictly required.

Exercise 11.3

a. Given a list whose members are assumed to be character string vectors of varying lengths, use an anonymous function with `lapply` to paste an exclamation mark onto the end of each element of each member, with an empty string as the separation character (see Section 4.2.2). Execute your line of code on the list given by the following:

```
foo <- list("a",c("b","c","d","e"),"f",c("g","h","i"))
```

b. Write a recursive version of a function implementing the factorial operator (see Exercise 10.4 on page 236 for details of the factorial operator). The recursion should operate by multiplying its argument, `x`, by the result of the function calling itself with a decremented value of `x-1`. The stopping rule should consist of returning the value `1` if the supplied argument value is `0`. Confirm that your function produces the same results as earlier, namely:

   i. 5 factorial is 120.

   ii. 12 factorial is 479,001,600.

   iii. 0 factorial is 1.

c. The *geometric mean* is a particular measure of centrality, different from the more common arithmetic mean. Given n observations denoted $x_1, x_2, \ldots, x_n$, you compute their geometric mean $\overline{g}$ as follows:

$$g = (x_1 \times x_2 \times \ldots \times x_n)^{1/n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n}$$

For example, to find the geometric mean of the data 4.3, 2.1, 2.2, 3.1, calculate the following:

$$g = (4.3 \times 2.1 \times 3.1)^{1/4} = 61.5846^{0.25} = 2.8$$

(This is rounded to 1 d.p.)

Write a function named `geolist` that can search through a specified list and compute the geometric means of each member per the following guidelines:

- Your function must define an internal helper function (Section 11.3.1) that returns the geometric mean of a vector argument.
- Assume the list can have only numeric vectors or numeric matrices as its members. Your function should contain a `for` loop (Section 10.2.1) to inspect each member in turn.
- If the member is a vector, compute the geometric mean of that vector using your internally defined helper function, overwriting the member with the result, which should be a single number.
- If the member is a matrix, use an implicit loop (Section 10.4.1) with your internally defined helper function to compute the geometric mean *of each row* of the matrix, overwriting the member with the results.
- The final list should be returned to the user.

Now check that your function, running on the objects `foo` and `bar` as defined here, matches these results:

i.

```
R> foo <-
list(1:3,matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),

4,2),matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),
              2,4))
R> geolist(foo)
[[1]]
```

```
[1] 1.817121

[[2]]
[1] 3.896152 3.794733 2.946184 4.442972

[[3]]
[1] 3.388035 4.106080
```

ii.

```
R> bar <- list(1:9,matrix(1:9,1,9),matrix(1:9,9,1),
               matrix(1:9,3,3))
R> geolist(bar)
[[1]]
[1] 4.147166

[[2]]
[1] 4.147166

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

[[4]]
[1] 3.036589 4.308869 5.451362
```

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`function`	Function creation	Section 11.1.1, p. 252
`return`	Function return objects	Section 11.1.1, p. 254
`missing`	Argument check	Section 11.2.3, p. 263

Function/operator	Brief description	First occurrence
…	Ellipsis (as argument)	Section 11.2.4, p. 264

12
EXCEPTIONS, TIMINGS, AND VISIBILITY



Now that you've seen how to write your own functions in R, let's examine some common function augmentations and behaviors. In this chapter, you'll learn how to make your functions throw an error or warning when they receive unexpected input. You'll also see some simple ways to measure execution time and check progress for computationally expensive functions. Finally, you'll see how R masks functions when two have the same name but reside in different packages.

12.1 Exception Handling

When there's an unexpected problem during execution of a function, R will notify you with either a *warning* or an *error*. In this section, I'll demonstrate how to build these constructs into your own functions where appropriate. I'll also cover how to instruct your code to *try* a calculation to check whether it's possible without an error (that is, to see whether it'll even work).

12.1.1 Errors and Warnings

In Chapter 11, you made your functions print a string (for example, `"no valid matrices"`) when they couldn't perform certain operations. Warnings and errors are more formal mechanisms designed to convey these types of messages and handle subsequent operations. An *error* forces the function to immediately terminate at the point it occurs. A *warning* is less severe; it indicates that the function is being run in an atypical way but tries to work around the issue and continue executing. In R, you can issue warnings with the `warning` command, and you can throw errors with the `stop` command. The following two functions show an example of each:

```
warn_test <- function(x){
    if(x<=0){
        warning("'x' is less than or equal to 0 but setting it to 1 and
                continuing")
        x <- 1
    }
    return(5/x)
}

error_test <- function(x){
    if(x<=0){
        stop("'x' is less than or equal to 0... TERMINATE")
    }
    return(5/x)
}
```

Both `warn_test` and `error_test` divide 5 by the argument `x`, which they expect to be positive. In `warn_test`, if `x` is non-positive, the function issues a warning, and `x` is overwritten to be `1` prior to performing the division. In `error_test`, on the other hand, if `x` is non-positive, the function throws an error and terminates immediately. The two commands `warning` and `stop` are used with a character string argument, which becomes the message printed to the console.

You can see these notifications by importing and calling the functions as follows:

```
R> warn_test(0)
[1] 5
Warning message:
In warn_test(0) :
  'x' is less than or equal to 0 but setting it to 1 and continuing
R> error_test(0)
Error in error_test(0) : 'x' is less than or equal to 0... TERMINATE
```

Notice that `warn_test` continues to execute and returns the value `5`, which is the result of `5/1` after setting `x` to `1`. The call to `error_test` did not return anything because R terminated the function at the `stop` command.

As a general guide, warnings are appropriate when there's a natural way for an operation to try to "save itself" even when it doesn't get the input it expects. In other situations—that is, when there's no natural or easy way to circumvent incorrect usage—it's more appropriate to throw an error and stop execution altogether.

Let's go back to the `myfibrec` function shown in Section 11.3.3, and create a version that can better handle potential misuse. Remember that a non-negative integer (whole number), giving the position of the desired Fibonacci number, is the only valid type of argument in this case. To enforce this, we can use the `is.numeric` function (Section 6.2.3) to check whether the user has supplied a numeric value, throwing an error if they haven't. Furthermore, assume that if the user supplies a negative number, the user actually means the positive version of that term; we will use a warning in this situation. Finally, we should ensure the number itself is whole. This can be handled with the appropriate "as-dot" coercion function (Section 6.2.4) `as.integer`, which strips away any decimal places from a non-integer (also called a *floating point* number) if they are present.

With all this in mind, consider these modifications:

```
myfibrec2 <- function(n){

    if(!is.numeric(n)){
        stop("'n' must be a numeric value")
```

```
    }

    if(n<0){
        warning("Assuming you meant 'n' to be positive -- doing that instead")
        n <- n*(-1)
    }

    n <- as.integer(n)

    if(n==0||n==1){
        return(n)
    } else {
        return(myfibrec2(n-1)+myfibrec2(n-2))
    }
}
```

In `myfibrec2`, the first check assesses the numeric status of the argument `n` (note the use of the NOT logical operator `!`; see Section 4.1.3); an appropriate error message results in the event the user supplies any kind of value not superficially identifiable as numeric. If this check passes (that is, if `n` *is* numeric), the function encounters another check, this time examining the negativity of the argument; if this is the case, a warning is issued and `n` is overwritten by the positive version of itself (obtained by multiplying itself by –1). If execution has made it past that point, then we know the function will be dealing with a non-negative numeric value. To ensure the value is whole, the value of `n` is overwritten by itself after having `as.integer` applied, thereby ensuring the validity of the term ultimately used for the recursive calculation. For this step, the user is not provided with a notification of any kind, since `as.integer` won't have any effect on an appropriate numeric value (one that is already decimal-free).

After making the function available by sending the definition to the R console, here you can see `myfibrec2` responding appropriately in a few different scenarios:

```
R> myfibrec2(6)
[1] 8
```

```
R> myfibrec2("this isn't right")
Error in myfibrec2("this isn't right") : 'n' must be a numeric value
R> myfibrec2(-4)
[1] 3
Warning message:
In myfibrec2(-4) :
  Assuming you meant 'n' to be positive -- doing that instead
R> myfibrec2(4.323134)
[1] 3
```

Notice that the call to `myfibrec2(-4)` returns the fourth Fibonacci term as instructed, and that this result is equivalent to the call `myfibrec2(4.323134)`. This is because in both cases, the function reconciles the (poorly) supplied argument value to be exactly 4.

Broadly speaking, both errors and warnings signal that something has gone wrong. If you're using a certain function or running chunks of code and you encounter these kinds of messages, you'll need to look carefully at what has been run and what may have occurred to spark them.

NOTE

Debugging *is the process of identifying and repairing erroneous code, for which there are various strategies. One of the most basic strategies involves including* `print` *or* `cat` *commands to inspect various quantities as they are calculated during live execution. If you're interested in R's more sophisticated debugging tools, check out the excellent introductory discussion in Chapter 19 of Peng (2022). Further technical detail is provided in Chapter 13 of* The Art of R Programming *by Matloff (2011), and a more general discussion can be found in* The Art of Debugging *by Matloff & Salzman (2008). As you gain more experience in R, understanding error messages or locating potential problems in code before they arise becomes easier, a benefit you get partly because of R's interpretative style.*

12.1.2 try Statements

When a function terminates from an error, it also terminates any parent functions. For example, if function A calls function B and function B halts because of an error, this halts execution of A at the same point. To avoid this severe consequence, you can use a `try` statement to attempt a function call and check whether it produces an error. You can also use an `if` statement to specify alternative operations, rather than forcing all processes to cease.

For example, if you call the `myfibrec2` function from Section 12.1.1 and pass it a non-numeric value, the function throws an error and terminates. But watch what happens when you pass that erroneous function call as the first argument to `try` in the following way:

```
R> attempt1 <- try(myfibrec2("this isn't right"),silent=TRUE)
```

Nothing seems to happen. In fact, the error still occurs in the call to `myfibrec2`, but `try` prevents it from interfering with general execution in the console. The printing of an error message has also been suppressed because the optional argument `silent` was set to `TRUE`. The error information is now stored in the object `attempt1`, which is of class `"try-error"`.

To see the safely trapped error, simply print `attempt1` to the console:

```
R> attempt1
[1] "Error in myfibrec2(\"this isn't right\") : 'n' must be a numeric value\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in myfibrec2("this isn't right"): 'n' must be a numeric value>
```

While you would have seen this message printed to the console if you'd left `silent` set to its default `FALSE` value, the adverse effect of the error would still be safely contained. Catching an error this way can be really handy, especially when a function produces the error in the body of another function. Using `try`, you can therefore handle the error without terminating that parent function. On the other hand, if you

pass an expression to `try` and it *doesn't* throw an error, then `try` has no effect, and you simply get the expected result of the expression. This facilitates the interpretation of `try` as a "safety net" for catching potential errors. It won't interfere with normal operation:

```
R> attempt2 <- try(myfibrec2(6),silent=TRUE)
R> attempt2
[1] 8
```

Here, you execute `myfibrec2` with a valid argument, `n=6`. Since this call doesn't result in an error, the result passed to `attempt2` is the normal return value from `myfibrec2`, in this case `8`.

Using try in the Body of a Function

Let's see a more complete example of how you could use `try` in a function. Recall the `error_test` function defined at the beginning Section 12.1.1, intended to divide the number 5 by some single value provided as the argument `x`, but throwing an error if `x` was supplied as less than or equal to zero. Consider now the function `error_testvec`, designed to operate on a vector of values:

```
error_testvec <- function(xvec){
    n <- length(xvec)
    result <- rep(0,n)
    for(i in 1:n){
        result[i] <- error_test(xvec[i])
    }
    return(result)
}
```

This function uses a `for` loop to work through `xvec` element by element, executing `error_test` on each one in turn. Given the definition of `error_test` (you'll need to make sure this is available in your workspace for the next examples to work), as long as all the values in `xvec` are positive, `error_testvec` works just fine:

```
R> et1 <- error_testvec(xvec=c(1,2,10,8))
R> et1
[1] 5.000 2.500 0.500 0.625
```

Suppose, however, there's a mistake and one of the entries in `xvec` ends up being negative:

```
R> et2 <- error_testvec(xvec=c(1,2,-10,8))
Error in error_test(xvec[i]) :
'x' is less than or equal to 0... TERMINATE
```

The internal call to `error_test` throws its custom error when it's called on the value `x=-10`, and this in turn terminates execution of `error_testvec`. Nothing is returned, and the entire call has failed.

Let's prevent this outright failure by using `try` within the `for` loop to check each call to `error_test` and have it catch any errors. The following function, `error_testvecTRY`, does just that:

```
error_testvecTRY <- function(xvec){
    n <- length(xvec)
    result <- rep(0,n)
    for(i in 1:n){
        attempt <- try(error_test(xvec[i]),silent=TRUE)
        if(inherits(attempt,"try-error")){
            result[i] <- NA
        } else {
            result[i] <- attempt
        }
    }
    return(result)
}
```

Here, you first set up a `result` vector of equal length to the input argument `xvec`, filled with zeros. Within each iteration of the `for` loop, you define the `attempt` object to store the result of trying an individual call to `error_test` and inspect the class of `attempt` using `inherits` (Section 6.2.2). If this object's class is `"try-error"`, that means `error_test` produced an error, and you fill the corresponding slot in the `result` vector with `NA`.

Otherwise, `attempt` will represent the valid, unmolested return value from `error_test`, so you place it in the corresponding slot of the `result` vector.

Now, if you import and call `error_testvecTRY` on the most recently used `xvec` vector, you see a complete set of results:

```
R> et3 <- error_testvecTRY(c(1,2,-10,8))
R> et3
[1] 5.000 2.500    NA 0.625
```

The error that would have otherwise terminated everything was silently caught, and the alternative response in this situation, `NA`, was inserted into the `result` vector. (It's worth remarking that `error_testvecTRY` represents a very simple set of operations and there are more efficient ways to do the same thing; see Exercise 12.1 , for example.)

NOTE

The `try` *command is a simplification of R's more complex* `tryCatch` *function, which is beyond the scope of this book. It provides more precise control over how you test and execute chunks of code. To learn more, enter* `?tryCatch` *in the console.*

Suppressing Warning Messages

In all the `try` calls I've shown so far, I've set the `silent` argument to `TRUE`, which stops any error messages from being printed. If you leave `silent` set to `FALSE` (the default value), the error message will print, but the error will still be caught without terminating execution.

Note that setting `silent=TRUE` suppresses only error messages, not warnings. Observe the following, which uses `warn_test` from Section 12.1.1:

```
R> wt1 <- try(warn_test(-10),silent=TRUE)
Warning message:
In warn_test(-10) :
```

```
  'x' is less than or equal to 0 but setting it to 1 and continuing
R> wt1
[1] 5
```

Although `silent` was `TRUE`, the warning (for negative values of `n` in this example) is still issued and printed. R treats warnings differently to errors in this type of scenario; this makes sense, as warnings can highlight other unforeseen issues with your code during execution. If you are absolutely sure you don't want to see any warnings, you can use `suppressWarnings`:

```
R> wt2 <- suppressWarnings(warn_test(-10))
R> wt2
[1] 5
```

The `suppressWarnings` function should be used *only if you are certain* that every warning in a given call can be safely ignored and you want to keep the output tidy.

Exercise 12.1

a. In Exercise 11.3 (b) on page 274, your task was to write a recursive R function to compute integer factorials, given some supplied non-negative integer `x`. Now modify your function so that it throws an error (with an appropriate message) if `x` is negative. Test your new function responses by using the following:
   i. `x` as `5`
   ii. `x` as `8`
   iii. `x` as `-8`

b. The idea of *matrix inversion*, briefly discussed in Section 3.1.3, is possible only for certain square matrices (those

with an equal number of columns as rows). You can compute these inversions using the `solve` function, for example:

```
R> solve(matrix(1:4,2,2))
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Note that `solve` throws an error if the supplied matrix cannot be inverted. With this in mind, write an R function that attempts to invert each matrix in a list, according to the following guidelines:

- The function should take four arguments:
    - The list `x` whose members are to be tested for matrix inversion
    - A value `noninv` to fill in results where a given matrix member of `x` cannot be inverted, defaulting to `NA`
    - A character string `nonmat` to be the result if a given member of `x` is not a matrix, defaulting to `"not a matrix"`
    - A logical value `silent`, defaulting to `TRUE`, to be passed directly to the `silent` argument of the `try` function when it is used
- The function should first check whether `x` is in fact a list using the appropriate "is-dot" function. If not, it should throw an error with an appropriate message.
- Then, the function should ensure that `x` has at least one member (that is, has length greater than zero). If not, it should throw an error with an appropriate message.

- Next, the function should check whether `nonmat` is a character string. If not, it should try to coerce it to a character string using an appropriate "as-dot" function (see Section 6.2.4), and it should issue an appropriate warning.
- After these checks, a loop should search each member `i` of the list `x`.
    - If member `i` is a matrix, attempt to invert it with `try`. If it's invertible without error, overwrite member `i` of `x` with the result. If an error is caught, then member `i` of `x` should be overwritten with the value of `noninv`.
    - If member `i` is not a matrix, then member `i` of `x` should be overwritten with the value of `nonmat`.
- Finally, the modified list `x` should be returned.

Now test your function using the following argument values to make sure it responds as expected:

i. `x` as

```
list(1:4,matrix(1:4,1,4),matrix(1:4,4,1),matrix(1:4,2,2))
```

and all other arguments at default.

ii. `x` as in (i), `noninv` as `Inf`, `nonmat` as `666`, and `silent` at default.

iii. Repeat (ii), but now with `silent=FALSE`.

iv. `x` as

```
list(diag(9),matrix(c(0.2,0.4,0.2,0.1,0.1,0.2),2,3),
    rbind(c(5,5,1,2),c(2,2,1,8),c(6,1,5,5),c(1,0,2,0)),
    matrix(1:6,2,3),cbind(c(3,5),c(6,5)),as.vector(diag(2)))
```

and `noninv` as `"unsuitable matrix"`; all other values at default.

Finally, test the error messages by attempting calls to your function with the following:

v. `x` as `"hello"`

vi. `x` as `list()`

12.2 Progress and Timing

We often use R for complex, time-consuming numeric operations, such as simulation or random variate generation, to track progress or to see how long a certain task takes to complete. For example, you may want to compare the speed of two different programming approaches to a given problem. In this section, you'll look at ways to time code execution and show its progress, including a glimpse at how to *profile* your functions.

12.2.1 Tracking Progress: Are We There Yet?

A *progress bar* shows how far along R is as it executes a set of operations. To show how this works, we'll mimic code that takes a while to execute by making R *sleep*. The `Sys.sleep` function makes R pause for a specified amount of time, in seconds, before continuing:

```
R> Sys.sleep(3)
```

If you run this command, R will pause for three seconds before you can continue using the console.

Now consider the following:

```
sleep_test <- function(n){
    result <- 0
    for(i in 1:n){
        result <- result + 1
        Sys.sleep(0.5)
```

```
    }
    return(result)
}
```

The `sleep_test` function takes a positive integer `n` and adds `1` to the `result` value for `n` iterations. At each iteration, you also tell the loop to sleep for a half second; this will act as a surrogate for something computationally expensive. Because of that sleep command, executing the following code takes about four seconds to return a result:

```
R> sleep_test(8)
[1] 8
```

Say you want to track the progress of this type of function as it executes. You can implement a textual progress bar with three steps: Initialize the progress bar object with `txtProgressBar`, update the bar with `setTxtProgressBar`, and terminate the bar with `close`. The next function, `prog_test`, modifies `sleep_test` to include those three commands:

```
prog_test <- function(n){
    result <- 0
    progbar <- txtProgressBar(min=0,max=n,style=1,char="=")
for(i in 1:n){
        result <- result + 1
        Sys.sleep(0.5)
        setTxtProgressBar(progbar,value=i)
    }
    close(progbar)
    return(result)
}
```

Here, you create an object named `progbar` by calling `txtProgressBar` with four arguments. The `min` and `max` arguments are numeric values that define the limits of the bar. In this case, you set `min=0` (this is the typical starting value) and `max=n`, which matches the number of iterations of the impending `for` loop. The `style` argument (integer, either `1`, `2`, or `3`) and the `char` argument (character string, usually a single character) govern

the appearance of the bar. Setting `style=1` means the bar will simply display a line of `char`; with `char="="`, it'll be a series of equal signs.

Once this object is created, you have to instruct the bar to actually *progress* during execution. This is done with a call to `setTxtProgressBar` at the desired point in the loop. You pass in the progress bar object to be updated (`progbar` in this case) and the `value` it should update to (this is simply taken to be the index position of the loop, `i`). Once complete (that is, immediately after the loop is exited), the progress bar must be terminated with a call to `close`, again passing in the target bar object.

Import and execute `prog_test`, and you'll see the line of `"="` drawn in steps as the loop completes:

```
R> prog_test(8)
==============================================================
[1] 8
```

The width of the bar is, by default, determined by the width of the R console pane upon execution of the `txtProgressBar` command. As noted, you can customize the bar a bit by changing the `style` and `char` arguments. Choosing `style=3`, for example, shows the bar as well as a "percent completed" counter. Some packages offer more elaborate options too, such as pop-up widgets, but the textual version is the simplest and most universally compatible version across different systems.

12.2.2 Measuring Completion Time: How Long Did It Take?

If you want to know how long a computation or chunk of code takes to complete, the most literal way to do so is to wrap the code with calls to the `Sys.time` function. In isolation, this command simply outputs an object that details current date and time information based on your system:

```
> Sys.time()
[1] "2024-07-30 13:31:15 NZST"
```

You can store objects like these immediately before and after some code and then compare them to see how much time has passed. Enter this in the editor:

```
t1 <- Sys.time()
Sys.sleep(3)
t2 <- Sys.time()
t2-t1
```

Now highlight all four lines and execute them in the console:

```
R> t1 <- Sys.time()
R> Sys.sleep(3)
R> t2 <- Sys.time()
R> t2-t1
Time difference of 3.012889 secs
```

By executing this entire code block at once, you get an easy measure of the total "wall clock" completion time in a nicely formatted string printed to the console. Exact results will vary based on your individual system, but all should be very close to three seconds. While rather crude, this time-object differencing technique is easy to interpret and readily provides an indication of the computational expense.

If you need more detailed timing reports, there are more sophisticated tools. For example, you can use `proc.time()` to receive not just the elapsed wall-clock time but also computer-related CPU timings (see the definitions in the help file `?proc.time`). To time a single expression, you can also use the `system.time` function (which uses the same detail of output as `proc.time`).

12.2.3 Profiling Your Code: Where's the Bottleneck?

R's built-in *profiler* provides a more sophisticated method of measuring computational expense in large or complicated sets of code. For example, say your code involves calling multiple functions and you want to determine which function is taking the longest time to complete execution.

The R profiler works by recording the time spent in each function that's called during a given execution of code. It approximates the result by recording which function is "active" based on repeatedly querying the system at a fine sequence of time points during execution.

We'll work through a simple example to see the basic workflow of using the R profiler. Recall that in Section 10.4.1 we found the sum of each row of a numeric matrix using an explicit `for` loop and compared this to using an implicit loop via `apply`. In that discussion, I noted that the "best" way is to use the built-in R function `rowSums`: It's faster and more efficient. Indeed, if you navigate to the help file for this function and its column-based counterpart, `colSums` (accessible by typing `?rowSums` or `?colSums` at the prompt), you'll see this sentence under Details:

> These functions are equivalent to use of `apply` with `FUN = mean` or `FUN = sum` with appropriate margins, but are a lot faster.

So let's establish just how different the computational burden is for these approaches!

First, we'll construct functions for both the explicit- and implicit-loop versions of the operation:

```
for_test <- function(mat){
    nr <- nrow(mat)
    result <- rep(NA,nr)
    for(i in 1:nr){
        result[i] <- sum(mat[i,])
    }
    return(result)
}

app_test <- function(mat){
    return(apply(mat,MARGIN=1,FUN=sum))
}
```

Next, let's create a parent function that will call `for_test` and `app_test`, as well as compute the row sums using `rowSums`, so we can do all three together:

```
threesums <- function(mat){
    s1 <- for_test(mat)
    s2 <- app_test(mat)
    s3 <- rowSums(mat)
    return(s3)
}
```

(Note that we've chosen to return only the `s3` object in `threesums`; since `s1`, `s2`, and `s3` will be exactly the same, it doesn't matter which one we pick.)

Now let's ensure these functions are available for use by sending them to the R console. The following code creates a tall, skinny numeric matrix (one million rows and three columns) and uses the time-difference approach from Section 12.2.2 to see what the overall execution time of `threesums` looks like on my machine:

```
R> val <- 1000000
R> bigmat <- matrix(rep(1:val,each=3),nrow=val,ncol=3)
R> t1 <- Sys.time()
R> foo <- threesums(bigmat)
R> t2 <- Sys.time()
R> t2-t1
Time difference of 2.429564 secs
```

This took just under two and a half seconds, but it doesn't tell us how and where that execution time was spent.

Invoke the R profiler with a call to the `Rprof` function. It creates a file where the results are stored; by default, this is named *Rprof.out*, is readable by a plaintext editor, and is placed in the working directory of the R session. You subsequently run all the code you want to profile, and then must stop the profiler with another call to `Rprof` with a `NULL` argument. Observe the following execution, which starts the profiler with all default settings, executes the code we're interested in (in this case, the line comprising the call to `threesums`), and then terminates the profiler with `Rprof(NULL)`:

```
R> Rprof()
R> foo <- threesums(bigmat)
```

```
R> Rprof(NULL)
```

If you've run this, go to your file manager and navigate to your working directory (remember you can execute `getwd()` at the R prompt to remind yourself what this is for any given R session). Therein, you should now see the aforementioned *Rprof.out* file that the profiler created.

Rather than inspect the generated file in a text editor, the details are best summarized within R itself. To do this, we use the `summaryRprof` function. By default, this function will look in the working directory for the file *Rprof.out*; if you use the default filename when you run the profiler as we've done here, you don't need to do anything else, with `summaryRprof` retrieving the most recently created profile report. Here's what the output looks like for the current example:

```
R> summaryRprof()
$by.self
                self.time self.pct total.time total.pct
"apply"              1.02    45.54       1.40     62.50
"for_test"           0.66    29.46       0.82     36.61
"FUN"                0.26    11.61       0.26     11.61
"sum"                0.16     7.14       0.16      7.14
"unlist"             0.06     2.68       0.06      2.68
"lengths"            0.04     1.79       0.04      1.79
"aperm.default"      0.02     0.89       0.02      0.89
"rowSums"            0.02     0.89       0.02      0.89

$by.total
                total.time total.pct self.time self.pct
"threesums"           2.24    100.00      0.00     0.00
"apply"               1.40     62.50      1.02    45.54
"app_test"            1.40     62.50      0.00     0.00
"for_test"            0.82     36.61      0.66    29.46
"FUN"                 0.26     11.61      0.26    11.61
"sum"                 0.16      7.14      0.16     7.14
"unlist"              0.06      2.68      0.06     2.68
"lengths"             0.04      1.79      0.04     1.79
"aperm.default"       0.02      0.89      0.02     0.89
"rowSums"             0.02      0.89      0.02     0.89
```

```
"aperm"               0.02      0.89      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 2.24
```

We are provided with a list of members detailing timing reports at different levels, arranged in different ways. The most useful output is provided in the very first table, corresponding to the `by.self` member. Here, the entries are sorted from highest to lowest based on the "self" time (the time spent running each function required by your code in isolation), both in terms of real time and as a proportion of total execution time. The `by.total` member sorts values based on overall encapsulated execution time (that is, itself and any other functions it calls internally). Notice therefore that in each table, the `self.pct` column will sum to exactly 1, but the `total.pct` won't. We also see the time interval used to capture the results in the `sample.interval` entry; the default is 20 milliseconds (0.02s). Finally, the complete execution time is reported in `sampling.time`, which is the same kind of result you'd get by using the time-differencing approach via `Sys.time`.

Looking to the current results, we can see from the `by.self` table that the implicit loop was actually the slowest technique; it is in the `apply` function where much of this run's 2.24 seconds was spent—1.02s, or 45.54 percent of the time, to be precise. The explicit loop gets the silver medal, with `for_test` taking 0.66s. We see now why it's best to use the built-in `rowSums` function for this task: It contributed less than 1 percent of the execution time of `threesums`! Finally, note that there are other entries in the results tables that were not explicitly written in the code of our custom functions, for example, `unlist` or `aperm.default`. These are functions used internally by the functions we *did* make explicit use of; any such utilities will also form part of the profile.

There are a number of options we can configure when profiling our code with `Rprof`, as well as precisely how the resulting profile is

summarized with `summaryRprof`. For instance, for more precise results of particularly fast functions, we may need to decrease the sampling interval; this can be done by setting the `interval` argument in the initial call to `Rprof`. In the current example, given the `rowSums` function is so fast that its contribution of 0.02s was equal to the default sampling interval, we could refine things by using a sampling interval of 10 milliseconds instead:

```
R> Rprof(interval=0.01)
R> foo <- threesums(bigmat)
R> Rprof(NULL)
R> summaryRprof()$by.self
                self.time self.pct total.time total.pct
"apply"              1.11    48.68       1.31     57.46
"for_test"           0.61    26.75       0.95     41.67
"sum"                0.34    14.91       0.34     14.91
"FUN"                0.12     5.26       0.12      5.26
"unlist"             0.04     1.75       0.04      1.75
"aperm.default"      0.02     0.88       0.02      0.88
"lengths"            0.02     0.88       0.02      0.88
"c"                  0.01     0.44       0.01      0.44
"rowSums"            0.01     0.44       0.01      0.44
```

Indeed, we see in this run that `rowSums` is very, very fast! Results will vary based on your individual system.

There are multiple contributed packages that cater to the appraisal of code. One elegant extension to the R profiler is provided by the `profvis` package by Wickham et al. (2024), which adds tools to visualize and interact with a set of profile results. Indeed, this functionality is built into the RStudio IDE under the Profile menu (see Section B.2.3).

NOTE

We use the term profiling *to refer to a detailed description of where computational bottlenecks exist in your function(s). You may also come across the term* benchmarking, *which refers to the comparison of different ways of doing the same thing. As such, the example in this section is*

technically a profile *of code executing a* benchmark *comparison of calculating row sums! There are formal benchmarking tools available for R; see, for example, the* `microbenchmark` *(Mersmann 2024) or* `rbenchmark` *(Kusnierczyk 2012) packages. For further reading on topics related to appraising your code, see the books by Peng (2022) and Wickham (2019).*

Exercise 12.2

a. Modify `prog_test` from Section 12.2.1 to include an ellipsis in its argument list, intended to take values for the additional arguments in `txtProgressBar`; name the new function `prog_test_fancy`. Use the time-differencing approach to time how long it takes a call to `prog_test_fancy` to complete execution. Set `50` as `n`, instruct the progress bar (through the ellipsis) to use `style=3`, and set the bar character to be `"r"`.

b. In Section 12.1.1 (page 279), you defined the function `myfibrec2`, which uses recursion to compute the `n`th term of the Fibonacci sequence and includes some basic checks on the argument `n`. Ensure this function is available in your workspace, and complete the following:

    i. Write a new function, `myfibvector`, which takes a single argument `nvec`. This function should use a `for` loop that cycles through each element of `nvec` and uses `myfibrec2` on each of these elements to find the corresponding Fibonacci term. The function `myfibvector` should include a progress bar of `style=3` and a character of your choosing that updates at each pass of the aforementioned `for` loop.

    ii. Use your new function to find the first 33 terms of the Fibonacci sequence (set `nvec` to be a sequence

of integers from 0 to 32). Use the time-differencing method to determine how long execution takes.

iii. Using a stand-alone `for` loop without recursion, compute these same 33 terms. Time it. Which approach—recursion or explicit `for` loop—would you prefer?

c. In Section 11.3.1, we defined the `multiples5` function (see page 271). Ensure this function is available in your workspace. Using an interval of 10 milliseconds, profile an instance of its execution with the following arguments. Set `x` as

```
list(matrix(1:2048^2,2048,2048),"not a matrix",
     "definitely not a matrix",matrix(2048^2:1,2048,2048),
     matrix(1:8,4,2))
```

and set `mat` as:

```
matrix((-2048^2):(-1),2048,2048)
```

Summarize the profile to determine how the computational expense is distributed. What is the most individually significant contributor to the execution time?

12.3 Masking

With the plethora of built-in and contributed data and functionality available for R, it is inevitable that at some point you will come across objects, usually functions, that share the same name in distinctly different loaded packages. You might also wish to define your own function that happens to share the name of a function in an R package that you've already loaded.

In these instances, R responds by *masking* one of the objects: one object or function will take precedence over the other and assume the object or function name, while the masked function must be called with an additional command. This protects objects from overwriting one another while still enabling access to all functionality. In this section, we'll look at the two most common masking situations in R.

12.3.1 Function and Object Distinction

When two functions or objects in different environments have the same name, the object that comes earlier in the search path will mask the later one. That is, when the object is sought, R will use the object or function it finds first, and you'll need extra code to access the other, masked object. Remember from Section 9.1.2 that you can see the current search path by executing `search()`. Doing so on my local system, using the base R GUI, results in the following:

```
R> search()
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

When R searches, the function or object that falls closest to the start of the search path (the global environment) is reached first and masks the function or object of the same name that occurs somewhere later in the search path (see Section 9.1.2). As a simple example of masking, let's define a function with the same name as a function in the `base` package: `sum`. Remember that the built-in version of `sum` simply adds up all the elements in a supplied vector:

```
R> foo <- c(4,1.5,3)
R> sum(foo)
[1] 8.5
```

Now suppose you were to create the following function:

```
sum <- function(x){
    result <- 0
    for(i in 1:length(x)){
        result <- result + x[i]^2
    }
    return(result)
}
```

This version of `sum` takes in a vector `x` and uses a `for` loop to square each element before summing them and returning the result. This can be imported from your editor into the R console without any problem, but it doesn't offer the same functionality as the (original) built-in version of `sum`. After importing the function, if you make a call to `sum`, your version is used:

```
R> sum(foo)
[1] 27.25
```

This happens because your user-defined function is stored in the global environment (`.GlobalEnv`), which always comes first in the search path. R's built-in function is part of the `base` package, which comes at the end of the search path. In this case, the user-defined function is masking the original.

If we want to use the original version of `sum`, we'll need to explicitly tell R where to find it. To run the `base` version of `sum`, you have to precede the function call with the name of its package and a double colon as follows:

```
R> base::sum(foo)
[1] 8.5
```

This approach will work for any masked function that would otherwise be normally accessible by the user; we just need to know which package owns the masked object we want.

Before moving on, let's now remove our custom `sum` function from the global environment to avoid any potential confusion!

```
R> rm(sum)
```

Clashing Package Objects

When you load a package, R will notify you if any objects in the package clash with other objects that are accessible in the present session. To illustrate this, I'll make use of two contributed packages: the `car` package (see Section 8.1) and the `faraway` package (you'll use this in Part IV). After ensuring these two packages are installed (Section 1.3.5), I load them in the following order and see this message:

```
R> library(faraway)
R> library(car)
Loading required package: carData

Attaching package: 'car'

The following objects are masked from 'package:faraway':

    logit, vif
```

This indicates that the packages have two objects with the same name: `logit` and `vif`. R has automatically notified you that these objects are being masked. In this instance, calling `logit` (or `vif`) at the prompt will access `car`'s object since that package was loaded more recently; hence it will precede `faraway` in your search path. To use `faraway`'s version, you must type `faraway::logit` (or `faraway::vif`). These rules also apply to accessing the respective help files.

A similar notification occurs when you load a package with an object that's masked by a global environment object (a global environment object will always take precedence over a package object). To see an example, load the `MASS` package, which is included with a standard installation of R as a "recommended package" (see Appendix A.2.2) but isn't automatically loaded. Create the following object:

```
R> cats <- "meow"
```

Now suppose you need to load `MASS`:

```
R> library("MASS")

Attaching package: 'MASS'

The following object is masked _by_ '.GlobalEnv':

    cats
```

Upon loading the package, you're informed that the `cats` object you've just created is masking an object of the same name in `MASS`. (As you can see with `?MASS::cats`, this object is a data frame with weight measurements of household felines.) If `MASS` shares any object names with any other packages you have loaded in any given session, you'll see the same kind of "package masking" message as shown earlier between `faraway` and `car`.

We can check maskings at any time with the `conflicts` function. Given the current state of my workspace, here's what I see:

```
R> conflicts(detail=TRUE)
$.GlobalEnv
[1] "cats"

$`package:MASS`
[1] "cats" "npk"

$`package:car`
[1] "logit" "vif"

$`package:faraway`
[1] "logit" "vif"

$`package:graphics`
[1] "plot"

$`package:datasets`
[1] "npk"
```

```
$`package:methods`
[1] "body<-"    "kronecker"

$`package:base`
[1] "body<-"    "kronecker" "plot"
```

I've used the option `detail=TRUE` to pinpoint not only the masked object names but also the packages responsible for them. In addition to the `logit`, `vif`, and `cats` maskings we've encountered, there are a couple of additional maskings among the objects of the automatically loaded packages (`graphics`, `datasets`, `methods`, and `base`).

Unmounting Packages

You can unmount loaded packages from your active R session. This is necessary only if you specifically want to remove the package from the current search path (and don't want to open a new session).

With the packages loaded in this discussion, my current search path looks like this:

```
R> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:car"
 [4] "package:carData"   "package:faraway"   "tools:RGUI"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:utils"     "package:datasets"  "package:methods"
[13] "Autoloads"         "package:base"
```

Now let's say I don't want `car` there anymore. I unmount it with the `detach` function as follows:

```
R> detach("package:car",unload=TRUE)
R> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:carData"
 [4] "package:faraway"   "tools:RGUI"        "package:stats"
 [7] "package:graphics"  "package:grDevices" "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

The command has removed the specified package from the search path of the current R session, also unloading its namespace (note that the `carData` package, a dependency that was automatically loaded when `car` itself was loaded, remains present). Now the functionality of `car` is no longer immediately available, and `faraway`'s `logit` and `vif` objects are no longer masked (you can confirm this with another call to `conflicts`).

NOTE

As contributed packages get updated by their maintainers, they may include new objects that spark new maskings or remove or rename objects that previously caused maskings (when compared with other contributed packages). The specific maskings illustrated here with `car`, `faraway`, *and* `MASS` *occur at the time of writing with the versions available and may change in the future.*

12.3.2 Data Frame Variable Distinction

There's one other common situation in which you'll be explicitly notified of masking: when you add a data frame to the search path. Let's see how this works. Define the following data frame:

```
R> foo <- data.frame(surname=c("a","b","c","d"),
                     sex=c(0,1,1,0),height=c(170,168,181,180),
                     stringsAsFactors=F)
R> foo
  surname sex height
1       a   0    170
2       b   1    168
3       c   1    181
4       d   0    180
```

The data frame `foo` has three column variables: `surname`, `sex`, and `height`. To access one of these columns, normally you need to use the `$` operator and enter something like `foo$surname`. However, you can *attach* a data frame directly to your search path, which makes it easier to access its

variables. Assume I'm continuing on in my current workspace. Here's what I see when using the `attach` function:

```
R> attach(foo)
R> search()
 [1] ".GlobalEnv"        "foo"               "package:MASS"
 [4] "package:carData"   "package:faraway"   "tools:RGUI"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:utils"     "package:datasets"  "package:methods"
[13] "Autoloads"         "package:base"
```

Now the `surname` variable is directly accessible at the R prompt because the `foo` data frame is itself part of the search path.

```
R> surname
[1] "a" "b" "c" "d"
```

This saves you from having to enter `foo$` every time you want to access a variable, which can be a handy shortcut if your analysis deals exclusively with one static, unchanging data frame. However, if you forget about your attached objects, they can cause problems later, especially if you continue to mount more objects onto the search path in the same session. For example, say you enter another data frame:

```
R> bar <- data.frame(surname=c("e","f","g","h"),
                     sex=c(1,0,1,0),weight=c(55,70,87,79),
                     stringsAsFactors=F)
R> bar
  surname sex weight
1       e   1     55
2       f   0     70
3       g   1     87
4       h   0     79
```

Then add it to the search path, too:

```
R> attach(bar)
The following objects are masked from foo:
```

```
    sex, surname
```

The notification tells you that the `bar` object now precedes `foo` in the search path:

```
R> search()
 [1] ".GlobalEnv"        "bar"               "foo"
 [4] "package:MASS"      "package:spatstat"  "tools:RGUI"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:utils"     "package:datasets"  "package:methods"
[13] "Autoloads"         "package:base"
```

As a result, any direct use of either `sex` or `surname` will now access `bar`'s contents, not `foo`'s. Meanwhile, the unmasked variable `height` from `foo` is still directly accessible:

```
R> height
[1] 170 168 181 180
```

This is a pretty simple example, but it highlights the potential for confusion when data frames, lists, or other objects are added to the search path. Mounting objects this way can quickly become difficult to track, especially for large datasets with many different variables. Therefore, it's best to avoid `attach`ing objects like this as a general guideline unless, as stated earlier, you're working exclusively with one data frame.

You can use `detach` to remove objects like data frames from the search path, similar to what you saw with packages:

```
R> detach(foo)
R> search()
 [1] ".GlobalEnv"        "bar"               "package:MASS"
 [4] "package:spatstat"  "tools:RGUI"        "package:stats"
 [7] "package:graphics"  "package:grDevices" "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

Note that in this scenario, the optional `unload` argument you saw used in an earlier example is not needed: That argument is relevant only when detaching a package.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`warning`	Issue warning	Section 12.1.1, p. 278
`stop`	Throw error	Section 12.1.1, p. 278
`try`	Attempt error catch	Section 12.1.2, p. 281
`Sys.sleep`	Sleep (pause) execution	Section 12.2.1, p. 286
`txtProgressBar`	Initialize progress bar	Section 12.2.1, p. 286
`setTxtProgressBar`	Increment progress bar	Section 12.2.1, p. 286
`close`	Close progress bar	Section 12.2.1, p. 286
`Sys.time`	Get local system time	Section 12.2.2, p. 287
`rowSums/colSums`	Matrix row/column totals	Section 12.2.3, p. 288
`Rprof`	Start/stop R profiler	Section 12.2.3, p. 290

Function/operator	Brief description	First occurrence
`summaryRprof`	Summarize profile results	Section 12.2.3, p. 290
`::`	Explicit package stipulation	Section 12.3.1, p. 294
`conflicts`	Identify object maskings	Section 12.3.1, p. 296
`detach`	Remove library/object from path	Section 12.3.1, p. 297
`attach`	Attach object to search path	Section 12.3.2, p. 298

PART III
STATISTICS AND PROBABILITY

13
ELEMENTARY STATISTICS



Statistics is the practice of turning *data* into *information*. We do this to identify trends and understand features of populations. This chapter will cover some basic terminology, definitions, and concepts, and use R to demonstrate their application.

13.1 Describing Raw Data

Often, the first thing statistical analysts are faced with is raw data—in other words, the records or observations that make up a sample. Depending on the nature of the intended analysis, these data could be stored in a specialized R object, often a data frame (Chapter 5), possibly read in from an external file using techniques from Chapter 8. Before you can even begin summarizing or exploring your data, however, it is important to clearly identify your available variables.

A *variable* is a characteristic of an individual in a population, the value of which can differ between entities within that population. For example, in Section 5.2, you experimented with an illustrative data frame `mydata`. You recorded the age, sex, and humor level for a sample of

people. These characteristics are your variables; the specific values you record differ between the individuals. Different types of variables are determined by the very nature of the values they may take. Before jumping into R, you'll look at some standard kinds of variables.

13.1.1 Numeric Variables

A *numeric* variable is one whose observations are naturally recorded as numbers. There are two types of numeric variables: continuous and discrete.

We define a *continuous* variable as one that can be recorded as any value in some interval, up to any number of decimals (which technically gives an infinite number of possible values, even if the continuum's range is restricted). For example, if you were observing rainfall amount, a value of 15 mm would make sense, but so would a value of 15.42135 mm. Any degree of measurement precision gives a valid observation.

A *discrete* variable, on the other hand, may take on only distinct numeric values, and if the range is restricted, then the number of possible values is finite. For example, if you were observing the number of heads in 20 flips of a coin, only whole numbers would make sense. It would not make sense to observe 15.42135 heads; the possible outcomes are restricted to the integers (whole numbers) from 0 to 20 (inclusive).

13.1.2 Categorical Variables

Though numeric observations are common for many variables, it's also important to consider *categorical* variables. Like some discrete variables, categorical variables may take only one of a finite number of possibilities. Unlike discrete variables, however, categorical observations are not always recorded as numeric values.

There are two types of categorical variables. We call those that cannot be logically ranked *nominal*. A good example of a categorical-nominal variable is sex. In most datasets, it has two fixed possible values, male and female, and the order of these categories is irrelevant. We call categorical variables that can be naturally ranked *ordinal*. An example of

a categoricalordinal variable would be the dose of a drug, with the possible values low, medium, and high. You can order these values in either increasing or decreasing amounts, and the ordering might be relevant to the research.

NOTE

Some statistical texts blur the definitions of discrete and categorical variables or even use them interchangeably. I prefer to keep the definitions separate, for clarity. That is, I'll use discrete *when referring to a naturally numeric variable that cannot be expressed on a continuous scale (such as a count), and* categorical *when the possible outcomes for a given individual are not necessarily numeric and the number of possible values is always finite.*

13.1.3 Examples

Once you know what to look for, identifying the types of variables in a given dataset is straightforward. Take the data frame `chickwts`, which is available in the automatically loaded `datasets` package. At the prompt, directly entering the following gives you the first five records of this dataset:

```
R> chickwts[1:5,]
  weight      feed
1    179 horsebean
2    160 horsebean
3    136 horsebean
4    227 horsebean
5    217 horsebean
```

R's help file (`?chickwts`) describes these data as comprising the weights of 71 chicks (in grams) after six weeks, based on the type of food provided to them. Now let's take a look at the two columns in their entirety as vectors:

```
R> chickwts$weight
 [1] 179 160 136 227 217 168 108 124 143 140 309 229 181 141 260 203 148 169
[19] 213 257 244 271 243 230 248 327 329 250 193 271 316 267 199 171 158 248
[37] 423 340 392 339 341 226 320 295 334 322 297 318 325 257 303 315 380 153
[55] 263 242 206 344 258 368 390 379 260 404 318 352 359 216 222 283 332
R> chickwts$feed
 [1] horsebean horsebean horsebean horsebean horsebean horsebean horsebean
 [8] horsebean horsebean horsebean linseed   linseed   linseed   linseed
[15] linseed   linseed   linseed   linseed   linseed   linseed   linseed
[22] linseed   soybean   soybean   soybean   soybean   soybean   soybean
[29] soybean   soybean   soybean   soybean   soybean   soybean   soybean
[36] soybean   sunflower sunflower sunflower sunflower sunflower sunflower
[43] sunflower sunflower sunflower sunflower sunflower sunflower meatmeal
[50] meatmeal  meatmeal  meatmeal  meatmeal  meatmeal  meatmeal  meatmeal
[57] meatmeal  meatmeal  meatmeal  casein    casein    casein    casein
[64] casein    casein    casein    casein    casein    casein    casein
[71] casein
Levels: casein horsebean linseed meatmeal soybean sunflower
```

Here, `weight` is a numeric measurement that can fall anywhere on a continuum, so this is a numeric-continuous variable. The fact that the chick weights appear to have been rounded or recorded to the nearest gram does not affect this definition because in reality the weights can be any figure (within reason). The `feed` column is a categorical variable because it has only six possible outcomes that aren't numeric. The absence of any natural or easily identifiable ordering leads to the conclusion that `feed` is a categoricalnominal variable.

13.1.4 Univariate and Multivariate Data

Univariate data is associated with only one dimension. For example, the `weight` variable in the earlier example is univariate since you can express each measurement with one component: a single number.

When considering data with respect to variables that exist in more than one dimension (in other words, with more than one component or measurement associated with each observation), your data are *multivariate*. Multivariate measurements are arguably most relevant

when the individual components aren't as useful on their own (in other words, as univariate quantities) in any given statistical analysis.

An ideal example is that of spatial coordinates, defined in terms of at least two components: a horizontal x-coordinate and a vertical y-coordinate. The univariate data alone—for example, the x-axis values only—aren't especially useful. Consider the `quakes` dataset (like `chickwts`, this is automatically available through the `datasets` package), which contains observations on 1,000 seismic events recorded off the coast of Fiji. If you look at the first five records and read the descriptions in the help file `?quakes`, you'll get a good understanding of what's presented:

```
R> quakes[1:5,]
     lat   long depth mag stations
1 -20.42 181.62   562 4.8       41
2 -20.62 181.03   650 4.2       15
3 -26.00 184.10    42 5.4       43
4 -17.97 181.66   626 4.1       19
5 -20.42 181.96   649 4.0       11
```

The columns `lat` and `long` provide the latitude and longitude of the event, `depth` provides the depth of the event (in kilometers), `mag` provides the magnitude on the Richter scale, and `stations` provides the number of observation stations that detected the event. If you're interested in the spatial dispersion of these earthquakes, then examining only the latitude or the longitude is rather uninformative. Two components describe the location of each event: a latitude *and* a longitude value. You can plot these 1,000 events like so:

```
R> plot(quakes$long,quakes$lat,xlab="Longitude",ylab="Latitude",asp=1.069)
```

Figure 13-1 shows the result.

Notice the use of the argument for locking the plot's aspect ratio: `asp=1.069`. In Section 7.2.6, I noted this is usually an important part of creating geographical plots. You'll see why the value 1.069 is roughly appropriate for this particular dataset in Section 14.6.4.

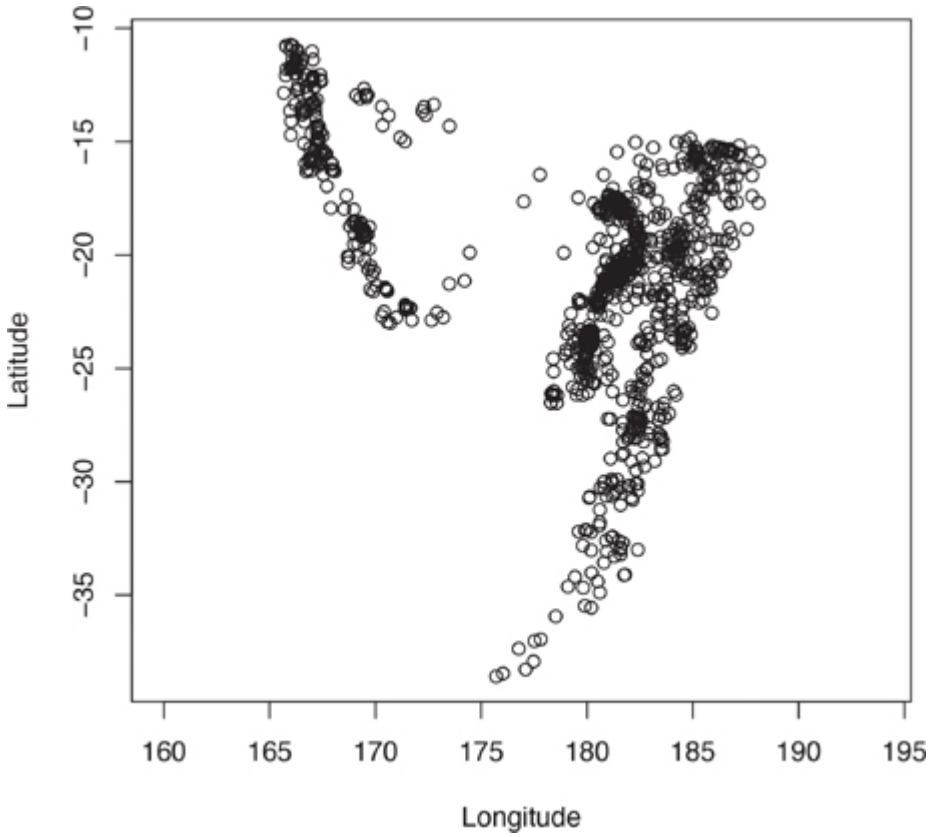


Figure 13-1: Plotting the spatial locations of earthquakes using a bivariate (multivariate with two components) variable

13.1.5 Parameter or Statistic?

Statistics as a discipline is concerned with understanding features of an overall *population*, defined as the entire collection of individuals or entities of interest. We refer to the characteristics of that population as *parameters*. Because researchers are rarely able to access relevant data on every single member of the population of interest, they typically collect a *sample* of entities to represent the population and record relevant data from these entities. They may then estimate the parameters of interest using the sample data, and those estimates are the *statistics*.

For example, if you were interested in the average age of adult women in the United States who own cats, the population of interest would be all adult women residing in the United States who own at least one cat. The parameter of interest is the true mean age of adult women in the United States who own at least one cat. As obtaining the age of

every single adult female American with a cat would be a difficult feat, a more feasible approach would be to randomly identify a smaller number of cat-owning American women and take data from them. This is your sample, and the mean age of the women in the sample is your statistic.

In summary, the key difference between a statistic and a parameter is whether the characteristic refers to the sample you drew your data from or to the wider population. Figure 13-2 illustrates this, with the unknown parameter of interest being the population mean μ. The corresponding statistic, $\bar{x}$, is taken to be the mean of a sample of individuals collected from that population. We use the latter to make inference about the former.

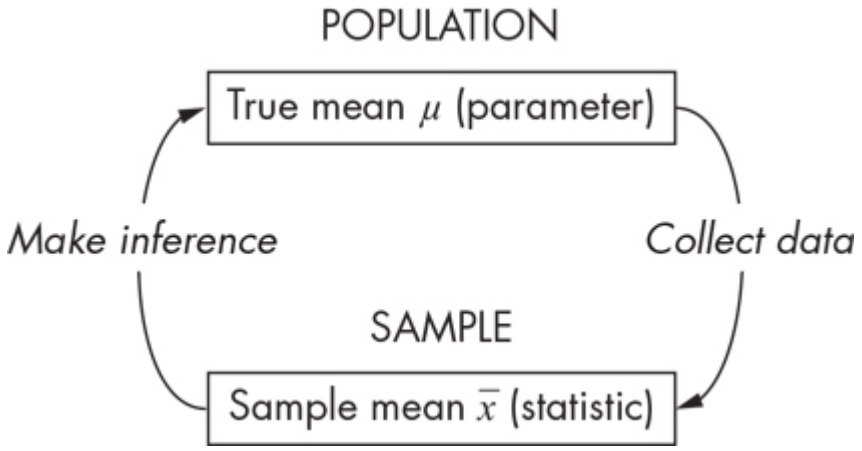


Figure 13-2: A conceptualization of statistical practice to illustrate the definitions of parameter and statistic, using the mean as an example

Exercise 13.1

a. For each of the following, identify the type of variable described: numeric-continuous, numeric-discrete, categoricalnominal, or categorical-ordinal:

   i. The number of blemishes on the hood of a car coming off a production line

ii. A survey question that asks the participant to select from Strongly agree, Agree, Neutral, Disagree, and Strongly disagree

iii. The noise level (in decibels) at a concert

iv. The noise level out of three possible choices: high, medium, low

v. A choice of primary color

vi. The distance between a cat and a mouse

b. For each of the following, identify whether the quantity discussed is a population parameter or a sample statistic. If the latter, also identify what the corresponding population parameter is.

i. The percentage of 50 New Zealanders who own a gaming console

ii. The average number of blemishes on the hoods of three cars in the No Dodgy Carz yard

iii. The proportion of domestic cats in the United States that wear a collar

iv. The average number of times per day a vending machine is used in a year

v. The average number of times per day a vending machine is used in a year, based on data collected on three distinct days in that year

13.2 Summary Statistics

Now that you've learned the basic terminology, you're ready to calculate some statistics with R. In this section, you'll look at the most

common types of statistics used to summarize the previously defined variables.

13.2.1 Measures of Centrality

We commonly use *measures of centrality* to explain large collections of data by describing where numeric observations are centered.

Mean

One of the most common measures of centrality is the *arithmetic mean* (or just the *mean*). It's considered to be the central "balance point" of a collection of observations.

For a sample of n numeric measurements labeled $x = \{x_1, x_2, \ldots, x_n\}$, you find the mean $\overline{x}$ as follows:

$$\overline{x} = \frac{(x_1 + x_2 + \ldots + x_n)}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (13.1)$$

So, for example, if you observe the data 2, 4.4, 3, 3, 2, 2.2, 2, 4, you calculate the mean like this:

$$\frac{2 + 4.4 + 3 + 3 + 2 + 2.2 + 2 + 4}{8} = 2.825$$

Median

The *median* is the "middle magnitude" of your observations, so if you place your observations in order from smallest to largest, you can find the median by either taking the middle value (if there's an odd number of observations) or finding the mean of the two middle values (if there's an even number of observations). Using the notation for n measurements labeled $x = \{x_1, x_2, \ldots, x_n\}$, you find the sample median m_x as follows:

1. Sort the observations from smallest to largest to give the "order statistics" $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, where $x^{(t)}$ denotes the tth smallest observation.

2. Then, do the following:

$$\overline{m}_x = \begin{cases} x^{\left(\frac{n+1}{2}\right)}, & \text{if } n \text{ is odd} \\ \left(x^{\left(\frac{n}{2}\right)} + x^{\left(\frac{n}{2}+1\right)}\right)/2, & \text{if } n \text{ is even} \end{cases} \quad (13.2)$$

For the same data, sorting them from smallest to largest yields 2, 2, 2, 2.2, 3, 3, 4, 4.4. With $n = 8$ observations, you have $n/2 = 4$. The median is therefore as follows:

$$\left(x_i^{(4)} + x_j^{(5)}\right)/2 = (2.2 + 3)/2 = 2.6$$

Mode

The *mode* is the "most common" observation. We use this statistic more often with numeric-discrete data than with numeric-continuous, though we use it with reference to *intervals* of the latter (commonly when discussing probability density functions; see Chapters 15 and 16). It's possible for a collection of n numeric measurements $x_1, x_2, \ldots, x_n$ to have no mode (where each observation is unique) or to have more than one mode (where more than one particular value occurs the largest number of times). To find the mode d_x, simply tabulate the frequency of each measurement.

Again using the eight observations from the example, you can see the frequencies here:

Observation	2	2.2	3	4	4.4
Frequency	3	1	2	1	1

The value 2 occurs three times, which is more frequent than any other value, so the single mode for these data is the value 2.

R Implementation

In R, it's easy to compute the arithmetic mean and the median with built-in functions of the same names. First, store the observations as the numeric vector `xdata`:

```
R> xdata <- c(2,4.4,3,3,2,2.2,2,4)
```

Then compute the statistics:

```
R> x.bar <- mean(xdata)
R> x.bar
[1] 2.825
R> m.bar <- median(xdata)
R> m.bar
[1] 2.6
```

Finding a mode is perhaps most easily achieved by using R's `table` function, which gives you the frequencies you need:

```
R> xtab <- table(xdata)
R> xtab
xdata
  2 2.2   3   4 4.4
  3   1   2   1   1
```

Though this clearly shows the mode for a small dataset, it's good practice to use code that can automatically identify the most frequent observations for any result of `table`. First, note that the `min` and `max` functions will report the smallest and largest values of any numeric vector, with `range` returning both in a vector of length 2:

```
R> min(xdata)
[1] 2
R> max(xdata)
[1] 4.4
R> range(xdata)
[1] 2.0 4.4
```

When you apply these functions to the result of calling `table`, they operate on the reported frequencies:

```
R> max(xtab)
[1] 3
```

Finally, therefore, you can construct a logical flag vector to return just the mode(s) of interest:

```
R> d.bar <- xtab[xtab==max(xtab)]
R> d.bar
2
3
```

Here, 2 is the value and 3 is the frequency of that value.

For a real-data example, let's return to the `chickwts` object explored earlier in Section 13.1.3. The mean and median weights of the chicks are as follows:

```
R> mean(chickwts$weight)
[1] 261.3099
R> median(chickwts$weight)
[1] 258
```

You can also look at the `quakes` dataset explored in Section 13.1.4. The most common magnitude of earthquake in the dataset is identified with the following:

```
R> Qtab <- table(quakes$mag)
R> Qtab[Qtab==max(Qtab)]
4.5
107
```

This indicates that there were 107 occurrences of a 4.5 magnitude event.

NOTE

Several methods are available to compute medians, though the impact on results is usually negligible for most practical purposes. Here, I've simply described and used the default "sample" version used by R.

Many of the functions R uses to compute statistics from a numeric structure will be affected by the presence of special values discussed in Section 6.1 (such as `NA`s or `NaN`s). Here's a toy example:

```
R> mean(c(1,4,NA))
[1] NA
R> mean(c(1,4,NaN))
[1] NaN
```

This behavior prevents unintended `NaN`s or forgotten `NA`s from being ignored without the user's knowledge; therefore, R will not by default return the intended numeric results for the non-missing entries. You can, however, set an optional argument `na.rm` to `TRUE`, which will force the function to operate only on the numeric values that are present:

```
R> mean(c(1,4,NA),na.rm=TRUE)
[1] 2.5
R> mean(c(1,4,NaN),na.rm=TRUE)
[1] 2.5
```

You should use this argument *only if you're aware* there might be missing values and that the result will be computed based on solely those values that *have* been observed. Functions including `sum`, `prod`, `mean`, `median`, `max`, `min`, and `range`—essentially anything that calculates a numeric statistic based on a numeric vector—all have the `na.rm` argument available to them. Lastly, in calculating simple summary statistics, it's useful to remember the `tapply` function, which computes statistics grouped by a specific categorical variable (see Section 10.4.2). Suppose, for example, you wanted to find the mean weight of the chicks grouped by feed type. You might use the `mean` function on each specific subset:

```
R> mean(chickwts$weight[chickwts$feed=="casein"])
[1] 323.5833
R> mean(chickwts$weight[chickwts$feed=="horsebean"])
[1] 160.2
R> mean(chickwts$weight[chickwts$feed=="linseed"])
[1] 218.75
R> mean(chickwts$weight[chickwts$feed=="meatmeal"])
```

```
[1] 276.9091
R> mean(chickwts$weight[chickwts$feed=="soybean"])
[1] 246.4286
R> mean(chickwts$weight[chickwts$feed=="sunflower"])
[1] 328.9167
```

This is cumbersome and lengthy. Using `tapply`, however, you can calculate the same values by category using just one line of code:

```
R> tapply(chickwts$weight,INDEX=chickwts$feed,FUN=mean)
   casein horsebean   linseed  meatmeal   soybean sunflower
 323.5833  160.2000  218.7500  276.9091  246.4286  328.9167
```

Here, the first argument is the numeric vector upon which to operate, the `INDEX` argument specifies the grouping variable, and the `FUN` argument gives the name of the function to be performed on the data in the first argument as per the subsets defined by `INDEX`. Like other functions you've seen that request the user to specify *another* function to govern operations, `tapply` includes an ellipsis (see Sections 9.2.5 and 11.2.4) to allow the user to supply further arguments directly to `FUN` if required.

13.2.2 Counts, Percentages, and Proportions

In this section, you'll look at summarizing data that aren't necessarily numeric. It makes little sense, for example, to ask R to compute the mean of a categorical variable, but it's sometimes useful to count the number of observations that fall within each category; these *counts* or *frequencies* represent the most elementary summary statistic of categorical data.

Tabulating Counts

This uses the same count summary that was necessary for the mode calculation in Section 13.2.1, so again you can use the `table` command to obtain frequencies. Recall there are six feed types making up the diet of the chicks in the `chickwts` data frame. Getting these factor-level counts is straightforward:

```
R> table(chickwts$feed)

   casein horsebean   linseed  meatmeal   soybean sunflower
       12        10        12        11        14        12
```

Calculating Proportions and Percentages

You can gather more information from these counts by identifying the *proportion* of observations that fall into each category. This will give you comparable measures across multiple datasets. Proportions represent the fraction of observations in each category, usually expressed as a decimal (floating-point) number between 0 and 1 (inclusive). To calculate proportions, you need only to modify the previous count function by dividing the count (or frequency) by the overall sample size. Here, this is obtained by using `nrow` on the appropriate data frame object; see Section 5.2.1:

```
R> table(chickwts$feed)/nrow(chickwts)

   casein horsebean   linseed  meatmeal   soybean sunflower
0.1690141 0.1408451 0.1690141 0.1549296 0.1971831 0.1690141
```

Of course, you needn't do everything associated with counts via `table`. A simple `sum` of an appropriate logical flag vector can be just as useful; recall that R implicitly treats `TRUE`s as `1` and `FALSE`s as `0` in any arithmetic operation on logical structures (refer to Section 4.1.4). Such a `sum` will provide you with the desired frequency, but to get a proportion, you still need to divide by the total sample size. Furthermore, this is actually equivalent to finding the `mean` of a logical flag vector. For example, to find the proportion of chicks fed soybean, note that the following two calculations give identical results of around 0.197:

```
R> sum(chickwts$feed=="soybean")/nrow(chickwts)
[1] 0.1971831
R> mean(chickwts$feed=="soybean")
[1] 0.1971831
```

You can use this approach to calculate the proportion of entities in combined groups, achieved easily through logical operators (see Section 4.1.3). The proportion of chicks fed either soybean *or* horsebean is as follows:

```
R> mean(chickwts$feed=="soybean"|chickwts$feed=="horsebean")
[1] 0.3380282
```

An alternative way to get the proportions of chicks on each diet is to appeal to `tapply` and define the `FUN` argument to be an anonymous function (refer to Section 11.3.2) that performs the required calculation. Notice that

```
R> tapply(chickwts$weight,INDEX=chickwts$feed,
          FUN=function(x) length(x)/nrow(chickwts))
   casein horsebean   linseed  meatmeal   soybean sunflower
0.1690141 0.1408451 0.1690141 0.1549296 0.1971831 0.1690141
```

returns the same results we found using `table`. Here, the anonymous function is defined with a dummy argument `x`, which you're using to represent the vector of weights in each feed group to which `FUN` applies. Finding the desired proportion is therefore simply a case of dividing the number of observations in `x` by the total number of observations.

It's worth briefly remarking on the relationship between a proportion and a percentage. The two represent the same thing. The only difference is the scale; the *percentage* is merely the proportion multiplied by 100. The percentage of chicks on a soybean diet is therefore approximately 19.7 percent:

```
R> mean(chickwts$feed=="soybean")*100
[1] 19.71831
```

Since proportions always lie in the interval [0, 1], percentages always lie within [0, 100].

Most statisticians use proportions as opposed to percentages in part due to the role proportions play in the direct representation of *probabilities* (discussed in Chapter 15). However, in some situations,

percentages are more useful, such as in basic data summaries or in the definition of *percentiles*, which will be detailed in Section 13.2.4.

13.2.3 Rounding, Floors, Ceilings, and Absolute Values

Sometimes you'll need to manipulate results for use in subsequent calculations or simply for a neater display. Common numeric operations such as rounding, directional rounding via the floor and ceiling operators, and taking the absolute value of a given number to ignore its sign, are all readily accessible in R.

The round Function

The `round` function is of particular use when summarizing numeric values. As its name suggests, you use this to round numeric values to a certain number of decimal places (d.p.). You need only supply to `round` your numeric vector (or matrix or any other appropriate data structure) and however many decimal places (as the argument `digits`) you want your figures rounded to. For example:

```
R> round(table(chickwts$feed)/nrow(chickwts),digits=3)

   casein horsebean   linseed  meatmeal   soybean sunflower
    0.169     0.141     0.169     0.155     0.197     0.169
```

This provides output that's easier to read at a glance and may be nicer to display than something with many decimal places if you're using these results to, say, annotate a plot. If you set `digits=0` (the default), output rounds to the nearest integer.

You should use rounding with care. When working through a series of calculations, be careful not to round preemptively; this can adversely impact the precision of any final results. For example, consider the following:

```
R> 4.5743 * sqrt(0.9732 * 0.0268)
[1] 0.7387425
```

Here, we have the result of a calculation involving three specific values recorded to 4 d.p.–precision. Suppose we performed the same calculation as $4.57 \times \sqrt{0.97 \times 0.03}$, that is, using 2 d.p.–rounded versions of the original three values:

```
R> round(4.5743,2) * sqrt(round(0.9732,2) * round(0.0268,2))
[1] 0.7795836
```

There is a notable difference in the final result, due purely to the premature rounding of the numbers. The lesson is clear: you should use rounding only on a final result, and interim values in any given calculation must always retain as much precision as we have available.

Floor and Ceiling Operations

Think of floor and ceiling operations as special cases of the more general rounding operation, which round a value either strictly down to the nearest integer (taking the *floor*) or strictly up to the nearest integer (taking the *ceiling*). Mathematically, we typically use the notation $\lfloor x \rfloor$ to denote the floor of some value x, and $\lceil x \rceil$ to denote the ceiling.

The R functions `floor` and `ceiling` perform these operations. Note that the directional nature of the rounding is strictly enforced regardless of the value at hand. For instance, if you were asked to round the value 9.83 to the nearest whole number, you would naturally answer 10. A floor operation will round strictly down, shown here by the fact that $\lfloor 9.83 \rfloor = 9$ as well as the value $\lfloor 5.12 \rfloor = 5$:

```
R> x <- 9.83
R> y <- 5.12
R> floor(x)
[1] 9
R> floor(y)
[1] 5
```

Similarly, the ceiling operator forces an upward round, so that $\lceil 9.83 \rceil = 10$ and $\lceil 5.12 \rceil = 6$:

```
R> ceiling(x)
[1] 10
R> ceiling(y)
[1] 6
```

In a familiar fashion to many R functions, `floor` and `ceiling` will work en masse when supplied a vector of numeric values.

Absolute Values

You can think of the *absolute value* of a number—formally known as yielding the overall magnitude of the value at hand—as a way to make the argument value positive. For some number x, we denote its absolute value as $|x|$, where $|x|$ simply returns x itself if it is positive, or returns $-x$ if x is negative. For example, $|4.48| = 4.48$; at the same time, $|-4.48| = 4.48$.

Let's use the relevant R command, `abs`, to confirm the previous results:

```
R> x <- 4.48
R> y <- -4.48
R> abs(x)
[1] 4.48
R> abs(y)
[1] 4.48
```

Just like `round`, `floor`, and `ceiling`, you can also supply a vector of values to `abs` if you want to find the absolute value of several numbers at once.

Exercise 13.2

a. With respect to the built-in `quakes` data frame (Section 13.1.4), complete the following tasks:

i. Obtain the proportion of seismic events in the `quakes` data frame that occurred at a depth of 300 km or deeper. Round the result to two decimal places.

ii. Calculate the mean and median magnitudes of the events that occurred at a depth of 300 km or deeper.

b. Turn your attention back to the built-in `chickwts` data frame (see Section 13.1.3). Write a `for` loop that prints the mean weight of chicks for each feed type to the R console—the same as the results given by the `tapply` function in Section 13.2.1. When printing, display the results rounded to one decimal place and ensure each mean is in some way annotated with the corresponding feed type.

c. Another ready-to-use dataset in the automatically loaded `datasets` package is `InsectSprays`. It contains data on the number of insects found on various agricultural units, as well as the type of insect spray that was used on each unit. Ensure you can access the data frame at the prompt, then study the help file `?InsectSprays` to get an idea of R's representation of the two variables. Complete the following tasks:

i. Identify the two variable types in `InsectSprays` (as per the definitions in Section 13.1.1 and Section 13.1.2).

ii. Calculate the modes of the distribution of insect counts (ignoring spray type).

iii. Use `tapply` with `sum` to report the total insect counts by each spray type.

iv. Using the same kind of `for` loop in (b), compute the percentage of agricultural units in each spray

type group that had at least five bugs on them. When printing to the screen, round the percentages to the nearest whole number.
Hint: Within the loop, for each spray type, you can use `mean` to obtain the proportion of agricultural units that satisfy the condition (as you saw in Section 13.2.2 on page 314) before converting these to percentages and rounding.

v. Repeat the previous task using `tapply` and an anonymous function to obtain the same numeric results with rounding. You should base this anonymous function on the same operation used in each iteration of your previously coded loop.

13.2.4 Quantiles, Percentiles, and the "Five-Number Summary"

Understanding how a collection of numeric observations are *distributed* is an important statistical concept that will form a key feature of discussions in Chapter 15 onward.

You can gain more insight into the distribution of a set of observations by examining quantiles. A *quantile* is a value computed from a collection of numeric measurements that indicates an observation's rank when compared to all the other present observations. For example, the median (Section 13.2.1) is a quantile, as it gives you a value below which half of the measurements lie; it's the 0.5th quantile.

Alternatively, you can express quantiles as *percentiles*; these are equivalent on a "percent scale" of 0 to 100. In other words, the pth quantile is equivalent to the $100 \times p$th percentile. The median, therefore, is the 50th percentile.

You can use a number of different algorithms to compute quantiles and percentiles. They all work by sorting the observations from smallest to largest and using some form of weighted average to find the numeric

value that corresponds to p, but results may vary slightly in other statistical software.

To obtain quantiles and percentiles in R, use the `quantile` function. Using the eight observations stored as the vector `xdata`, you can confirm the 0.8th quantile (or 80th percentile) as 3.6:

```
R> xdata <- c(2,4.4,3,3,2,2.2,2,4)
R> quantile(xdata,prob=0.8)
80%
3.6
```

As you can see, `quantile` takes the data vector of interest as its first argument, followed by a numeric value supplied to `prob`, giving the quantile of interest. In fact, `prob` can take a numeric vector of quantile values. This is convenient when multiple quantiles are desired:

```
R> quantile(xdata,prob=c(0,0.25,0.5,0.75,1))
  0%  25%  50%  75% 100%
2.00 2.00 2.60 3.25 4.40
```

Here, you use `quantile` to obtain the *five-number summary* of `xdata`, consisting of the 0th percentile (the minimum), the 25th percentile, the 50th percentile, the 75th percentile, and the 100th percentile (the maximum). We refer to the 0.25th quantile as the *first* or *lower quartile* and the 0.75th quantile as the *third* or *upper quartile*. Also note that the 0.5th quantile of `xdata` is equivalent to the median (2.6, calculated in Section 13.2.1 using `median`). The median is the second quartile, with the maximum value being the fourth quartile.

There are ways to obtain the five-number summary other than using `quantile`; when applied to a numeric vector, the `summary` function also provides these statistics, along with the mean, automatically:

```
R> summary(xdata)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.000   2.600   2.825   3.250   4.400
```

To look at some examples using real data, let's compute the lower and upper quartiles of the weights of the chicks in the `chickwts`:

```
R> quantile(chickwts$weight,prob=c(0.25,0.75))
  25%   75%
204.5 323.5
```

This indicates that 25 percent of the weights lie at or below 204.5 grams and that 75 percent of the weights lie at or below 323.5 grams.

Let's also compute the five-number summary (along with the mean) of the magnitude of the seismic events off the coast of Fiji that occurred at a depth of less than 400 km, using the `quakes` data frame:

```
R> summary(quakes$mag[quakes$depth<400])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4.00    4.40    4.60    4.67    4.90    6.40
```

This begins to highlight the utility of quantiles when interpreting the distribution of numeric measurements. From these results, you can see that most of the magnitudes of events at a depth of less than 400 km lie around 4.6, the median, and the first and third quartiles are just 4.4 and 4.9, respectively. But you can also see that the maximum value is much further away from the upper quartile than the minimum is from the lower quartile, suggesting a *skewed* distribution, one that stretches more positively (in other words, to the right) from its center than negatively (in other words, to the left). This notion is also supported by the fact that the mean is greater than the median; the mean is being "dragged upward" by the extended tail of larger values.

You'll explore this further in Chapter 14 when you investigate datasets using basic statistical plots, and Chapter 15 will formalize some of the associated terminology.

13.2.5 Measures of Spread

The measures of centrality explored in Section 13.2.1 offer a good indication of where your numeric measurements are massed, but the

mean, median, and mode do nothing to describe how *dispersed* your data are. This requires measures of *spread*.

An Illustrative Example

In addition to your vector of eight hypothetical observations, which is given again here,

```
R> xdata <- c(2,4.4,3,3,2,2.2,2,4)
```

you'll also look at another eight observations stored as follows:

```
R> ydata <- c(1,4.4,1,3,2,2.2,2,7)
```

Although these are two different collections of numbers, note that they have an identical arithmetic mean:

```
R> mean(xdata)
[1] 2.825
R> mean(ydata)
[1] 2.825
```

Now let's plot these two data vectors side by side, each one on a horizontal line, by executing the following:

```
R> plot(xdata,type="n",xlab="",ylab="data vector",yaxt="n",bty="n")
R> abline(h=c(3,3.5),lty=2,col="gray")
R> abline(v=2.825,lwd=2,lty=3)
R> text(c(0.8,0.8),c(3,3.5),labels=c("x","y"))
R> points(jitter(c(xdata,ydata)),c(rep(3,length(xdata)),
                                   rep(3.5,length(ydata))))
```

You saw how to use these base R graphics functions in Chapter 7, though because some of the observations in `xdata` and in `ydata` occur more than once, you may wish to randomly alter them slightly to prevent overplotting, which aids in the visual interpretation. We call this step *jittering*, which is achieved by passing the numeric vector of interest to the `jitter` function prior to plotting with `points`. Additionally, note that you can use `yaxt="n"` in any call to `plot` to suppress the y-axis;

similarly, `bty="n"` removes the typical box that's placed around a plot (you'll focus more on this type of plot customization in Chapter 23).

Figure 13-3 shows the result, which provides you with valuable information. Though the mean is the same for both `xdata` and `ydata`, you can see that the observations in `ydata` are more "spread out" around the measure of centrality than the observations in `xdata`. Common descriptors of this variability include the *variance*, the *standard deviation*, and the *interquartile range*.

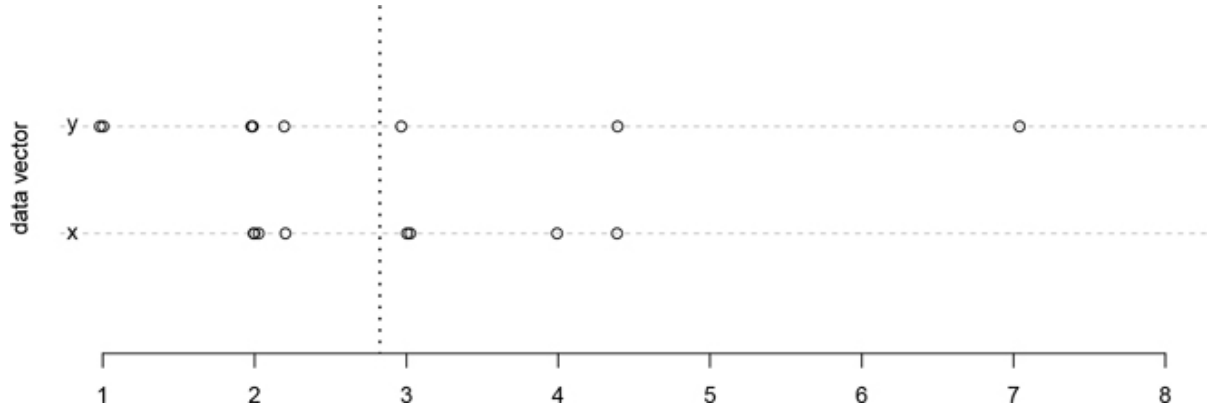


Figure 13-3: Comparing two hypothetical data vectors that share an identical arithmetic mean (marked by the vertical dotted line) but have different magnitudes of spread. Identical observations are jittered slightly.

Variance

The sample variance measures the degree of the spread of numeric observations around their arithmetic mean. The variance is a particular representation of the average squared distance of the observations away from their mean. For a set of n numeric measurements labeled $x = \{x_1, x_2, \ldots, x_n\}$, the sample variance s_x^2 is given by the following, where $\bar{x}$ is the sample mean described in Equation (13.1):

$$s_x^2 = \frac{(x_1-\bar{x})^2+(x_2-\bar{x})^2+\ldots+(x_n-\bar{x})^2}{n-1} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \qquad (13.3)$$

For example, if you take the eight illustrative observations 2, 4.4, 3, 3, 2, 2.2, 2, 4, their sample variance is as follows when rounded to three decimal places (some terms are hidden with . . . for readability):

$$\begin{aligned}
&\frac{(2-2.825)^2+(4.4-2.825)^2+\ldots+(4-2.825)^2}{7} \\
&= \frac{(-0.825)^2+(1.575)^2+\ldots+(1.175)^2}{7} \\
&= \frac{6.355}{7} = 0.908
\end{aligned}$$

Standard Deviation

The standard deviation is simply the square root of the variance; it provides a value interpretable on the scale of the original observations. With the same notation for a sample of *n* observations, we calculate the sample standard deviation *s* by taking the square root of Equation (13.3):

$$s_x = \sqrt{s^2} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - x\right)^2} \qquad (13.4)$$

For the sample variance calculated earlier, the standard deviation of the hypothetical observations is as follows (to three decimal places):

$$\sqrt{0.908} = 0.953$$

Thus, a rough way to interpret this is that 0.953 effectively represents the average distance points will fall away from the mean.

Interquartile Range

Unlike the variance and standard deviation, you don't compute *interquartile range (IQR)* with respect to the sample mean. The IQR measures the width of the "middle 50 percent" of the data, that is, the range of values that lie within a 25 percent quartile on either side of the median. As such, you calculate the IQR as the difference between the upper and lower quartiles of your data. Formally, where $Q_x(\cdot)$ denotes the quantile function (as defined in Section 13.2.4), the IQR is given as:

$$\text{IQR}_x = Q_x\left(0.75\right) - Q_x\left(0.25\right) \qquad (13.5)$$

R Implementation

The direct R commands for computing these three measures of spread are `var` (variance), `sd` (standard deviation), and `IQR` (interquartile range):

```
R> var(xdata)
[1] 0.9078571
R> sd(xdata)
[1] 0.9528154
R> IQR(xdata)
[1] 1.25
```

You can confirm the relationship between the sample variance and standard deviation using the square root function `sqrt` on the result from `var`, and you can reproduce the IQR by calculating the difference between the third and first quartiles:

```
R> sqrt(var(xdata))
[1] 0.9528154
R> as.numeric(quantile(xdata,0.75)-quantile(xdata,0.25))
[1] 1.25
```

Note that `as.numeric` (see Section 6.2.4) is just used to strip away the percentile annotations from the returned object of `quantile` (which is a named vector; see Section 2.3.1).

Now do the same with the `ydata` observations that had the same arithmetic mean as `xdata`. The calculations give you the following:

```
R> sd(ydata)
[1] 2.012639
R> IQR(ydata)
[1] 1.6
```

Remember that `ydata` is on the same scale as `xdata`, so the results confirm what you can see in Figure 13-3: that the observations in the former are more spread out than in the latter.

For two quick final examples, let's return again to the `chickwts` and `quakes` datasets. In Section 13.2.1, you saw that the mean weight of all the chicks is 261.3099 grams. You can now find that the standard deviation of the weights is as follows:

```
R> sd(chickwts$weight)
[1] 78.0737
```

Informally, this implies that the weights of chicks are on average around 78.1 grams away from the mean weight (technically, though, remember it is merely the square root of a function of the squared distances).

NOTE

The definition of the variance (and hence the standard deviation) here has referred exclusively to the "sample estimator," the default in R, which uses the divisor of n – *1 in the formula. This is the formula used when the observations at hand represent a sample of an assumed larger population. In these cases, use of the divisor* n – *1 is more accurate, providing what's known as an* unbiased *estimate of the true population value. Thus, you aren't exactly calculating the "average squared distance" of the present observations, though it can loosely be thought of as such and does indeed approach this as the sample size increases.*

Lastly, in Section 13.2.4 you used `summary` to obtain the five-number summary of the magnitudes of some of the earthquakes in the `quakes` dataset. Looking at the first and third quartiles in these earlier results (4.4 and 4.9, respectively), you can quickly determine that the IQR of this subset of the events is 0.5. Confirm this with `IQR`:

```
R> IQR(quakes$mag[quakes$depth<400])
[1] 0.5
```

This gives you the width, in units of the Richter scale, of the middle 50 percent of the observations.

Exercise 13.3

a. Consider again the built-in `chickwts` data frame.

   i. Compute the 10th, 30th, and 90th percentiles of all the chick weights.

   ii. Use `tapply` to determine which feed type is associated with the highest sample variance of weights.

b. Turn to the seismic event data in the built-in `quakes` object and complete the following tasks:

   i. Find the IQR of the recorded depths.

   ii. Find the five-number summary of all magnitudes of seismic events that occur at a depth of 400 km or deeper. Compare this to the summary values found in Section 13.2.4 of those events occurring at less than 400 km and briefly comment on what you notice.

   iii. Use your knowledge of `cut` (Section 4.3.7) to create a new factor vector called `depthcat` that identifies four evenly spaced categories of `quakes$depth` spanning the range of the recorded depths (remember the functions `min`, `max`, and `range` seen in Section 13.2.1). Use the appropriate arguments to `cut` such that when executing `levels(depthcat)`, it gives the following:

```
R> levels(depthcat)
[1] "[40,200)"  "[200,360)" "[360,520)" "[520,680]"
```

   iv. Use `tapply` to find the sample mean and standard deviation of the magnitudes of the events associated with each category of depth according to `depthcat`.

v. Use `tapply` to compute the 0.8th quantile of the magnitudes of the seismic events in `quakes`, split by `depthcat`.

13.2.6 Relationships Between Numeric Variables

When analyzing data, it's often useful to be able to investigate the relationship between two numeric variables to assess trends. For example, you might expect height and weight observations to have a noticeable positive relationship; taller people tend to weigh more. Conversely, you might imagine that handspan and length of hair would have less of an association. One of the simplest and most common ways to quantify and compare such associations is through the idea of correlation, for which you need the covariance.

Covariance

The *covariance* expresses how much two numeric variables "change together" and the nature of that relationship, whether it is positive or negative. Suppose for n individuals you have a sample of observations for two variables, labeled $x = \{x_1, x_2, \ldots, x_n\}$ and $y = \{y_1, y_2, \ldots, y_n\}$, where x_i corresponds to y_i for $i = 1, \ldots, n$. Compute the sample covariance r_{xy} with the following, where x and y represent the respective sample means of both sets of observations:

$$r_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - x)(y_i - y) \qquad (13.6)$$

When you get a positive result for r_{xy}, it shows that there is a positive linear relationship: As x increases, y increases. When you get a negative result, it shows a negative linear relationship: As x increases, y decreases, and vice versa. When $r_{xy} = 0$, this indicates that there is no linear relationship between the values of x and y. Note that the order of the variables in the formula itself doesn't matter; in other words, $r_{xy} \equiv r_{yx}$.

To demonstrate, let's use the original eight illustrative observations, which I'll denote here with x = {2, 4.4, 3, 3, 2, 2.2, 2, 4}, and the additional eight observations denoted with y = {1, 4.4, 1, 3, 2, 2.2, 2, 7}. Remember that both x and y have sample means of 2.825. The sample covariance of these two sets of observations is as follows (rounded to three decimal places):

$$\begin{aligned}&\frac{(2-2.825)\times(1-2.285)+\ldots+(4-2.825)\times(7-2.825)}{7}\\ &= \frac{(-0.825)(-1.825)+\ldots+(1.175)(4.175)}{7}\\ &= \frac{10.355}{7} = 1.479\end{aligned}$$

The figure is a positive number, so this suggests there is a positive relationship based on the observations in x and y.

Correlation

Correlation allows you to interpret the covariance further by identifying both the direction and the strength of any association. There are different types of correlation coefficients, but the most common of these is *Pearson's product-moment correlation coefficient*, the default implemented by R and the estimator for which I will use in this chapter. Compute Pearson's sample correlation coefficient ρ_{xy} by dividing the sample covariance by the product of the standard deviation of each dataset. Formally, where r_{xy} corresponds to Equation (13.6) and s_x and s_y to Equation (13.4),

$$\rho_{xy} = \frac{r_{xy}}{s_x s_y} \qquad (13.7)$$

which ensures that $-1 \leq \rho_{xy} \leq 1$.

When $\rho_{xy} = -1$, a perfect negative linear relationship exists. Any result less than zero shows a negative relationship, and the relationship gets weaker the nearer to zero the coefficient gets, until $\rho_{xy} = 0$, implying no relationship between x and y. As the coefficient increases

above zero, a positive relationship is shown, until $\rho_{xy} = 1$, which is a perfect positive linear relationship.

If you take the standard deviations already computed for x and y in Section 13.2.5 ($s_x = 0.953$ and $s_y = 2.013$ to three decimal places), you find the following to three decimal places:

$$\frac{1.479}{0.953 \times 2.013} = 0.771$$

The value ρ_{xy} is positive just like r_{xy}; 0.771 indicates a moderate-to-strong positive association between the observations in x and y. Again, $\rho_{xy} \equiv \rho_{yx}$.

R Implementation

We use the R commands `cov` and `cor` for the sample covariance and correlation; you need only to supply the two corresponding vectors of data:

```
R> xdata <- c(2,4.4,3,3,2,2.2,2,4)
R> ydata <- c(1,4.4,1,3,2,2.2,2,7)
R> cov(xdata,ydata)
[1] 1.479286
R> cov(xdata,ydata)/(sd(xdata)*sd(ydata))
[1] 0.7713962
R> cor(xdata,ydata)
[1] 0.7713962
```

You can plot these bivariate observations as a coordinate-based plot, or *scatterplot* (see more examples in Section 14.6). Executing the following

```
R> plot(xdata,ydata,pch=13,cex=1.5)
```

gives you Figure 13-4.

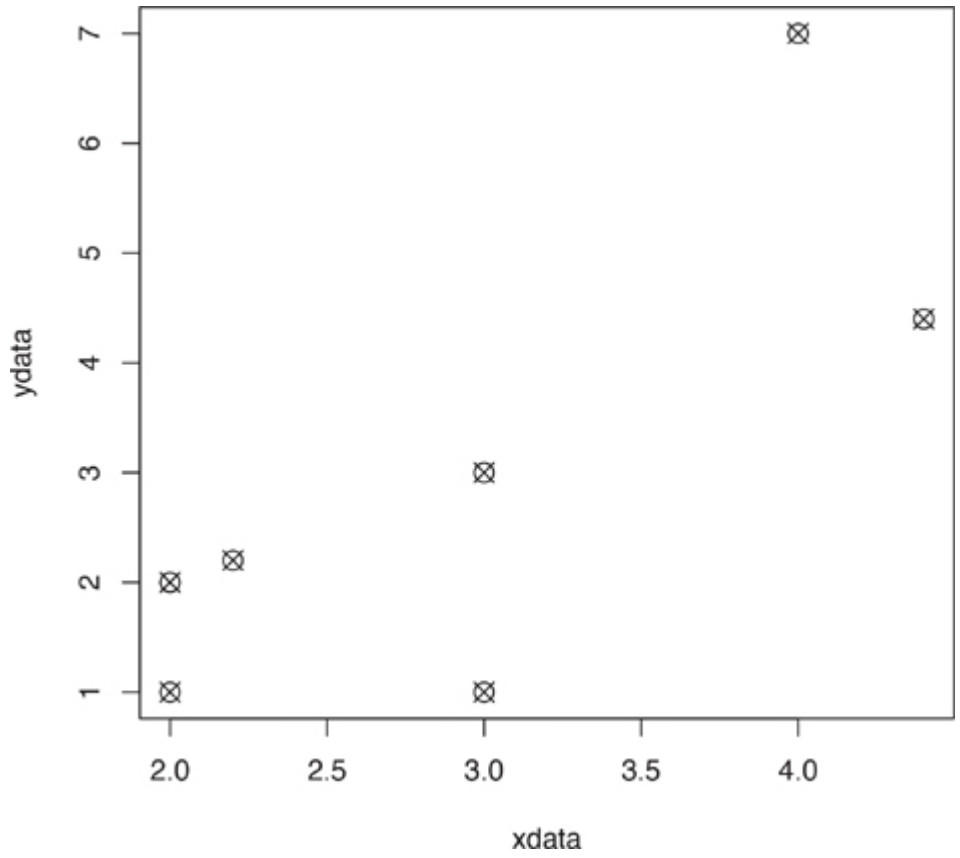


Figure 13-4: Plotting the `xdata` *and* `ydata` *observations as bivariate data points to illustrate the interpretation of the correlation coefficient*

As discussed earlier, the correlation coefficient estimates the nature of the *linear* relationship between two sets of observations, so if you look at the pattern formed by the points in Figure 13-4 and imagine drawing a perfectly straight line that best represents all the points, you can determine the strength of the linear association by how close those points are to your line. Points closer to a perfect straight line will have a value of ρ_{xy} closer to either –1 or 1. The direction is determined by how the line is sloped: An increasing trend, with the line sloping upward toward the right, indicates positive correlation; a negative trend would be shown by the line sloping downward toward the right. Considering this, you can see that the estimated correlation coefficient for the data plotted in Figure 13-4 makes sense. The points do appear to increase together as a rough straight line in terms of the values in `xdata` and `ydata`, but this linear association is by no means perfect. We discuss how to

compute the "ideal" or "best" straight line to fit such data in Chapter 20.

To further aid your understanding of the idea of correlation, Figure 13-5 displays nine different scatterplots, each showing exactly 100 points. These observations have been randomly and artificially generated to follow preset "true" values of ρ_{xy}, labeled above each plot.

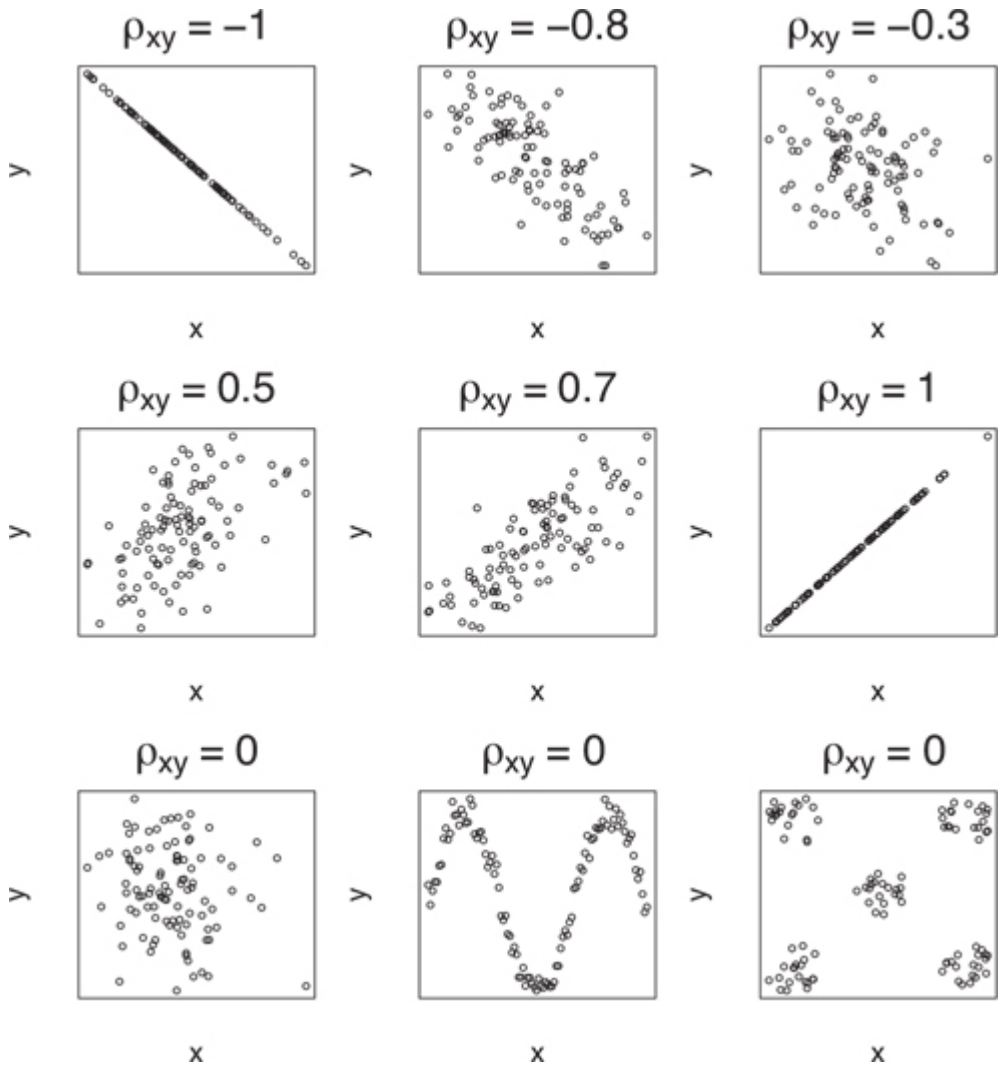


Figure 13-5: Artificial x *and* y *observations, generated to illustrate a given value of the correlation coefficient*

The first row of scatterplots shows negatively correlated data; the second shows positively correlated data. These match what you expect to see: The direction of the line shows the negative or positive

correlation of the trend, and the extremity of the coefficient corresponds to the closeness to a "perfect line." The final row shows datasets generated with a correlation coefficient set to zero, implying no linear relationship between the observations in *x* and *y*. The middle and rightmost plots are particularly important because they highlight the fact that Pearson's correlation coefficient identifies only "straight-line" relationships; these last two plots clearly show some kind of trend or pattern, but you cannot use this particular statistic to detect such a trend.

To wrap up this section, look again at the `quakes` data. Two of the variables are `mag` (the magnitude of each event) and `stations` (the number of stations that reported detection of the event). A plot of `stations` on the y-axis against `mag` on the x-axis can be created with the following line

```
R> plot(quakes$mag,quakes$stations,xlab="Magnitude",ylab="No. of stations")
```

which produces Figure 13-6.

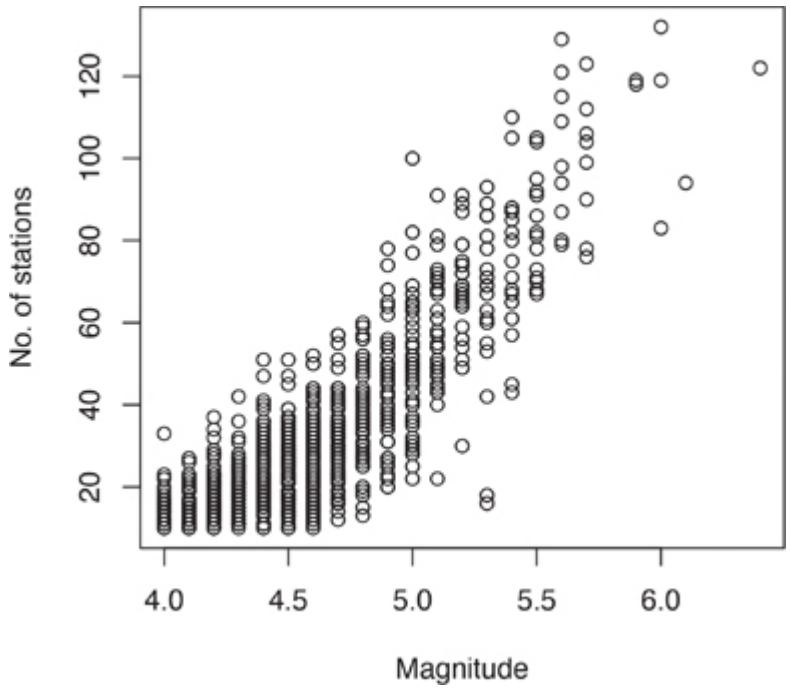


*Figure 13-6: Plotting the number of stations reporting the event (*y*) and the magnitude (*x*) of each event in the* `quakes` *data frame*

The vertical patterning shows that the magnitudes appear to have been recorded to a certain specific level of precision (this is owed to the difficulty associated with measuring earthquake magnitudes exactly). Nevertheless, a positive relationship (more stations tend to detect events of higher magnitude) is clearly visible in the scatterplot, a feature that is confirmed by a positive covariance:

```
R> cov(quakes$mag,quakes$stations)
[1] 7.508181
```

This directly corresponds to a strong linear relationship gauged by Pearson's correlation coefficient:

```
R> cor(quakes$mag,quakes$stations)
[1] 0.8511824
```

NOTE

Now is an apt time to remember the golden rule that correlation does not imply causation. *When you detect a high correlative effect between two variables, this does not mean that one* causes *the other. Causation is difficult to prove in even the most controlled situations. Correlation merely allows you to measure* association.

As mentioned earlier, you can use other representations of correlation; *rank* coefficients, such as *Spearman's* and *Kendall's* correlation coefficients, differ from Pearson's estimate in that they do not require the relationship to be linear. These are also available through the `cor` function by accessing the optional `method` argument (see `? cor` for details). Pearson's correlation coefficient is the most commonly used, however, and is linked directly to linear regression methods, which you'll start to examine in Chapter 20.

13.2.7 Outliers

In simple terms, an *outlier* is an observation that does not appear to "fit" with the story being told by the rest of the data. It is a noticeably extreme value when compared with the bulk of the data—in other words, an anomaly. In some cases, you might suspect that such an extreme observation has not actually come from the same mechanism that generated the other observations, but there is no hard-and-fast numeric rule as to what constitutes an outlier. For example, consider the 10 hypothetical data points in `foo`:

```
R> foo <- c(0.6,-0.6,0.1,-0.2,-1.0,0.4,0.3,-1.8,1.1,6.0)
```

Using skills from Chapter 7 (and from creating Figure 13-3), you can plot `foo` on a line as follows:

```
R> plot(foo,rep(0,10),yaxt="n",ylab="",bty="n",cex=2,cex.axis=1.5,cex.lab=1.5)
R> abline(h=0,col="gray",lty=2)
R> arrows(5,0.5,5.9,0.1,lwd=2)
R> text(5,0.7,labels="outlier?",cex=3)
```

The left-hand side of Figure 13-7 shows the result.

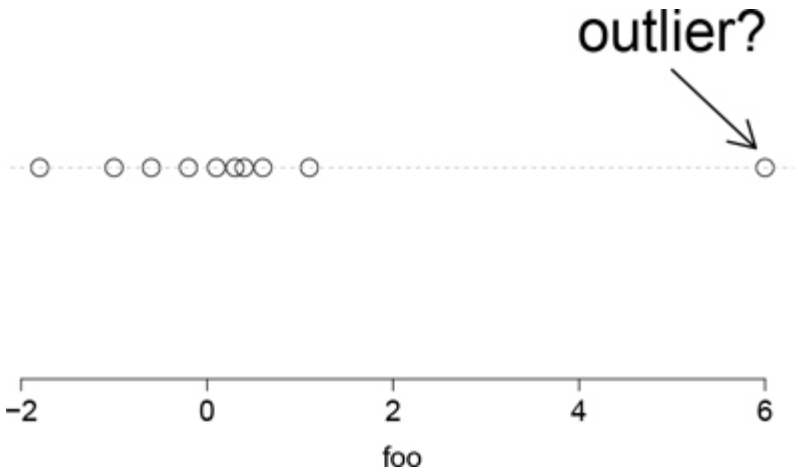


Figure 13-7: Illustrating the identification of outliers for univariate (left) and bivariate (right) data. Should you include such values in your statistical analysis? The answer can be difficult to determine.

From this plot, you see that most of the observations are centered around 0, but one value is way out at 6. To give a bivariate example, I'll use two further vectors, `bar` and `baz`:

```
R> bar <- c(0.1,0.3,1.3,0.6,0.2,-1.7,0.8,0.9,-0.8,-1.0)
R> baz <- c(-0.3,0.9,2.8,2.3,1.2,-4.1,-0.4,4.1,-2.3,-100.0)
```

I'll plot these data using the following code:

```
R> plot(bar,baz,axes=T,cex=2,cex.axis=1.5,cex.lab=1.5)
R> arrows(-0.5,-80,-0.94,-97,lwd=2)
R> text(-0.45,-74,labels="outlier?",cex=3)
```

The result is on the right of Figure 13-7.

It's important to identify outliers because of the potential impact they can have on any statistical calculations or model fitting. For this reason, many researchers will try to identify possible outliers before computing results by conducting an "exploratory" analysis of their data using basic summary statistics and data visualization tools (like those you'll look at in Chapter 14).

Outliers can occur naturally, where the outlier is a "true" or accurate observation recorded from the population, or unnaturally, where something has "contaminated" that particular contribution to the sample, such as a mistake made during data entry; the natural response is to simply omit any outliers occurring through unnatural sources prior to analysis. In practice this is not always easy because the true cause of an outlier may not be able to be determined. In some cases, researchers conduct their analysis both ways: presenting results including and excluding any perceived outliers.

With this in mind, if you return to the univariate example on the left in Figure 13-7, including all observations results in the following output for the sample mean:

```
R> mean(foo)
[1] 0.49
```

However, when the extreme value of 6 (the 10th observation) is deleted, you get the following:

```
R> mean(foo[-10])
[1] -0.1222222
```

This highlights the impact a single extreme observation can have. Without any additional information about the sample, however, it would be difficult to say whether it's sensible to exclude the outlier 6. The same kind of effect is noticeable if you compute, say, the correlation coefficient of `bar` with `baz`, shown on the right in Figure 13-7 (again, it's the 10th observation that is the possible outlier):

```
R> cor(bar,baz)
[1] 0.4566361
R> cor(bar[-10],baz[-10])
[1] 0.8898639
```

The estimated correlation becomes much stronger without that outlier.

Again, knowing whether to delete the outlier can be hard to correctly gauge in practice. At this stage, it's important simply to be aware of the impact outliers can have on an analysis and to perform at least a cursory inspection of the raw data before beginning more rigorous statistical investigations.

NOTE

The extent of the effect that extreme observations have on your data analysis depends not only on their extremity but also on the statistics you intend to calculate. The sample mean, for example, is highly sensitive to outliers and will differ greatly when including or excluding them, so any statistic that depends on the mean, like the variance or covariance, will be affected too. Quantiles and related statistics, such as the median or IQR, are relatively unaffected by outliers. In statistical parlance this property is referred to as robustness.

Exercise 13.4

a. In Exercise 7.1 (b) on page 159, you plotted height against weight measurements. Compute the correlation coefficient based on the observed data of these two variables.

b. For this task, you'll use the built-in `mtcars` dataset first seen in Section 9.3.1.

    i. Ensure you can access this data frame by entering `mtcars` at the prompt. Then inspect its help file to remind yourself of the variables present.

    ii. Two of the variables describe a vehicle's horsepower and shortest time taken to travel a quarter-mile distance. Using base R graphics, plot these two data vectors with horsepower on the x-axis and compute the correlation coefficient.

    iii. Through the help file, identify the variable in `mtcars` that corresponds to transmission type. Use your knowledge of factors in R to create a new factor from this variable called `tranfac`, where manual cars should be labeled `"manual"` and automatic cars `"auto"`.

    iv. Next, use `ggplot2` functionality to produce the same scatterplot as in (ii), but now use aesthetic mapping with `tranfac` to visually distinguish between manual and automatic cars (see Section 7.4.3).

    v. Finally, compute separate correlation coefficients for horsepower and quarter-mile time based on the transmission of the vehicles and, comparing these estimates with the overall value from (ii), briefly comment on what you find.

c. Return to the `chickwts` dataset to complete the following tasks:

i. Produce a plot like the left panel of Figure 13-7, based on the weights of chicks on the sunflower diet only. Note that one of the sunflower-fed chicks has a far lower weight than the others.

ii. Compute the standard deviation and IQR of the weights of the sunflower-fed chicks.

iii. Now suppose you're told that the lowest weight of the sunflower-fed chicks was caused by a certain illness, irrelevant to your research. Delete this observation and recalculate the standard deviation and IQR of the remaining sunflower chicks. Briefly comment on the difference in calculated values.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`mean`	Arithmetic mean	Section 13.2.1, p. 310
`median`	Median	Section 13.2.1, p. 310
`table`	Tabulate frequencies	Section 13.2.1, p. 310
`min`, `max`, `range`	Minimum and maximum	Section 13.2.1, p. 310
`round`	Round numeric values	Section 13.2.3, p. 315

Function/operator	Brief description	First occurrence
`floor`	Floor (round down) operator	Section 13.2.3, p. 316
`ceiling`	Ceiling (round up) operator	Section 13.2.3, p. 316
`abs`	Absolute value operator	Section 13.2.3, p. 316
`quantile`	Quantiles/percentiles	Section 13.2.4, p. 318
`summary`	Five-number summary	Section 13.2.4, p. 318
`jitter`	Jitter points in plotting	Section 13.2.5, p. 320
`var`, `sd`	Variance, standard deviation	Section 13.2.5, p. 322
`IQR`	Interquartile range	Section 13.2.5, p. 322
`cov`, `cor`	Covariance, correlation	Section 13.2.6, p. 325

14
BASIC DATA VISUALIZATION



Data visualization is an important part of statistical analysis. The visualization tools appropriate for a given dataset are dependent upon the types of variables (as per the definitions in Sections 13.1.1 and 13.1.2) for which you've made observations. In this chapter, you'll look at the most common types of plots used in data exploration. You'll study several examples, beginning each case by using base R graphics before examining the corresponding `ggplot2` functionality.

14.1 Bar Plots and Pie Charts

We commonly use bar plots and pie charts to visualize qualitative data by category frequency. In this section you'll learn how to generate both using R, though note that the former is generally preferable to the latter.

14.1.1 Bar Plot Creation

A *bar plot* draws either vertical or horizontal bars, typically separated by whitespace, to visualize frequencies according to the relevant categories.

Though the raw frequencies themselves are usually displayed, we can also use bar plots to visualize other quantities, such as means or proportions, which directly depend upon these frequencies.

As an example, let's use the `mtcars` dataset first seen in Section 9.3.1, and used in Exercise 13.4 (b) on page 331. Detailing various characteristics of 32 classic performance cars in the mid-1970s, the first five records are as follows:

```
R> mtcars[1:5,]
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

The documentation in `?mtcars` explains the variables. Of these, `cyl` provides the number of cylinders in each engine: four, six, or eight. To find out how many cars were observed with each number of cylinders, use `table`:

```
R> cyl.freq <- table(mtcars$cyl)
R> cyl.freq
 4  6  8
11  7 14
```

These frequencies are nicely displayed as a bar plot; the result of the following line

```
R> barplot(cyl.freq)
```

is on the left of Figure 14-1.

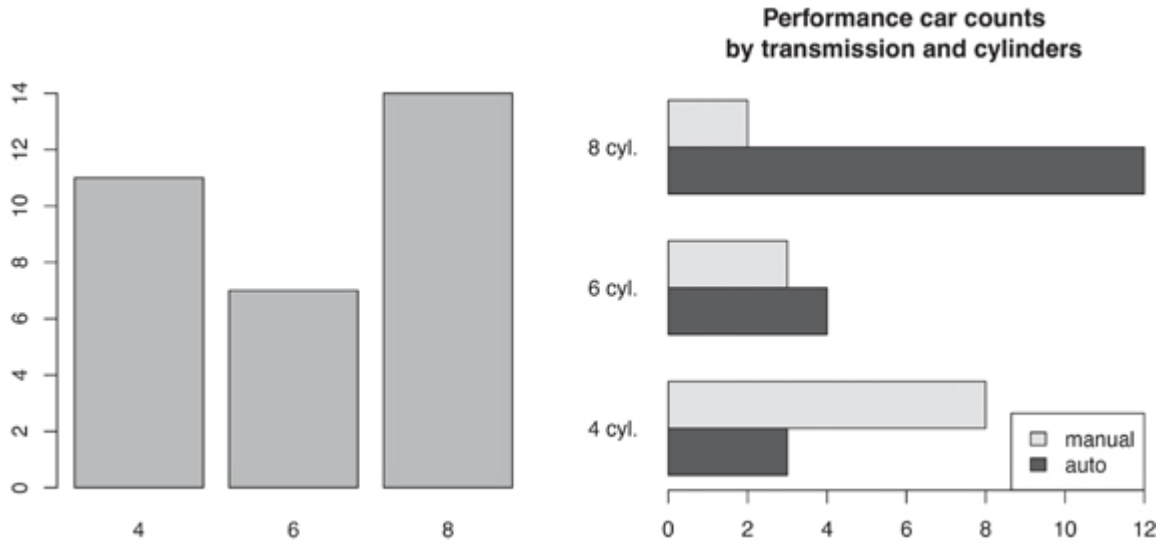


Figure 14-1: Two examples of bar plots of data from `mtcars` using base R graphics: the simplest, default version using one categorical variable (left) and a "dodged" bar plot illustrating various visual options and using two categorical variables (right)

This plot displays the number of four-, six-, and eight-cylinder cars in the dataset, but without annotations it isn't clear what's being summarized to anyone but yourself. Fortunately, it's easy to annotate such plots and, should we wish to convey more information, further split up the frequencies of each bar according to an additional categorical variable. Consider the following code where, this time, you're finding the counts associated with each category of `cyl` by transmission type (the variable `am`—remember to look at the help file `?mtcars` to understand the variables and their values):

```
R> table(mtcars$cyl[mtcars$am==0])
4  6  8
3  4 12

R> table(mtcars$cyl[mtcars$am==1])
4 6 8
8 3 2
```

If you aim to produce a bar plot that's *stacked* (where bars are split up vertically) or *dodged* (where bars are broken up and placed beside each other), `barplot` requests its first argument as a suitably arranged matrix.

You could construct it from the previous results using `matrix` or `rbind` (Section 3.1.1), but it's easier to just make use of a single call to `table` right off the bat:

```
R> cyl.freq.matrix <- table(mtcars$am,mtcars$cyl)
R> cyl.freq.matrix

   4  6  8
0  3  4 12
1  8  3  2
```

As you can see, you can cross-tabulate counts by supplying two categorical or discrete vectors of equal length to `table`; the first vector stipulates row counts, and the second defines the columns. The outcome is a matrix object; here, it's a 2×3 structure providing the quantities of the four-, six-, and eight-cylinder automatic cars in the first row and the quantities of the manual cars in the second. The rule is that each column of the bar plot will correspond to a column of the supplied matrix; these will be further split with respect to each row of the supplied matrix. The plot on the right of Figure 14-1 is the result of the following code:

```
R> barplot(cyl.freq.matrix,beside=TRUE,horiz=TRUE,las=1,
           main="Performance car counts\nby transmission and cylinders",
           names.arg=c("4 cyl.","6 cyl.","8 cyl."),
           legend.text=c("auto","manual"),args.legend=list(x="bottomright"))
```

The help file `?barplot` explains the options here in detail. To label the bars according to the categories of the column variable of the matrix that was initially passed to `barplot`, use a character vector of the appropriate length passed to `names.arg`. The options `beside=TRUE` and `horiz=TRUE` select a dodged, horizontal bar plot. If both options were `FALSE`, a stacked, vertical bar plot would be created. The argument `las=1` forces the labels on the vertical axis to appear horizontally, rather than parallel to it. You use the final two arguments, `legend.text` and `args.legend`, for the legend. While you could draw a legend separately via `legend` (see Section 7.3), using `legend.text` and `args.legend` automates the color assignment to

ensure the reference keys correctly match the shading of the bars themselves.

14.1.2 In ggplot2

You may produce similar plots using `ggplot2` via a `geom_bar` layer. Loading the installed package with `library("ggplot2")` and entering the following

```
R> ggplot(mtcars,mapping=aes(x=factor(cyl))) + geom_bar()
```

will produce the most basic bar plot, on the left of Figure 14-2.

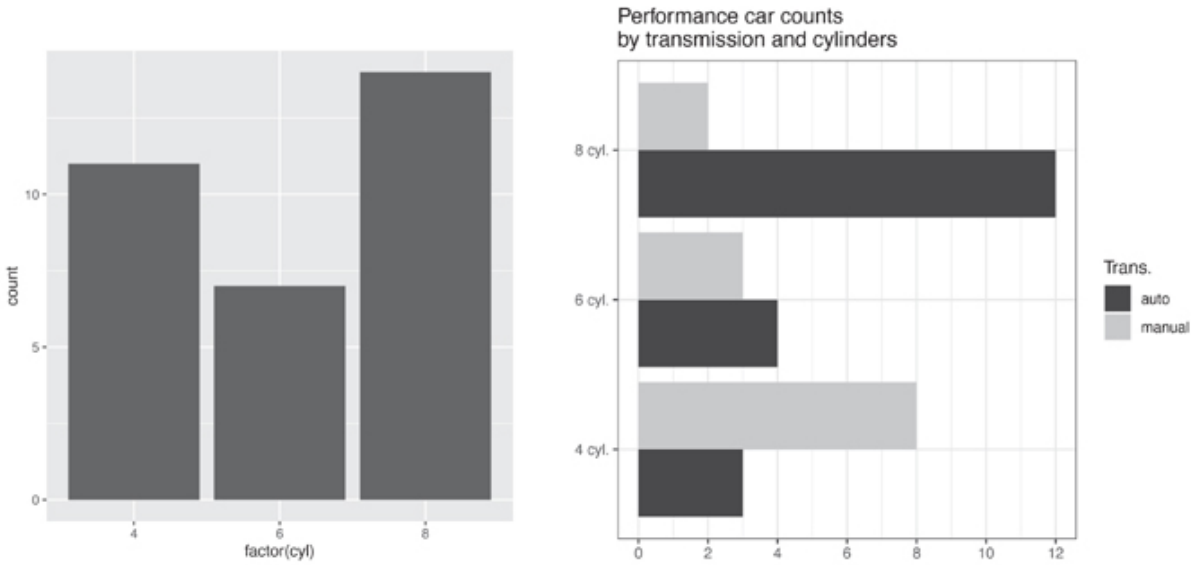


Figure 14-2: Two examples of bar plots of data from `mtcars` using `ggplot2` functionality: the most simple `qplot` version using one categorical variable (left) and a "dodged" bar plot, the same as in Figure 14-1, based on the supply of various additional geoms and scaling options (right)

Note here that you must supply the default mapping variable in `ggplot` as a factor for creating a bar plot (in `mtcars` the vector `mtcars$cyl` is just numeric, which is fine for the base R `barplot`, but `ggplot2` functionality is a bit more strict).

Again, you can create far more complicated images depending upon what you want to display. To produce a `ggplot2` version of the dodged bar plot from 14-1, call the following:

```
R> ggplot(mtcars,mapping=aes(x=factor(cyl),fill=factor(am))) +
       geom_bar(position="dodge") + coord_flip() + theme_bw()  +
       scale_x_discrete(labels=c("4 cyl.","6 cyl.","8 cyl.")) +
       scale_y_continuous(breaks=seq(0,12,2)) +
       scale_fill_grey(name="Trans.",labels=c("auto","manual")) +
       xlab("") + ylab("") +
       ggtitle("Performance car counts\nby transmission and cylinders")
```

The result is on the right in Figure 14-2.

Note a number of new additions to the basic `ggplot2` setup. The default x-axis mapping, by the factorized version of `cyl`, remains the same as earlier. You further specify that the bars should be `fill`ed according to a factor created by using the transmission variable `am`; so, each `cyl` bar is instructed to split according to that variable. It becomes a dodged bar plot through the use of `position="dodge"` when adding the `geom_bar` geom; as in base R graphics, the default behavior is to generate a stacked plot. Then, adding `coord_flip()` flips the axes to provide horizontal bars rather than the default vertical style.

The remaining additions control the colors and axis annotations. Adding `theme_bw()` changes the visual theme of the image; in the current example, I've chosen to remove the gray background because it's too similar in color to the manual car bars. The `scale_x_discrete` modifier specifies labels for each category of the default `cyl` mapping; the `scale_y_continuous` modifier is employed to control the axis labels for the frequencies (note that you should interpret the calls to the `scale_` functions with respect to the unflipped image). The default behavior of `fill` is to use colors, so you use the `scale_fill_grey` modifier to force this to be grayscale and to alter the labels of the automatically generated legend to match at the same time. Lastly, we customize a title via `ggtitle`.

While the overall volume of code for this version of the bar plot appears greater than that of the base R implementation, the most prominent advantage of using `ggplot2` in this case lies in the fact that you don't need to manually tabulate counts or design specific matrix structures of these frequencies; the variable mappings do this automatically. For practice, I encourage you to experiment with this

code example, omitting or modifying the additions to the `ggplot2` object we just created to assess the impact on the resulting image.

14.1.3 A Quick Pie Chart

The *pie chart* is an alternative option for visualizing frequency-based quantities across levels of categorical variables, with appropriately sized "slices" representing the relative counts of each categorical variable. For example:

```
R> pie(table(mtcars$cyl),labels=c("4 cyl.","6 cyl.","8 cyl."),
       col=c("white","gray","black"),main="Performance cars by cylinders")
```

Figure 14-3 shows the resulting plot.

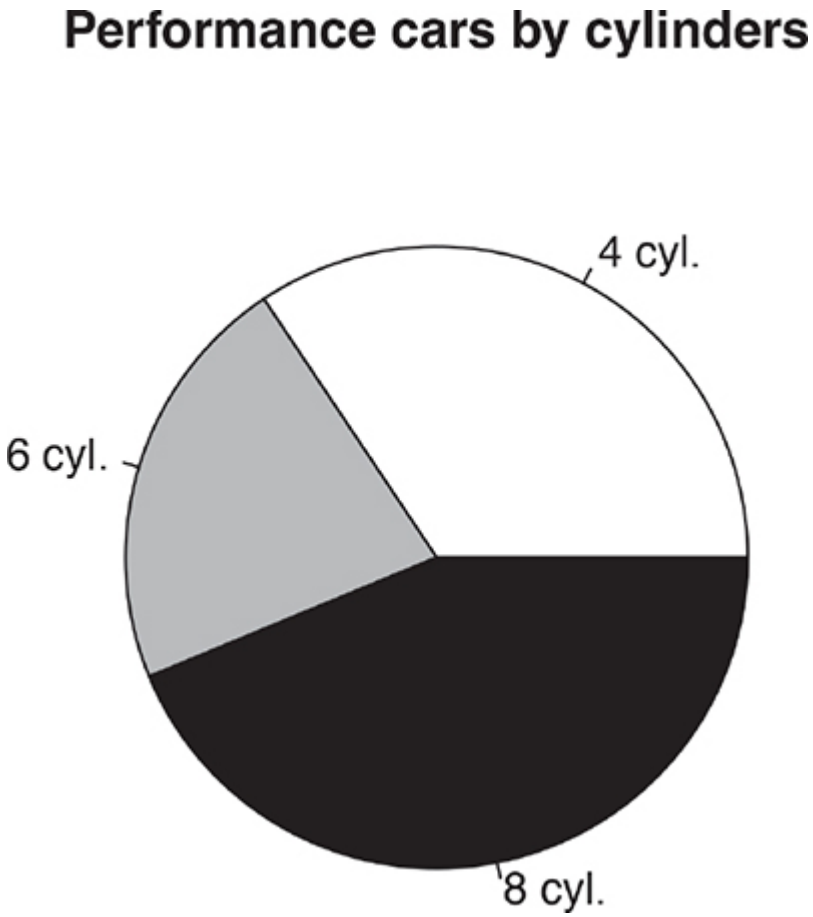


Figure 14-3: A pie chart of the frequencies of total cylinders of the cars in the `mtcars` data frame

Though it's possible to achieve (by changing the coordinate system used for a bar plot), there is currently no direct "pie" geom in `ggplot2`. This is in part reflective of the recommendation by statisticians and data scientists to simply avoid pie charts in general. That fact is even summarized in the help file `?pie`:

> Pie charts are a bad way of displaying information. The eye is good at judging linear measures and bad at judging relative areas.

Bar plots are far more useful than pie charts; the utility of bar plots is further increased if you want frequencies split by a second categorical variable or if the levels of a factor have some meaningful order.

Exercise 14.1

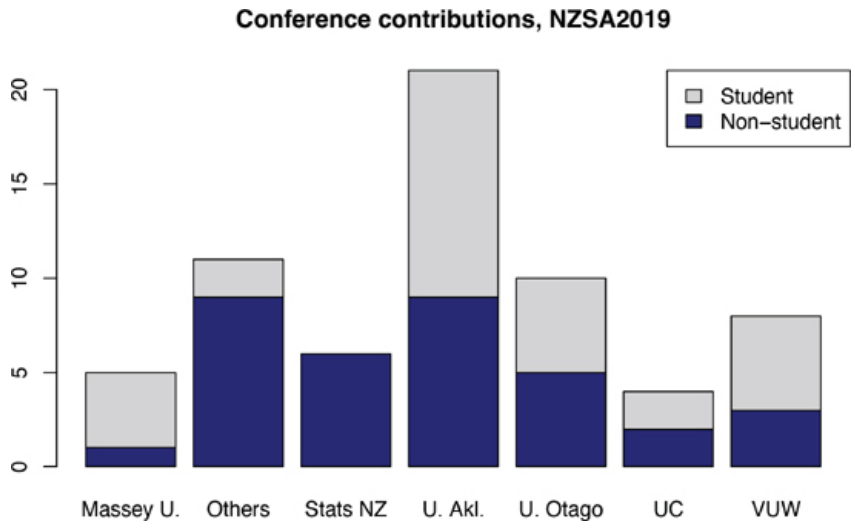
a. Recall the built-in `InsectSprays` data frame, which contains counts of insects on various agricultural units treated with one of six sprays. Obtain the total number of insects found according to each spray (see Exercise 13.2).
    i. Use base R graphics to produce a vertical bar plot of these totals. Appropriately label the axes.
    ii. Use base R graphics to produce a pie chart of these totals. Appropriately label the plot.

b. Among the downloadable resources for this book (see *https://nostarch.com/book-of-r-2nd-edition*) is the file *nzsa2019.rds*, a serialized data frame containing data on contributions to the New Zealand Statistical Association conference held in Dunedin in 2019. Obtain the data file and read it in using `readRDS` (covered in Section 8.4.2). The data comprise two columns of factor variables: `Status` (whether the presenter was a student or non-student) and

`Inst` (the host institution: either the University of Auckland, the University of Otago, Victoria University of Wellington, University of Canterbury, Massey University, Statistics New Zealand, or Other).

i. Create a two-dimensional table yielding the total number of contributions split by `Status` (as the rows) and `Inst` (as the columns). Your resulting table should thus have two rows and seven columns.

ii. Use these tabulated counts and base R graphics to generate a stacked bar plot of the contributions per institution, with the "stacking" colored with respect to the registration status. Ensure you have a neat title. Include a legend linking the two different colors within each bar to the appropriate status. Your final plot should look something like this:



iii. Reproduce this stacked bar plot using `ggplot2` functionality.

14.2 Histograms

The bar plot is intuitively sensible for counting observations in relation to categorical variables but is of virtually no use if the variable you're interested in is numeric-continuous. To visualize the distribution of continuous measurements, you can use a *histogram*, a tool that's sometimes confused with a bar plot owing to its similar appearance. A histogram also measures frequencies, but in targeting a numeric-continuous variable, it's first necessary to *bin* the observed data, meaning to define intervals and then count the number of continuous observations that fall within each one. The size of this interval is the *bin width*; the set of values that delineate the bins are the *breakpoints* or simply the *breaks*.

14.2.1 A Simple Example

For a simple example of a histogram, consider the horsepower data of the 32 cars in `mtcars`, given in the fourth column, named `hp`:

```
R> mtcars$hp
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66
[19]  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

For the purpose of discussion, consider the horsepowers of all performance cars from that era as your population and assume that these 32 observations represent a sample from that population (see Section 13.1.5). Using base R graphics, the `hist` command takes a vector of numeric-continuous observations and produces a histogram:

```
R> hist(mtcars$hp)
```

The result is on the left in Figure 14-4.

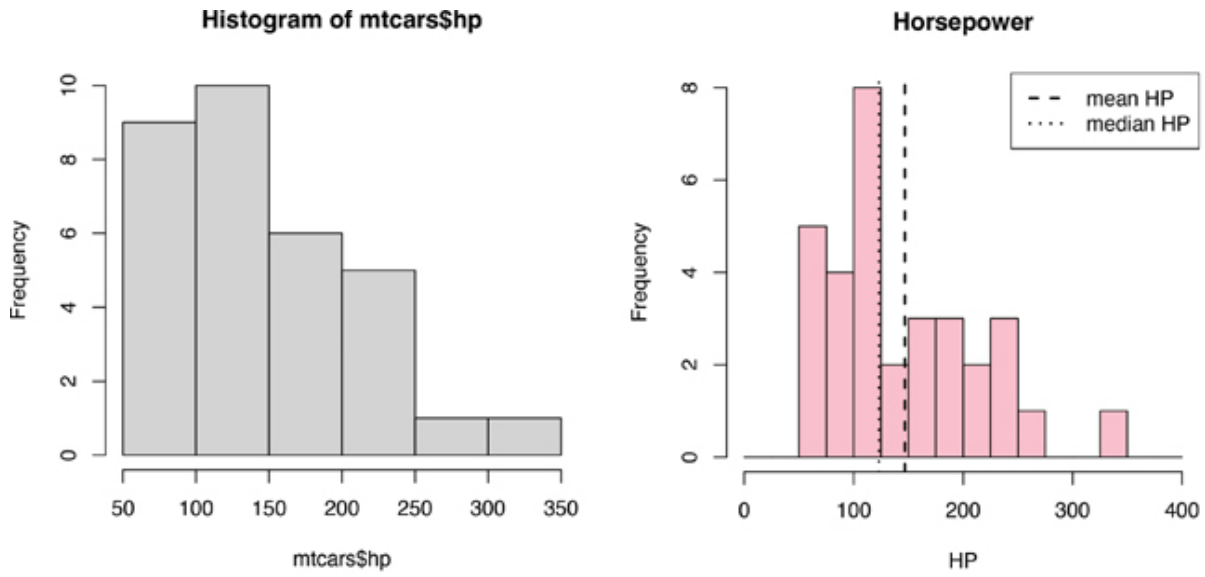


Figure 14-4: Illustrating the default behavior of `hist` on the `mtcars` horsepower data (left); customizing bin width, color, and title options, as well as adding markers of centrality (right)

The result appears to use a bin width of 50 units spanning the range of the data, providing you with a quick and useful first impression of the distribution of horsepower measurements. It seems to be centered roughly in the range of 75 to 150, tapering off on the right (we call this a *right* or *positive skew*; Section 15.2.4 will cover more terminology).

14.2.2 Bin Width and Customization

The accuracy of a histogram as a representation of the shape of a distribution of measurements depends solely upon the widths of the intervals used to bin the data. You control bin width in `hist` via the `breaks` argument. One way to use this is to supply a numeric vector giving each breakpoint to `breaks`. This is done in the following code with the help of the `seq` command (Section 2.3.2), implying a halving of the width of each bin from 50 (in the default result) to 25 here:

```
R> hist(mtcars$hp,breaks=seq(0,400,25),col="pink",main="Horsepower",xlab="HP")
R> abline(v=c(mean(mtcars$hp),median(mtcars$hp)),lty=c(2,3),lwd=2)
R> legend("topright",legend=c("mean HP","median HP"),lty=c(2,3),lwd=2)
```

The new set of breakpoints also implies a slightly widened overall domain upon which the histogram is defined.

Given on the right of Figure 14-4, this code shows the result of using the narrower bins, coloring the bars pink and adding a more readable title. It also includes vertical lines denoting the mean and median, using `abline`, and a `legend` (refer back to Section 7.3).

With the smaller bin width, more detail is visible in the distribution. However, using narrower bins risks highlighting "unimportant features" (in other words, features of the histogram that represent natural variation as a consequence of the finite-sized sample). These typically occur at locations on the scale where data are scarce. For example, the single 335-horsepower car has produced an isolated bar on the right of the scale, but you might reasonably conclude that this is not a precise, accurate reflection of a "true bump" at that location in terms of the distribution of the overall population.

It's therefore important to note that choosing the interval width (or equivalently, the breakpoints) is a balancing act of sorts. You want to choose a width that gives you a good idea of the distribution of measurements without emphasizing unimportant detail by using too small a bin width. Equivalently, you also want to avoid hiding important features by using too *large* a bin width. To address this, there are data-driven algorithms that use the scale of the recorded observations to try to calculate an appropriately balanced bin width. You can supply a character string to `breaks`, giving the name of the algorithm that you want to employ. The default `breaks="Sturges"` often works well, though it's worth trying a small number of alternative widths when exploring data in this way. For further details on this and other ways to use `breaks`, the documentation `?hist` provides clear and concise instruction.

14.2.3 In ggplot2

The `ggplot2` package emphasizes the issues surrounding intervals and their widths in a different way. To produce the default histogram, we use a `geom_histogram` layer atop the initial `ggplot` declaration (with the variable of interest aesthetically mapped to the `x` margin):

```
R> ggplot(mtcars,mapping=aes(x=hp)) + geom_histogram()
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

The left-hand image in Figure 14-5 shows the result of this code. Note, however, that a notification concerning the default histogram bins is printed to the console.

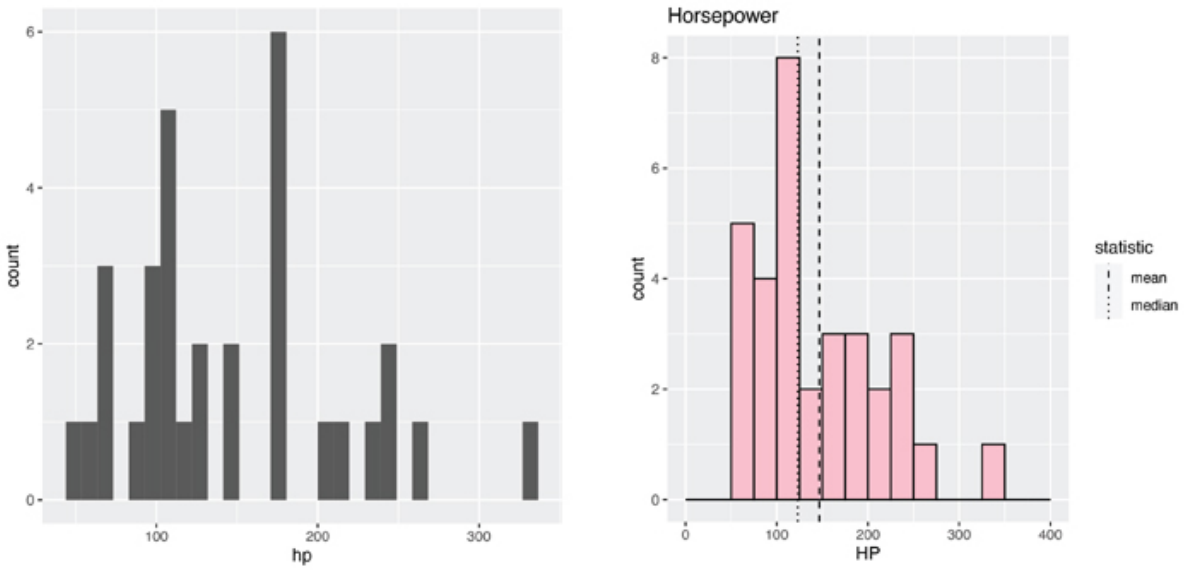


Figure 14-5: Illustrating the default behavior of `geom_histogram` on the `mtcars` horsepower data (left); customizing bin width, color, and title options, as well as adding markers of centrality (right)

Specifically, if you don't specify the bins or breakpoints, exactly 30 intervals will be used to span the range of the data. Inspecting the relevant documentation given with a call to `?geom_histogram` tells you the following:

> By default, the underlying computation (`stat_bin()`) uses 30 bins; this is not a good default, but the idea is to get you experimenting with different number of bins. You can also experiment modifying the `binwidth` with `center` or `boundary` arguments. `binwidth` overrides `bins` so you should do one change at a time. You may need to look at a few options to uncover the full story behind your data.

So, rather than defaulting to a data-driven algorithm like the base R function `hist` does, `ggplot2` encourages users to become aware of the issue and actively set their own bins. You can see that 30 bins yields

inappropriately narrow intervals for this example; there are many gaps where no observations have fallen. There are a number of ways to choose histogram intervals in `geom_histogram`, one of which is to use the `breaks` argument in the same way as in `hist`, supplying it with an appropriate numeric vector of interval endpoints. To re-create the plot on the right of Figure 14-4 using `ggplot2` functionality, we can use the object given by the following code:

```
gghist <- ggplot(mtcars,mapping=aes(x=hp)) +
             geom_histogram(color="black",fill="pink",breaks=seq(0,400,25)) +
             ggtitle("Horsepower") + xlab("HP")
```

Simply printing `gghist` will produce much the same histogram as appears on the right of Figure 14-4. In addition to using the same breakpoints, the `color` argument governs the bar outline color and `fill` the internal color of the bars. However, to fully match our previously created base R histogram, we also wish to incorporate vertical lines for the mean and median horsepower. This requires a little more work and is arguably best done by first creating a separate, smaller data frame with the values we want to use:

```
R> mm <- data.frame(val=c(mean(mtcars$hp),median(mtcars$hp)),
                    statistic=factor(c("mean","median")))
R> mm
       val statistic
1 146.6875      mean
2 123.0000    median
```

Then, we use the `geom_vline` function. The right-hand side of Figure 14-5 shows the result of the following code:

```
gghist + geom_vline(mapping=aes(xintercept=val,linetype=statistic),data=mm) +
    scale_linetype_manual(values=c(2,3))
```

Here, the aesthetic `mapping` must be instructed to change using `aes` as usual. The locations of the desired vertical lines, which we've just stored in the `val` column of the `mm` data frame, are passed to the `xintercept` argument. To ensure a correctly labeled legend is created for the mean

and median, you must also instruct `linetype` in the call to `aes` to be mapped to the corresponding descriptive labels; these are stored as a factor in the `statistic` column of `mm`. The `mm` object itself is passed to the `data` argument. We need to do this to ensure `aes` knows where to look for its ingredients; this layer is independent of the raw horsepower data declared in the initial call to `ggplot`.

By default, the two lines will be drawn mimicking base R line types `lty=1` (solid) and `lty=2` (dashed), but to match the earlier plot, you'd want `lty=2` and `lty=3` (dotted). You can further add the `scale_linetype_manual` modifier to make this change; the desired line type numbers are passed as a vector to `values`.

Again, all of this might appear to be a bit fiddly on first glance. But when handled correctly, it frees us from mistakes that might otherwise be made when we need to keep track of variable mappings "manually," as base R typically requires of us. Here, for instance, the legend for the two vertical lines was automatically produced based on our newly declared aesthetic mappings, and it is therefore much harder to accidentally mix up labels and linetypes.

Choosing between `ggplot2` and base R graphics often comes down to your intended goal. For automated handling of graphics, especially where categorical variables are used to separate subsets of the dataset, `ggplot2` is particularly powerful. On the other hand, if you require fine, non-standard control over the creation of a given plot and its annotations, traditional base R graphics can be easier to handle.

14.3 Density Plots

Think of a *density plot* as a more sophisticated histogram. Used for the same purpose—to visualize the distribution of a collection of continuous observations—it provides a smooth impression of where the observations fall. This is a more natural representation of how statisticians think about the manifestation of continuous observations (see Section 16.2).

14.3.1 Construction

Let's remain with the horsepower observations of the `mtcars` data. We use the `density` function and need to explicitly plot the resulting object to obtain the density plot. The result of the following line

```
plot(density(mtcars$hp))
```

is on the left of Figure 14-6.

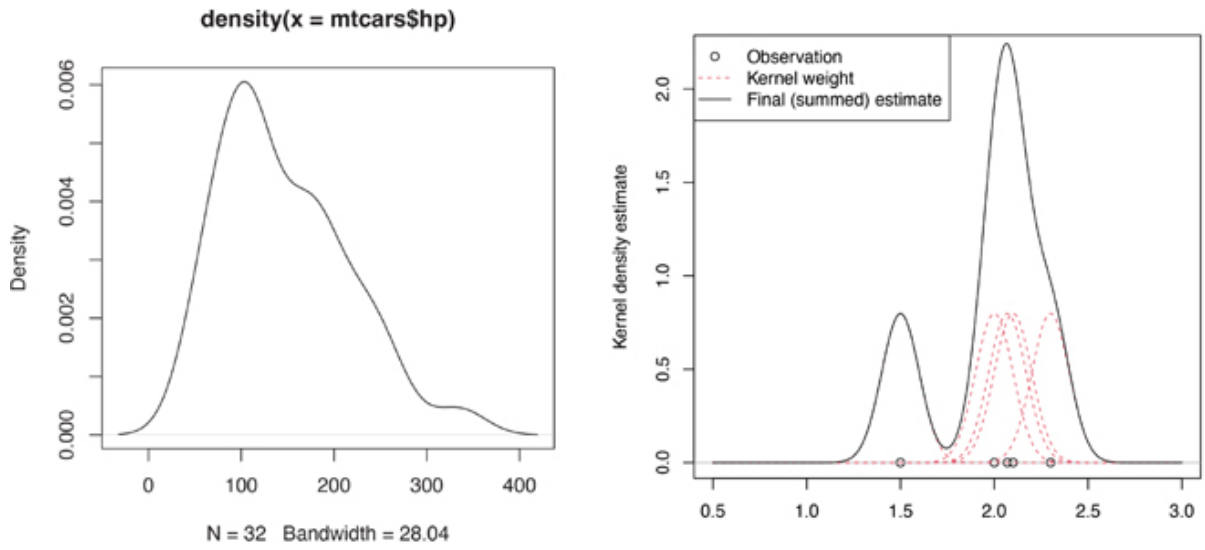


Figure 14-6: The default density plot for the horsepower data (left); an illustration of the construction of a density plot via kernel smoothing (right)

In the same way as a histogram, peaks in a density plot reflect higher concentrations of data; areas of the domain with relatively few observations are associated with a small density.

To better understand how these smooth density plots are produced, it's worth briefly explaining how the underlying methodology, *kernel density estimation*, works. Rather than counting how many observations fall in some pre-specified set of bins (as we would in constructing a histogram), we assign to each individual observation a special *kernel weight*, which is highest near to the observation and declines the farther away we look. The final value of the smoothed density curve at a given location is calculated by simply summing up the kernel weights from each observation; the density plot itself results by evaluating this sum

everywhere in the domain of the data. The right side of Figure 14-6 illustrates this idea, where five hypothetical data points are subjected to this smoothing process to yield a density plot.

14.3.2 Bandwidths and Customization

Key to a density plot is how how much we choose to "smooth out" each kernel. This is controlled by the all-important *bandwidth* (sometimes called the *smoothing parameter* or the *window width*). The bandwidth is specific to the scale of the data, and its effect on a density plot is analogous to the bin width when creating a histogram. A very large bandwidth will result in an over-smoothed final density plot because our kernel weights will be wide and flat, and we might miss out on important detail. A very small bandwidth will have the opposite effect: Our kernel weights will be tall and skinny, and the resultant under-smoothing will offer a density that is too *noisy* (too sensitive to individual observations).

Looking back at Figure 14-6, we can see `density` automatically chose a bandwidth value of around 28.04 units for the horsepower data. The leftmost panel of Figure 14-7 gives the result of the following code, where we use the argument `bw` to set a much smaller bandwidth:

```
R> plot(density(mtcars$hp,bw=10),main="")
```

The remaining two plots in Figure 14-7 are a result of the same line of code, using `bw=30` and `bw=50`.

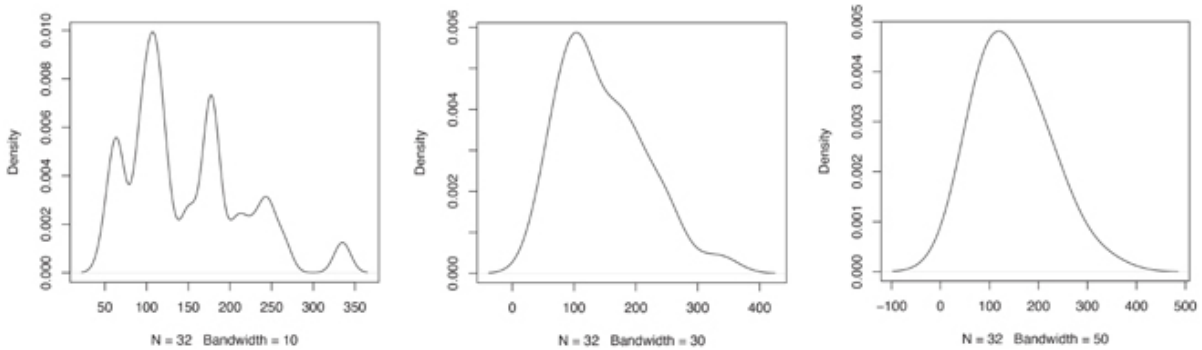


Figure 14-7: Three density plots of the `mtcars` horsepower data using increasing bandwidths (left to right)

Looking from left to right, note the increasing smoothness of the final density plot, so much so that the left-hand tail of the rightmost plot somehow suggests the possibility of encountering negative-horsepower cars! This is an artifact of over-smoothing and does not reflect reality.

An inspection of the help file `?density.default` reveals a variety of further options for specifying a given density, such as the specific functional form of the kernel weight itself and the choice of automated bandwidth selection method to be used (the defaults are often sensible for rough, exploratory analyses). Familiar graphical parameters such as `col` (for color), `lty` (for line type), and `xlab`/`ylab`/`main` (for axis labels and titles) are available to customize the plot itself.

NOTE

One important aspect of kernel density estimation is not immediately available in the base R `density` *function: that of* boundary correction. *This is a modification to the methodology designed to assist realistic estimation of the curve at "boundaries" of the data, such as zero for the horsepower data. Without it, as we saw on the right of Figure 14-7, the final density curve can encapsulate unrealistic values. Some contributed R packages include boundary correction for density estimation, such as the* `evmix` *package (Hu & Scarrott 2018). It is also available in the* `ggplot2` *version for density plots (Section 14.3.4).*

14.3.3 When Multiple Densities Overlap

We are often interested in comparing several distributions when exploring data. Doing this with histograms can be visually problematic when the desire is to overlay the different distributions atop one another on a single image. Density plots make this easier.

Recall from Section 14.1 that the `mtcars` data includes a variable giving the number of cylinders for each car. Suppose we're interested in comparing the distributions of horsepower for the three sub-groups formed by considering four-, six-, and eight-cylinder cars separately. To do this via density plots with base R graphics, we'll first need to compute the density estimates for each cylinder-specific collection of horsepower observations:

```
R> hp4 <- mtcars$hp[mtcars$cyl==4]
R> hp6 <- mtcars$hp[mtcars$cyl==6]
R> hp8 <- mtcars$hp[mtcars$cyl==8]
R> den4 <- density(hp4,bw=15,from=10,to=400)
R> den6 <- density(hp6,bw=15,from=10,to=400)
R> den8 <- density(hp8,bw=15,from=10,to=400)
```

The first three lines are used to store the group-specific horsepower values for four-, six-, and eight-cylinder cars as `hp4`, `hp6`, and `hp8`, respectively (using vector subsetting with the appropriate relational operation on `cyl`; see Sections 2.3.3 and 4.1.2). The next three compute the corresponding densities (note no actual plotting has taken place yet); these are all instructed to use a bandwidth of 15 units and be evaluated on the same domain (between 10 and 400 horsepower).

We'll produce the plot by drawing the curve associated with the fourcylinder cars first via `plot` and then adding the two remaining curves with `lines` (refer to Section 7.3). Note the xand y-coordinates defining the domain of the estimate and the density itself are stored as the `$x` and `$y` components of any object returned by `density`.

```
R> plot(den4$x,den4$y,type="l",ylim=c(0,max(c(den4$y,den6$y,den8$y))),
       xlab="HP",ylab="Density",main="Horsepower by no. cylinders")
R> lines(den6$x,den6$y,lty=2)
R> lines(den8$x,den8$y,lty=3)
R> legend("topright",legend=c("4 cyl.","6 cyl.","8 cyl."),lty=1:3)
```

Figure 14-8 shows the result of this code. The `ylim` argument in the initial call to `plot` is necessary because the first plot will define the limits of the axes. We want to ensure the full height of all density curves will

be displayed; without this, the line for the six-cylinder cars would've been chopped off, since the y-axis limits would have been defined only based on the line for the four-cylinder cars.

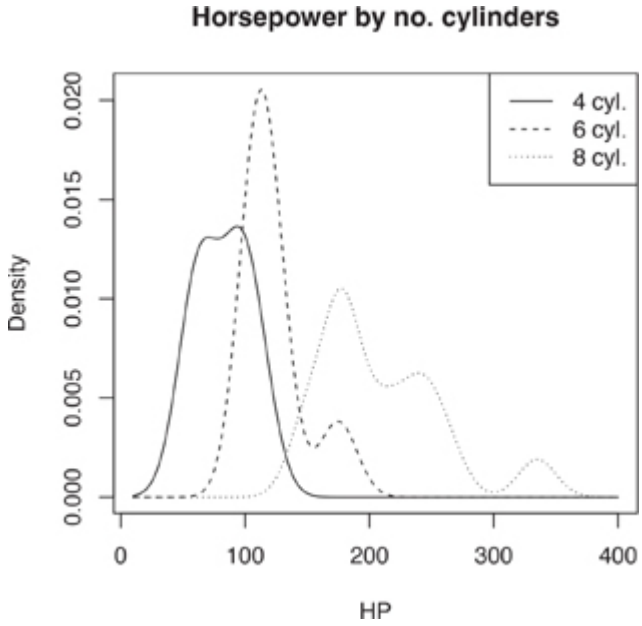


Figure 14-8: Overlapping multiple density plots to compare distributions of horsepower by number of cylinders using base R graphics and the `mtcars` dataset

14.3.4 In ggplot2

The `ggplot2` functionality includes density plot geoms, namely `geom_density`. The left of Figure 14-9 shows the default appearance, operating on the `mtcars` horsepower data.

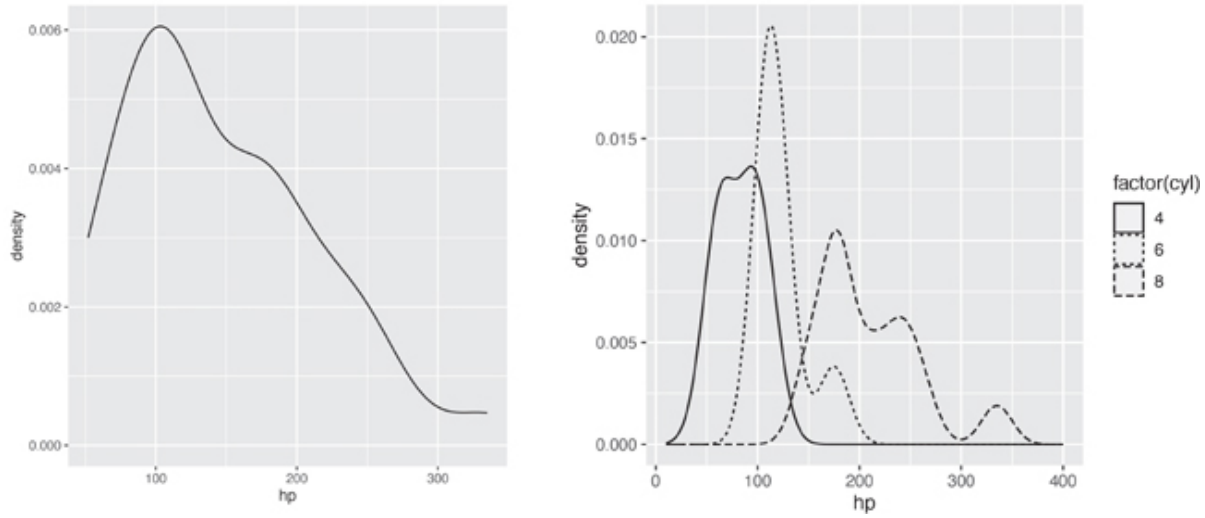


Figure 14-9: Density plots using `ggplot2`: the default plot of horsepower from the `mtcars` dataset (left) and the densities of horsepower values, split by number of cylinders (right)

It is provided by the following line (remembering the `ggplot2` library must already be loaded in your current R session):

```
R> ggplot(mtcars,mapping=aes(x=hp)) + geom_density()
```

Naturally, we can tailor the estimation of the density itself (for example, bandwidth / kernel weight function), as we saw when using base R to plot; study the help file `?geom_density` to see what's possible.

The functionality of `ggplot2` really gets a chance to shine if, as earlier, we wish to estimate and display multiple densities split by some factor. Consider the result of the following code, given on the right of Figure 14-9, which repeats the previous exercise of overlapping separate density curves of the horsepower observations according to number of cylinders:

```
R> ggplot(mtcars,mapping=aes(x=hp,linetype=factor(cyl))) +
       geom_density(bw=15,bounds=c(0,Inf)) + xlim(10,400)
```

In contrast to the base R version, we have achieved much the same result with far less code. Here, the line type itself is mapped to the cylinder variable (which must be explicitly converted to a factor given the `cyl` column in `mtcars` is purely numeric in its original form). This means the aesthetic mapping takes care of all the work for us; there is

no need to explicitly produce individual density estimates and carefully construct the plot one curve at a time like we needed to do for the base R version in Figure 14-8. As before, we also set the bandwidth to be the same (15 units) for all three density estimates (done by supplying `bw=15` to `geom_density`). The domain of evaluation is also set to be the same as in the base R version; adding the tweak `xlim(10,400)` takes care of that. The `bounds` argument in `geom_density` controls boundary correction calculations, a feature that is not presently available in the base R `density` function. We know it's impossible for horsepower to be negative, but there isn't an obvious upper bound. Thus, setting `bounds=c(0,Inf)` will correct for any notable density that might fall below zero on the left but will leave the right-hand tail of the estimates untouched. In this particular example, given the data at hand and the relatively small bandwidth, the effect of any boundary correction is negligible; it would be more obvious if we chose `bw` to be large (see Figure 14-7). Regardless, it is generally recommended to make use of `bounds` whenever you're dealing with a naturally bounded variable.

Exercise 14.2

a. Turn your attention back to the built-in `InsectSprays` data frame used in Exercise 14.1, which contains counts of insects on various agricultural units treated with different insecticide sprays.

    i. Produce a histogram of the counts of insects using base R graphics using default settings.

    ii. Re-do the histogram using a smaller bin width. Specifically, set new breakpoints to be the sequence of values from 0 to the maximum insect count in steps of 2. Tidy up the title and axis labels

as appropriate, and set the color of the refined histogram to be `"springgreen"`.

iii. Produce a default histogram of the counts of insects using `ggplot2`.

iv. Repeat (ii) with `ggplot2` to make it look like the refined base R version; that is, set the custom breakpoints as in the previous task, refine the axis labels and title, and color the plot.

v. Use base R graphics to produce a density plot of the insect counts. Use default settings for everything. What is the value of the smoothing bandwidth that was used? Produce two more density plots, one with a bandwidth of 1 unit, another with a bandwidth of 4. How do the changing bandwidths affect your interpretations of the distribution of the insect counts? Are the density curves "realistic" near counts of zero insects?

vi. Use `ggplot2` to produce a plot of six overlapping density curves of the insect counts, one for each spray type. Use a common bandwidth of 2.5 units, and map the `spray` category to line color. Use appropriate bounds to ensure the density curves are corrected near the natural boundary of zero. Are there any features of this plot that stand out to you with respect the distributions of insect counts for the different spray types?

b. Yet another of R's ready-to-use datasets is `USArrests`. This object contains data on the number of arrests for murder, rape, and assault per 100,000 individuals in each of the 50 states of the United States, recorded in 1973 (for example, see McNeil 1977). It also includes a variable giving the

percentage of urban-based population in each state. Briefly inspect the data frame object and the accompanying documentation `?USArrests`. Then complete the following tasks:

i. Use base R functionality to generate a histogram of the proportions of urban populations for the states. Set your breaks to be 10 units each, between 0 and 100. Have the histogram show the first quartile, the median, and the third quartile; then provide a matching legend. Use colors as you like and include appropriate axis annotation.

ii. Re-do the prior histogram using `ggplot2`. To ensure a legend is produced, you'll need to store the three statistics being marked with vertical lines as a small data frame object like that created in Section 14.2.3 on page 342.

14.4 Box Plots

An especially popular alternative to the histogram is the *box-and-whisker plot*, or *box plot* for short. These are essentially a visual representation of the fivenumber summary discussed in Section 13.2.4.

14.4.1 Stand-Alone

Let's revisit the built-in `quakes` data frame of the 1,000 seismic events near Fiji (first seen in Exercise 8.1 on page 183 and subsequently used in Chapter 13). For the sake of comparison, you can examine both a histogram and a box plot of the magnitudes of these events using default base R behavior. The following code

```
R> hist(quakes$mag)
R> boxplot(quakes$mag)
```

produces the plots in Figure 14-10. Note that the labels superimposed onto the box plot on the right-hand side of the figure point out the key information displayed.

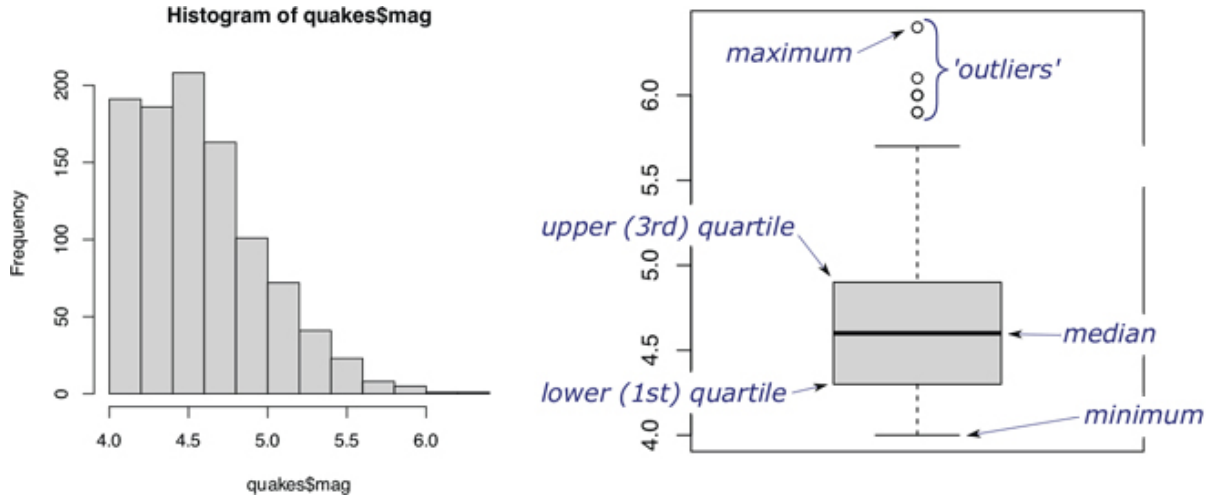


Figure 14-10: The default histogram (left) and box plot (right) of the magnitude data from `quakes`

Like the histogram, a box plot shows important features of the distribution, such as overall centrality, spread, and skew (these terms will be clarified in Section 15.2.4).

As the labeling arrows point out, the line in the middle of the box represents the median, and the lower and upper edges of the box show the respective quartiles. If there are no "extreme" observations, perpendicular lines (the *whiskers*) extending from the box indicate the minimum and the maximum; any dots drawn beyond the whiskers are deemed to be "outlier points." By default, `boxplot` defines an outlier as an observation that lies more than 1.5 times the IQR below the lower quartile or above the upper quartile. This prevents the whiskers from extending too far and overemphasizing any asymmetry in the distribution. Thus, the maximum and minimum values marked by the whiskers themselves are not always the raw, overall maximum or minimum values in the dataset because a value that has been deemed an outlier might actually represent the highest or lowest value. You can control the nature of this classification via the `range` argument in `boxplot`,

though the default of `range=1.5` is usually sensible for basic data exploration.

14.4.2 Side-by-Side

One particularly pleasing aspect of these plots is the ease with which you can compare the five-number summary distributions of different groups with *side-by-side* box plots. Again using the `quakes` data, define the following corresponding factor and inspect the first five elements (see Section 4.3.7 to review use of the `cut` command):

```
R> stations.fac <- cut(quakes$stations,breaks=c(0,50,100,150))
R> stations.fac[1:5]
[1] (0,50] (0,50] (0,50] (0,50] (0,50]
Levels: (0,50] (50,100] (100,150]
```

Recall that the `stations` variable records how many monitoring stations detected each event. This code has produced a factor breaking up these observations into one of three groups: events detected by 50 stations or fewer, between 51 and 100 stations, and between 101 and 150 stations. Thus, you can compare the distributions of the magnitudes of the events according to these three groups. The following line

```
R> boxplot(quakes$mag~stations.fac,
           xlab="# stations detected",ylab="Magnitude",col="peachpuff")
```

produces Figure 14-11.

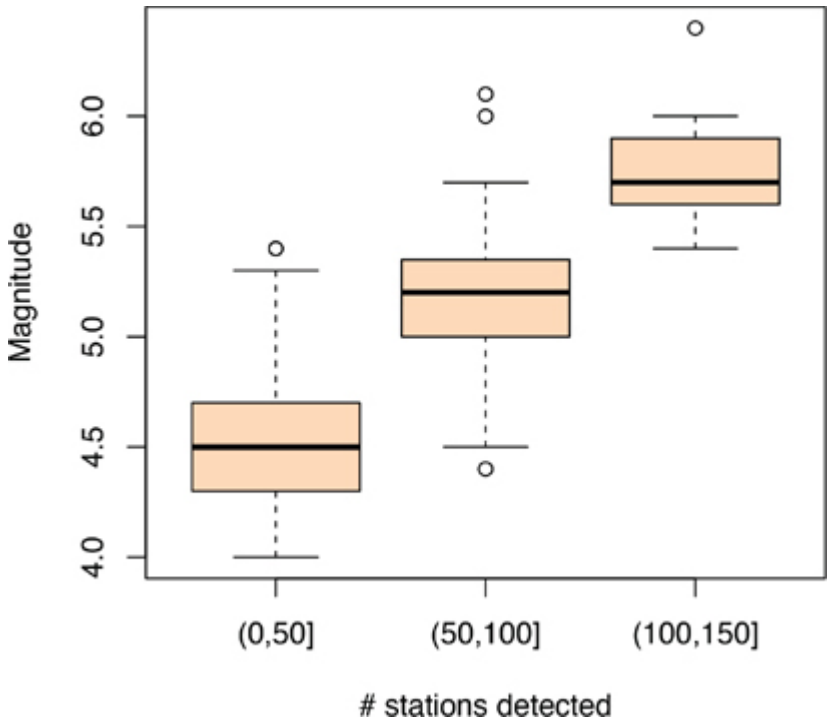


Figure 14-11: Side-by-side box plots of the `quakes` *magnitudes, split by the three groups identified by* `station.fac`*, using base R graphics*

With this line of code, you should note new syntax in the form of a tilde (~), shown here in `quakes$mag~stations.fac`. You can read the ~ as "on," "by," or "according to" (you'll use the tilde notation frequently in Chapters 20 through 22). Here, you're instructing `boxplot` to plot `quakes$mag` *according to* `station.fac` so that a separate box plot is produced for each group, naturally given in the order listed in the grouping factor. Optional arguments are also employed to control axis labeling and box color. You might prefer to flip the axes to have horizontal box plots instead of the default vertical ones shown here; to do this, simply set the optional argument `horizontal=TRUE`. Either way, your interpretation of this plot should mirror what you've previously seen in Figure 13-6: The higher the recorded magnitude, the more stations detected a given seismic event.

14.4.3 In ggplot2

To produce a box plot with the `ggplot2` package, use `geom_boxplot`. A default vertical box plot of the magnitude data from the `quakes` object is given by

the following line,

```
R> ggplot(quakes,mapping=aes(x=factor(1),y=mag)) + geom_boxplot()
```

the result of which is shown on the left of Figure 14-12.

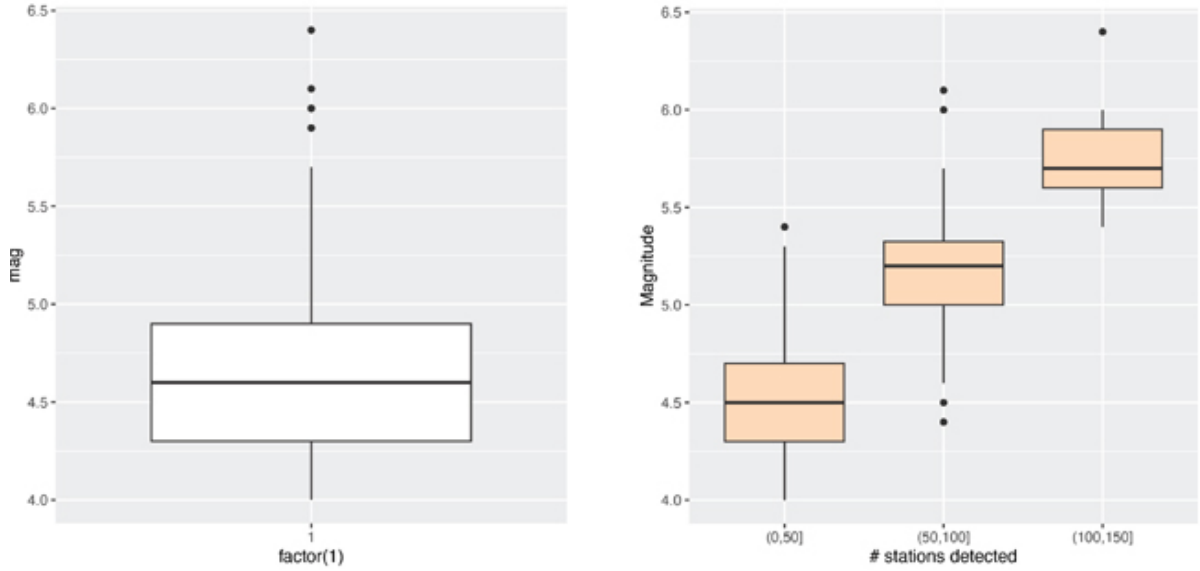


Figure 14-12: A stand-alone, default vertical box plot of the `quakes` *magnitude data (left), and side-by-side box plots of magnitude split by "number of detecting stations" category (right) using* `ggplot2` *functionality*

Mapping the appropriate variables to the desired axes permits us to produce side-by-side box plots. For the stand-alone box plot, we created a "dummy" x-axis map `x=factor(1)`; we weren't looking to split the observations up into different groups. If we instead make use of the `stations.fac` object from Section 14.4.2, the following code produces the `ggplot2` version of Figure 14-11; the result is shown on the right of Figure 14-12:

```
R> ggplot(quakes,mapping=aes(x=stations.fac,y=mag)) +
       geom_boxplot(fill="peachpuff") + xlab("# stations detected") +
       ylab("Magnitude")
```

Here, the split by `station.fac` is sparked by mapping it to the x-axis. The `fill` argument in `geom_boxplot` is used to color the boxes; the previously seen `xlab` and `ylab` additions neaten up the axis labels.

NOTE

At the time of writing, the default `geom_boxplot` *calculation of quantiles (used to compute the upper and lower quartiles of each "box") is slightly different from that used in base R. As such, the calculated IQR and hence the identification of outlier points may differ a little between the two versions.*

14.5 Violin Plots

Box plots are unable to capture local features of the distributions of interest, which is relevant if, say, there are irregularly spaced "clusters" of observations in your dataset. The *violin plot* was proposed to address this issue, and, in tandem with the ever-increasing interest in statistics and data science, it is becoming more popular today.

14.5.1 Design and Interpretation

The violin plot combines a basic box plot (Section 14.4) with a (kernel) density plot (Section 14.3), the latter superimposed and laid against either side of the former. In so doing, we seek to retain the convenience of visualization of the classic five-number summary that a box plot provides, while adding the ability to assess finer structure in the distribution of the observations at hand.

You can generate violin plots using the `vioplot` function; this function is part of a contributed package with the same name (Adler et al. 2022). Install the package (Section 1.3.5) and ensure it's loaded with a call to `library("vioplot")`.

As a motivating example of where violin plots might be helpful, we'll consider another of R's ready-to-use datasets. The `faithful` object contains observations on the durations of eruptions, and the waiting times between eruptions, of the Old Faithful geyser in Yellowstone National Park, Wyoming. See the documentation in `?faithful` for details.

Figure 14-13 shows a histogram, a box plot, and a violin plot of the waiting times (recorded in seconds) between eruptions.

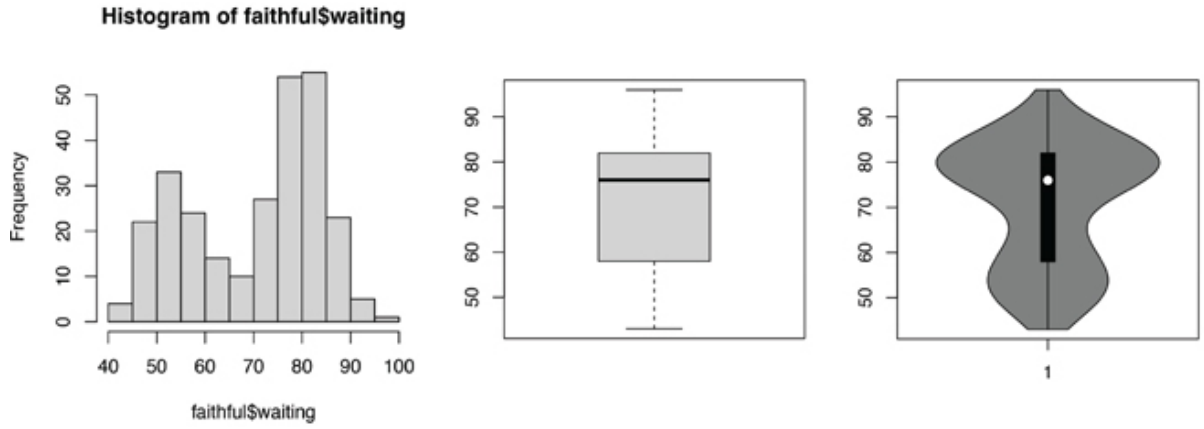


Figure 14-13: The default histogram (left), box plot (middle), and violin plot (right, produced using the `vioplot` *package) of the Old Faithful waiting times*

With the `vioplot` package loaded, produce these three plots by calling the following lines in turn:

```
R> hist(faithful$waiting)
R> boxplot(faithful$waiting)
R> vioplot(faithful$waiting)
```

Looking at the histogram first, it seems appArent there are two "groups" of waiting times; one centered at around 55 seconds, the other centered around 80 seconds. This leads to a "twin-peak" or *bimodal* appearance in the histogram. However, this important information is completely hidden if we focus on the box plot, as it is unable to capture these localized features. The violin plot is, on the other hand, able to reveal the structure of the distribution. Here, the black box along the central "spine" represents the same box plot, but reflected on either side is the density plot of same observations, permitting this detail.

There are important things to remember when generating violin plots. Just as with stand-alone density estimation, we must be mindful of the smoothing bandwidth used to produce the density curves (Section 14.3.2). By default, the bandwidth used for a violin plot density is set according to a similar rule of thumb as is used in the base R function

`density`. In `vioplot` it is controllable via the optional argument `h`. Also consider that violin plots typically don't separate out perceived outliers like the standard box plot functions do because the entire set of data forms the corresponding density estimates. In turn, this means we should be wary of over-interpretation of the density curves themselves, especially when we are working with relatively small sets of observations.

14.5.2 Side-by-Side Violins

To compare multiple distributions, use side-by-side violin plots. These are constructed in much the same way as you saw with side-by-side box plots. Consider again the built-in `quakes` dataset along with the custom `stations.fac` factor we created in Section 14.4.2 (page 351). The result of the following code (again, remember the `vioplot` package is needed)

```
R> vioplot(quakes$mag~stations.fac,xlab="# stations detected",
           ylab="Magnitude",col="peachpuff")
```

is presented in Figure 14-14.

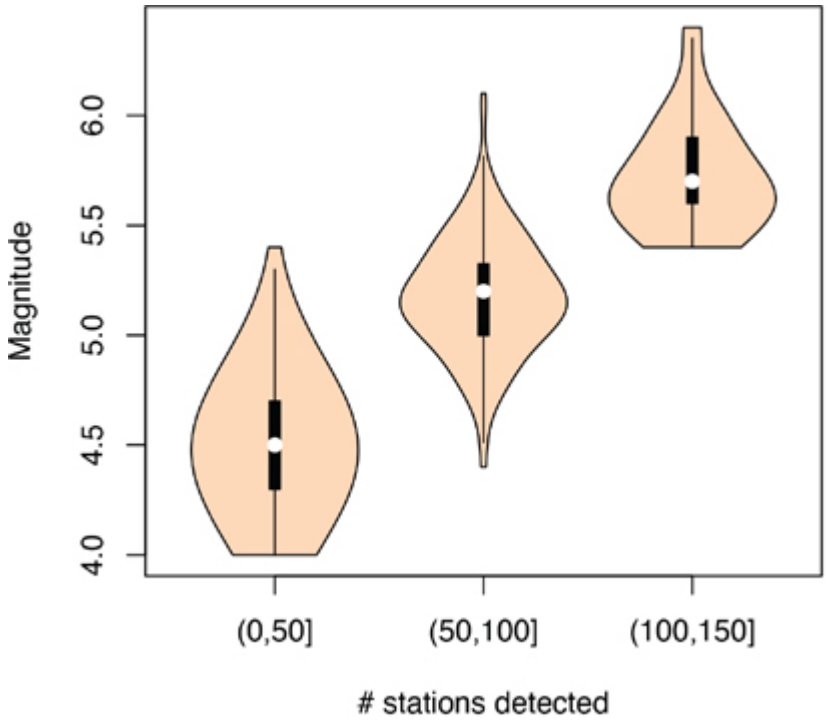


Figure 14-14: Side-by-side violin plots of the `quakes` magnitude data split by the "number of detecting stations" category

Therein, the plot conveys the same overall information as provided by the side-by-side box plots in Figures 14-11 and 14-12, with the density curves highlighting the relative symmetry of the distribution corresponding to the (50,100] category compared to the other two.

14.5.3 In ggplot2

As we might expect, `ggplot2` also includes functionality for violin plots via `geom_violin`, which behaves similarly to `geom_boxplot`. Indeed, for a stand-alone violin plot we can run exactly the same line of code seen in Section 14.4.3, exchanging `geom_boxplot` for `geom_violin`:

```
R> ggplot(quakes,mapping=aes(x=factor(1),y=mag)) + geom_violin()
```

This produces the default violin plot of the seismic event magnitudes, as shown on the left of Figure 14-15.

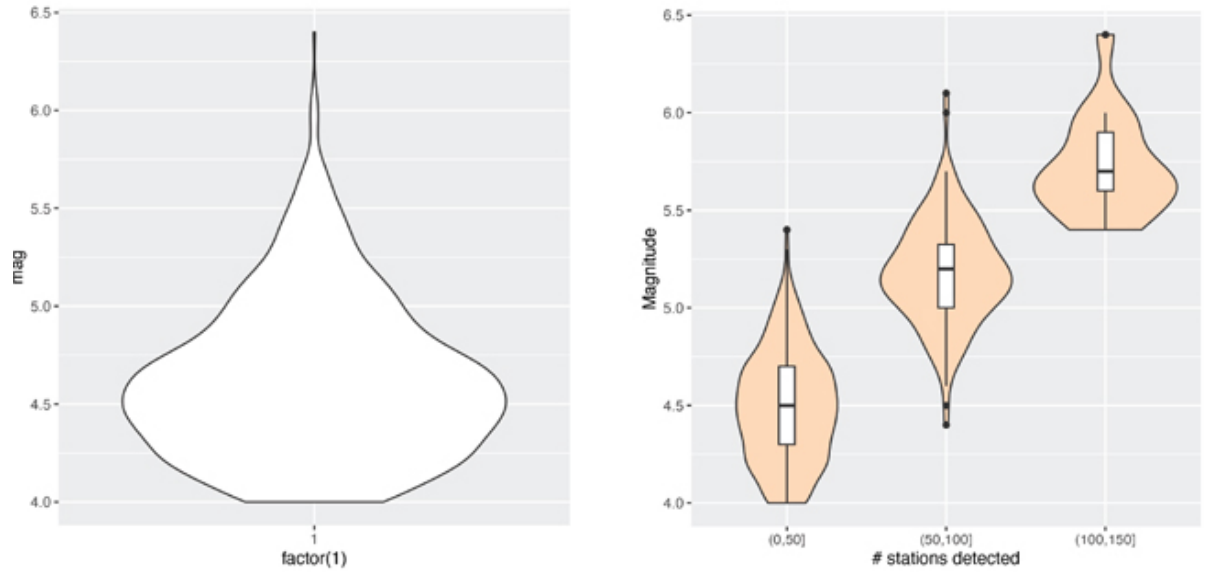


Figure 14-15: A stand-alone, default vertical violin plot of the `quakes` magnitude data (left) and side-by-side violin plots of magnitude split by the "number of detecting stations" category (right) using `ggplot2` functionality

Notice that `geom_violin` by itself does not include internal box plots like those produced earlier using `vioplot`. However, `ggplot2` operates in a

"layerlike" fashion, meaning we can superimpose the corresponding `geom_boxplot` atop the violin plots to get a little closer to the results obtained using base R graphics. To illustrate this, let's stick with the `quakes` data and once more create side-by-side plots of the event magnitudes split by the `stations.fac` object (page 351). The following code yields the plot shown on the right of Figure 14-15:

```
R> ggplot(quakes,mapping=aes(x=stations.fac,y=mag)) +
       geom_violin(fill="peachpuff") + geom_boxplot(width=0.1) +
       xlab("# stations detected") + ylab("Magnitude")
```

The addition of the `geom_boxplot` layer has placed box plots within each violin plot; the optional argument `width=0.1` serves to reduce the width of each box so they fit within each violin (and don't obscure any of the corresponding density curves). Unlike `vioplot`, we can see the box plots here retain the default display of classified outliers as individual points.

Exercise 14.3

a. Revisit the built-in `InsectSprays` data, and complete the following tasks:

   i. Create a stand-alone base R box plot of the insect counts.

   ii. Use the `vioplot` package to create a stand-alone violin plot of the insect counts, using default settings for everything. Does this plot offer anything more in terms of interpretation than the box plot?

   iii. Use base R to produce side-by-side box plots with insect counts separated by spray type. Separately, use `vioplot` to create a set of violin plots that shows

the same. Use the `h` argument to set the smoothing bandwidth used in the latter to be 2 units, and ensure both plots have neat axis labels.

iv. Use `ggplot2` tools to re-create the side-by-side violin plots from (iii). Ensure the smoothing bandwidth is set to 2 units and that internal box plots, with appropriately decreased width for visibility, are superimposed.

b. You started this chapter by exploring the `mtcars` dataset (see Section 14.1.1). Bring up the help file to remember the available variables, then complete the following:

i. Use `vioplot` to create side-by-side violin plots of MPG against number of forward gears (the latter has only three possible values: 3, 4 and 5). Make sure axis labels are clear and informative.

ii. Use `ggplot2` functionality to create side-by-side boxplots of MPG against number of forward gears. Note the latter variable is not stored as a factor in `mtcars`, so `factor` will need to be used when including that variable in the relevant command. As before, make sure axis labels are tidy.

14.6 Scatterplots and Coordinate-Based Plots

We most frequently use a *scatterplot* to identify a relationship between the observed values of two different numeric-continuous variables, displayed as x-y coordinate plots. The coordinate-wise nature of base R graphics lends itself naturally to the creation of scatterplots; you've seen several examples already in this book. However, not every x-y coordinate–based plot is always called a scatterplot. For example, a plot of spatial coordinates like that shown in Figure 13-1 would generally

not be regarded as a scatterplot, but a plot of the earthquake magnitude against the number of stations detecting the event, like in Figure 13-6, would be. This is because an assessment of any relationship between the two variables is of reasonable interest in the latter but makes less sense in the former.

I'll finish this chapter by expanding on how you can use scatterplots to explore continuous variables and visualize relationships between them. To do this, let's access another ready-to-use R dataset: the famous `iris` data. Collected in the mid-1930s, this data frame of 150 rows and 5 columns consists of petal and sepal measurements for three species of perennial iris flowers: *Iris setosa*, *Iris virginica*, and *Iris versicolor* (Anderson 1935, Fisher 1936). Here's what the first five records look like:

```
R> iris[1:5,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
```

Looking through the documentation file (`?iris`), you can see that the data consist of 50 observations for each of three species, with the four numeric variables recorded in centimeters (cm).

14.6.1 Single Plot

You can modify a simple scatterplot to split the plotted points according to a categorical variable, exposing potential differences between any visible relationships with respect to the continuous variables. For example, using base R graphics, you can examine the petal measurements according to the three species. Using the "stepping-stone" approach first explained in Chapter 7, one option is to manually build up this plot by first using `type="n"` to generate an empty plotting region of the correct dimensions and subsequently using `points` to add

the points corresponding to each species separately, altering point character and color as desired:

```
R> plot(iris[,4],iris[,3],type="n",xlab="Petal Width (cm)",
        ylab="Petal Length (cm)")
R> points(iris[iris$Species=="setosa",4],
          iris[iris$Species=="setosa",3],pch=19,col="black")
R> points(iris[iris$Species=="virginica",4],
          iris[iris$Species=="virginica",3],pch=19,col="gray")
R> points(iris[iris$Species=="versicolor",4],
          iris[iris$Species=="versicolor",3],pch=1,col="black")
R> legend("topleft",legend=c("setosa","virginica","versicolor"),
          col=c("black","gray","black"),pch=c(19,19,1))
```

You can find the plot in Figure 14-16.

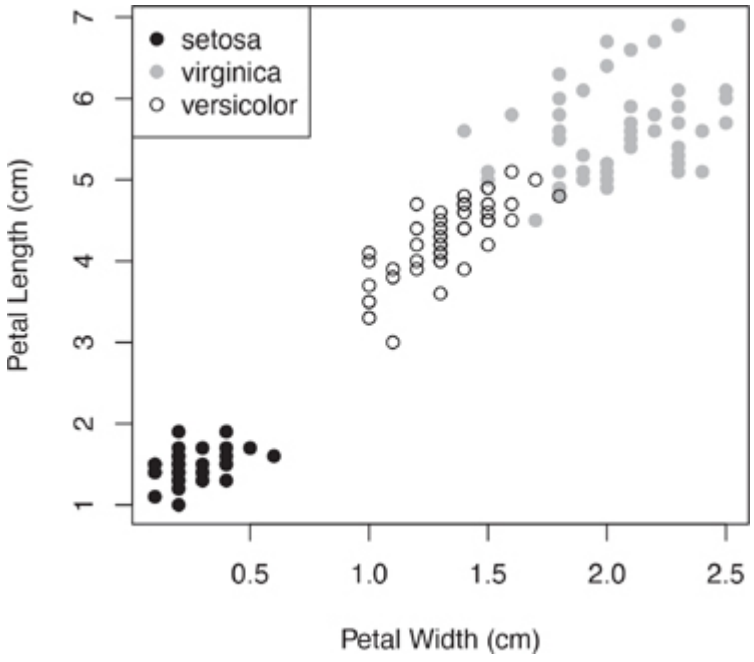


Figure 14-16: A scatterplot of petal measurements split by species from the built-in `iris` data frame

Note that the *Iris virginica* species has the largest petals, followed by *Iris versicolor*, and the smallest petals belong to *Iris setosa*. However, this code, while functional, is fairly cumbersome. You can generate the same image more simply by first setting up vectors that specify the desired

point character and color to correspond to each individual observation. To this end, consider the two objects created here:

```
R> iris_pch <- rep(19,nrow(iris))
R> iris_pch[iris$Species=="versicolor"] <- 1
R> iris_col <- rep("black",nrow(iris))
R> iris_col[iris$Species=="virginica"] <- "gray"
```

The first line creates a vector `iris_pch` of equal length to the number of observations in `iris`, with every entry being `19`. Vector subsetting then overwrites the entries corresponding to *Iris versicolor* and sets the point character to `1`. The same steps are followed to create `iris_col`; first, an appropriately sized vector is filled with the character strings `"black"`, and then those entries corresponding to *Iris virginica* are overwritten and set to `"gray"`. With that, note that the single line

```
R> plot(iris[,4],iris[,3],col=iris_col,pch=iris_pch,
       xlab="Petal Width (cm)",ylab="Petal Length (cm)")
```

will produce an identical plot when followed by the same call to `legend` as earlier.

14.6.2 Matrix of Plots

The "single" type of planar scatterplot is really useful only when comparing *two* numeric-continuous variables. When there are many more than two continuous variables of interest, it isn't possible to display this information satisfactorily on a single plot. A simple and common solution is to generate a two-variable scatterplot for each pair of variables and show them together in a structured way; this is referred to as a *scatterplot matrix*. Making use of the `iris_pch` and `iris_col` vectors created earlier, you can generate a scatterplot matrix for all four continuous variables in `iris`, retaining the distinction between species. Working with base R graphics, use the `pairs` function; the result of the line

```
R> pairs(iris[,1:4],pch=iris_pch,col=iris_col,cex=0.75)
```

is shown in Figure 14-17.

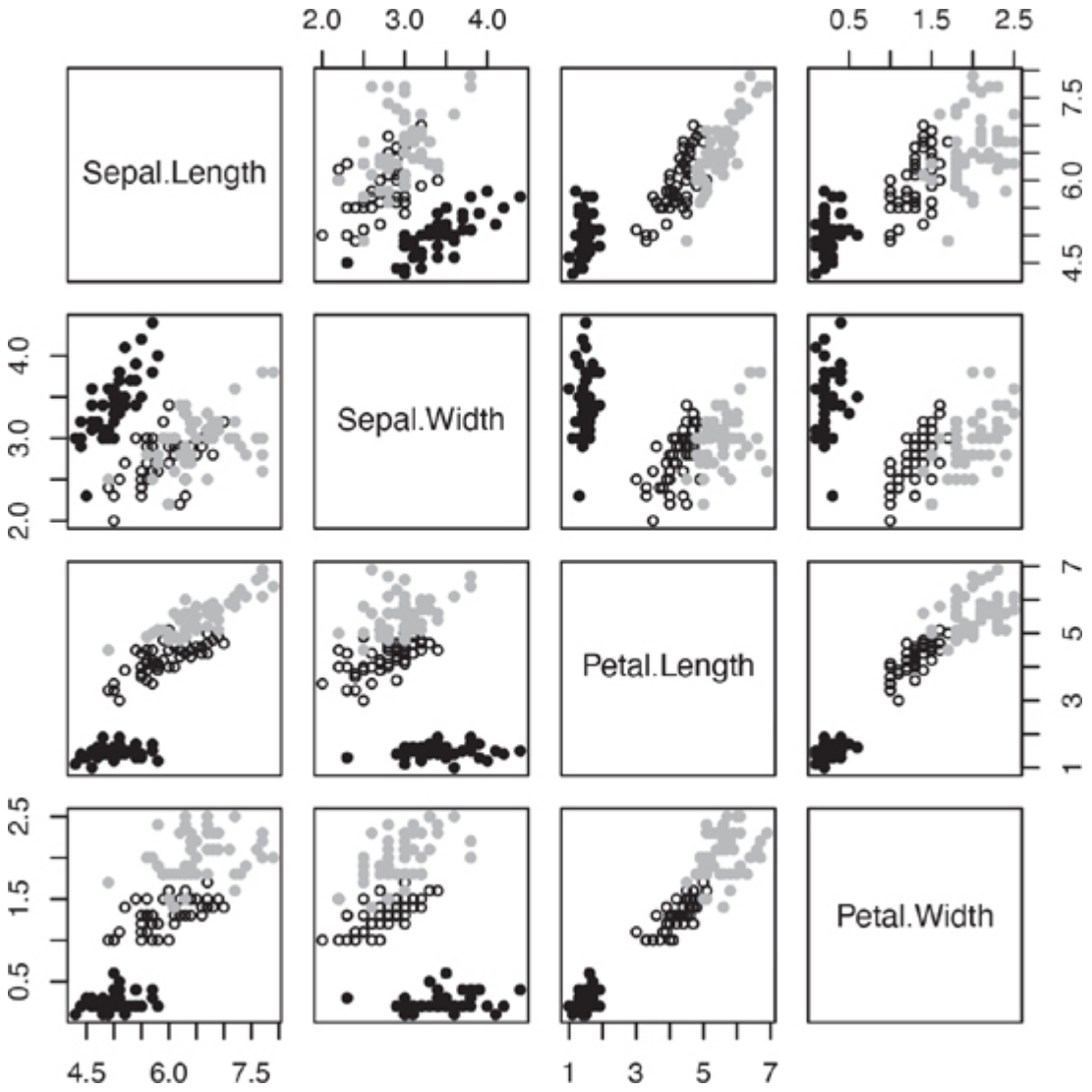


Figure 14-17: A scatterplot matrix with respect to all four continuous measurements in the data frame

The easiest way to use `pairs` is to supply a matrix or data frame of the raw observations as its first argument, done here by selecting all columns of `iris` except the `Species` column (that is, the first four, with `iris[,1:4]`). The interpretation of the plots depends upon the labeling of the diagonal panels, running from the top left to the bottom right. They will appear in the same order as the columns given as the first argument. These "label panels" allow you to determine which individual plot in the matrix corresponds to each pair of variables. For example, the first

column of the scatterplot matrix in Figure 14-17 corresponds to an x-axis variable of Sepal Length; the third row of the matrix corresponds to a y-axis variable of Petal Length, and each row and column displays a scale that is constant moving left/right or up/down, respectively. This means that plots above the diagonal are mirrored in those below it; the plot of Petal Width (y) on Sepal Width (x) at position row 4, column 2 displays the same data as the scatterplot at position row 2, column 4 but flipped on its axes. As such, `pairs` includes an option to display only those scatterplots above *or* below the diagonal, by setting either `lower.panel=NULL` or `upper.panel=NULL` to suppress one or the other.

Scatterplot matrices allow for convenient simultaneous comparison of the collection of pairwise relationships formed by observations made on multiple continuous variables. In the scatterplot matrix of the `iris` data, note there's a strong positive linear association between petal measurements but a weaker relationship between the sepal measurements. Furthermore, although *Iris setosa* may reasonably be considered the smallest flower in terms of its petals, the same can't be said of its sepals.

14.6.3 In ggplot2

Turning to `ggplot2`, recall the `geom_point` function that was introduced back in Section 7.4.1. By now, you know that `ggplot2` works particularly well when we seek to categorize our data by way of some corresponding factor object. The following code re-creates the scatterplot of the `iris` petal measurements split by species:

```
R> ggplot(iris,mapping=aes(x=Petal.Width,y=Petal.Length,shape=Species)) +
       geom_point() + scale_shape_manual(values=4:6) +
       xlab("Petal width (cm)") + ylab("Petal length (cm)")
```

Figure 14-18 shows the resulting plot.

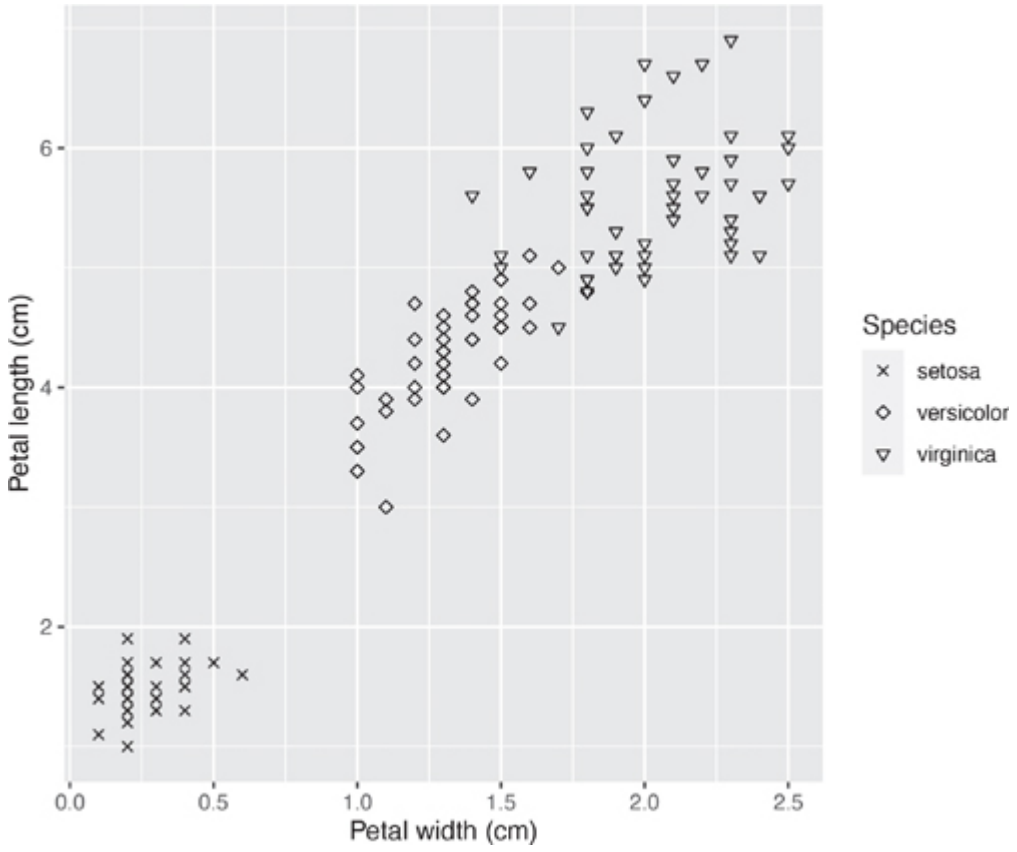


Figure 14-18: Using `ggplot2` functionality to plot petal dimensions for the three `iris` species, with point `shape`as the aesthetic modifier

Here, I've split up the points using the `Species` variable mapped to `shape` (the equivalent of the base R terminology `pch`), and I've modified the point types using the `scale_shape_manual` modifier (their defaults are filled shapes; open shapes make groups of points a bit easier to distinguish from one another).

To generate a scatterplot matrix in a `ggplot2` style, it's recommended that you download the `GGally` package (Schloerke et al. 2024) to access the `ggpairs` function. This package is designed to be an extension or add-on to `ggplot2`. It's installed from CRAN as per usual—for example, by running `install.packages("GGally")` at the prompt—and must be loaded via `library("GGally")` prior to use. After this is done, as a quick example, the following code produces the plot in Figure 14-19:

```
R> ggpairs(iris,mapping=aes(col=Species),axisLabels="internal")
```

Though you might see familiar warnings related to the histogram bin widths, `ggpairs` offers an impressive visual array for such a short line of code. The output not only gives you the same (lower half) scatterplot matrix you could produce with `pairs` but also provides variable-by-species histograms and box plots along the bottom and right, respectively. It also displays the estimates of the correlation coefficients. As shown, you can map a variable to an aesthetic modifier to split up the plotted observations based on factor levels. In Figure 14-19, this is done by color, and you instruct `ggpairs` to operate on the `Species` factor. Documentation found in `?ggpairs` gives concise information on the various options that control the presence and appearance of the individual plots.

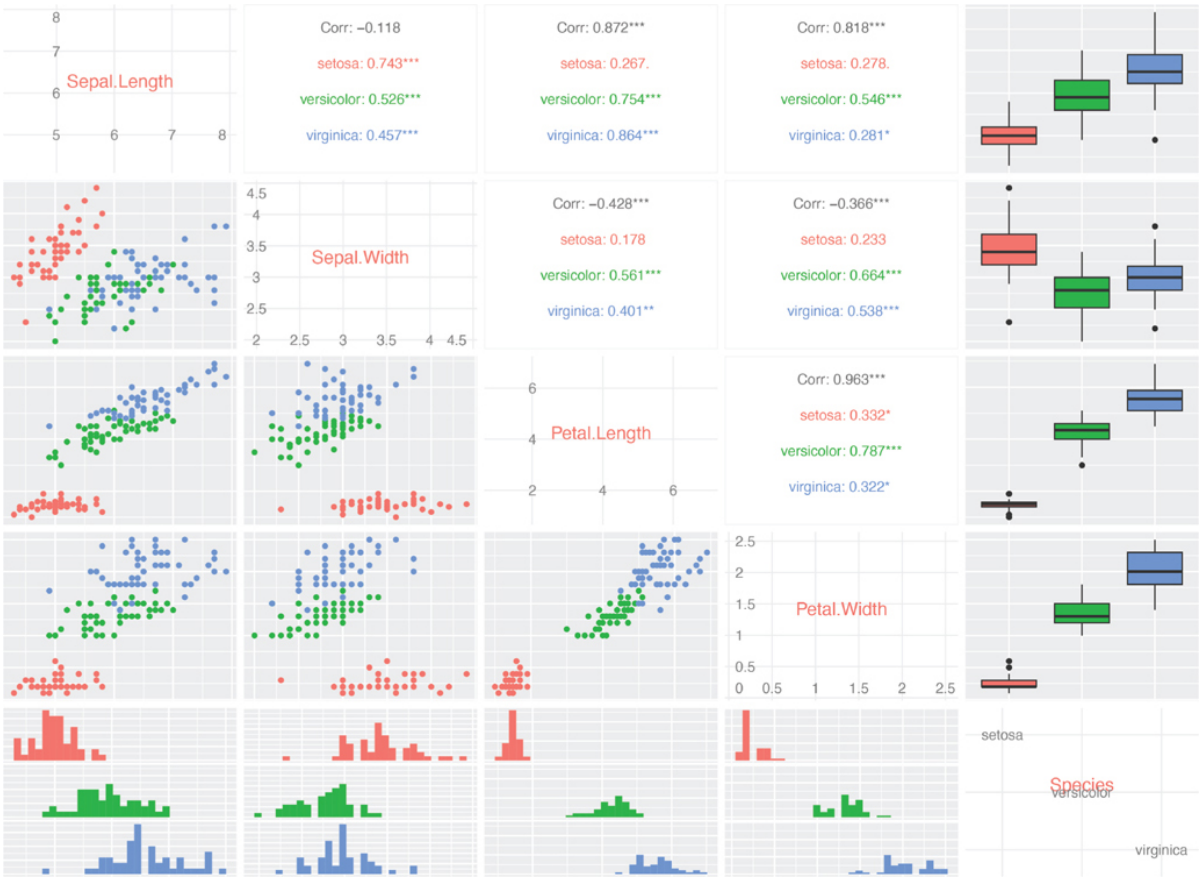


Figure 14-19: The generation of a scatterplot matrix using `ggpairs` from the `GGally` package, using color to separate the species. Note the useful addition of estimated correlation coefficients and distributional plots.

Exercise 14.4

a. Revisit the built-in `mtcars` dataset, used in several examples and exercises in this chapter thus far. Complete the following:

   i. Use the help file accessible by running `?mtcars` to identify the variables corresponding to engine displacement and quarter-mile time. Produce a base R scatterplot of these two variables, with the latter on the y-axis.

   ii. Identify the variable corresponding to number of cylinders. Re-do the scatterplot of quarter-mile time against displacement, using different point characters for four-, six-, and eight-cylinder cars (use skills from Section 14.6.1 to set up a vector, to be supplied to the `pch` argument of `plot`, which contains `1`, `2`, and `3` for the four-, six-, and eight-cylinder cars, respectively). Tidy the axis labels up and include a legend that defines the three types of points.

   iii. Repeat the quarter-mile time vs. displacement plot, with point characters differing by number of cylinders, using `ggplot2` (remember to convert the variable corresponding to number of cylinders to a `factor` to work with these tools).

b. In Exercise 14.2 you explored the built-in `USArrests` data, which contains observed rates of violent crimes (`Murder`, `Assault`, `Rape`) in 1973. These are provided by state, with the percentage of urban population also present as `UrbanPop`.

i. Create a base R scatterplot of assault rates on the x-axis against murder rates on the y-axis.

ii. Find the median urban population percentage. Based on this, create a new binary-valued vector `urbancat` that contains `1`s for those states with an `UrbanPop` value greater than the median, and `0`s otherwise. Ensure `urbancat` is stored as (or converted to) a factor object.

iii. Create a copy of the `USArrests` object with the `UrbanPop` column omitted; name it `uscopy`. Append the `urbancat` factor object you created in (ii) to `uscopy` as its fourth column (for details on handling data frames in this way, refer back to Section 5.2).

iv. Reproduce your scatterplot from (i) with points colored differently according to `urbancat`. Tidy up the axis labels and include a legend.

v. Use `GGally` functionality and the `uscopy` data frame from (iii) (with `urbancat` factor in place as the fourth column) to create a scatterplot matrix with accompanying plots, globally categorized by color to reflect the dichotomized urban population value.

14.6.4 Plots of Spatial Coordinates

In Section 7.2.6, you considered a plot's aspect ratio. There, the production of an accurate depiction of spatial data was highlighted as a common scenario where locking the aspect ratio might be desirable. Here, we'll consider this special case in a little more detail. Although plots of spatial points aren't usually referred to as "scatterplots," in base R we use the same fundamental x-y coordinate–based mechanism to create such plots; hence, it's worth discussing here.

In what follows, mentions of longitude and latitude are assumed to refer to *decimal degrees* of longtitude and latitude. Software implementations that demand numerical manipulation of such coordinates will use this format (as opposed to longitude and latitude expressed in degrees-minutes-seconds).

Pre-Existing Distance-Based Units

If your coordinates are already expressed in terms of any readily interpretable distance-based units (like miles, feet, kilometers, or meters), then it is entirely acceptable to plot your spatial coordinates with a 1:1 aspect ratio, by setting `asp=1`. This makes sense because, as noted in Section 7.2.6, such an aspect ratio treats "one unit" as being the same distance horizontally as well as vertically (one foot, for instance, is the same in any direction). In turn, your resulting plot will be scale-locked appropriately, and thus will faithfully represent the relative spatial appearance of your points.

Longitude and Latitude

Complications arise when dealing with raw geographic coordinates, typically recorded in terms of degrees of longitude and latitude. In general, it isn't appropriate to use a 1:1 aspect ratio in these cases, for the simple reason that "one degree" of longitude is not typically equal to "one degree" of latitude in terms of familiar measures of distance (due in part to the curvature of the Earth). At a minimum, this means that creation of a two-dimensional plot based on coordinates referenced by longitude and latitude ought to ensure the axes possess a suitable aspect ratio in order to preserve (in an approximate sense) the actual planar appearance of the points.

This leads us into the territory of *geographic projections*, a collective term for methodologies by which geo-located positions on the Earth's surface can be appropriately transformed to lie on a two-dimensional "canvas" for subsequent analysis. Geographers, surveyors, and researchers in different areas of spatial science are adept at working with

these kinds of projections, of which there are many. This is a rich and complex area, a dutiful coverage of which warrants entire books (and contributed R packages!). As such, it is beyond the scope of this book.

Fortunately, however, there are relatively simple rules of thumb that you may use to service rough approximations from longitude and latitude to planar, distance-based measures. The underpinning concept is that units of distance per degree of latitude is roughly the same everywhere (one degree of latitude is approximately equal to 69 miles), but distance per degree of longitude depends on what the value of latitude is. Let's begin by assuming you want to produce a plot of coordinates expressed in longitude and latitude, but simply want to ensure you use a roughly appropriate aspect ratio to ensure the relative two-dimensional spatial positioning of your points is sensible. To find an appropriate value for `asp`, which we'll denote with a, use the following rule:

$$a = \frac{1}{\cos\left(\bar{l}\frac{\pi}{180}\right)} \qquad (14.1)$$

Here, cos is the standard *cosine* function of trigonometry available in R via the built-in `cos` command; π is the familiar geometric value 3.14 (to 2 d.p.), available in R as the built-in constant `pi`; and $\bar{l}$ is a "reference" value of latitude that represents your data well (simply using the mean latitude value of your data is one reasonable possibility for this). In most software packages (R included) the cosine function is programmed to accept angles expressed in *radians*, not degrees. This is why we need the $\frac{\pi}{180}$ in the denominator of (14.1) to scale the decimal-degrees latitude value accordingly.

Using skills from Chapter 11, here's the definition of a quick function you can use to find an appropriate value of a given a set of latitude values:

```
lat.asp <- function(lat){
  latmid <- mean(lat)
  p <- base::pi
```

```
  return(1/cos(latmid*p/180))
}
```

Note that `lat.asp` uses the mean value of the supplied latitude values as the reference point, as suggested; other options are just as valid, provided they reflect the scope of the data at hand.

After storing this function in your workspace, let's apply it to the latitude values of the built-in `quakes` data visualized in Figure 13-1 on page 307:

```
R> lat.asp(quakes$lat)
[1] 1.068608
```

With this result, you can now see why I recommended use of `asp=1.069` in the creation of Figure 13-1!

More generally, we can use the same rule to help convert longitudes and latitudes to distance units, like miles or kilometers. This is arguably more useful than leaving the units in degrees when you plan to perform some kind of spatial analysis. To perform a conversion to x-y coordinates expressed in miles, for instance, use the following information:

- Close to the equator, there are approximately 68.71 miles (or 110.57 kilometers) per degree of latitude, regardless of longitude. That is, 1° (lat) = 68.71 mi.
- At the equator, miles per degree of longitude is the widest: approximately 69.17 miles (111.32 kilometers). This shrinks according to the reference value of latitude. That is, 1° (long) = 69.17/*a* mi, where *a* is the result of (14.1).

There are many different ways to implement these rules, but one possible example of an R function to perform such a conversion is as follows:

```
longlat.conv <- function(long,lat,miles=TRUE){
  a <- lat.asp(lat)
  longzero <- long-min(long)
```

```
  latzero <- lat-min(lat)

  x <- 69.17/a*longzero
  y <- 68.71*latzero

  if(!miles){
    x <- 1.60934*x
    y <- 1.60934*y
  }

  return(list(x=x,y=y))
}
```

The function `longlat.conv` takes in your vectors of longitude and latitude measurements. It first uses the `lat.asp` function defined earlier to derive the appropriate relative scaling, storing the result as `a`. Then, the function creates "zeroed" versions of the longitude and latitude values by subtracting from these the minimum value of each. This forces the *origin coordinate*, (0, 0), to sit at the lower-left corner of the box that bounds all the points (this facilitates the conversion to distances). New vectors `x` and `y` are then created according to the aforementioned rules using the zeroed coordinate vectors to convert these to miles. If the optional argument `miles` is set to `FALSE`, the function further converts `x` and `y` to kilometers (1 mile is approximately 1.61 kilometers).

Operating on the `quakes` data, we can therefore obtain the coordinates in units of kilometers as follows:

```
R> quak.km <- longlat.conv(quakes$long,quakes$lat,miles=FALSE)
```

The resulting object is a list with components `x` and `y` containing the converted coordinates. Since these are now represented as measurable distance in any direction, any resulting plot should be produced with a 1:1 aspect ratio (that is, we shouldn't use the previously calculated value of *a*, as this has already been adjusted for).

Plotting the longitude and latitude values with the appropriate aspect ratio should therefore yield much the same spatial image as

plotting the converted coordinates. Consider now Figure 14-20.

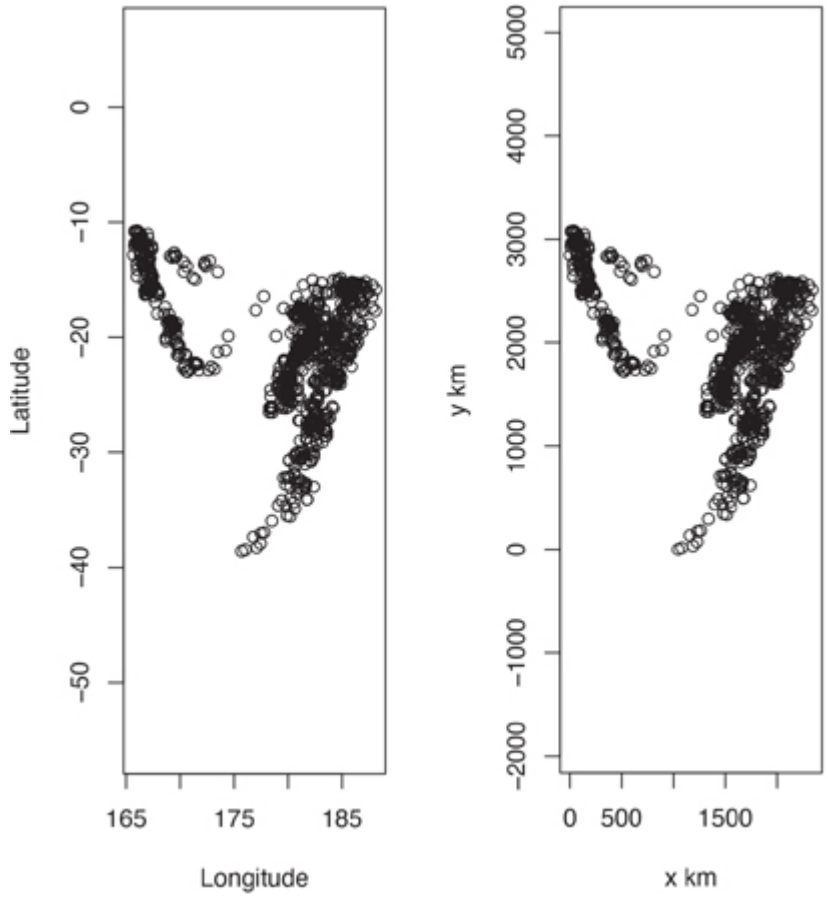


Figure 14-20: Using the built-in `quakes` *data to illustrate the presentation of geographic coordinates with a sensible approximation to spatial distances: plotting longitude and latitude values with an appropriate 1.069:1 aspect ratio (left) and plotting coordinates converted from longitude and latitude to distance-based values with a 1:1 aspect ratio (right)*

The plot on the left is a replica of Figure 13-1 (in a taller, skinnier window); on the right we see the plot of the distance-based coordinates in `quak.km` produced in a plotting window of identical size, with a 1:1 aspect ratio. Use the following two lines to create these plots:

```
R> plot(quakes$long,quakes$lat,asp=1.069,xlab="Longitude",ylab="Latitude")
```

and

```
R> plot(quak.km$x,quak.km$y,asp=1,xlab="x km",ylab="y km")
```

Although the units are quite different in both of these plots, either way of presenting the geographic coordinates retains the same appropriate appearance in terms of spatial distances.

It's worth reiterating that the conversions described here are an approximation. While they're fairly accurate for relatively small regions on the Earth's surface, and as such are sufficient for data exploration and many statistical pursuits, if you require highly accurate representations of distance you'll need to learn about geographic projections. Kessler & Battersby (2019) is a good general text on this topic and accompanying implementations in R are described in Comber & Brunsdon (2021) and Pebesma & Bivand (2023). The latter is written by authors of the contributed R package `sf`, a prominent package for spatial data analysis.

NOTE

While we've considered only base R plots of geographic coordinates here, similar tools are available in `ggplot2`*, typically linked to objects defined by the aforementioned* `sf` *package. As a good place to start, inspect* `ggplot2`*'s documentation for the* `coord_sf` *function (*`?coord_sf`*), as well as the book by Pebesma & Bivand (2023).*

Exercise 14.5

Store the following two vectors, which contain 11 mysterious coordinates in longitude (`longvals`) and latitude (`latvals`) in your R workspace:

```
longvals <- c(170.468,170.4697,170.469,170.4682,170.4684,
              170.4687,170.4691,170.4694,170.4698,170.4701,
              170.4702)
latvals <- c(-45.8634,-45.8634,-45.8639,-45.8641,-45.8642,
```

```
-45.8643,-45.8644,-45.8644,-45.8643,-45.8641,
-45.864)
```

Furthermore, ensure the two functions created in this section, `lat.asp` and `longlat.conv`, are available in your workspace.

a. Use `lat.asp` to identify the appropriate aspect ratio value to use if you were to plot these 11 coordinates.

b. Convert these 11 coordinates to distances in feet using `longlat.conv`. Note that by default, this function will return distances in miles, so you'll need to multiply the resulting coordinate values by the number of feet in a mile.

c. Create a plot of the raw longitude/latitude coordinate values using the appropriate value for `asp` as identified in (a). Also, create a plot of the coordinate values expressed in feet as found in (b). What's the correct value to use for the `asp` argument in the latter plot? Ensure your two plots yield points of identical appearance.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`barplot`	Create a bar plot	Section 14.1.1, p. 334
`geom_bar`	Bar plot geom (`ggplot2`)	Section 14.1.2, p. 336
`scale_x_discrete`	Modify discrete x-axis (`ggplot2`)	Section 14.1.2, p. 336
`scale_y_continuous`	Modify continuous y-axis (`ggplot2`)	Section 14.1.2, p. 336

Function/operator	Brief description	First occurrence
`theme_bw`	Black/white color theme (`ggplot2`)	Section 14.1.2, p. 336
`coord_flip`	Switch xand y-axes (`ggplot2`)	Section 14.1.2, p. 336
`scale_fill_grey`	Grayscale coloring (`ggplot2`)	Section 14.1.2, p. 336
`pie`	Create a pie chart	Section 14.1.3, p. 337
`hist`	Create a histogram	Section 14.2.1, p. 340
`geom_histogram`	Histogram geom (`ggplot2`)	Section 14.2.3, p. 342
`geom_vline`	Add vertical lines (`ggplot2`)	Section 14.2.3, p. 343
`scale_linetype_manual`	Alter line types (`ggplot2`)	Section 14.2.3, p. 343
`density`	Construct (kernel) density estimate	Section 14.3.1, p. 344
`geom_density`	Density geom (`ggplot2`)	Section 14.3.4, p. 348
`boxplot`	Create box plots	Section 14.4.1, p. 350
`~`	Plot “according to”	Section 14.4.2, p. 351
`geom_boxplot`	Box plot geom (`ggplot2`)	Section 14.4.3, p. 352

Function/operator	Brief description	First occurrence
`vioplot`	Create violin plots (`vioplot`)	Section 14.5.1, p. 354
`geom_violin`	Violin plot geom (`ggplot2`)	Section 14.5.3, p. 356
`pairs`	Scatterplot matrix	Section 14.6.2, p. 360
`scale_shape_manual`	Alter point characters (`ggplot2`)	Section 14.6.3, p. 361
`ggpairs`	Scatterplot matrix (`GGally`)	Section 14.6.3, p. 362

15
PROBABILITY



The concept of *probability* is central to statistical reasoning. Even the most complicated statistical techniques and models ultimately seek to make a probabilistic statement about some phenomenon. In this chapter, I'll use simple, everyday examples to illustrate this key idea in preparation for the remaining chapters. If you're already familiar with the basics of probability, random variables, and the associated terminology, you may want to skip ahead to Chapter 16, where R functionality begins to feature more prominently.

15.1 What Is a Probability?

A *probability* is a number that describes the "magnitude of chance" associated with making a particular observation or statement. It's always a number between 0 and 1 (inclusive) and is often expressed as a fraction. How you calculate a probability depends on the definition of a corresponding event.

15.1.1 Events and Probability

In statistics, an *event* typically refers to a specific outcome of interest. To describe the chance of event A actually occurring, you use a probability, denoted by $\Pr(A)$. At the extremes, $\Pr(A) = 0$ suggests A cannot occur, and $\Pr(A) = 1$ suggests that A occurs with complete certainty.

Let's say you roll a six-sided, fair die. Let A be the event "you roll a 5 or a 6." You can assume that each outcome on a standard die has a probability of occurring 1/6 in any given roll. Under these conditions, you have this:

$$\Pr(A) = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$$

This is a *frequentist*, or *classical*, probability, and it is assumed to be the relative frequency with which an event occurs over many identical, objective trials.

As another example, say you're married and arrive home much later than usual. Let B be the event "your significant other is angry because of your tardiness." It's a relatively straightforward process to observe A in a mathematical sense, but B isn't so objectively observed, and the quantity can't be easily computed. Instead, you might assign a number to $\Pr(B)$ given your own past experience. For example, you might say "I think $\Pr(B) = 0.5$" if you think there's a 50-50 chance your partner will be mad, but this would be based on your personal impressions of the situation and knowledge of your spouse's temperament or mood, not on an impartial experiment that could be easily reproduced for any two individuals. This is a *Bayesian* probability, which uses prior knowledge or subjective belief to inform the calculations.

Owing to its naturally implied objectivity, the frequentist interpretation is the generally assumed definition of probability; this book focuses on frequentist probabilities. If you're interested in Bayesian analyses using R, Kruschke (2010) is a well-received text on the subject. For a gentle, example-driven introduction to the relevant ideas, see also the book by Kurt (2019).

NOTE

Though it's tempting to define the concept of probability in terms of likelihood *(and colloquially, many do), likelihood means something slightly different in statistical theory, so I'll avoid this term for now.*

The way in which you compute probabilities when considering multiple events is determined by several important rules. These are similar in nature to the concepts of AND and OR that are key to comparing the logical values `TRUE` and `FALSE` in R via `&&` and `||` (refer to Section 4.1.3). Just like these logical comparisons, calculation of probabilities based on several defined events can usually be broken down into a specific calculation concerning two distinct events.

To serve as a simple running example over the next few sections, assume you roll a standard die and define event A to be "you roll a 4 or more" and event B to be "you roll an even number." You can therefore conclude that in isolation, $\Pr(A) = \frac{1}{2}$ and $\Pr(B) = \frac{1}{2}$.

15.1.2 Conditional Probability

A *conditional* probability is the probability of one event occurring after taking into account the occurrence of another event. The quantity $\Pr(A \mid B)$ represents "the probability that A occurs, *given* that B has occurred," and vice versa if you write $\Pr(B \mid A)$.

If $\Pr(A \mid B) = \Pr(A)$, then the two events are *independent*; if $\Pr(A \mid B) \neq \Pr(A)$, then the two events are *dependent*. Generally, you can't assume that $\Pr(A \mid B)$ is equal to $\Pr(B \mid A)$.

Turn to the events A (you roll a 4 or more) and B (you roll an even number) as defined previously. You already know that $\Pr(A) = \frac{1}{2}$. Now think of $\Pr(A \mid B)$. What is the probability your outcome is a 4 or more, *given* an even number has occurred? Since there are three even numbers (2, 4, and 6) the probability that you obtain a 4 or more, assuming you know an even number has occurred, is $\frac{2}{3}$. Thus, $\Pr(A \mid B) \neq \Pr(A)$ in this

context, and the two events are therefore not independent (that is, they are *de*pendent).

15.1.3 Intersection

You write the *intersection* of two events as $\Pr(A \cap B)$ and read it as "the probability that both A and B occur simultaneously." It is common to represent this as a Venn diagram, as Figure 15-1 shows.

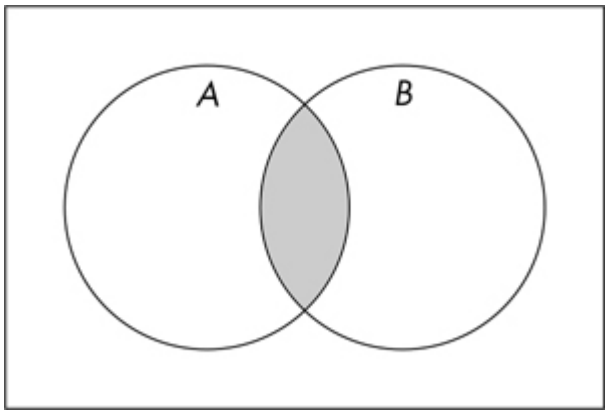


Figure 15-1: A Venn diagram representing the intersection of two events (shaded area)

The circle labeled A represents the outcome(s) satisfying A, and disc B represents the outcome(s) for B. The shaded area represents the specific outcome(s) that satisfies both A and B—the intersection—and the area outside both discs represents the outcome(s) satisfying neither A nor B. Theoretically, the intersection between two events is derived from the following equation:

$$\Pr(A \cap B) = \Pr(A|B) \times \Pr(B) \quad \text{or} \quad \Pr(B|A) \times \Pr(A) \qquad (15.1)$$

If $\Pr(A \cap B) = 0$, then the two events are *mutually exclusive*. In other words, they cannot occur simultaneously. Note that if the two events are independent, Equation (15.1) simplifies to $\Pr(A \cap B) = \Pr(A) \times \Pr(B)$.

Returning to the die example, what is the probability that on a single toss you roll an even number *and* it's a 4 or more? Using the fact that

$\Pr(A|B) = \frac{2}{3}$ and that $\Pr(B) = \frac{1}{2}$, you can compute $\Pr(A \cap B) = \frac{2}{3} \times \frac{1}{2} = \frac{1}{3}$ and confirm this in R:

```
R> (2/3)*(1/2)
[1] 0.3333333
```

We conclude that the two events are not mutually exclusive because $\Pr(A \cap B) \neq 0$. This makes sense; it's perfectly possible in a die roll to observe a number that's both even and at least 4.

15.1.4 Union

You write the *union* of two events as $\Pr(A \cup B)$ and read it as "the probability that A or B occurs." Figure 15-2 illustrates this concept.

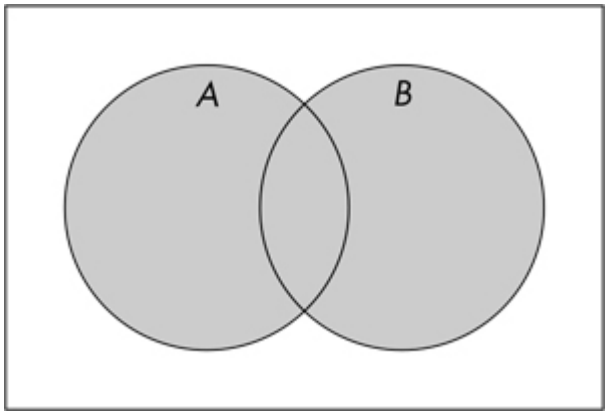


Figure 15-2: A Venn diagram representing the union of two events (shaded area)

Theoretically, we find the union using the following equation:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B) \qquad (15.2)$$

You need to subtract the intersection in this equation because summing $\Pr(A)$ and $\Pr(B)$ alone would incorrectly count $\Pr(A \cap B)$ twice. Note, though, that in the special case where the two events are mutually exclusive, Equation (15.2) simplifies $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ because the intersection is exactly 0 in this circumstance.

So, in rolling the die, what's the probability that you observe an even number *or* a number that's at least 4? Use Equation (15.2) to find that $\Pr(A \cup B) = \frac{1}{2} + \frac{1}{2} - \frac{1}{3} = \frac{2}{3}$. To confirm this in R, enter the following:

```
R> (1/2)+(1/2)-(1/3)
[1] 0.6666667
```

15.1.5 Complement

Lastly, you write the probability of the *complement* of an event as $\Pr\left(A\right)$ and read it as 'the probability that A does *not* occur.'

Figure 15-3 illustrates the concept as a Venn diagram.

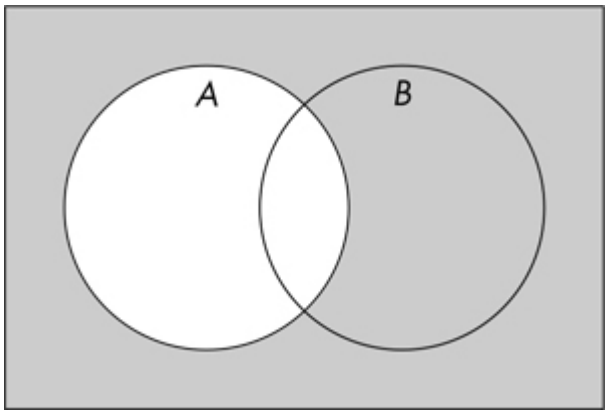


Figure 15-3: A Venn diagram representing the complement of an event (shaded area)

We calculate the complement using the following equation:

$$\Pr\left(A\right) = 1 - \Pr(A)$$

Wrapping up the running example, use this equation to find the probability that you do *not* roll a 4 or greater: $\Pr\left(A\right) = 1 - \frac{1}{2} = \frac{1}{2}$. If a 4, 5, or 6 is not obtained, then you must've rolled a 1, 2, or 3, so there are three possible outcomes left out of the six.

While the die-rolling example may not represent the most pressing need facing statistical researchers today, it provides some clear illustrations of the behavior and terminology associated with the rules of probability. These rules apply across the board and play an important role in the interpretation of arguably more pressing endeavors in statistical modeling.

Exercise 15.1

You have a standard deck of 52 playing cards. There are two colors (black and red) and four suits (black spades, black clubs, red hearts, and red diamonds). Each suit has 13 cards, in which there is an Ace, numbered cards from 2 to 10, and three face cards (Jack, Queen, and King).

a. You randomly draw and then replace a card. What's the probability it's an Ace? What's the probability it's the 4 of spades?

b. You randomly draw a card, and after replacing it, you shuffle and randomly draw another. Let A be the event that a drawn card is red; let B be the event that a drawn card is a club. What is $\Pr(B|A)$? That is, what is the probability the second card is a club, *given* the first one was a red card? Are the two events independent?

c. Repeat (b), this time assuming that when the first card is drawn, it is not replaced. Would this change your answer to (b) in terms of independence?

d. Let C be the event a card is a face card, and let D be the event a card is black. You draw a single card. Evaluate $\Pr(C \cap D)$. Are the two events mutually exclusive?

15.2 Random Variables and Probability Distributions

A *random variable* is a variable whose specific outcomes are assumed to arise by chance or according to some random or *stochastic* mechanism.

You encountered the definition of a *variable*—a characteristic that describes an individual entity based on data you've observed—in Section 13.1. *Random* variables are more of a theoretical concept, whereby we declare or define an event of interest without having yet made an observation. We then state as a probability the chances of observing a specific value, or one within a specific interval, for that defined random variable.

Therefore, you should think of random variables as being tied to a mathematical function that defines how these probabilities are distributed across all possible outcomes; this mathematical function is a *probability distribution*. In this section, you'll look at some elementary ways in which random variables are summarized and how their corresponding probability distributions are dealt with in a statistical sense.

15.2.1 Realizations

So, the concept of a random variable revolves around the consideration of the possible outcomes of a variable in a probabilistic fashion. We refer to the data collected by observing outcomes of a random variable as *realizations*.

For example, suppose you roll your beloved die. Define the random variable Y to be the result. The possible outcomes (and hence the possible realizations we might make) are $Y = 1$, $Y = 2$, $Y = 3$, $Y = 4$, $Y = 5$, and $Y = 6$.

Now let's say you routinely monitor the maximum daily temperature at your preferred picnic spot. Let the random variable W be the temperature in degrees Fahrenheit you observe there. Theoretically,

you might say that the possible realizations of W will lie somewhere in the interval $-\infty < W < \infty$.

The two preceding examples serve to illustrate two types of random variables. While both are numerically valued, Y is a *discrete random variable*, where outcomes are restricted to a finite, countable set, and W is a *continuous random variable*, where there is technically an infinite number of possible outcomes. Whether any given random variable is discrete or continuous has consequences for the way in which you think about, and may utilize, the probabilities associated with making realizations.

15.2.2 Discrete Random Variables

A discrete random variable follows the same definitions as the variables covered in Chapter 13. Its realizations can take on only certain precise values, for which no other degree of measurement accuracy is possible or interpretable. Rolling a standard die can result in only those six distinct possibilities described previously by Y, and it would make no sense to observe, for example, "5.91."

A probability is directly tied to defined outcomes known as *events* (see Section 15.1.1). When discussing a discrete random variable, events are therefore defined with respect to the distinct possible values the variable can take, and the corresponding probability distribution is formed when you jointly consider the collection of all the probabilities associated with all possible realizations.

We call probability distributions tied to discrete random variables *probability mass functions*. Since these define the probabilities of all possible outcomes, the sum of the probabilities in any complete probability mass function must always equal exactly 1.

Let's consider another example. Suppose you go into a casino and play a simple gambling game. At each turn, you can either lose \$4 with probability 0.32, break even (win or lose nothing) with probability 0.48, win \$1 with probability 0.15, or win \$8 with probability 0.05. Because these are the only four possible outcomes, the probabilities sum to 1.

Define the discrete random variable X as the "amount earned" at each turn you have. Table 15-1 expresses the distribution of these probabilities.

Table 15-1: The Probabilities and Cumulative Probabilities for the Amount Won, *X*, in a Hypothetical Gambling Game

x	–4	0	1	8
$\Pr(X = x)$	0.32	0.48	0.15	0.05
$\Pr(X \leq x)$	0.32	0.80	0.95	1.00

Note that the loss of $4 is represented as a negative earning as per the definition of X.

Cumulative Probability Distributions

The *cumulative probability* is also an important part of the general idea of a probability distribution. A cumulative probability for a random variable X is "the probability of observing less than or equal to x" and you write it as $\Pr(X \leq x)$. In the discrete case, you obtain the distribution of cumulative probabilities by summing the individual probabilities of the mass function up to and including any given value of x. See the bottom row of Table 15-1. For example, though $\Pr(X = 0)$ is 0.48, $\Pr(X \leq 0) = 0.32 + 0.48 = 0.80$.

Visualizing probability distributions is always useful, and because of the discrete nature of X, it's easy to use the `barplot` function for this. Using skills from Section 14.1, the following code first stores vectors of the possible outcomes and corresponding probabilities (`X.outcomes` and `X.prob`, respectively) and then produces the left image in Figure 15-4:

```
R> X.outcomes <- c(-4,0,1,8)
R> X.prob <- c(0.32,0.48,0.15,0.05)
R> barplot(X.prob,ylim=c(0,0.5),names.arg=X.outcomes,space=0,
          xlab="x",ylab="Pr(X = x)")
```

The optional argument `space=0` eliminates the gaps between the bars.

Next, you can use the built-in `cumsum` function to progressively sum the entries in `X.prob`, giving you the cumulative probabilities:

```
R> X.cumul <- cumsum(X.prob)
R> X.cumul
[1] 0.32 0.80 0.95 1.00
```

Now using `X.cumul`, you can plot the cumulative probability distribution in the same way as earlier; the following line

```
R> barplot(X.cumul,names.arg=X.outcomes,space=0,xlab="x",ylab="Pr(X <= x)")
```

generates the right panel of Figure 15-4.

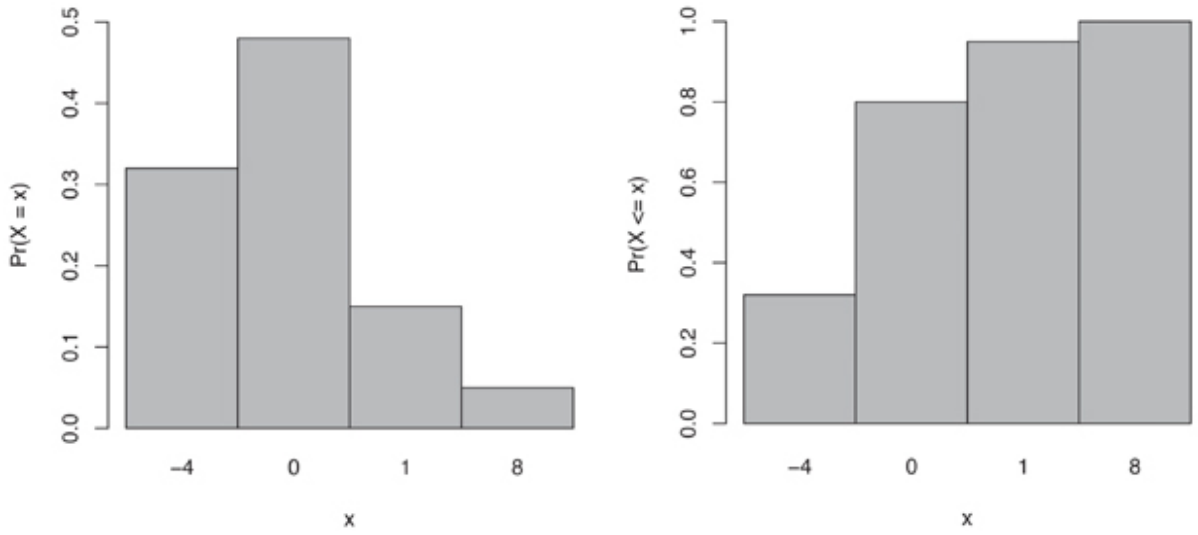


Figure 15-4: Visualizing the probability distribution associated with event-specific probabilities of a hypothetical gambling game (left) and the corresponding cumulative probability distribution (right)

Generally, it's important to remember the following three facts for any probability mass function based on a discrete random variable X:

- There are k distinct possible outcomes $x_1, \ldots, x_k$.
- For each x_i, where $i = \{1, \ldots, k\}$, $0 \leq \Pr(X = x_i) \leq 1$.
- $\sum_{i=1}^{k} \Pr(X = x_i) = 1$.

Mean and Variance

It's useful to be able describe or summarize properties of a random variable of interest as you would for raw data. The most useful two properties are the mean and variance, both of which depend upon the relevant distribution of probabilities associated with that random variable.

For some discrete random variable X, the *mean* μ_X (also referred to as the *expectation* or the *expected value* $\mathbb{E}[X]$) is the "average value of the outcomes" that you can expect over many realizations. Assuming X has k possible outcomes labeled $x_1, x_2, \ldots, x_k$, you find the mean of a discrete random variable using the following equation:

$$\begin{aligned} \mu_X = E[X] &= x_1 \times \Pr(X = x_1) + \ldots + x_k \times \Pr(X = x_k) \\ &= \textstyle\sum_{i=1}^{k} x_i \Pr(X = x_i) \end{aligned} \tag{15.3}$$

To find the mean, simply multiply the numeric value of each outcome by its corresponding probability, and sum the results.

For a discrete random variable X, the *variance* σ_X^2, also written as Var[X], quantifies the variability in the possible realizations of X. Theoretically, in terms of expectations, it can be shown that $\sigma_X^2 = \text{Var}[X] = E\left[X^2\right] - E[X]^2 = E\left[X^2\right] - \mu_X^2$. Calculation of the discrete random variable variance depends upon its mean μ_X and is given as follows:

$$\begin{aligned} \sigma_X^2 = \text{Var}[X] &= (x_1 - \mu_X)^2 \times \Pr(X = x_1) \\ &\quad + \ldots + (x_k - \mu_X)^2 \times \Pr(X = x_k) \\ &= \textstyle\sum_{i=1}^{k} (x_i - \mu_X)^2 \times \Pr(X = x_i) \end{aligned} \tag{15.4}$$

Again, the calculation is straightforward: You compute the variance by squaring the differences between each realization and mean and then multiplying by the corresponding probability of the occurrence before summing these products.

In practice, the probabilities associated with each outcome are often unknown and are estimated from observed data. Following that step, you apply the formulas in Equations (15.3) and (15.4) to obtain *estimates* of the respective properties. Also, note that the general descriptions of mean and variance are the same as in Section 13.2, only you're now quantifying centrality and spread with respect to a random phenomenon.

Let's consider the gambling game with the possible realizations of the amount earned, X, and the associated probabilities as specified in Table 15-1. R's vector-oriented behavior (refer to Section 2.3.4), leads to straightforward calculation of the mean and variance of X. With the objects `X.outcomes` and `X.prob` as defined previously, you can get the mean of X by summing the appropriate element-wise multiplication:

```
R> mu.X <- sum(X.outcomes*X.prob)
R> mu.X
[1] -0.73
```

So, $\mu_X = -0.73$.

By the same token, the following provides the variance of X:

```
R> var.X <- sum((X.outcomes-mu.X)^2*X.prob)
R> var.X
[1] 7.9371
```

You can also compute the standard deviation by taking the square root of the variance (recall the definitions in Section 13.2.5). This is done with the built-in `sqrt` command (Section 2.1.1):

```
R> sd.X <- sqrt(var.X)
R> sd.X
[1] 2.817286
```

Based on these results, we learn important things about the gambling game and its behavior. The expected outcome of –0.73 suggests that, on average, you'll lose $0.73 per turn, but that making individual realizations is relatively "noisy," with a standard deviation of

about $2.82. Remember these specific quantities are not, and need not be, one of the specifically defined outcomes; they are mathematical properties of a random process that essentially reflect long-run averages.

15.2.3 Continuous Random Variables

Again following from the definitions of variables from Chapter 13, a continuous random variable has no limit to the number of possible realizations. For a discrete random variable, it is natural to think of a specific outcome as an event and assign it a corresponding probability. Things are a little different when you're dealing with a continuous random variable, however.

General Concept

Recall the picnic example in Section 15.2.1, where we defined the random variable W to be the maximum temperature on any given day. We originally supposed $-\infty < W < \infty$, which clearly encapsulates an infinite number of possible realisations. However, even if you restrict the range of possible values to be something more realistic—say, that W could only take on a value between 40 and 90 degrees Fahrenheit ($40 \leq W \leq 90$)—there are still an infinite number of distinct values on that restricted continuum. This is because measuring 59.1 degrees makes as much sense as observing something like 59.16742 degrees, were we able to do so. The number of unique outcomes is still infinite because theoretically, any temperature value to an arbitrary level of precision is still a valid realization.

As such, it isn't possible to assign probabilities to specific, single temperatures; instead, we only think about probabilities being assigned to *intervals* of values. This means that asking something like "What is the probability the temperature is *exactly* 55.2 degrees Fahrenheit?"—$\Pr(W = 55.2)$—is not a valid question. However, asking "What is the probability W is *less than or equal to* 55.2 degrees?"—$\Pr(W \leq 55.2)$—is answerable because it defines an interval.

Probability Density Functions

With continuous random variables, the function that describes the distribution of probabilities must be *continuous* on the range of possible outcomes. You compute probabilities as "areas underneath" that continuous function, and the "total area" underneath a continuous probability distribution must evaluate to exactly 1. For outcomes that are more probable than others, this continuous function will have higher values because there will be a higher concentration or "density" of probability associated with them. For this reason, we call a probability distribution tied to a continuous random variable a *probability density function*.

These facts will become clearer when considering the following example. Suppose you're told the probabilities associated with the picnic temperature random variable $40 \leq W \leq 90$ follow the density function $f(w)$, where the following is true:

$$f(w) = \begin{cases} \frac{w-40}{625} & \text{if } 40 \leq w \leq 65; \\ \frac{90-w}{625} & \text{if } 65 \leq w \leq 90; \\ 0 & \text{otherwise} \end{cases} \qquad (15.5)$$

The division by 625 is needed in this particular function to ensure a total probability of 1. This makes more sense with visualization. To plot this density function, first consider the following code:

```
R> w <- seq(35,95,by=5)
R> w
[1] 35 40 45 50 55 60 65 70 75 80 85 90 95
R> lower.w <- w>=40 & w<=65
R> lower.w
 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[11] FALSE FALSE FALSE
R> upper.w <- w>65 & w<=90
R> upper.w
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[11]  TRUE  TRUE FALSE
```

The first assignment sets up an even sequence of values to represent certain realizations of w simply called `w`; the second assignment uses relational operators (see Section 4.1.2) and the element-wise logical operator `&` (see Section 4.1.3) to create a logical flag vector identifying those elements of `w` that form the "lower half" of values for $f(w)$ as defined by Equation (15.5); the third assignment does the same for the "upper half" of values.

The next lines make use of `lower.w` and `upper.w` to evaluate the correct result of $f(w)$ for the entries in `w`:

```
R> fw <- rep(0,length(w))
R> fw[lower.w] <- (w[lower.w]-40)/625
R> fw[upper.w] <- (90-w[upper.w])/625
R> fw
 [1] 0.000 0.000 0.008 0.016 0.024 0.032 0.040 0.032 0.024 0.016
[11] 0.008 0.000 0.000
```

Note that this code doesn't result in $f(w)$ for all possible values of w; you've merely created the vector `w` as a finite set of values and obtained the corresponding values of the mathematical function at those arguments, giving the vector `fw`. However, these two vectors are sufficient for plotting. Using skills from Chapter 7, you can plot a line representing the continuous density function $f(w)$ for $35 \leq w \leq 95$:

```
R> plot(w,fw,type="l",ylab="f(w)")
R> abline(h=0,col="gray",lty=2)
```

The plot is given as the left panel of Figure 15-5; note the addition of a dashed horizontal line at $f(w) = 0$ using `abline`.

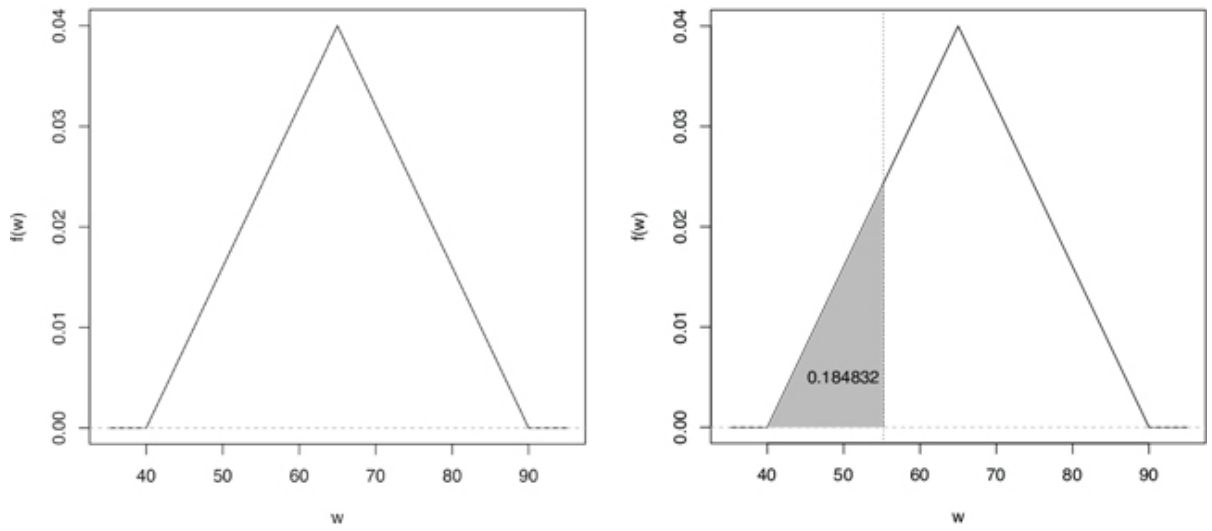


Figure 15-5: Visualizing the hypothetical probability density function as defined by Equation (15.5) for the picnic temperature random variable W *(left) and illustrating the computation of a specific probability from the text (right)*

The continuous function defined by Equation (15.5) yields a triangular shape, with an apex at $w = 65$. The increasing line from $w = 40$ to $w = 65$ represents the first of the three components of (15.5), the decreasing line represents the second component, and for all $w < 40$ and $w > 90$, the line sits at 0, which is the third and final component of (15.5).

Generally, any function $f(w)$ that defines a probability density for a random variable W must possess the following properties:

- $f(w) \leq 0$ for all $-\infty < w < \infty$
- $\int_{-\infty}^{\infty} f(w)\, dw = 1$ (the total area underneath the function must be 1)

In terms of the temperature example, you can see from Equation (15.5) that $f(w) \leq 0$ for any value of w. To calculate the total area underneath the function, you need be concerned only with the function evaluated at $40 \leq w \leq 90$, since it's 0 everywhere else.

You can do this geometrically by working out the area of the triangle formed by the function and the horizontal line at 0. For this triangle, you can use the elementary half-base-times-height rule. The base of the

triangle is 90 – 40 = 50, and the apex is at a value of 0.04. So, in R, you can calculate half the base width times the height with the following:

```
R> 0.5*50*0.04
[1] 1
```

This confirms it is indeed equal to 1; you can now see the reason behind my specific definition in Equation (15.5).

Let's return to the question of obtaining the probability that the maximum temperature is less than or equal to 55.2 degrees Fahrenheit on a given day. For this, you must find the area underneath $f(w)$, the probability density function, bounded by the horizontal line at 0 and an imaginary vertical line at 55.2. This particular area forms another triangle, for which it is again appropriate to use the half-base-times-height rule. In Cartesian coordinates, this is the triangle formed by the vertices at (40, 0), (55.2, 0), and (55.2, $f(55.2)$), as shown in the right panel of Figure 15-5; you'll see how to plot this in a moment.

You should first work out the value $f(55.2)$. From Equation (15.5), this is provided by creating the following object:

```
R> fw.specific <- (55.2-40)/625
R> fw.specific
[1] 0.02432
```

Note that this *isn't* a probability; it cannot be assigned to specific realizations. It's just the value of the continuous density function that you're going to need in order to calculate the interval-based probability $\Pr(W \leq 55.2)$.

You can determine that the base of the triangle of interest in this particular setting is 55.2 – 40 = 15.2. Then, along with `fw.specific`, note that half-base-times-height gives the following:

```
R> fw.specific.area <- 0.5*15.2*fw.specific
R> fw.specific.area
[1] 0.184832
```

You've shown geometrically, using $f(w)$, that $\Pr(W \leq 55.2) = 0.185$ (when rounded to three decimal places). In other words, you can say that there is roughly an 18.5 percent chance that the maximum temperature at the picnic spot will be less than or equal to 55.2 degrees Fahrenheit.

Again, all this is easier to digest visually. The following R code replots the density function $f(w)$ and marks off and shades the area of interest:

```
R> fw.specific.vertices <- rbind(c(40,0),c(55.2,0),c(55.2,fw.specific))
R> fw.specific.vertices
     [,1]    [,2]
[1,] 40.0 0.00000
[2,] 55.2 0.00000
[3,] 55.2 0.02432
R> plot(w,fw,type="l",ylab="f(w)")
R> abline(h=0,col="gray",lty=2)
R> polygon(fw.specific.vertices,col="gray",border=NA)
R> abline(v=55.2,lty=3)
R> text(50,0.005,labels=fw.specific.area)
```

The result is the right panel of Figure 15-5. The plotting commands should be familiar from Chapter 7, barring `polygon`. The built-in `polygon` function allows you to supply custom vertices in order to draw or shade a polygon upon an existing plot. Here, a matrix with two columns is defined using `rbind`, providing the x and y locations (first and second columns, respectively) of the three corners of the triangle to shade. Note that the creation of `fw.specific.vertices` has made use of `fw.specific`, the value of $f(w)$ at $w = 55.2$; this is the topmost vertex of the shaded triangle. Further arguments to `polygon` control the shading (`col="gray"`) and the color with which to draw a border around the defined polygon (`border=NA` requests no border).

Not all density functions can be appraised in this simple geometric fashion. Formally, *integration* is the mathematical operation used to find areas under a continuous function, denoted with the $\int$ symbol. That is, the mathematically inclined reader familiar with this technique should

find it straightforward to show that "the area under $f(w)$ from $w = 40$ to $w = 55.2$, providing $\Pr(W \leq 55.2)$," is yielded by the following:

$$\begin{aligned}
\int_{40}^{55.2} f(w)\,\mathrm{d}w &= \int_{40}^{55.2} \tfrac{w-40}{625}\,\mathrm{d}w \\
&= \left.\tfrac{w^2-80w}{1250}\right|_{40}^{55.2} \\
&= \tfrac{55.2^2-80\times 55.2-40^2+80\times 40}{1250} \\
&= 0.185 \qquad \text{(rounded to 3 d.p.)}
\end{aligned}$$

In R, the third line of this calculation looks like this:

```
R> (55.2^2-80*55.2-40^2+80*40)/1250
[1] 0.184832
```

And again, R finds the correct result. I'll leave the mathematical details aside, but it's nice to confirm that the more general integral does match the intuitive geometric solution based on the area of the triangle, computed earlier as `fw.specific.area`.

Hopefully it's becoming clearer why it doesn't make sense to assign probabilities to single, specific realizations associated with continuous random variables. For example, evaluating the "area under the function $f(w)$" at a single value is the same as finding the area of a polygon with a base width of *zero*, and hence, the probability itself is technically zero for *any* $\Pr(W = w)$. Furthermore, in the continuous setting, it makes no difference to your calculations if you use $<$ or $\leq$, or $>$ or $\leq$. So although you found $\Pr(W \leq 55.2)$ earlier, if you had been tasked to find $\Pr(W < 55.2)$, you would have gotten the same answer of 0.185. It may seem a little unnatural at first, but it all comes down to the idea of an infinite number of possible realizations such that there's no meaningful interpretation of "equality" to a distinct value.

Cumulative Probability Distributions

We interpret the cumulative probability distribution for a continuous random variable in the same way as for a discrete variable. Given a

certain value w, the cumulative distribution function provides the probability of observing w or less. This may seem familiar; the probability you worked out earlier, $\Pr(W \leq 55.2)$, based on the shaded triangle on the right of Figure 15-5 or using analytical methods, is itself a cumulative probability. More generally, you find a cumulative probability for a continuous random variable by calculating the area under the density function of interest from $-\infty$ to w. This general treatment therefore requires mathematical integration of the relevant probability density function. Looking at Figure 15-5, imagine a vertical line moving from left to right of the density plot and, at every location, evaluate the area under the density function to the left of that line.

For the picnic temperature density function defined in Equation (15.5), it can be shown that the cumulative distribution function F is:

$$\begin{aligned} F(w) &= \int_{-\infty}^{w} f(u)\,\mathrm{d}u \\ &= \begin{cases} 0 & \text{if } w < 40; \\ \frac{w^2 - 80w + 1600}{1250} & \text{if } 40 \leq w \leq 65; \\ \frac{180w - w^2 - 6850}{1250} & \text{if } 65 < w \leq 90; \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{15.6}$$

Making use of the sequence `w` and the logical flag vectors `lower.w` and `upper.w` created in "Continuous Random Variables" on page 381, you can use the same vector subset-and-overwrite approach to plot $F(w)$. The following code creates the required vector `Fw`:

```
R> Fw <- rep(0,length(w))
R> Fw[lower.w] <- (w[lower.w]^2-80*w[lower.w]+1600)/1250
R> Fw[upper.w] <- (180*w[upper.w]-w[upper.w]^2-6850)/1250
R> Fw[w>90] <- 1
R> plot(w,Fw,type="l",ylab="F(w)")
R> abline(h=c(0,1),col="gray",lty=2)
```

The final two lines produce the plot in Figure 15-6.

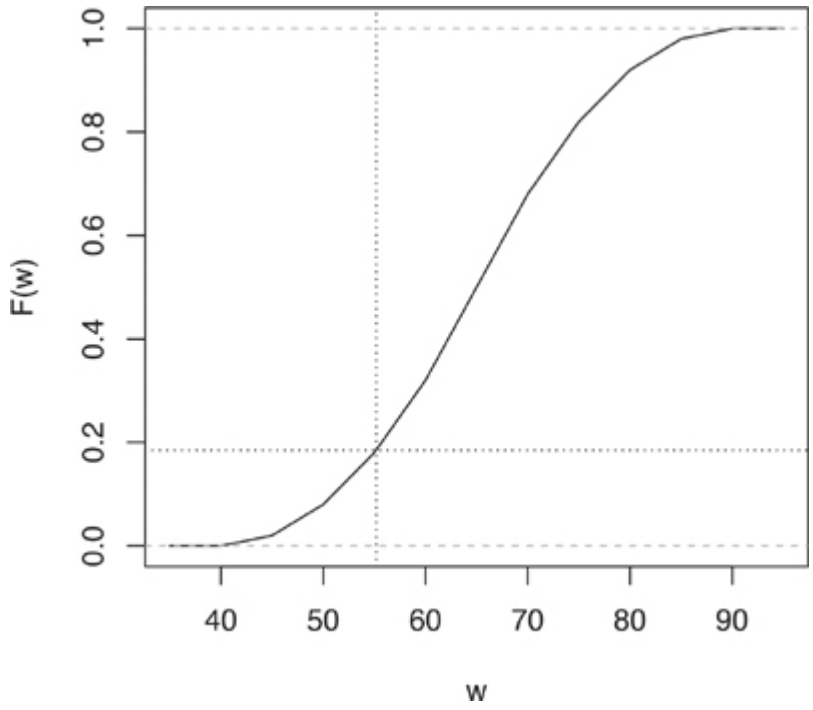


Figure 15-6: Plotting the cumulative distribution function of the picnic temperature example, which is given as Equation (15.6). The cumulative probability of observing a temperature less than (or equal to) 55.2 is marked with dotted vertical and horizontal lines.

Including the following two lines after the creation of this plot clearly identifies the fact that at $w = 55.2$, the cumulative probability you worked out earlier, $\Pr(W \leq 55.2)$, is located precisely on the curve of F:

```
R> abline(v=55.2,lty=3)
R> abline(h=fw.specific.area,lty=3)
```

Mean and Variance

Naturally it's also possible, and useful, to determine the mean and variance of a continuous random variable.

For a continuous random variable W with density f, you interpret the mean μ_W (or *expectation* or *expected value* $\mathbb{E}[X]$) as the "average outcome" that you can expect over many realizations (the same interpretation as for discrete random variables). The mathematical definition is as follows:

$$\mu w = E[W] = \int_{-\infty}^{\infty} w f(w)\, \mathrm{d}w \quad (15.7)$$

This equation represents the continuous analogue of Equation (15.3) and you can read it as "the total area underneath the function given by multiplication of the density $f(w)$ with the value of w itself."

For W, the variance σ_W^2, also written as Var[W], quantifies the variability inherent in realizations of W. Calculation of the continuous random variable variance depends upon its mean μ_W and is given as follows:

$$\sigma_W^2 = \mathrm{Var}[W] = \int_{-\infty}^{\infty} (w - \mu W)^2 f(w)\, \mathrm{d}w \qquad (15.8)$$

Again, the procedure is to find the area under the density function multiplied by a certain quantity—in this case, the squared difference of the value of w with the overall expected value μ_W.

Evaluation of the mean and variance of the picnic temperature random variable must follow (15.7) and (15.8), respectively. These calculations become rather complex, so I won't reproduce them here. Figure 15-5, however, shows that the mean of W must be $\mu_W = 65$; it is the perfect center of the symmetric density function $f(w)$.

In terms of the required integrals, you can therefore use the previously stored `w` and `fw` objects to view the two functions, $wf(w)$ and $(w - \mu_W)^2 f(w)$, by executing the following:

```
R> plot(w,w*fw,type="l",ylab="wf(w)")
R> plot(w,(w-65)^2*fw,type="l",ylab="(w-65)^2 f(w)")
```

This produces the two images in Figure 15-7.

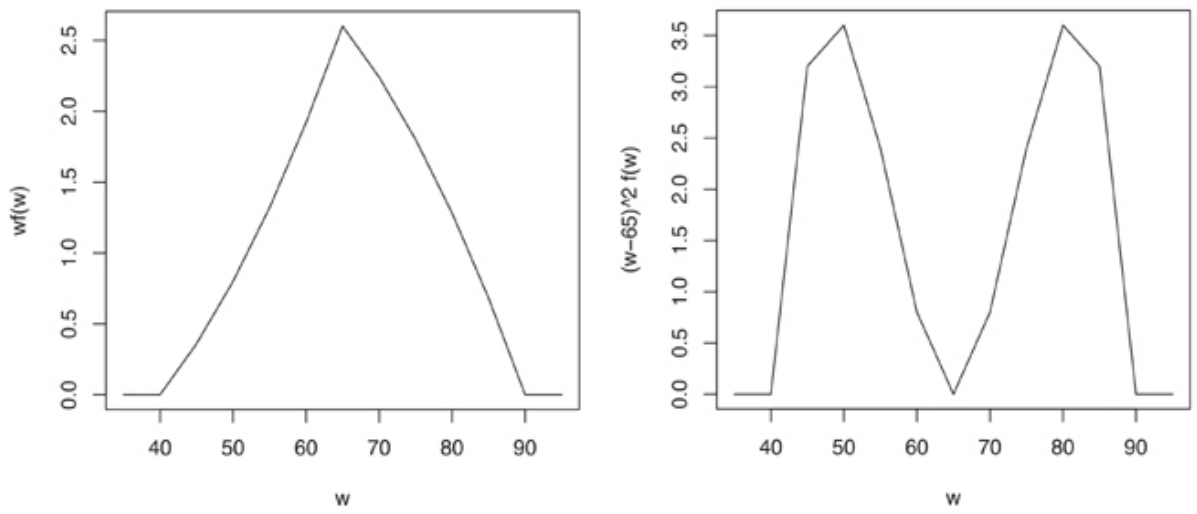


Figure 15-7: Integrands for the expected value (left) and variance (right) of the probability density function for the temperature example

Rest assured that the following results can be shown mathematically, using Equations (15.7) and (15.8):

$$\mu_W = 65 \quad \text{and} \quad \text{Var}\left[W\right] = 104.1667$$

By visually approximating the area underneath each of the drawn lines in Figure 15-7, you should find these results are consistent. As earlier, the standard deviation of the distribution of W is given by the square root of the variance, and the following readily provides this value:

```
R> sqrt(104.1667)
[1] 10.20621
```

15.2.4 Shape, Skew, and Modality

At this point, you're familiar with both continuous and discrete random variables and their natural pairings with a distribution of probabilities, and you've had a look at visualizations of distributions of probability mass and density functions. In this section, I'll define some terminology commonly used to describe the appearance of these distributions; being

able to describe your visual impressions is just as important as being able to readily compute them.

Symmetry A distribution is *symmetric* if you can draw a vertical line down the center and it is equally reflected, with 0.5 probability falling on either side of this center line (see e.g. Figure 15-5). A symmetric probability distribution implies that the mean and the median of the distribution are identical.

Skew If a distribution is *asymmetric*, you can often qualify your description further by discussing *skew*. When the "tail" of a distribution (in other words, moving away from its measures of centrality) tapers off longer in one direction than the other, we identify this as the direction of the skew. *Positive* or *right* skew indicates a tail extending longer to the right of center; *negative* or *left* skew refers to a tail extending longer to the left of center. You could also qualify the strength or prominence of the skew.

Modality A probability distribution is not required to have a single peak. *Modality* describes the number of easily identifiable peaks in the distribution of interest. For example, we use the terms *unimodal*, *bimodal*, and *trimodal* to describe distributions with one, two, and three peaks, respectively. More generally, we could describe distributions with more than one clear peak as being *multi*-modal.

Figure 15-8 provides some visual interpretations of symmetry, asymmetry, skew, and modality. (Note that although they are drawn with a continuous line, you can assume they represent the general shape of either a discrete probability mass function *or* a continuous density function.)

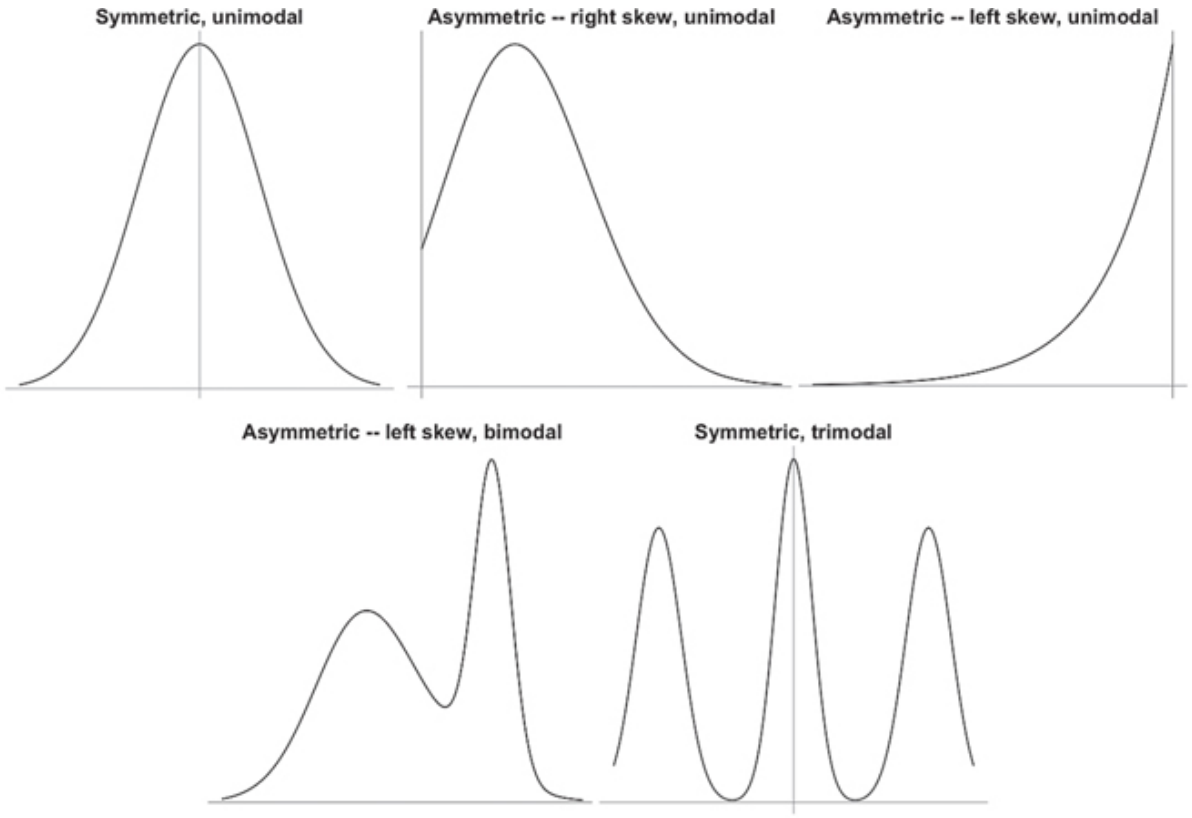


Figure 15-8: General examples of the terms used to describe probability distributions. The top three images are unimodal and highlight the notion of symmetry versus asymmetric skew; the bottom two images emphasize the idea of modality.

You can use these descriptors when discussing the probability distributions for the gambling game and picnic temperature examples. The mass function for X, on the left in Figure 15-4, is unimodal and asymmetric; it appears to have a mild but noticeable right skew. The density function for W, given in Figure 15-5, is also unimodal, though as noted earlier it is perfectly symmetric.

Exercise 15.2

a. For each of the following definitions, identify whether it is best described as a random variable or as a *realization* of a

random variable. Furthermore, identify whether each statement describes a continuous or a discrete quantity.

i. The number of coffees x made by your local shop on June 3, 2016

ii. The number of coffees X made by your local shop on any given day

iii. Y, whether or not it rains tomorrow

iv. Z, the amount of rain that falls tomorrow

v. How many crumbs k are on your desk right now

vi. Total collective weight W of the crumbs on your desk at any specified time

b. Suppose you construct the following table providing probabilities associated with the random variable S, the total stars given to any movie in a particular genre by a certain critic:

s	**1**	**2**	**3**	**4**	**5**
Pr(S = s)	0.10	0.13	0.21	???	0.15

i. Assuming this table describes the complete set of outcomes, evaluate the missing probability Pr(S = 4).

ii. Obtain the cumulative probabilities.

iii. What is the mean of S, the expected number of stars this critic will award any given movie in this genre?

iv. What is the standard deviation of S?

v. What is the probability that any given movie in this genre will be given at least three stars?

vi. Visualize, and briefly comment on the appearance of, the probability mass function.

c. Return to the picnic temperature example based on the random variable W defined in Section 15.2.3.

i. Write an R function as per Equation (15.5) to return $f(w)$ for any numeric vector of values supplied as w. Try to avoid using a loop in favor of vector-oriented operations (you can do this with the help of the vector-valued form of the AND logical operator, `&`, introduced in Section 4.1.3).

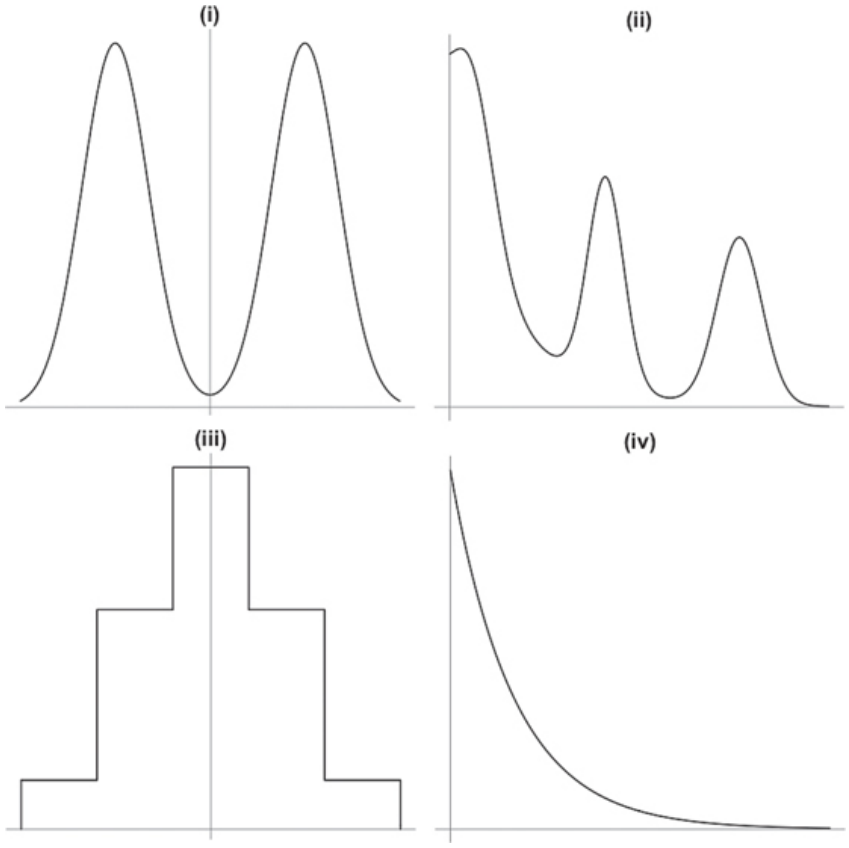
ii. Write an R function as per Equation (15.6) to return $F(w)$ for any numeric vector of values supplied as w. Again, try to avoid using a loop, either explicit or implicit.

iii. Use your functions from (i) and (ii) to confirm the results from the text, in other words, that $f(55.2) = 0.02432$ and that $F(55.2) = 0.184832$.

iv. Make use of your function for $F(w)$ to compute $\Pr(W > 60)$.
Hint: Note that because the total area underneath $f(w)$ is 1, $\Pr(W > 60) = 1 - \Pr(W \leq 60)$.

v. Find $\Pr(60.3 < W < 76.89)$.
Hint: Note that for any two values a and b, where $a < b$, $\Pr(a < W < b) = \Pr(W < b) - \Pr(W < a)$.

vi. Assume each of the following plots labeled (i)–(iv) shows the general appearance of a probability distribution. Use terminology from Section 15.2.4 to describe the shape of each.



Important Code in This Chapter

Function/operator	Brief description	First occurrence
`cumsum`	Calculate cumulative sum	Section 15.2.2, p. 378
`polygon`	Add shaded polygon to plot	Section 15.2.3, p. 383

16
COMMON PROBABILITY DISTRIBUTIONS



In this chapter, you'll look at a number of univariate probability distributions that exist to deal with commonly occurring random phenomena in statistical modeling. These distributions follow the same basic rules as the illustrative examples presented in Chapter 15, and their properties are well understood and documented. In fact, they are so ubiquitous that most statistical software packages have corresponding built-in functionality for their evaluation, and R is no exception. Several of these distributions represent an essential ingredient in traditional statistical hypothesis testing, which you'll explore in Chapters 17 and 18.

Just like the random variables they model, the common distributions you'll examine here are broadly categorized as either discrete or continuous. Each distribution has four core R functions tied to it: a `d`-function, providing specific mass or density function values; a `p`-function, providing cumulative distribution probabilities; a `q`-function, providing quantiles; and an `r`-function, providing a facility to generate random realizations.

16.1 Common Probability Mass Functions

Let's start by looking at definitions and examples of some common probability mass functions for discrete random variables. Section 16.2 will cover common continuous distributions.

16.1.1 Bernoulli Distribution

The *Bernoulli* distribution is the probability distribution of a discrete random variable that has only two possible outcomes, such as success or failure. We refer to this type of variable as *binary* or *dichotomous*.

Let's say you've defined a binary random variable X for the success or failure of an event, where $X = 0$ is failure, $X = 1$ is success, and p is the known probability of success. Table 16-1 shows the probability mass function for X.

Table 16-1: The Bernoulli Probability Mass Function

x	0	1
$\Pr(X = x)$	$1 - p$	p

From Section 15.2.2 you know that the probabilities associated with all possible outcomes must sum to 1. Therefore, if the probability of success is p for a binary random variable, the only other alternative outcome, failure, must occur with probability $1 - p$.

In mathematical terms, for a discrete random variable $X = x$, the Bernoulli mass function f is

$$f(x) = p^x(1-p)^{1-x}; \quad x \in \{0, 1\} \qquad (16.1)$$

where p is a parameter of the distribution. The notation

$$X \sim \text{BERN}(p)$$

is often used to indicate that "X follows a Bernoulli distribution with parameter p."

The following are the key points to remember:

- X is dichotomous and can take only the values 1 ("success") or 0 ("failure").
- p is interpreted as "the probability of success," and thus $0 \leq p \leq 1$.

The mean and variance are defined as follows, respectively:

$$\mu_X = p \qquad \text{and} \qquad \sigma_X^2 = p\,(1-p)$$

Say you use the common example of rolling a die, with success defined as getting a 4, and you roll once. You therefore have a binary random variable X that can be modeled using the Bernoulli distribution, with the probability of success $p = \frac{1}{6}$. For this example, X~BERN $\left(\frac{1}{6}\right)$. Using (16.1), you can determine that

$$\Pr(\text{rolling a 4}) = \Pr(X = 1) = f(1) = \left(\tfrac{1}{6}\right)^1 \left(\tfrac{5}{6}\right)^0 = \tfrac{1}{6}$$

and, in much the same way, that $f(0) = \frac{5}{6}$. Furthermore, you'd have $\mu_X = \frac{1}{6}$ and $\sigma_X^2 = \frac{1}{6} \times \frac{5}{6} = \frac{5}{36}$.

16.1.2 Binomial Distribution

The *binomial distribution* is the distribution of successes in n number of trials involving binary discrete random variables. The role of the Bernoulli distribution is typically as a "building block" for more complicated distributions, like the binomial, which are used in more realistic scenarios.

Define a random variable $X = \sum_{i=1}^{n} Y_i$, where $Y_1, Y_2, \ldots, Y_n$ are each Bernoulli random variables corresponding to the same event. The new random variable X, a sum of Bernoulli random variables, now describes *the number of successes in* n *trials* of the defined action. Provided

that certain reasonable assumptions are satisfied, the probability distribution that describes this success count is the binomial distribution.

For a discrete, binomially distributed random variable X and a realization x, the binomial probabilities are found via the probability mass function f defined as

$$f(x) = \Pr(X = x) = \binom{n}{x} p^x (1-p)^{n-x}; \quad x \in \{0, 1, \ldots, n\} \qquad (16.2)$$

where:

$$\binom{n}{x} = \frac{n!}{x!(n-x)!} \qquad (16.3)$$

is known as the *binomial coefficient*. (Recall the integer factorial operator !, first discussed in Exercise 10.4, page 236.) This coefficient, also referred to as a *combination*, accounts for all different orders in which you might observe x successes throughout n trials.

The parameters of the binomial distribution are n and p, and the notation

$$X \sim \text{BIN}(n, p)$$

is often used to indicate that X follows a binomial distribution for n trials with parameter p.

The following are the key points to remember:

- X can take only the integer values 0, 1, . . . , n, which represents the total number of observed successes.
- You interpret p as "the probability of success at each trial." Therefore, $0 \leq p \leq 1$, and $n > 0$ is an integer that represents "the number of trials."
- Each of the n trials is a Bernoulli success/failure random variable, the trials are independent (in other words, the outcome of one

doesn't affect the outcome of any other), and p is constant for all trials.

The mean and variance of a binomially distributed random variable X are defined as follows:

$$\mu_X = np \qquad \text{and} \qquad \sigma_X^2 = np\,(1-p)$$

Counting the number of successes for repeated trials of a binary-valued test is one of the common random phenomena mentioned at the start of this section. Consider the specific situation in which there's only one "trial," that is, $n = 1$. Examining Equations (16.2) and (16.3), it should become clear that (16.2) simplifies to (16.1). In other words, the Bernoulli distribution is just a special case of the binomial. This makes sense with respect to the definition of a binomial random variable as a sum of n Bernoulli random variables. As such, R provides functionality for the Bernoulli distribution through the built-in functions for the binomial distribution.

To illustrate this, I'll return to the example of rolling a die with success defined as getting a 4. If you perform eight separate die rolls, what is the probability of observing exactly five successes (five 4s) in total? Well, you'd have $X \sim \text{BIN}\left(8, \frac{1}{6}\right)$, and you can work through this probability mathematically using Equation (16.2).

$$\begin{aligned} \Pr(\text{get five 4s}) = \Pr(X = 5) &= f(5) \\ &= \tfrac{8!}{5!\times 3!}\left(\tfrac{1}{6}\right)^5\left(\tfrac{5}{6}\right)^3 \\ &= 0.004 \qquad \text{(rounded to 3 d.p.)} \end{aligned}$$

The result tells you there is approximately a 0.4 percent chance that you'll observe exactly five 4s in eight rolls of the die. This makes sense; it's far more probable that you might observe zero to two 4s in eight rolls of a die.

Fortunately, R functions will handle the arithmetic in these situations. The built-in functions `dbinom`, `pbinom`, `qbinom`, and `rbinom` are all

relevant to the binomial and Bernoulli distributions and are summarized in one help file indexed by each of these function names. We can describe them as follows:

- `dbinom` directly provides the mass function probabilities $\Pr(X = x)$ for any valid x, that is, $0 \le x \le n$.
- `pbinom` provides the cumulative probability distribution: Given a valid x, it yields $\Pr(X \le x)$.
- `qbinom` provides the *inverse* cumulative probability distribution (also known as the *quantile function* of the distribution): Given a valid probability $0 \le p \le 1$, it yields the value of x that satisfies $\Pr(X \le x) = p$.
- `rbinom` generates any number of realizations of X given a specific binomial distribution.

The dbinom Function

With this knowledge, you can use R to confirm the result of $\Pr(X = 5)$ for the die-roll example described a moment ago:

```
R> dbinom(x=5,size=8,prob=1/6)
[1] 0.004167619
```

To the `dbinom` function, you provide the specific value of interest as `x`; the total number of trials, n, as `size`; and the probability of success at each trial, p, as `prob`. True to R, a vector argument is possible for `x`. If you want the full probability mass function table for X for this example, you can supply the vector `0:8` to `x`:

```
R> X.prob <- dbinom(x=0:8,size=8,prob=1/6)
R> X.prob
[1] 2.325680e-01 3.721089e-01 2.604762e-01 1.041905e-01 2.604762e-02
[6] 4.167619e-03 4.167619e-04 2.381497e-05 5.953742e-07
```

You can confirm these sum to 1:

```
R> sum(X.prob)
[1] 1
```

The resulting vector of probabilities, which corresponds to the specific outcomes $x \in \{0, 1, \ldots, 8\}$, is returned using e-notation (refer to Section 2.1.3). You can tidy this up by rounding the results using the `round` function introduced in Section 13.2.3. Rounding to three decimal places, the results are easier to read:

```
R> round(X.prob,3)
[1] 0.233 0.372 0.260 0.104 0.026 0.004 0.000 0.000 0.000
```

The achievement of one success in eight trials has the highest probability, at approximately 0.372. Furthermore, the mean (expected value) and variance of X in this example are $\mu_X = np = 8 \times \frac{1}{6} = 8/6$ and $\sigma_X^2 = np\,(1-p) = 8 \times \frac{1}{6} \times \frac{5}{6}$:

```
R> 8/6
[1] 1.333333
R> 8*(1/6)*(5/6)
[1] 1.111111
```

You can plot the corresponding probability mass function in the same way as for the example in Section 15.2.2:

```
R> barplot(X.prob,names.arg=0:8,space=0,xlab="x",ylab="Pr(X = x)")
```

This line produces Figure 16-1.

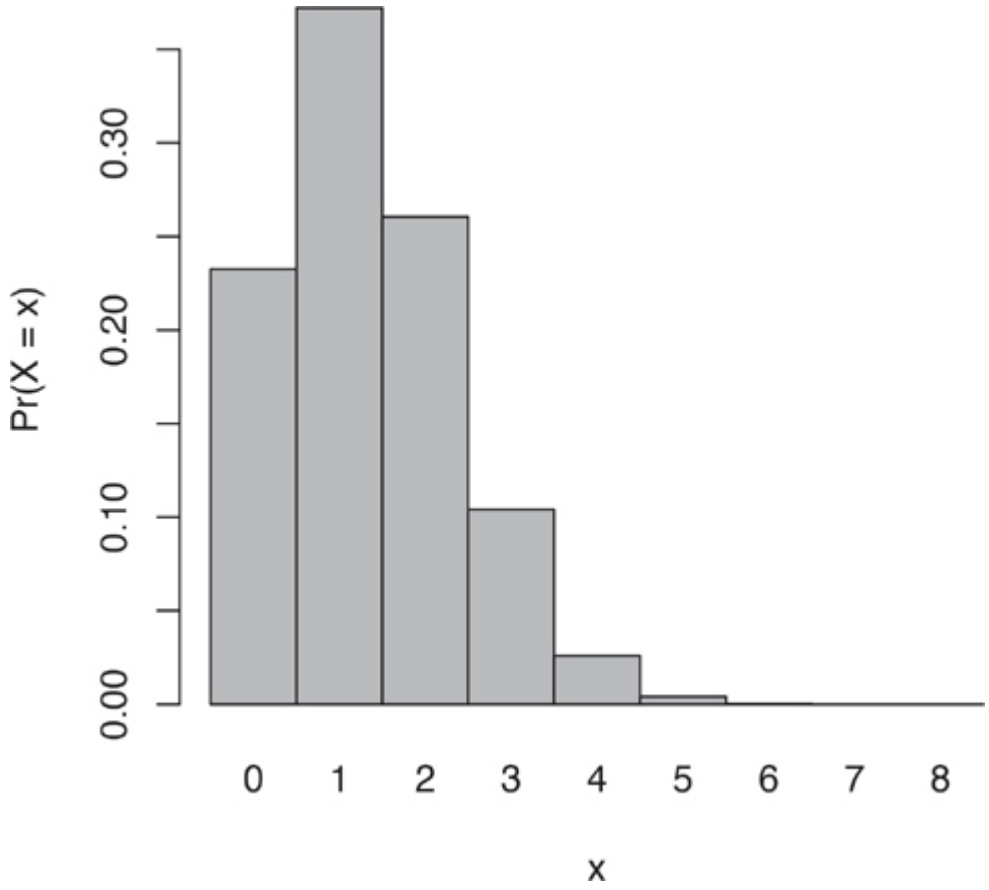


Figure 16-1: The probability mass function associated with the binomial distribution of the die-rolling example; $X \sim BIN\left(8, \frac{1}{6}\right)$

The pbinom Function

The other R functions for the binomial distribution work in much the same way. The first argument is always the value (or values) of interest; n is supplied as `size` and p as `prob`. For example, to find the probability that you observe three or fewer 4s, $\Pr(X \leq 3)$, you either sum the relevant individual entries from `dbinom` as earlier or use `pbinom`:

```
R> sum(dbinom(x=0:3,size=8,prob=1/6))
[1] 0.9693436
R> pbinom(q=3,size=8,prob=1/6)
[1] 0.9693436
```

Note that the pivotal argument to `pbinom` is tagged `q`, not `x`; this is because, in a cumulative sense, you are searching for a probability based

on a quantile. You can use the cumulative distribution results from `pbinom` in the same way to search for "upper-tail" probabilities (probabilities to the right of a given value) since you know that the total probability mass is always 1. To find the probability that you observe *at least* three 4s in eight rolls of the die, $\Pr(X \geq 3)$ (which is equivalent to $\Pr(X > 2)$ in the context of this discrete random variable), the following finds the correct result because it's the complement of $\Pr(X \leq 2)$ that you're looking for:

```
R> 1-pbinom(q=2,size=8,prob=1/6)
[1] 0.1348469
```

The qbinom Function

Less frequently used is the `qbinom` function, which is the inverse of `pbinom`. Where `pbinom` provides a cumulative probability when given a quantile value `q`, the function `qbinom` provides a quantile value when given a cumulative probability `p`. The discrete nature of a binomial random variable means `qbinom` will return the nearest value of x below which `p` lies. For example, note that

```
R> qbinom(p=0.95,size=8,prob=1/6)
[1] 3
```

provides 3 as the quantile value, even though you know from earlier that the exact probability that lies at or below 3, $\Pr(X \leq 3)$, is 0.9693436. You'll look at `p`- and `q`-functions more when dealing with continuous probability distributions; see Section 16.2.

The rbinom Function

Lastly, you can use the `rbinom` function to generate random realizations of a binomially distributed variable. Again, going with the BIN(8, $\frac{1}{6}$) distribution, note the following:

```
R> rbinom(n=1,size=8,prob=1/6)
[1] 0
R> rbinom(n=1,size=8,prob=1/6)
```

```
[1] 2
R> rbinom(n=1,size=8,prob=1/6)
[1] 2
R> rbinom(n=3,size=8,prob=1/6)
[1] 2 1 1
```

The initial argument `n` doesn't refer to the number of trials. The number of trials is still provided to `size` with p given to `prob`. Here, `n` requests the number of realizations you want to generate for the random variable $X \sim \text{BIN}\left(8, \frac{1}{6}\right)$. The first three lines each request a single realization; in the first eight rolls, you observe zero successes (4s), and in the second and third sets of eight rolls, you observe two 4s in each. The fourth line highlights the fact that multiple realizations of X are easily obtained and stored as a vector by increasing `n`. As these are *randomly generated realizations*, if you run these lines now, you'll probably observe some different values.

Though not used often in standard statistical testing methods, the `r`-functions for probability distributions, either discrete or continuous, play an important role when it comes to simulation and various advanced numeric algorithms in computational statistics.

Exercise 16.1

A forested nature reserve has 13 bird-viewing platforms scattered throughout a large block of land. Ecologists claim that at any point in time, there is a 75 percent chance of seeing birds at any given platform. Suppose you walk through the reserve and visit every platform. If you assume that all relevant conditions are satisfied, let X be a binomial random variable representing the total number of platforms at which you see birds.

a. Visualize the probability mass function of the binomial distribution of interest.

b. What is the probability you see birds at all sites; Pr(X = 13)?

c. What is the probability that you see birds at more than 9 platforms; Pr($X > 9$)?

d. What is the probability of you seeing birds at between 8 and 11 platforms (inclusive); Pr($8 \leq X \leq 11$)? Confirm your answer by using only the `d`-function and then again using only the `p`-function.

e. Say that, before your visit, you decide that if you see birds at fewer than 9 sites, you'll make a scene and demand your entry fee back. What's the probability that you embarrass yourself in this way?

f. Simulate realizations of X that represent 10 different visits to the reserve; store your resulting vector as an object.

g. Compute the mean and standard deviation of the distribution of interest.

16.1.3 Poisson Distribution

In this section, you'll use the *Poisson* distribution to model a slightly more general, but just as important, discrete random variable: a *count*. For example, the variable of interest might be the number of seismic tremors a certain station detects in a given year or the number of imperfections found on square-foot pieces of sheet metal coming off a factory production line.

Importantly, we assume the events or items being counted manifest independently of one another. In mathematical terms, for a Poisson-distributed discrete random variable X and a realization x, the probability mass function f is given as follows, where λ_p is a parameter of the distribution (this will be explained further momentarily):

$$f(x) = \frac{\lambda_p^x \exp(-\lambda_p)}{x!}; \quad x \in \{0, 1, \ldots\} \qquad (16.4)$$

We use the notation

$$X \sim \text{POIS}(\lambda_p)$$

to indicate that "X follows a Poisson distribution with parameter λ_p." The following are the keys points to remember:

- The entities, features, or events being counted occur independently in a well-defined interval at a constant rate.
- X can take only non-negative integers: 0, 1, ...
- You interpret λ_p as the "mean number of occurrences"; it must therefore be a finite and strictly positive value, that is, $0 < \lambda_p < \infty$.

The mean and variance are both equal to the parameter λ_p:

$$\mu_X = \lambda_p \qquad \text{and} \qquad \sigma_X^2 = \lambda_p$$

Like the binomial random variable, the values taken by a Poisson random variable are discrete, non-negative integers. Unlike the binomial, however, there's typically no upper limit on a Poisson count. While this implies that an "infinite count" is allowed to occur, it's a distinct feature of the Poisson distribution that the probability mass associated with some value x goes to zero as x itself goes to infinity.

As noted in Equation (16.4), any Poisson distribution depends upon the specification of a single parameter, denoted here with λ_p. This parameter describes the mean number of occurrences, which impacts the overall shape of the mass function, as Figure 16-2 shows.

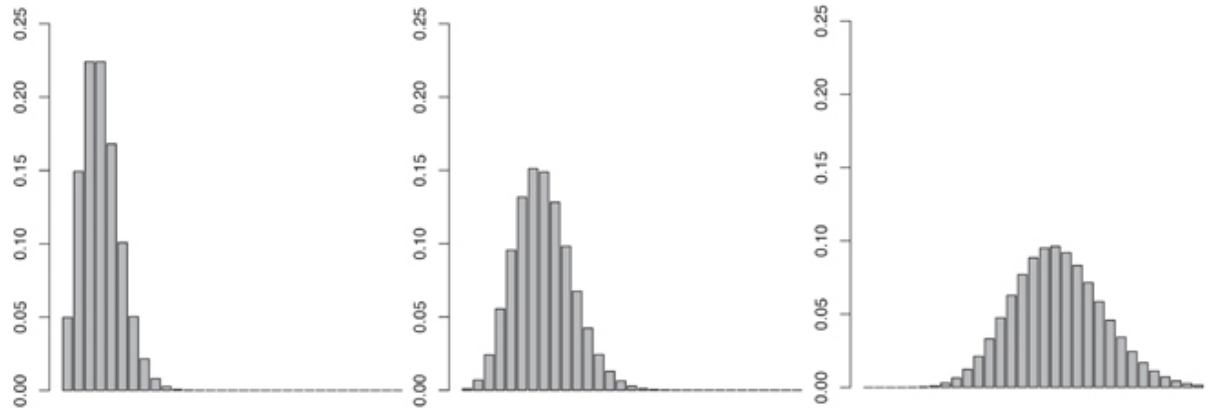


Figure 16-2: Three examples of the Poisson probability mass function, plotted for 0 ≤ x ≤ 30. The "expected count" parameter λ0.3emp is altered from 3.00 (left) to 6.89 (middle) and to 17.20 (right).

Again, it's worth noting that the total probability mass over all possible outcomes is 1, no matter what the value of λ_p is and regardless of the fact that possible outcomes can, technically, range from 0 to infinity.

For illustration of the relevant R functions, let's consider the example of blemishes on 1-foot-square sheets of metal coming off a production line, mentioned in the opening of this section. Suppose you're told that the number of blemishes found, X, is thought to follow a Poisson distribution with $\lambda_p = 3.22$, as in $X \sim \text{POIS}(3.22)$. In other words, you'd expect to see an average of 3.22 blemishes on your 1-foot sheets.

The dpois and ppois Functions

The R `dpois` function provides the individual Poisson mass function probabilities $\Pr(X = x)$ for the Poisson distribution. The `ppois` function provides the left cumulative probabilities, as in $\Pr(X \leq x)$. Consider the following lines of code:

```
R> dpois(x=3,lambda=3.22)
[1] 0.2223249
R> dpois(x=0,lambda=3.22)
[1] 0.03995506
```

```
R> round(dpois(0:10,3.22),3)
[1] 0.040 0.129 0.207 0.222 0.179 0.115 0.062 0.028 0.011 0.004 0.001
```

The first call finds that $\Pr(X = 3) = 0.22$ (to 2 d.p.); in other words, the probability that you observe exactly three blemishes on a randomly selected piece of sheet metal is equal to about 0.22. The second call indicates a less than 4 percent chance that the piece is flawless. The third line returns a rounded version of the relevant mass function for the values $0 \leq x \leq 10$. You can confirm the first result like this:

```
R> (3.22^3*exp(-3.22))/prod(3:1)
[1] 0.2223249
```

The following creates a visualization of the mass function:

```
R> barplot(dpois(x=0:10,lambda=3.22),ylim=c(0,0.25),space=0,names.arg=0:10,
           ylab="Pr(X=x)",xlab="x")
```

This is shown on the left of Figure 16-3.

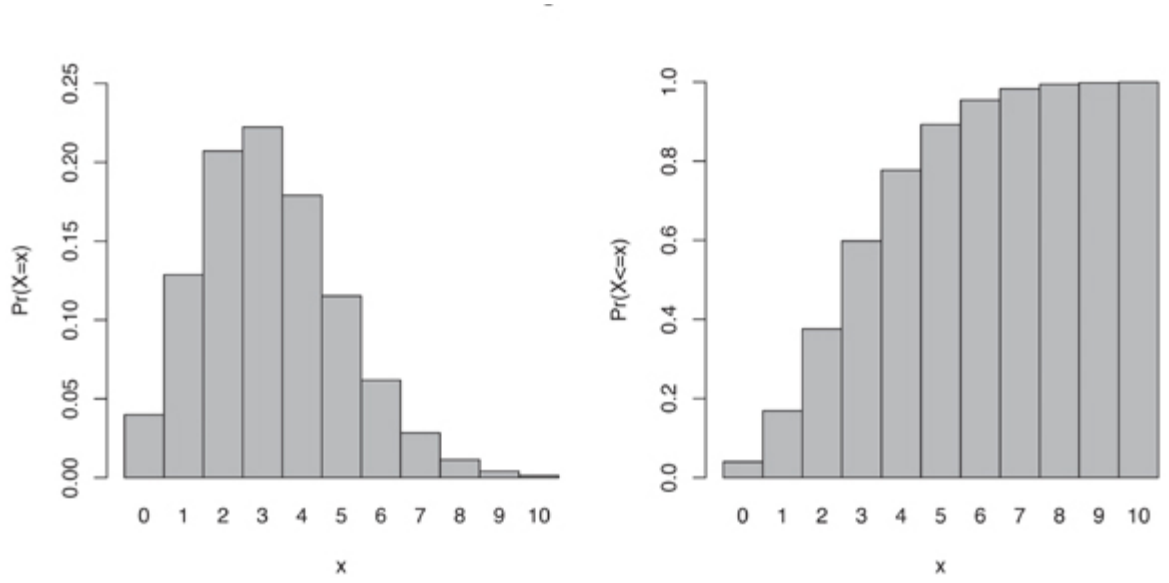


Figure 16-3: The Poisson probability mass function (left) and cumulative distribution function (right) for $\lambda_p = 3.22$ plotted for the integers $0 \leq x \leq 10$, with reference to the sheet metal example

To calculate cumulative results, use `ppois`:

```
R> ppois(q=2,lambda=3.22)
[1] 0.3757454
R> 1-ppois(q=5,lambda=3.22)
[1] 0.1077005
```

These lines find that the probability you observe at most two imperfections, $\Pr(X \leq 2)$, is about 0.38, and the probability you observe strictly more than five blemishes, $\Pr(X \leq 6)$, is roughly 0.11.

Create a visualization of the cumulative mass function, as given on the right of Figure 16-3, with the following:

```
R> barplot(ppois(q=0:10,lambda=3.22),ylim=0:1,space=0,names.arg=0:10,
           ylab="Pr(X<=x)",xlab="x")
```

The qpois Function

The `q`-function for the Poisson distribution, `qpois`, provides the inverse of `ppois`, just as `qbinom` provides the inverse of `pbinom` in Section 16.1.2.

The rpois Function

To produce random variates, use `rpois`; supply the number of realizations you want as `n` and supply the all-important parameter as `lambda`. You can imagine

```
R> rpois(n=15,lambda=3.22)
[1] 0 2 9 1 3 1 9 3 4 3 2 2 6 3 5
```

as selecting fifteen 1-foot-square metal sheets from the production line at random and counting the number of blemishes on each. (Note again that this is random generation; your specific results are likely to vary.)

Exercise 16.2

Every Saturday, at the same time, a researcher stands by the side of a road and tallies the number of cars going by within a 120-minute window. Based on previous knowledge, she believes that the mean number of cars going by during this time is exactly 107. Let X represent the appropriate Poisson random variable of the number of cars passing her position in each Saturday session.

a. What is the probability that more than 100 cars pass her on any given Saturday; $\Pr(X > 100)$?

b. Determine the probability that no cars pass; $\Pr(X = 0)$.

c. Plot the relevant Poisson mass function over the values in $60 \leq x \leq 150$.

d. Simulate 260 results from this distribution (about five years of weekly Saturday monitoring sessions). Plot the simulated results using `hist`; use `xlim` to set the horizontal limits from 60 to 150. Compare your histogram to the shape of your mass function from (c).

16.1.4 Other Mass Functions

There are many other well-defined probability mass functions in R's built-in suite of statistical calculations. All model a discrete random variable in a certain way under certain conditions and are defined with at least one parameter, and most are represented by their own set of `d-`, `p-`, `q-`, and `r`-functions. Here are a few more:

- The *geometric* distribution counts the number of failures before a success is recorded and is dependent on a "probability of success parameter" `prob`. Its functions are `dgeom`, `pgeom`, `qgeom`, and `rgeom`.
- The *negative binomial* distribution is a generalization of the geometric distribution, dependent upon parameters `size` (number of trials) and `prob`. Its functions are `dnbinom`, `pnbinom`, `qnbinom`, and `rnbinom`.

- We use the *hypergeometric* distribution to model sampling without replacement (in other words, a "success" can change the probabilities associated with further successes), dependent upon parameters `m`, `n`, and `k` describing the nature of sampled items. Its functions are `dhyper`, `phyper`, `qhyper`, and `rhyper`.
- The *multinomial* distribution is a generalization of the binomial, where a success can occur in one of multiple categories at each trial, with parameters `size` and `prob` (this time, `prob` must be a vector of probabilities corresponding to the multiple categories). Its built-in functions are limited to `dmultinom` and `rmultinom`.

As noted earlier, some familiar probability distributions are just simplifications or special cases of functions that describe a more general class of distributions.

16.2 Common Probability Density Functions

Continuous random variables are tied to probability density functions. There are a number of common density functions we encounter in practice. In this section, you'll learn about some of these and R's accompanying `d-`, `p-`, `q-`, and `r-`functions.

16.2.1 Uniform

The *uniform* distribution is a simple density function that describes a continuous random variable whose interval of possible values offers no fluctuations in probability.

For a continuous random variable $a < X < b$, the uniform density function f is

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a < x < b; \\ 0 & \text{otherwise} \end{cases} \qquad (16.5)$$

where a and b are parameters of the distribution defining the limits of the possible values X can take. We use the notation

$$X \sim \text{UNIF}(a, b)$$

to indicate that "X follows a uniform distribution on the interval defined by the limits a and b."

The following are the key points to remember:

- X can take any value in the interval bounded by a and b.
- a and b can be any values, provided that $a < b$, and they represent the lower and upper limits, respectively, of the interval of possible values.

The mean and variance are as follows:

$$\mu_X = \frac{a+b}{2} \qquad \text{and} \qquad \sigma_X^2 = \frac{(b-a)^2}{12}$$

Especially for the more complicated densities in this section, it's useful to visualize the functions in order to understand the probabilistic structure that underpins the corresponding random variable. For the uniform distribution, given by Equation (16.5), Figure 16-4 shows two examples using two different sets of limits (I'll provide the code to produce these types of plots shortly).

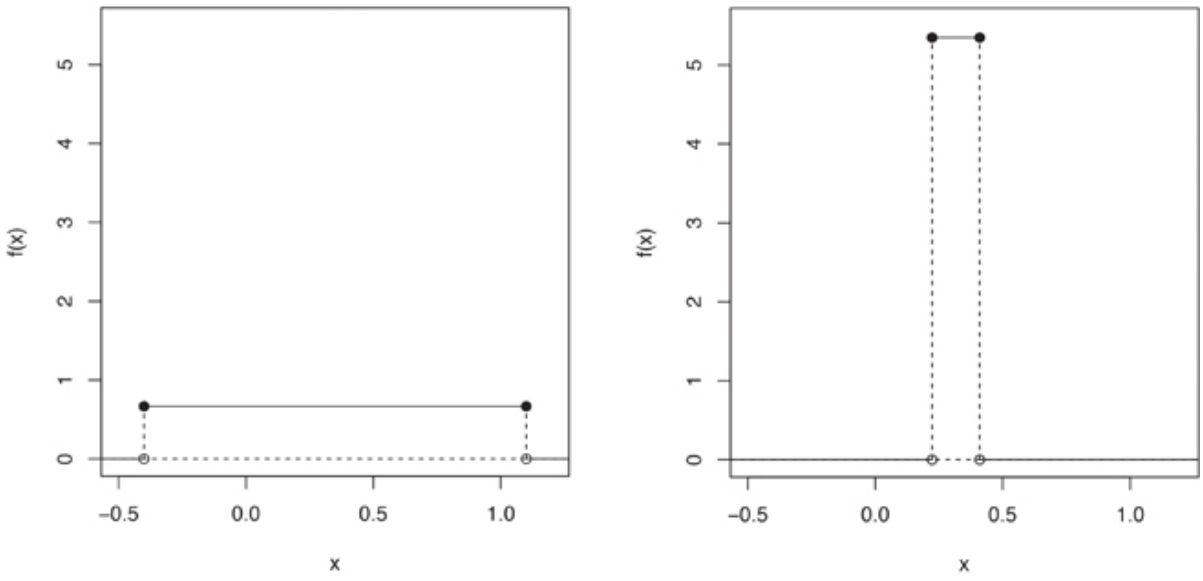


Figure 16-4: Two uniform distributions plotted on the same scale for comparability. Left: X *~ UNIF(–0.4,1.1); right:* X *~ UNIF(0.223,0.410). The total area underneath each density function is, as always, 1.*

For the left plot in Figure 16-4, you can use Equation (16.5) to confirm the exact height of the X ~ UNIF(–0.4, 1.1) density by hand: $1/\{1.1-(-0.4)\} = 1/1.5 = \frac{2}{3}$. For the plot on the right, based on X ~ UNIF(0.223, 0.410), you can use R to find that its height is roughly 5.35:

```
R> 1/(0.41-0.223)
[1] 5.347594
```

The dunif Function

You can use the built-in `d`-function for the uniform distribution, `dunif`, to return these heights for any value within the defined interval. The `dunif` command returns zero for values outside of the interval. The parameters of the distribution, *a* and *b*, are provided as the arguments `min` and `max`, respectively. For example, the line

```
R> dunif(x=c(-2,-0.33,0,0.5,1.05,1.2),min=-0.4,max=1.1)
[1] 0.0000000 0.6666667 0.6666667 0.6666667 0.6666667 0.0000000
```

evaluates the uniform density function of $X \sim \text{UNIF}(-0.4, 1.1)$ at the values given in the vector passed to `x`. Because the first and last values fall outside the bounds defined by `min` and `max`, they correspond to a density of zero. All others evaluate to the height value of $\frac{2}{3}$, as previously calculated.

As a second example, the line

```
R> dunif(x=c(0.3,0,0.41),min=0.223,max=0.41)
[1] 5.347594 0.000000 5.347594
```

confirms the correct density values for the $X \sim \text{UNIF}(0.223, 0.410)$ distribution, with the second value, 0, falling outside the defined interval.

This most recent example in particular should remind you that probability density functions for continuous random variables, unlike mass functions for discrete variables, *do not* directly provide probabilities, as mentioned in Section 15.2.3. In other words, the results just returned by `dunif` represent points on the density function itself and not any notion of chance attached to the specific values of x at which it was evaluated.

To calculate some probabilities based on the uniform density function, let's use an example of a faulty drill press. In a woodworker's shop, imagine there's a drill press that cannot keep to a constant alignment when in use; instead, it randomly hits the intended target at up to 0.4 cm to the left or 1.1 cm to the right. Let the random variable $X \sim \text{UNIF}(-0.4, 1.1)$ represent where the drill hits the material relative to the target at 0. Figure 16-5 replots the left image of Figure 16-4 on a more appropriate y-axis scale. You have three versions, each marking off a different area under the density function: from left to right, $\Pr(X \leq -0.21)$, $\Pr(-0.21 \leq X \leq 0.6)$, and $\Pr(X \geq 0.6)$.

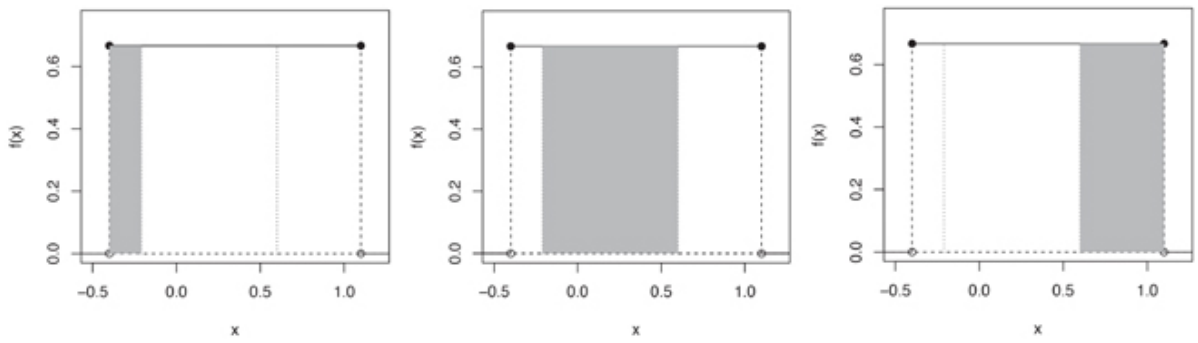


Figure 16-5: Three areas underneath the X ~ UNIF(–0.4,1.1) *density function for the drill press example. Left:* Pr(X ≤ –0.21)*; middle:* Pr(–0.21 ≤ X ≤ 0.6)*; right:* Pr(X ≤ 0.6)*.*

These plots are created using the coordinate-based plotting skills covered in Chapter 7. You plot the density itself with the following:

```
R> a1 <- -4/10
R> b1 <- 11/10
R> unif1 <- 1/(b1-a1)
R> plot(c(a1,b1),rep(unif1,2),type="o",pch=19,xlim=c(a1-1/10,b1+1/10),
       ylim=c(0,0.75),ylab="f(x)",xlab="x")
R> abline(h=0,lty=2)
R> segments(c(a1-2,b1+2,a1,b1),rep(0,4),rep(c(a1,b1),2),rep(c(0,unif1),each=2),
           lty=rep(1:2,each=2))
R> points(c(a1,b1),c(0,0))
```

You can use much of the same code to produce the plots in Figure 16-4 by modifying the `xlim` and `ylim` arguments to adjust the scale of the axes.

You add the vertical lines denoting $f(-0.21)$ and $f(0.6)$ in Figure 16-5 with another call to `segments`:

```
R> segments(c(-0.21,0.6),c(0,0),c(-0.21,0.6),rep(unif1,2),lty=3)
```

Finally, you can shade the areas with the `polygon` function (Section 15.2.3). For example, in the leftmost plot in Figure 16-5, use the previous plotting code followed by this:

```
R> polygon(rbind(c(a1,0),c(a1,unif1),c(-0.21,unif1),c(-0.21,0)),col="gray",
           border=NA)
```

As mentioned earlier, the three shaded areas in Figure 16-5 represent, from left to right, $\Pr(X < -0.21)$, $\Pr(-0.21 < X < 0.6)$, and $\Pr(X > 0.6)$, respectively. In terms of the drill press example, you can interpret these as the probability that the drill hits the target 0.21 cm to the left or more extreme (up to the limit of –0.4), the probability that the drill hits the target between 0.21 cm to the left and 0.6 cm to the right, and the probability that the drill hits the target 0.6 cm to the right or more (up to the limit of 1.1), respectively. (Remember from Section 15.2.3 that it makes no practical difference if you use ≤ or <, or ≤ or >, for probabilities associated with continuous random variables.)

The punif Function

We define probabilities associated with continuous random variables as areas under the function: *integrals*. Your study therefore focuses on the appropriate intervals of X rather than any specific value. The `p`-function for probability densities, just like the `p`-function for probability mass functions, provides the cumulative probability $\Pr(X \leq x)$. In the context of the uniform density, this means that given a specific value of x (supplied as a "quantile" argument `q`), `punif` will provide the left-directional area underneath the function from that specific value back to the lower limit.

Accessing `punif`, the line

```
R> punif(-0.21,min=a1,max=b1)
[1] 0.1266667
```

tells you that the leftmost area in Figure 16-5 represents a probability of about 0.127. The line

```
R> 1-punif(q=0.6,min=a1,max=b1)
[1] 0.3333333
```

tells you that $\Pr(X > 0.6) = \frac{1}{3}$. Find the final result of $\Pr(-0.21 < X < 0.6)$, giving a 54 percent chance, with

```
R> punif(q=0.6,min=a1,max=b1) - punif(q=-0.21,min=a1,max=b1)
[1] 0.54
```

since the first call provides the area under the density from 0.6 all the way left and the second call provides the area from –0.21 all the way left. Therefore, this difference will provide the desired middle area.

It's essential to be able to manipulate cumulative probability results like this when working with probability distributions in R; you might find it useful to sketch out the desired area before using `p`-functions, not unlike the images in Figure 16-5.

The qunif Function

The `q`-functions for densities tend to be used more often than those for mass functions because the continuous nature of the random variable means that a unique quantile value can be found for any unique and valid probability `p`.

The `qunif` function is the inverse of `punif`:

```
R> qunif(p=0.1266667,min=a1,max=b1)
[1] -0.21
R> qunif(p=1-1/3,min=a1,max=b1)
[1] 0.6
```

These lines confirm the values of X used earlier to get the lower- and uppertail probabilities $\Pr(X < -0.21)$ and $\Pr(X > 0.6)$, respectively. Any `q`-function expects a *cumulative* (also called *left-hand*) probability as its first argument, which is why you need to supply `1-1/3` in the second line to recover the quantile value of `0.6`. (The total area is 1. You want the area to the *right* of 0.6 to be $\frac{1}{3}$; thus, the area on the left that defines the same quantile must be $1 - \frac{1}{3}$.)

The runif Function

Lastly, to generate random realizations of a specific uniform distribution, use `runif`. Let's say the woodworker drills 10 separate holes

using the faulty press; you can simulate one instance of the position of each of these holes relative to their target with the following call:

```
R> runif(n=10,min=a1,max=b1)
[1] -0.2429272 -0.1226586  0.9318365  0.4829028  0.5963365
[6]  0.2009347  0.3073956 -0.1416678  0.5551469  0.4033372
```

Again, note that your values will differ when executing functions that generate random values.

Exercise 16.3

You visit a national park and are informed that the height of a certain species of tree found in the forest, represented by the random variable X, is uniformly distributed between 3 and 70 feet.

a. What is the probability you encounter a tree shorter than $5\frac{1}{2}$ feet; $\Pr(X < 5.5)$?

b. For this probability density function, what is the height h that marks the cutoff point of the tallest 15 percent of trees; $0.15 = \Pr(X > h)$?

c. Evaluate the mean and standard deviation of the tree height distribution.

d. Using the results in (c), confirm that the chance that you encounter a tree with a height within half a standard deviation (that is, below or above) of the mean height is roughly 28.9 percent.

e. At what constant value is the density function itself? Show it in a plot such as those in Figure 16-4.

f. Simulate 10 observed tree heights. Based on these data, use `quantile` (refer to Section 13.2.4) to estimate the answer you

arrived at in (b). Repeat your simulation, this time generating 1,000 variates, and estimate (b) again. Do this a handful of times, taking a mental note of your two estimates each time. Overall, what do you notice of your two estimates (one based on 10 variates at a time and the other based on 1,000) with respect to the "true" value in (b)?

16.2.2 Normal

The *normal distribution* is one of the most well-known and commonly applied probability distributions in modeling continuous random variables. Characterized by a distinctive "bell-shaped" curve, it's also referred to as the *Gaussian* distribution.

For a continuous random variable $-\infty < X < \infty$, the normal density function f is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \qquad (16.6)$$

where μ and σ are parameters of the distribution, π is the familiar geometric value 3.1415] . . . , and $\exp\{\cdot\}$ is the exponential function (Section 2.1.2). We use the notation

$$X \sim \mathrm{N}(\mu, \sigma)$$

to indicate that "X follows a normal distribution with mean μ and standard deviation σ."

The following are the key points to remember:

- Theoretically, X can take any value from $-\infty$ to ∞.
- As hinted at earlier, the parameters μ and σ directly describe the mean and the standard deviation of the distribution, with the square of the latter, σ^2, being the variance.

- In practice, the mean parameter is finite $-\infty < \mu < \infty$, and the standard deviation parameter is strictly positive and finite $0 < \sigma < \infty$.
- If you have a random variable $X \sim N(\mu, \sigma)$, then you can create a new random variable $Z = (X - \mu)/\sigma$, which means $Z \sim N(0, 1)$. We call this *standardization* of X.

The two parameters noted earlier fully define a particular normal distribution. Normal distributions are always perfectly symmetric, unimodal, and centered on the mean μ. The degree of "spread" of the curve is defined using the standard deviation σ.

The left plot of Figure 16-6 provides the density functions for four specific normal distributions. Altering the mean results in a translation, where the center of the distribution is simply shifted to sit on the specific value of μ. A smaller standard deviation reduces the spread, resulting in a taller, skinnier appearance of the density. Increasing σ flattens the density out around the mean.

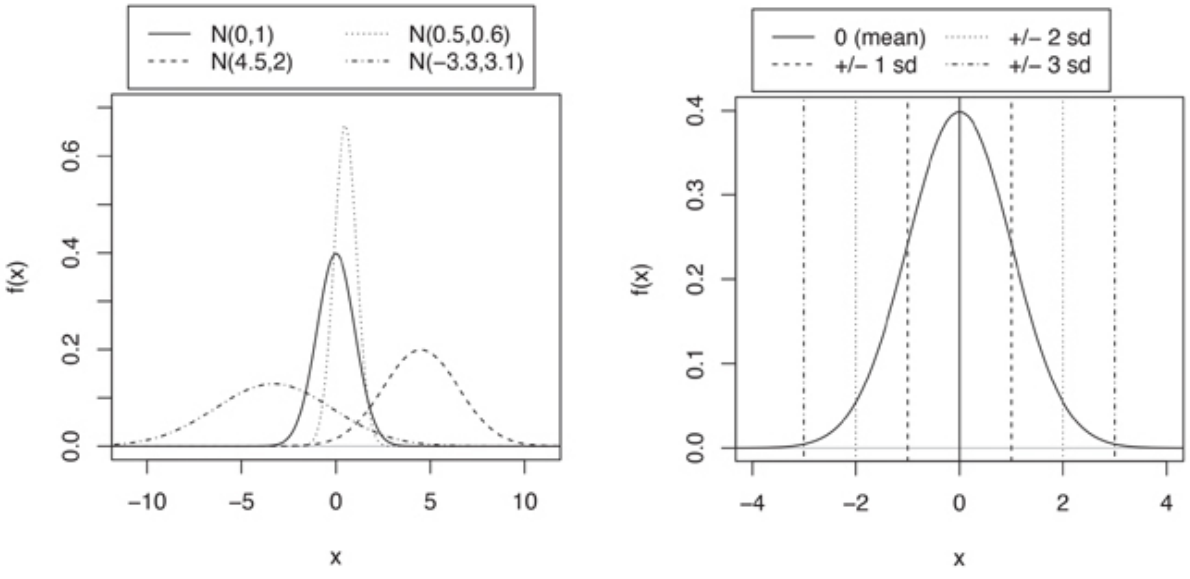


Figure 16-6: Illustrating the normal distribution: four different instances of the density achieved by varying the mean μ and standard deviation σ (left) and the "standard normal" distribution, N(0,1), marking off the mean ±1σ, ±2σ, and ±3σ (right)

The right plot of Figure 16-6 zooms in on the N(0, 1) distribution when you have a normal density centered on $\mu = 0$ and with a standard deviation of $\sigma = 1$. This distribution, known as the *standard normal*, is frequently used as a reference with which to compare different normally distributed random variables with one another on the same scale of probabilities. It's common practice to rescale, or *standardize*, some variable $X \sim N(\mu_X, \sigma_X)$ to a new variable Z such that $Z \sim N(0, 1)$ (you'll see this in Chapter 18). Vertical lines in the plot show plus or minus one, two, and three times the standard deviation away from the mean of zero. This serves to highlight the fact that for *any* normal distribution, a probability of exactly 0.5 lies either above or below the mean. Furthermore, note that there's a probability of approximately 0.683 of a value falling within one standard deviation of the mean, approximately 0.954 probability under the curve from -2σ to $+2\sigma$, and approximately 0.997 probability between -3σ and $+3\sigma$.

NOTE

The mathematical definition of the normal density means that as you move further away from the mean, the value of the density function itself will approach zero. In fact, any normal density function never actually touches the horizontal line at 0; it just gets closer and closer as you move to positive or negative infinity. This behavior is formally referred to as asymptotic; *in this case, you'd say that the normal distribution* f(x) *has a* horizontal asymptote *at* f(x) *= 0. In discussing probabilities as areas under the curve, you'd refer to the fact that "the total area under the curve from negative to positive infinity" is 1, in other words,* $\int_{-\infty}^{\infty} f(x)\,dx = 1$.

The dnorm Function

Being a probability density function, the `dnorm` function doesn't directly provide probabilities, but merely the value of the desired normal function curve $f(x)$ at any x. To plot a normal density, you'd therefore be able to use `seq` (refer to Section 2.3.2) to create a fine sequence of values

for x, evaluate the density at these values with `dnorm`, and then plot the result as a line. For example, to produce an image of the standard normal distribution curve similar to that in the bottom image of Figure 16-6, the following code will create the desired x values as `xvals`:

```
R> xvals <- seq(-4,4,length=50)
R> fx <- dnorm(xvals,mean=0,sd=1)
R> fx
 [1] 0.0001338302 0.0002537388 0.0004684284 0.0008420216 0.0014737603
 [6] 0.0025116210 0.0041677820 0.0067340995 0.0105944324 0.0162292891
[11] 0.0242072211 0.0351571786 0.0497172078 0.0684578227 0.0917831740
[16] 0.1198192782 0.1523049307 0.1885058641 0.2271744074 0.2665738719
[21] 0.3045786052 0.3388479358 0.3670573564 0.3871565916 0.3976152387
[26] 0.3976152387 0.3871565916 0.3670573564 0.3388479358 0.3045786052
[31] 0.2665738719 0.2271744074 0.1885058641 0.1523049307 0.1198192782
[36] 0.0917831740 0.0684578227 0.0497172078 0.0351571786 0.0242072211
[41] 0.0162292891 0.0105944324 0.0067340995 0.0041677820 0.0025116210
[46] 0.0014737603 0.0008420216 0.0004684284 0.0002537388 0.0001338302
```

Then `dnorm`, which includes specification of μ as `mean` and σ as `sigma`, produces the precise values of $f(x)$ at those `xvals`. Finally, a call such as `plot(xvals,fx,type="l")` achieves the bare bones of a density plot, which you can easily enhance by adding titles and using commands such as `abline` and `segments` to mark locations off (I'll produce another plot in a moment, so this basic one isn't shown here). Refer back to Chapter 7 for basic plotting skills.

Note that if you don't supply any values to `mean` and `sd`, the default behavior of R is to implement the standard normal distribution; the object `fx` shown earlier could have been equally created with an even shorter call using just `dnorm(xvals)`.

The pnorm Function

The `pnorm` function obtains cumulative probabilities under the specified normal density. As with `dnorm`, if no parameter values are supplied, R automatically sets `mean=0` and `sd=1`. In the same way you used `punif` in Section 16.2.1, you can find any probabilities you want from the normal

distribution by calculating differences between the appropriate results from `pnorm`. For example, I mentioned earlier that a probability of approximately 0.683 lies within one standard deviation of the mean. Assuming a standard normal distribution, you can confirm this as follows:

```
R> pnorm(q=1)-pnorm(q=-1)
[1] 0.6826895
```

Here, the first call to `pnorm` evaluates the area under the curve from positive 1 left (in other words, all the way to $-\infty$) and then finds the difference between that and the area from –1 left. The result reflects the proportion between the two dashed lines in the rightmost plot of Figure 16-6.

These kinds of probabilities will be the same for *any* normal distribution. Consider the distribution where $\mu = -3.42$ and $\sigma = 0.2$. The following provides the same value:

```
R> mu <- -3.42
R> sigma <- 0.2
R> mu.minus.1sig <- mu-sigma
R> mu.minus.1sig
[1] -3.62
R> mu.plus.1sig <- mu+sigma
R> mu.plus.1sig
[1] -3.22
R> pnorm(q=mu.plus.1sig,mean=mu,sd=sigma) -
   pnorm(q=mu.minus.1sig,mean=mu,sd=sigma)
[1] 0.6826895
```

It takes a little more work to specify the distribution of interest since it's not standard, but the principle is the same: plus and minus exactly one standard deviation away from the mean.

The symmetry of the normal distribution is also useful when calculating probabilities. Sticking with the N(3.42, 0.2) distribution, the probability you make an observation greater than $\mu + \sigma = -3.42 + 0.2 = -3.22$ (an uppertail or right-hand probability) is identical to the

probability of making an observation less than $\mu - \sigma = -3.42 - 0.2 = -3.62$ (a *lower-tail* probability):

```
R> 1-pnorm(mu.plus.1sig,mu,sigma)
[1] 0.1586553
R> pnorm(mu.minus.1sig,mu,sigma)
[1] 0.1586553
```

You can also evaluate these values by hand, given the result you computed earlier that says $\Pr(\mu - \sigma < X < \mu + \sigma) = 0.6826895$. The remaining probability *outside* of this middle area must be as follows:

```
R> 1-0.6826895
[1] 0.3173105
```

So, in each of the lower- and upper-tail areas marked off by $\mu - \sigma$ and $\mu + \sigma$, respectively, there must be a probability of the following:

```
R> 0.3173105/2
[1] 0.1586552
```

This corresponds to the same value found using the most recent call to `pnorm` (note that there can be some minor rounding errors in these types of calculations).

We can visualize all this in a plot such as the one in Figure 16-7. Start this plot with the following code:

```
R> xvals <- seq(-5,-2,length=300)
R> fx <- dnorm(xvals,mean=mu,sd=sigma)
R> plot(xvals,fx,type="l",xlim=c(-4.4,-2.5),main="N(-3.42,0.2) distribution",
        xlab="x",ylab="f(x)")
R> abline(h=0,col="gray")
R> abline(v=c(mu.plus.1sig,mu.minus.1sig),lty=3:2)
R> legend("topleft",legend=c("-3.62\n(mean - 1 sd)","\n-3.22\n(mean + 1 sd)"),
          lty=2:3,bty="n")
```

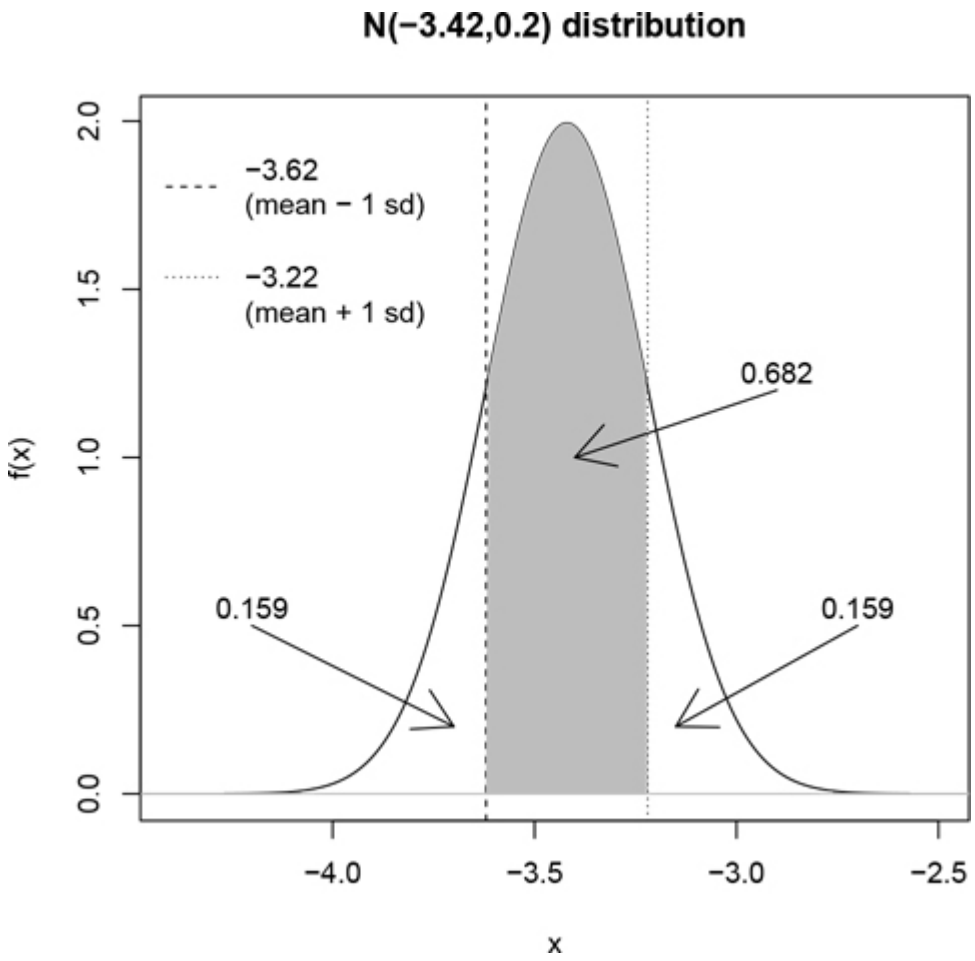


Figure 16-7: Illustrating the example in the text, which uses the symmetry of the normal distribution to point out features of probabilities under the curve. Note that the total area under the density is 1, which in conjunction with symmetry is useful for making calculations.

To add the shaded area between $\mu \pm \sigma$, use `polygon`, for which you need the vertices of the shape of interest. To get a smooth curve, make use of the fine sequence `xvals` and corresponding `fx` as defined in the code, and use logical subsetting (Section 4.1.5) to restrict attention to those locations of x such that $-3.62 \leq x \leq -3.22$:

```
R> xvals.sub <- xvals[xvals>=mu.minus.1sig & xvals<=mu.plus.1sig]
R> fx.sub <- fx[xvals>=mu.minus.1sig & xvals<=mu.plus.1sig]
```

You can then "sandwich" these between the two corners at the bottom of the shaded polygon using the matrix structure that the `polygon` function expects:

```
R> polygon(rbind(c(mu.minus.1sig,0),cbind(xvals.sub,fx.sub),c(mu.plus.1sig,0)),
          border=NA,col="gray")
```

Finally, use `arrows` and `text` to annotate the plot, indicating the areas discussed in the text:

```
R> arrows(c(-4.2,-2.7,-2.9),c(0.5,0.5,1.2),c(-3.7,-3.15,-3.4),c(0.2,0.2,1))
R> text(c(-4.2,-2.7,-2.9),c(0.5,0.5,1.2)+0.05,
       labels=c("0.159","0.159","0.682"))
```

The qnorm Function

Let's turn to `qnorm`. To find the quantile value corresponding to a lower-tail probability of exactly 0.159, execute the following:

```
R> qnorm(p=0.159,mean=mu,sd=sigma)
[1] -3.619715
```

Given the earlier results (when exploring the `pnorm` function), it should be clear why the result is a value of approximately –3.62.

As another example, you can find the upper quartile (the value *above which* you'd find a probability of exactly 0.25) with this:

```
R> qnorm(p=1-0.25,mean=mu,sd=sigma)
[1] -3.285102
```

Remember that the `q`-function will operate based on the (left) lower-tail probability, so to find a quantile based on an upper-tail probability, you must first subtract said probability from the total probability of 1.

In some methods and models used in frequentist statistics, it's common to assume that your observed data are normal. You can test the validity of this assumption by using your knowledge of the theoretical quantiles of the normal distribution, found via `qnorm`: Calculate a range of sample quantile values for your observed data (Section 13.2.4) and plot these against the same quantiles for a correspondingly standardized normal distribution. We refer to this visual tool as a normal *quantile-quantile* or *QQ* plot; it's useful when viewed alongside a histogram. If the

plotted points don't lie on a straight line, then this means the quantiles estimated from your data sample do not match those corresponding to a normal curve. Hence, the assumption that your data are normal may not be valid.

The built-in `qqnorm` function takes in your raw data and produces a QQ plot. Recall the `chickwts` dataset, first used in Section 13.1.3. Say you want to find out whether it's reasonable to assume the weights of the chicks are normally distributed. To that end, you use

```
R> hist(chickwts$weight,main="",xlab="weight")
R> qqnorm(chickwts$weight,main="Normal QQ plot of weights")
R> qqline(chickwts$weight,col="gray")
```

to produce the histogram of the 71 weights and the normal QQ plot in Figure 16-8. The additional `qqline` command adds the "optimal" line that the points would lie close to if the sample data originated from a normal distribution.

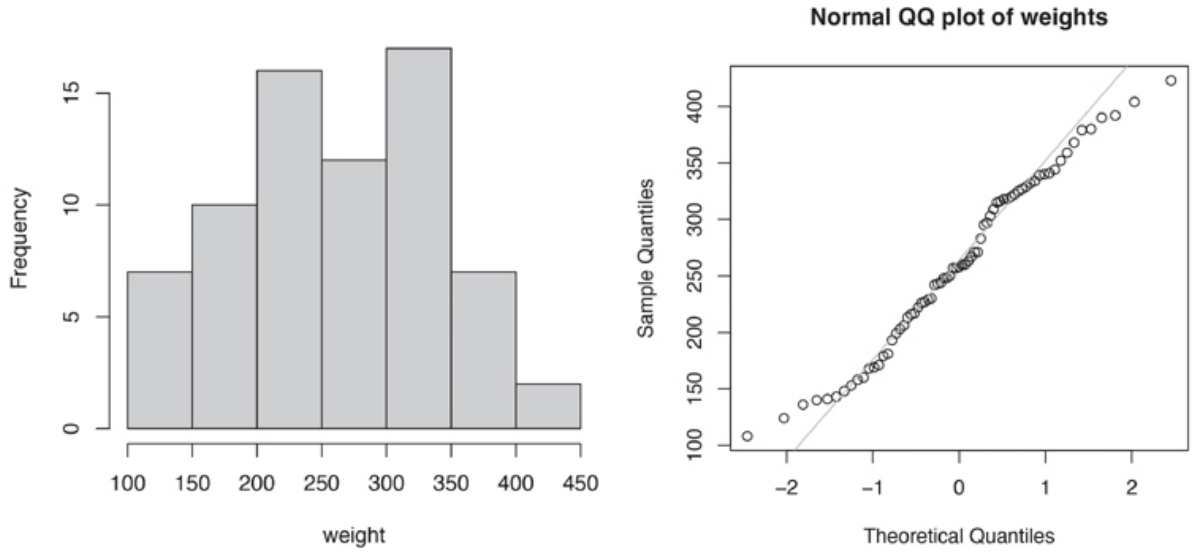


Figure 16-8: Histogram (left) and normal QQ plot (right) of the weights of chicks in the `chickwts` dataset. Are the data normally distributed?

If you inspect the histogram of the weights, you can see that the data match the general appearance of a normal distribution, with a roughly symmetric unimodal appearance. That said, it doesn't quite achieve the

smoothness and naturally decaying height that produces the familiar normal bell shape. The QQ plot on the right reflects this; the central quantile values appear to lie on the line relatively well, except for some relatively minor "wiggles." There are some clear discrepancies in the outer tails, but note that it's typical to observe discrepancies in these extreme quantiles in any QQ plot because fewer data points naturally occur there. Taking all of this into consideration, the assumption of normality for the chick weight data might be doubted, but isn't completely unreasonable.

NOTE

In addition to being mindful of the subjectivity of a visual inspection of a QQ plot, it's important to consider the sample size when assessing the validity of these kinds of assumptions. The larger the sample size, the less random variability will creep into the histogram and QQ plot, and you can more confidently reach a conclusion about whether your data follow the specified distribution. For instance, judging the assumption of normality with regards to the chick weight data may be complicated by the fact there's a relatively small sample size of only 71.

The rnorm Function

You generate random variates of any given normal distribution with `rnorm`; for example, the line

```
R> rnorm(n=7,mu,sigma)
[1] -3.764532 -3.231154 -3.124965 -3.490482 -3.884633 -3.192205 -3.475835
```

produces seven normally distributed values arising from N(–3.42, 0.2). In contrast to the QQ plot produced for the chick weights in Figure 16-8, you can use `rnorm`, `qqnorm`, and `qqline` to examine the degree to which hypothetically observed datasets that are truly normal vary in the context of a QQ plot.

The following code generates 71 standard normal values and produces a corresponding normal QQ plot and then does the same for a separate dataset of n = 710:

```
R> fakedata1 <- rnorm(n=71)
R> fakedata2 <- rnorm(n=710)
R> qqnorm(fakedata1,main="Normal QQ plot of generated N(0,1) data; n=71")
R> qqline(fakedata1,col="gray")
R> qqnorm(fakedata2,main="Normal QQ plot of generated N(0,1) data; n=710")
R> qqline(fakedata2,col="gray")
```

Figure 16-9 displays these plots.

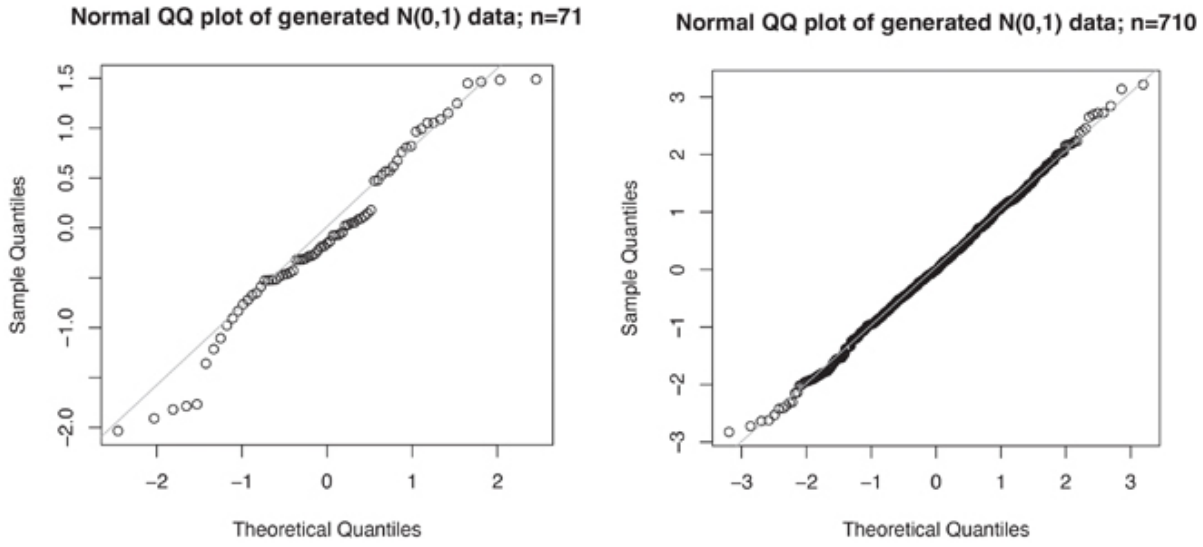


Figure 16-9: Normal QQ plots of 71 (left) and 710 (right) observations randomly generated from the standard normal distribution

The QQ plot for the simulated dataset of size n = 71 shows a similar deviation from the optimal line as did the QQ plot of the chick weights dataset. Bumping the sample size up by a factor of 10 shows that the QQ plot for the n = 710 normal observations offers up far less random variation, although visible discrepancies in the tails do still occur. To gain an impression of the magnitude of the deviation you might reasonably expect to see from sample to sample, rerun these lines of code several times (in other words, generating new datasets each time) and examine how each new QQ plot varies.

Normal Functions in Action: A Quick Example

Let's finish this section with one more working problem. Assume the manufacturer of a certain type of snack knows that the total net weight of the snacks in its 80-gram advertised package, X, is normally distributed with a mean of 80.2 grams and a standard deviation of 1.1 grams. The manufacturer weighs the contents of randomly selected individual packets. The probability a randomly selected packet is less than 78 grams (that is, $\Pr(X < 78)$) is as follows:

```
R> pnorm(78,80.2,1.1)
[1] 0.02275013
```

The probability that a packet is found to weigh some amount between 80.5 and 81.5 grams is:

```
R> pnorm(81.5,80.2,1.1)-pnorm(80.5,80.2,1.1)
[1] 0.2738925
```

The weight below which the lightest 20 percent of packets lie is:

```
R> qnorm(0.2,80.2,1.1)
[1] 79.27422
```

You can simulate the weights of five randomly selected packets with the following:

```
R> round(rnorm(5,80.2,1.1),1)
[1] 78.6 77.9 78.6 80.2 80.8
```

Exercise 16.4

a. A tutor knows that the length of time taken to complete a certain statistics question by first-year undergraduate students, X, is normally distributed with a mean of 17 minutes and a standard deviation of 4.5 minutes.

i. What is the probability a randomly selected undergraduate takes more than 20 minutes to complete the question; $\Pr(X > 20)$?

ii. What's the chance that a student takes between 5 and 10 minutes to finish the question; $\Pr(5 < X < 10)$?

iii. Find the time x that marks off the fastest 10 percent of students; $0.1 = \Pr(X < x)$.

iv. Plot the normal distribution of interest between $\pm 4\sigma$ and shade in the probability area of (iii), the fastest 10 percent of students (refer back to the code that produced Figure 16-7).

v. Generate a realization of times based on a class of 10 students completing the question.

b. A meticulous gardener is interested in the length of blades of grass on his lawn. He believes that blade length X follows a normal distribution centered on 10 mm with a variance of 2 mm.

i. Find the probability that a blade of grass is between 9.5 and 11 mm long; $\Pr(9.5 < X < 11)$.

ii. What are the standardized values of 9.5 and 11 in the context of this distribution? Using the standardized values, confirm that you can obtain the same probability you found in (i) with the standard normal density.

iii. Below which value x are the shortest 2.5 percent of blade lengths found; $0.025 = \Pr(X < x)$?

iv. Standardize your answer from (iii).

16.2.3 Student's t-Distribution

Student's t-*distribution* is a continuous probability distribution generally used when dealing with statistics estimated from a sample of data. It will become particularly relevant in the next two chapters, so I'll briefly explain it here.

A standard *t*-distribution looks a lot like the standard normal distribution: It's bell-shaped, symmetric, and unimodal, and it's centered on 0. The difference is that while we typically use a normal distribution to describe population-level characteristics, the *t*-distribution deals with inference based on a finite-sized *sample drawn from* a population.

Definition and the Degrees of Freedom

You usually don't have to define any parameters for the *t*-distribution, but you must choose the appropriate *t*-distribution by way of a strictly positive number $v > 0$; we call this the *degrees of freedom (df)* because it represents the number of individual components in the calculation of a given statistic that are "free to change." Expressed as such, we might further qualify v as being an integer; exceptions to this (for example, where the df is a decimal or floating-point number) are permissible when v must be approximated. Either way, you'll see in the upcoming chapters that this quantity is usually directly related to sample sizes.

For the moment, though, loosely think of the *t*-distribution as the representation of a *family* of curves and think of the degrees of freedom as the "selector" you use to tell you which particular version of the density to use. The precise equation for the density of the *t*-distribution is also not especially useful in an introductory setting, though it's useful to remember that the total probability underneath any *t* curve is naturally 1 (as it is for any density function).

Summarizing, the following two points are worth remembering:

- The *t*-distribution family is usually used in a standardized form, centered on 0 (non-standard *t*-distributions are possible, but rarer).

- You must specify a *t*-distribution in terms of a degrees-of-freedom value $v > 0$, and the density function becomes equal to the standard normal if you let v go to infinity.

The `dt`, `pt`, `qt`, and `rt` functions represent the R implementation of the *t*-distribution's density, cumulative distribution (left-hand probabilities under the curve), quantile, and random variate generation functions, respectively. The first arguments of each, `x`, `q`, `p`, and `n`, respectively, provide the relevant value (or values) of interest to these functions; the second argument in all is `df`, to which you must specify the degrees of freedom v.

The dt Function

The best way to get an impression of the *t* family is through visualization. Figure 16-10 plots the standard normal distribution, plus the *t*-distribution curve with $v = 1$, $v = 6$, and $v = 20$ df.

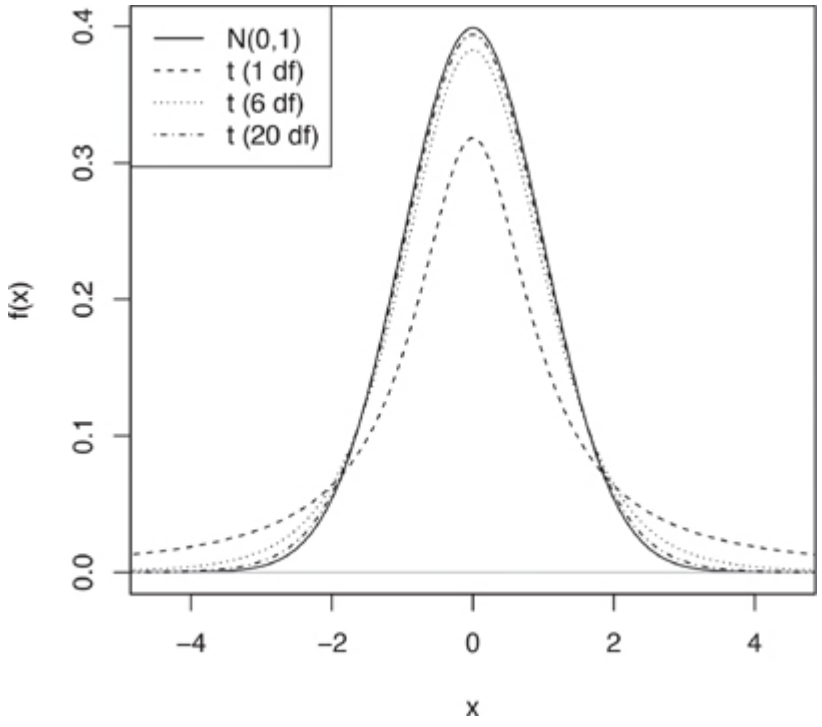


Figure 16-10: Contrasting the standard normal distribution with three instances of the `t`*-distribution. Note that the higher the degrees of freedom, the more the* `t`*-distribution resembles the normal.*

Create this plot with the following code:

```
R> xvals <- seq(-5,5,length=200)
R> plot(xvals,dnorm(x=xvals),xlim=c(-4.5,4.5),type="l",xlab="x",ylab="f(x)")
R> abline(h=0,col="gray")
R> lines(xvals,dt(x=xvals,df=1),lty=2)
R> lines(xvals,dt(x=xvals,df=6),lty=3)
R> lines(xvals,dt(x=xvals,df=20),lty=4)
R> legend("topleft",legend=c("N(0,1)","t (1 df)","t (6 df)","t (20 df)"),
          lty=1:4)
```

Perhaps the most important note to take away from Figure 16-10, and indeed from this section, is the aforementioned way in which the *t* density function changes with respect to the N(0, 1) distribution as you increase the df. For small values of v close to 1, the *t*-distribution is shorter, in terms of its mode, with more probability occurring in noticeably fatter tails.

The qt Function

It turns out that the *t* density approaches the standard normal density as $v \to \infty$. As a case in point, note that the upper 5 percent tail of the standard normal distribution is delineated by the following value:

```
R> qnorm(1-0.05)
[1] 1.644854
```

The same upper tail of the *t*-distribution is provided with df values of $v = 1$, $v = 6$, and $v = 20$, respectively:

```
R> qt(1-0.05,df=1)
[1] 6.313752
R> qt(1-0.05,df=6)
[1] 1.94318
R> qt(1-0.05,df=20)
[1] 1.724718
```

In direct comparison with the standard normal, the heavier weight in the tails of the *t* density leads naturally to more extreme quantile

values given the same probability value. Notice that this extremity, however, reduces as the df increases, which is a direct reflection of the fact that the corresponding *t* density is now "more like" the standard normal density.

The pt Function

The message conveyed by studying quantile values from the *t*-distribution is equivalently delivered by considering probabilities. Using the standard normal distribution as a reference, consider the following result:

```
R> 1-pnorm(1.87)
[1] 0.03074191
```

This tells you that for the standard normal density, a probability of around 0.03 lies in the upper tail delineated by a quantile value of 1.87 (see Section 16.2.2). Now, let's see what the upper-tail area is under the *t*-distribution for an increasing *v*:

```
R> 1-pt(1.87,df=1)
[1] 0.1563114
R> 1-pt(1.87,df=6)
[1] 0.05533941
R> 1-pt(1.87,df=20)
[1] 0.0380971
```

Notice that as we increase the degrees of freedom, the tail probability gets smaller (and begins to approach the corresponding probability from the standard normal). Again, this makes sense given what we know about the shape of the *t* density (refer back to Figure 16-10): The fatter tails of a *t* density with a smaller *v* mean a larger tail area will exist for a fixed quantile value.

The rt Function

If you wish to generate random realizations from a standard *t*-distribution, simply tell the `rt` function how many you want, along with

the degrees of freedom for the desired density. For example:

```
R> rt(n=4,df=1)
[1]  4.4520530 -3.9273791 -0.5805225 -0.2814131
R> rt(n=4,df=200)
[1]  0.5029836  0.9829744 -1.7806238 -1.0420139
```

16.2.4 Exponential

Probability density functions don't have to be symmetric like those you've encountered so far, nor do they need to allow for the random variable to be able to take values from negative infinity to positive infinity (like the normal or *t*-distributions). A good example of this is the *exponential distribution*, for which realizations of a random variable X are valid only for non-negative values, that is, $0 \leq X < \infty$.

For an exponential random variable $0 \leq X < \infty$, the density function f is

$$f(x) = \lambda_e \exp\{-\lambda_e x\}; \quad 0 \leq x < \infty \qquad (16.7)$$

where λ_e is a parameter of the distribution and $\exp\{\cdot\}$ is the exponential function. We use the notation

$$X \sim \text{EXP}(\lambda_e)$$

to indicate that "X follows an exponential distribution with rate λ_e."

The following are the key points to note:

- Theoretically, X can take any value in the range 0 to ∞, and $f(x)$ decreases as x increases.
- The "rate" parameter must be strictly positive; in other words, $\lambda_e > 0$. It defines $f(0)$ and the rate of decay of the function to the horizontal asymptote at 0.

The mean and variance are as follows, respectively:

$$\mu_X = \frac{1}{\lambda_e} \quad \text{and} \quad \sigma_X^2 = \frac{1}{\lambda_e^2}$$

The dexp Function

The density function for the exponential distribution is characterized by a steadily decreasing line beginning at $f(0) = \lambda$; the rate of this decay ensures a total area of 1 underneath the curve. You create Figure 16-11 with the relevant `d`-function in the following code:

```
R> xvals <- seq(0,10,length=200)
R> plot(xvals,dexp(x=xvals,rate=1.65),xlim=c(0,8),ylim=c(0,1.65),type="l",
        xlab="x",ylab="f(x)")
R> lines(xvals,dexp(x=xvals,rate=1),lty=2)
R> lines(xvals,dexp(x=xvals,rate=0.4),lty=3)
R> abline(v=0,col="gray")
R> abline(h=0,col="gray")
R> legend("topright",legend=c("EXP(1.65)","EXP(1)","EXP(0.4)"),lty=1:3)
```

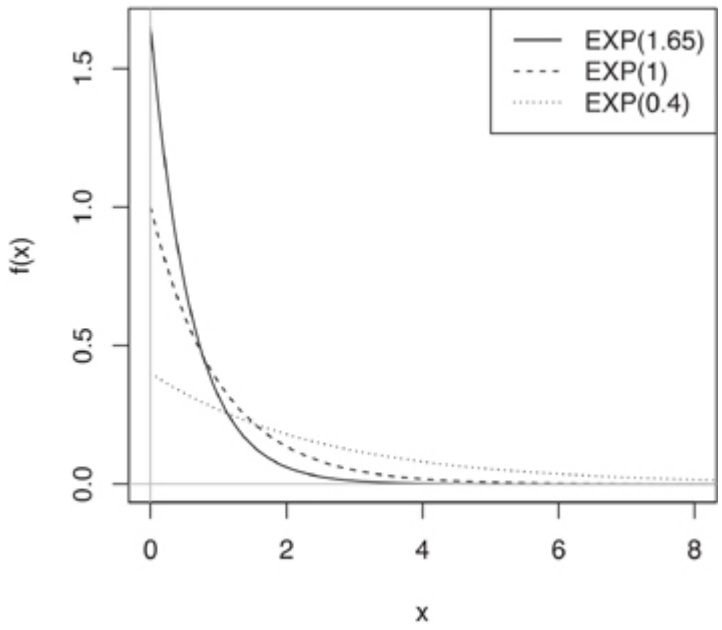


Figure 16-11: Three different exponential density functions. Decreasing λ_e lowers the mode and extends the tail.

The parameter λ_e is provided to `rate` in `dexp`, which is evaluated at x, provided to the first argument `x` (via the `xvals` object in this example). A

distinct feature of the exponential density function is that aforementioned decay to zero, with larger values of λ_e translating to a taller (yet sharper and more rapid) drop.

This naturally decreasing behavior helps identify the role the exponential distribution often plays in applications: one of a "time-until-event" nature. In fact, there's a special relationship between the exponential distribution and the Poisson distribution introduced in Section 16.1.3. When using the Poisson distribution to model the count of a certain event through time, you use the exponential distribution to model the time between these events. In such a setting, the exponential parameter λ_e defines the mean *rate* at which the events occur over time.

The pexp Function

To see the `pexp` function in action, let's revisit the example from Exercise 16.2, where the average number of cars passing an individual within a 120-minute window was said to be 107. Define the random variable X to be the waiting time between two consecutive cars passing and, using an exponential distribution for X on a per-minute time scale, set $\lambda_e = 107/120 \approx 0.89$ (rounded to 2 d.p.). If 107 cars are typically observed in a two-hour window, then you see cars at an average rate of 0.89 per minute.

Thus, you interpret λ_e as the "per-unit" measure of the λ_p parameter from the Poisson mass function. The interpretation of the mean as the reciprocal of the rate, $\mu_X = 1/\lambda_e$, is also intuitive. For example, when observing cars at a rate of about 0.89 per minute, note that the average waiting time between cars is roughly $1/0.89 \approx 1.12$ minutes.

So, in this example, you want to examine the density $X \sim \text{EXP}\left(\frac{107}{120}\right)$:

```
R> lambda.e <- 107/120
R> lambda.e
[1] 0.8916667
```

Say a car has just passed the individual's location and you want to find the probability that they must wait more than two and a half minutes before seeing another car; in other words, $\Pr(X > 2.5)$. You can do so using `pexp`:

```
R> 1-pexp(q=2.5,rate=lambda.e)
[1] 0.1076181
```

This indicates that you have just over a 10 percent chance of observing at least a 2-minute 30-second gap before the next car appears. Remember that the default behavior of the `p`-function is to find the cumulative, left-hand probability from the provided value, so you need to subtract the result from 1 to find an upper-tail probability.

As another example, you can find the probability of waiting less than 25 seconds for the next car with the following line:

```
R> pexp(25/60,lambda.e)
[1] 0.3103202
```

Note that you need to first convert the value of interest from seconds to minutes since you've defined $f(x)$ via $\lambda_e \approx 0.89$ on the scale of the latter.

The qexp Function

Use the appropriate quantile function `qexp` to find, say, the cutoff point for the shortest 15 percent of waits.

```
R> qexp(p=0.15,lambda.e)
[1] 0.1822642
```

This indicates that the value of interest is about 0.182 minutes or roughly $0.182 \times 60 = 10.9$ seconds.

The rexp Function

Finally, as usual, you can use the corresponding `r`-function to generate random variates of any specific exponential distribution:

```
R> rexp(n=5,rate=lambda.e)
[1] 1.6044782 0.4690638 0.2389020 0.0749131 1.2616632
```

NOTE

It's important to distinguish between the "exponential distribution," the "exponential family *of distributions," and the "exponential* function." *The first refers to the density function that's just been studied, whereas the second refers to a general class of probability distributions, including the Poisson, the normal, and the exponential itself. The third is just the standard mathematical exponential function upon which the exponential family members depend and is directly accessible in R via* `exp`.

Exercise 16.5

a. Situated in the central north island of New Zealand, the Pōhutu geyser is said to be the largest active geyser in the southern hemisphere. Suppose that it erupts an average of 3,500 times every year.

   i. With the intention of modeling a random variable X as the time between consecutive eruptions, evaluate the parameter value λ_e with respect to a time scale in days (assume 365.25 days per year to account for leap years).

   ii. Plot the density function of interest. What's the mean wait in days between eruptions?

   iii. What's the probability of waiting less than 30 minutes for the next eruption; $Pr(X < 0.5/24)$?

iv. What waiting time x defines the longest 10 percent of waits; $0.1 = Pr(X > x)$? Convert your answer to hours.

b. You can also use the exponential distribution to model certain product survival times, or "time-to-failure" types of variables. Say a manufacturer of a particular air conditioning unit knows that the product has an average life of 11 years before it needs any type of repair callout. Let the random variable X represent the time until the necessary repair of one of these units (in years) and assume X follows an exponential distribution with $\lambda_e = 1/11$.

   i. The company offers a five-year full repair warranty on this unit. What's the probability that a randomly selected air conditioner owner makes use of the warranty; $Pr(X < 5)$?

   ii. A rival company offers a six-year guarantee on its competing air conditioning unit but knows that its units last, on average, only nine years before requiring some kind of repair. What are the chances of making use of that warranty (note this also requires a change to λ_e)?

   iii. Determine the probabilities that the units in (i) and the units in (ii) last more than 15 years.

16.2.5 Chi-Squared

Another commonly seen continuous probability distribution is the *chi-squared* distribution. Pronounced *kai*, "chi" refers to the Greek symbol χ.

Description and Appearance

Like the *t*-distribution (Section 16.2.3), the chi-squared distribution represents a family of density functions and its use is dependent upon specification of a degrees-of-freedom value $v > 0$. Like the exponential distribution, it describes only positively valued random variables $0 < X < \infty$. Mathematically speaking, the chi-squared distribution with v degrees of freedom is the resulting distribution of a random variable X, where $X = \sum_{i=1}^{v} Z_i^2$ is the random variable that arises by summing v squared, standard normal random variables Z_i. The mean of a chi-squared distribution is equal to the degrees of freedom v, with the variance equal to $2v$.

To give you an idea of what these density functions actually look like, Figure 16-12 shows three particular curves defined with $v = 1$, $v = 5$, and $v = 10$ degrees of freedom. Note the positive domain of the function and the "flattening" and "right-extending" behavior as v is increased.

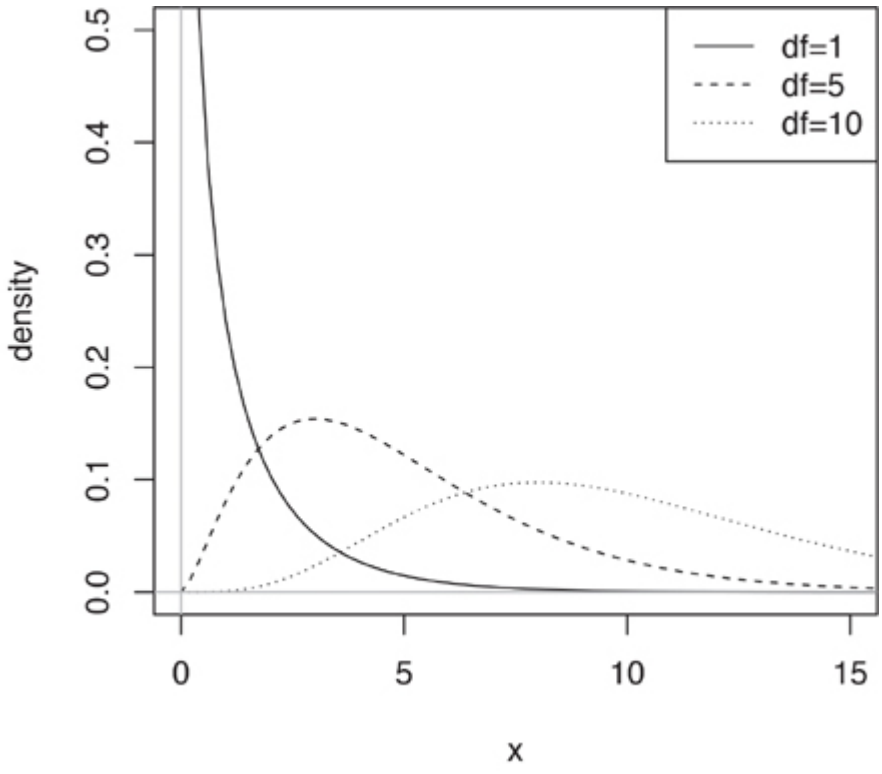


Figure 16-12: Three instances of the chi-squared density function using differing degrees of freedom values

R Implementation: dchisq, pchisq, qchisq, rchisq

I produced the plot in Figure 16-12 using the relevant `d`-function, `dchisq`, with v passed to the argument `df` as follows:

```
R> x <- seq(0,20,length=100)
R> plot(x,dchisq(x,df=1),type="l",xlim=c(0,15),ylim=c(0,0.5),ylab="density")
R> lines(x,dchisq(x,df=5),lty=2)
R> lines(x,dchisq(x,df=10),lty=3)
R> abline(h=0,col="gray")
R> abline(v=0,col="gray")
R> legend("topright",legend=c("df=1","df=5","df=10"),lty=1:3)
```

Probabilities are obtained in the cumulative/left-hand tail manner via `pchisq`, the corresponding `p`-function. For example, to find $\Pr(X < 9.42)$ when X follows a chi-squared distribution with 7 degrees of freedom, execute the following:

```
R> pchisq(q=9.42,df=7)
[1] 0.7761089
```

To find a right-hand/upper-tail probability, you can subtract any such `pchisq` result from the total area under the curve; of course this total area is, as usual for density functions, 1. Using positional matching of arguments (see Section 9.2.3), the following result reveals $\Pr(X > 24) \approx 0.155$ when X follows a chi-squared distribution with $v = 18$:

```
R> 1-pchisq(24,18)
[1] 0.1550278
```

The `qchisq` function provides quantiles for a pre-specified lower-tail probability value; the following example reveals that for a chi-squared distribution with 1 degree of freedom, the value that separates the uppermost 0.05 probability from the remaining 0.95 is approximately 3.84:

```
R> qchisq(p=0.95,df=1)
[1] 3.841459
```

As usual, you can use the corresponding `r`-function to generate random realizations of a given chi-squared random variable:

```
R> rchisq(n=5,df=7)
[1] 11.264984  6.829794  8.737887  4.044202  8.526810
```

16.2.6 F-Distribution

The last density function we'll examine here in any detail is the *F-distribution*. You'll see this distribution crop up in Part IV, as it forms a key part of assessing linear regression models.

Description and Appearance

Mathematically, an F-distribution arises when considering a random variable $X = \frac{S_1/\nu_1}{S_2/\nu_2}$ formed from the ratio of two chi-squared random variables S_1 and S_2 with respective df values ν_1 and ν_2. As such, the F-distribution family is also defined only for positive values $0 < X < \infty$ and requires specification of two degrees-of-freedom values corresponding to the two contributing chi-squared random variables (ν_1, ν_2).

Figure 16-13 shows three different curves representing three specific F-distributions; each is defined with a unique pair of degrees-of-freedom values. Unsurprisingly, given their mathematical derivation, the F densities still look somewhat like those of stand-alone, chi-squared random variables.

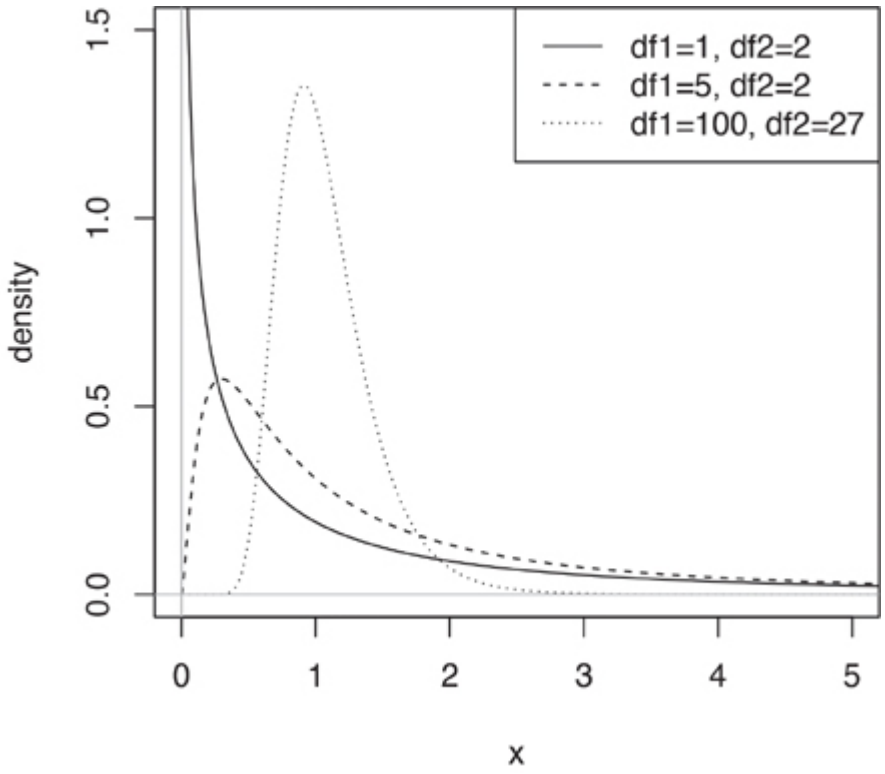


Figure 16-13: Three instances of the F *density function using differing degrees-of-freedom value pairs*

R Implementation: df, pf, qf, rf

The `df` (density), `pf` (cumulative probability), `qf` (quantile), and `rf` (random realization) functions represent R's implementation of the *F*-distribution. For example, I used the `df` function to produce Figure 16-13 with the following:

```
R> plot(x,df(x,df1=1,df2=2),type="l",xlim=c(0,5),ylim=c(0,1.5),ylab="density")
R> lines(x,df(x,df1=5,df2=2),lty=2)
R> lines(x,df(x,df1=100,df2=27),lty=3)
R> abline(h=0,col="gray")
R> abline(v=0,col="gray")
R> legend("topright",legend=c("df1=1, df2=2","df1=5, df2=2","df1=100, df2=27"),
          lty=1:3)
```

The remaining functions operate in exactly the same way as their chi-squared counterparts, other than the fact we must supply both `df1`

and `df2` arguments when dealing with the *F*-distribution. Two quick examples of the relevant `p`-and `q`-functions are:

```
R> pf(q=1.3,df1=1,df2=2)
[1] 0.6276459
R> qf(p=0.9,df1=100,df2=27)
[1] 1.539
```

Here, the call to `pf` tells us there is a probability of approximately 0.63 of observing a realized value less than 1.3 from an *F*-distribution on (1, 2) degrees of freedom; the call to `qf` reveals that for an *F*-distribution on (100, 27) degrees of freedom, a value of 1.539 separates an upper-tail probability of 0.1 from the remaining 0.9.

Exercise 16.6

Consider X to be a chi-squared random variable for the next three tasks.

a. Generate three random realizations of X based on $v = 8$ degrees of freedom.

b. With the same df, calculate the probability you would make a realization greater than each value from (a). In other words, if you denote the three realizations from (a) as x_1, x_2, and x_3, find $\Pr(X > x_1)$, $\Pr(X > x_2)$, and $\Pr(X > x_3)$.

c. Work out c, where $\Pr(X < c) = 0.99$, for a chi-squared distribution using three different df values of 3, 10, 300. In other words, for each of these three values of v, find the quantile value that delineates the upper 0.01 of probability from the lower 0.99. What do you notice?

Turn your attention to the *F*-distribution for the remaining two tasks.

d. What is the 0.75th quantile of the *F*-distribution on (35, 50) degrees of freedom? What is the 0.25th quantile?

e. Confirm via the appropriate `p`-function that the probability area between the two quantile values in (d) is exactly 0.5.

16.2.7 Other Density Functions

There are a number of common probability density functions used for a wide variety of tasks involving continuous random variables. Here are just three of many others you might come across:

- The *gamma distribution* is a generalization of both the exponential and chi-squared distributions. Its functions are `dgamma`, `pgamma`, `qgamma`, and `rgamma`, and it's dependent upon "shape" and "scale" parameters provided as the arguments `shape` and `scale`.
- The *Weibull distribution* also includes the exponential distribution as a special case, and is often used to model the time between events. Its R functions are `dweibull`, `pweibull`, `qweibull`, and `rweibull`; these too require specification of `shape` and `scale` parameter values.
- The *beta distribution* is often used in Bayesian modeling, and it has implemented functions `dbeta`, `pbeta`, `qbeta`, and `rbeta`. It's defined by two "shape" parameters α and β, supplied as `shape1` and `shape2`, respectively.

NOTE

In all of the common probability distributions you've examined over the past couple of sections, I've emphasized the need to perform "one-minus" operations to find probabilities or quantiles with respect to an upper- or right-tailed area. This is because of the cumulative nature of the `p`*and* `q`*-functions: By definition, it's the lower tail that's dealt with. However, most* `p`*and* `q`*-functions in R include an optional logical argument,* `lower.tail`*, which defaults to* `TRUE`*. Therefore, an alternative is to set* `lower.tail=FALSE` *in*

any relevant function call, in which case R will expect or return upper-tail areas specifically.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`dbinom`	Binomial mass function	Section 16.1.2, p. 397
`pbinom`	Binomial cumulative probabilities	Section 16.1.2, p. 398
`qbinom`	Binomial quantiles function	Section 16.1.2, p. 399
`rbinom`	Binomial random realizations	Section 16.1.2, p. 399
`dpois`	Poisson mass function	Section 16.1.3, p. 402
`ppois`	Poisson cumulative probabilities	Section 16.1.3, p. 402
`rpois`	Poisson random realizations	Section 16.1.3, p. 403
`dunif`	Uniform density function	Section 16.2.1, p. 406
`punif`	Uniform cumulative probabilities	Section 16.2.1, p. 408
`qunif`	Uniform quantiles	Section 16.2.1, p. 409
`runif`	Uniform random realizations	Section 16.2.1, p. 409

Function/operator	Brief description	First occurrence
`dnorm`	Normal density function	Section 16.2.2, p. 412
`pnorm`	Normal cumulative probabilities	Section 16.2.3, p. 421
`qnorm`	Normal quantiles	Section 16.2.2, p. 415
`qqnorm/qqline`	Normal QQ plots	Section 16.2.2, p. 416
`rnorm`	Normal random realizations	Section 16.2.2, p. 417
`dt`	Student's *t* density function	Section 16.2.3, p. 420
`qt`	Student's *t* quantiles	Section 16.2.3, p. 421
`pt`	Student's *t* cumulative probabilities	Section 16.2.3, p. 422
`rt`	Student's *t* random realizations	Section 16.2.3, p. 422
`dexp`	Exponential density function	Section 16.2.4, p. 423
`pexp`	Exponential cumulative probabilities	Section 16.2.4, p. 424
`qexp`	Exponential quantiles	Section 16.2.4, p. 425

Function/operator	Brief description	First occurrence
`rexp`	Exponential random realizations	Section 16.2.4, p. 425
`dchisq`	Chi-squared density function	Section 16.2.5, p. 427
`pchisq`	Chi-squared cumulative probabilities	Section 16.2.5, p. 427
`qchisq`	Chi-squared quantiles	Section 16.2.5, p. 428
`rchisq`	Chi-squared random realizations	Section 16.2.5, p. 428
`df`	*F* density function	Section 16.2.6, p. 429
`pf`	*F* cumulative probabilities	Section 16.2.6, p. 429
`qf`	*F* quantiles	Section 16.2.6, p. 429

PART IV
STATISTICAL TESTING AND MODELING

17
SAMPLING DISTRIBUTIONS AND CONFIDENCE



In Chapters 15 and 16, you applied the idea of a probability distribution to examples in which a corresponding random variable is defined as some measurement or observation of interest whose realizations are subject to chance. In this chapter, you'll treat sample statistics as random variables. Doing so introduces the concept of a *sampling distribution*, which is the term given to the probability distribution used to describe the variability, or "uncertainty," naturally present when you estimate population parameters using sample statistics. A *confidence interval* is a direct reflection of this uncertainty and provides an interval estimate of a population parameter. Overall, these concepts will form the foundation for formal hypothesis testing in Chapter 18.

17.1 Sampling Distributions

A sampling distribution is just like any other probability distribution but is so-named when it's specifically associated with a random variable that is itself a sample statistic. In Chapters 15 and 16, you assumed

knowledge of the parameters of the relevant example distribution (for example, the mean and the standard deviation of a normal distribution or the probability of success in a binomial distribution), but in practice these kinds of population-level quantities are, more often than not, unknown. In this situation, you'd typically estimate the quantities from a sample (see Figure 13-2 on page 308 for a conceptual illustration). You can treat any statistic estimated from a sample as a random variable, with the estimated value itself as the realization of that random variable. This makes sense, as it's entirely possible different samples drawn from the same population will provide a different value for the same statistic, and realizations of random variables are naturally subject to variability. Understanding and modeling this natural variability inherent in estimated sample statistics (using relevant sampling distributions) is a key part of many statistical analyses.

Like any other probability distribution, the central "balance" point of a sampling distribution is its mean, but the standard deviation of a sampling distribution is termed its *standard error*. The slight change in terminology reflects the fact that the probabilities of interest are no longer tied to raw measurements or observations per se, but rather to a quantity calculated from a *sample* of such observations. The theoretical formulas for various sampling distributions therefore depend upon (a) the original probability distributions that are assumed to have generated the raw data and (b) the size of the sample itself.

This section will explain the key ideas behind sampling distributions and provide some examples, focusing on two simple and easily recognized statistics: a single sample mean and a single sample proportion. I'll expand on this in Chapter 18 when covering hypothesis testing, and you'll need to understand the role of sampling distributions in assessing important model parameters when you look at regression methods in Chapters 20 to 22.

NOTE

The validity of the theory of sampling distributions as discussed in this chapter makes an important assumption. Whenever I talk about a sample of data from which a given statistic is calculated, I assume those observations are independent *of one another and that they are* identically distributed. *You'll often see "independent, identically distributed" observations abbreviated as* iid *in statistical material.*

17.1.1 For a Sample Mean

The arithmetic mean is arguably the most commonly used measure of centrality (see Section 13.2.1) when summarizing a dataset.

Mathematically, the variability inherent in a sample mean is described as follows: Formally, denote the random variable of interest as X. This represents the mean of a sample of n observations from the "raw observation" random variable X, as in $x_1, x_2, \ldots, x_n$. We assume those observations have a true finite mean $-\infty < \mu_X < \infty$ and a true finite standard deviation $0 < \sigma_X < \infty$. The conditions for finding the probability distribution of a sample mean vary depending on whether you know the value of the standard deviation.

Situation 1: Standard Deviation Known

When you know the true value of the standard deviation of the raw observation data, σ_X, then the following are true:

- If X is normally distributed, the sampling distribution of X is itself a normal distribution, with mean μ_X and standard error $\sigma_X/\sqrt{n}$.
- If X is not normally distributed, the sampling distribution of X is still approximately normal, with mean μ_X and standard error $\sigma_X/\sqrt{n}$, and this approximation improves as $n \to \infty$. This is known as the *central limit theorem (CLT)*.

Situation 2: Standard Deviation Unknown

In practice, you typically won't know the true value of the standard deviation of the raw measurement distribution that generated your sample data. In this case, we usually just replace σ_X with s_X, which is the standard deviation of the sampled data. However, this substitution introduces additional variability that affects the distribution associated with the sample mean random variable X:

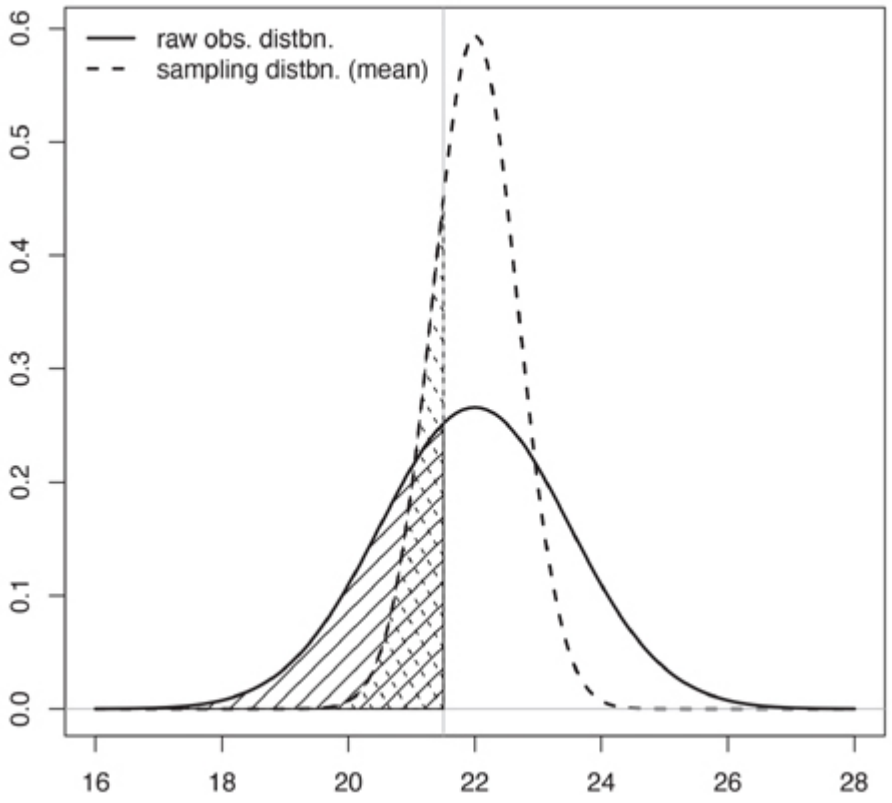
- Standardized values (Section 16.2.2) of the sampling distribution of X follow a t-distribution with $\nu = n - 1$ degrees of freedom; standardization is performed using the standard error $s_X/\sqrt{n}$.
- Additionally, if n is small, then it is necessary to assume the distribution of X is normal for the validity of this t-based sampling distribution of X.

The nature of the sampling distribution of X therefore depends upon whether you know the true standard deviation of the observations, as well as the sample size n. The CLT states that normality occurs even if the raw observation distribution is itself not normal, but this approximation is less reliable if n is small. It's a common rule of thumb to rely on the CLT only if $n \geq 30$. If you use s_X, the sample standard deviation, to calculate the standard error of X, then the sampling distribution is the t-distribution (following standardization). Again, this is generally taken to be reliable only if $n \geq 30$.

Example: Temperatures

As an example, suppose that the daily maximum temperature in the month of January in a certain New Zealand town follows a normal distribution with a mean of 22 degrees Celsius and a standard deviation of 1.5 degrees. Assume we have observed the temperatures on $n = 5$ random days (for the sake of simplicity, also assume the daily maximum temperatures are independent of one another). Then, in line with the comments for Situation 1, the sampling distribution of X will be normal, with mean 22 and standard error $1.5/\sqrt{5} \approx 0.671$.

The top image of Figure 17-1 shows the raw measurement distribution along with the sampling distribution of the mean for $n = 5$.


raw obs. distbn.
sampling distbn. (mean)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
16
18
20
22
24
26
28

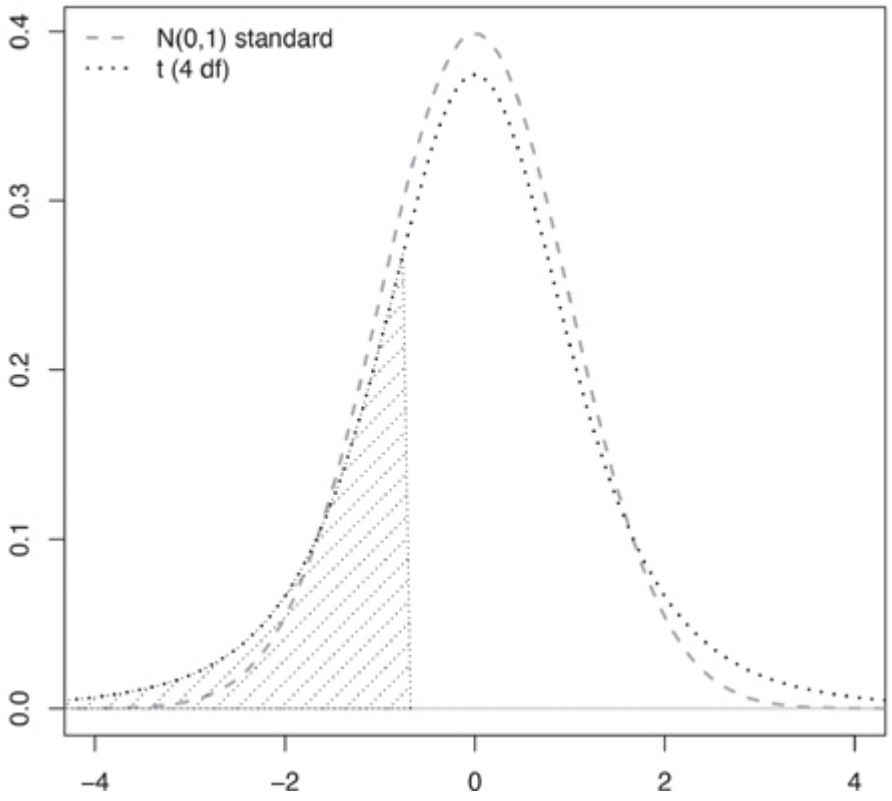

N(0,1) standard
t (4 df)
0.0
0.1
0.2
0.3
0.4
−4
−2
0
2
4

Figure 17-1: Illustrating the sampling distribution of a sample mean for n = *5, based on an N(22,1.5) raw observation distribution. Top: Comparing the normal-based version of the sampling distribution (assuming* σ_X *is known) to the observation distribution. Bottom: Comparing the centered* t*-based version of the sampling distribution, using 4 degrees of freedom (in other words, assuming* s *has been used to calculate the standard error), to a standard normal. Shaded areas represent Pr*(X < *21.5*), $Pr\left(\mathrm{X} < 21.5\right)$ *(solid and dashed, topmost plot) and* $Pr\left(\mathrm{T} < (21.5 - x)/\left(s/\sqrt{5}\right)\right)$ *(dotted, bottom plot).*

You can start this plot with code similar to the examples given in Section 16.2.2:

```
R> xvals <- seq(16,28,by=0.1)
R> fx.samp <- dnorm(xvals,22,1.5/sqrt(5))
R> plot(xvals,fx.samp,type="l",lty=2,lwd=2,xlab="",ylab="")
R> abline(h=0,col="gray")
R> fx <- dnorm(xvals,22,1.5)
R> lines(xvals,fx,lwd=2)
R> legend("topleft",legend=c("raw obs. distbn.","sampling distbn. (mean)"),
         lty=1:2,lwd=c(2,2),bty="n")
```

In this example, the sampling distribution of X is clearly a taller, skinnier normal distribution than the one tied to the observations themselves. This makes sense; you expect less variability in an *average* of several measurements as opposed to the raw, individual measurements. Functionally, the presence of n in the denominator of the standard error dictates a more precise distribution around the mean if you increase the sample size. Again, this makes sense; means will "vary less" between samples of a larger size.

You can now ask various probability questions; note the importance of distinguishing between the measurement distribution and the sampling distribution. For example, the following code provides Pr(X < 21.5), the probability that a randomly chosen day in January has a maximum temperature of less than 21.5 degrees:

```
R> pnorm(21.5,mean=22,sd=1.5)
[1] 0.3694413
```

The next bit of code provides the probability that the *sample mean* will be less than 21.5 degrees, $\Pr\left(\overline{X} < 21.5\right)$ based on a sample of five random days in January:

```
R> pnorm(21.5,mean=22,sd=1.5/sqrt(5))
[1] 0.2280283
```

The line-shaded areas in the topmost plot of Figure 17-1 show these two different probabilities. You can add these shaded areas to the plot by running the following lines directly after the earlier plotting code:

```
R> abline(v=21.5,col="gray")
R> xvals.sub <- xvals[xvals<=21.5]
R> fx.sub <- fx[xvals<=21.5]
R> fx.samp.sub <- fx.samp[xvals<=21.5]
R> polygon(cbind(c(21.5,xvals.sub),c(0,fx.sub)),density=10)
R> polygon(cbind(c(21.5,xvals.sub),c(0,fx.samp.sub)),density=10,
           angle=120,lty=2)
```

In previous uses of `polygon`, you've simply specified a `col` for color; in this example, I implemented shading lines, using the arguments `density` (number of lines per inch) and `angle` (slope of lines in degrees; defaults to `angle=45`).

To evaluate the probabilities, note that you've required knowledge of the parameters governing *X*. In practice, you'll rarely have these quantities (as noted in Situation 2). Instead, you obtain a sample of data and calculate summary statistics, which pose as estimates of the corresponding unknown parameters.

Running the following line produces five randomly generated temperatures from the $X \sim N(22, 1.5)$ distribution:

```
R> obs <- rnorm(5,mean=22,sd=1.5)
R>
[1] 22.92233 23.09505 20.98653 20.10941 22.33888
```

Now, for the sake of the example, say these five values constitute all the data you have for this particular problem; in other words, pretend

you don't know that $\mu_X = 22$ and $\sigma_X = 1.5$. Your best guesses of the true values of μ_X and σ_X—denoted x and s, respectively—are therefore as follows:

```
R> obs.mean <- mean(obs)
R> obs.mean
[1] 21.89044
R> obs.sd <- sd(obs)
R> obs.sd
[1] 1.294806
```

You can then calculate the estimated standard error of the sample mean:

```
R> obs.mean.se <- obs.sd/sqrt(5)
R> obs.mean.se
[1] 0.5790549
```

Because $n = 5$ is relatively small, you must assume the values in `obs` are realizations from a normal distribution, in line with the points made for Situation 2. This allows you to handle the sampling distribution of X using the t-distribution with 4 degrees of freedom. Recall from Section 16.2.3, though, that any t-distribution is typically placed on a standardized scale. Therefore, for you to find the probability that the mean temperature (in a sample of five days) is less than 21.5 based on your calculated sample statistics, you must first standardize this value using the formula outlined in Section 16.2.2, page 410. Label the corresponding random variable as T and the specific value as t_4, stored as the object `t4` in R:

```
R> t4 <- (21.5-obs.mean)/obs.mean.se
R> t4
[1] -0.6742706
```

This places the value of interest, 21.5, on the standardized scale, making it interpretable with respect to a standard normal distribution or, as is correct in this setting (because you are using the estimate s rather than the unknown σ_X in calculating the standard error), t_4 follows

the aforementioned *t*-distribution with 4 degrees of freedom. Using this distribution, the probability of interest $\Pr\left(T < t_4\right) = \Pr\left(T < -0.6742706\right) = \Pr\left(X < 21.5\right)$ is therefore given as:

```
R> pt(t4,df=4)
[1] 0.26855
```

Note that when you calculated this probability from the "true" normal sampling distribution of $\Pr\left(X < 21.5\right)$ on page 439, you got a result of about 0.23. However, using the *estimated* sampling distribution (arising from the five data points in `obs`) you arrive at roughly 0.27 (2 d.p.) for the same probability. This comparatively larger probability is due to the fatter tails of the *t*-distribution, which account for the additional uncertainty associated with using *estimates* of unknown quantities (see Section 16.2.3).

In order to visualize this, the bottom plot in Figure 17-1 provides the *t*-distribution with $\nu = 4$, marking off the probability described. The N(0, 1) density is also plotted for comparison; this represents the standardized version of the $\mathrm{N}\left(22, 1.5/\sqrt{5}\right)$ sampling distribution from Situation 1. You can create this plot with the following code:

```
R> xvals <- seq(-5,5,length=100)
R> fx.samp.t <- dt(xvals,df=4)
R> plot(xvals,dnorm(xvals),type="l",lty=2,lwd=2,col="darkgray",xlim=c(-4,4),
       xlab="",ylab="")
R> abline(h=0,col="gray")
R> lines(xvals,fx.samp.t,lty=3,lwd=2)
R> polygon(cbind(c(t4,-5,xvals[xvals<=t4]),c(0,0,fx.samp.t[xvals<=t4])),
          density=10,lty=3)
R> legend("topleft",legend=c("N(0,1) standard","t (4 df)"),
         col=c("darkgray","black"),lty=2:3,lwd=c(2,2),bty="n")
```

Consideration of probability distributions associated with sample means is clearly not a trivial exercise. Using sample statistics governs the

nature of the sampling distribution; in particular, it will be *t*-based if you use the sample standard deviation to calculate the standard error. However, as the examples here have shown, once that's been established, the calculation of various probabilities is easy and follows the same general rules and R functionality detailed in Section 16.2.

17.1.2 For a Sample Proportion

You interpret sampling distributions for sample proportions in much the same way as those for a sample mean. Indeed, you can think of a sample proportion as the mean value of a set of binary 0-1 observations. If n trials of a success/failure event are performed, you can obtain an estimate of the proportion of successes; if another n trials are performed, the new estimate could vary. It's this variability that you're investigating.

The random variable of interest, $\widehat{P}$, represents the estimated proportions of successes over any n trials, each resulting in some defined binary outcome. It is estimated as $\widehat{p} = \frac{x}{n}$, where x is the number of successes in a sample of size n. Let the corresponding true proportion of successes (often unknown) simply be denoted with π.

NOTE

Used in this setting, π doesn't refer to the common geometric value 3.14 (2 d.p.). Rather, the π symbol is by convention often used to denote a population proportion parameter.

The sampling distribution of $\widehat{P}$ is approximately normal with mean π and standard error $\sqrt{\pi(1-\pi)/n}$. The following are the key things to note:

- This approximation is valid if n is large or π is not too close to either 0 or 1.

- There are rules of thumb to determine this validity; one such rule is to assume the normal approximation is satisfactory if both $n\pi$ and $n(1 - \pi)$ are greater than 5.
- When the true π is unknown or is unassumed to be a certain value, it is typically replaced by $\widehat{p}$ in all of the previous formulas.

As long as you can deem the approximation to the normal distribution valid, this is the only probability distribution that you need to be concerned with. However, it's worth noting that the standard error of the sampling distribution of a sample proportion depends directly upon the proportion π. This becomes important when constructing confidence intervals and carrying out hypothesis tests, which you'll begin to explore in Chapter 18.

Let's look at a practical example. Suppose a political commentator in the United States is interested in the proportion of voting-age citizens in her home city that already know how they will vote in the next presidential election. She obtains a yes or no answer from 118 suitable randomly selected individuals. Of these individuals, 80 say they know how they'll vote. To investigate the variability associated with the proportion of interest, you'll therefore need to consider

$$\widehat{P} \sim \mathrm{N}\left(\widehat{p}, \sqrt{\frac{\widehat{p}(1-\widehat{p})}{n}}\right) \qquad (17.1)$$

where $\widehat{p} = \frac{80}{118}$. In R, the following gives you the estimate of interest:

```
R> p.hat <- 80/118
R> p.hat
[1] 0.6779661
```

In the sample, about 68 percent of the surveyed individuals know how they will vote in the next election. Note also that, according to the aforementioned rule of thumb, the approximation to the normal distribution is valid because both values are greater than 5:

```
R> 118*p.hat
[1] 80
R> 118*(1-p.hat)
[1] 38
```

Estimate the standard error of $\widehat{p}$ with the following:

```
R> p.se <- sqrt(p.hat*(1-p.hat)/118)
R> p.se
[1] 0.04301439
```

Then you can plot the corresponding sampling distribution:

```
R> pvals <- seq(p.hat-5*p.se,p.hat+5*p.se,length=100)
R> p.samp <- dnorm(pvals,mean=p.hat,sd=p.se)
R> plot(pvals,p.samp,type="l",xlab="",ylab="",
      xlim=p.hat+c(-4,4)*p.se,ylim=c(0,max(p.samp)))
R> abline(h=0,col="gray")
```

Figure 17-2 shows the curve it produces.

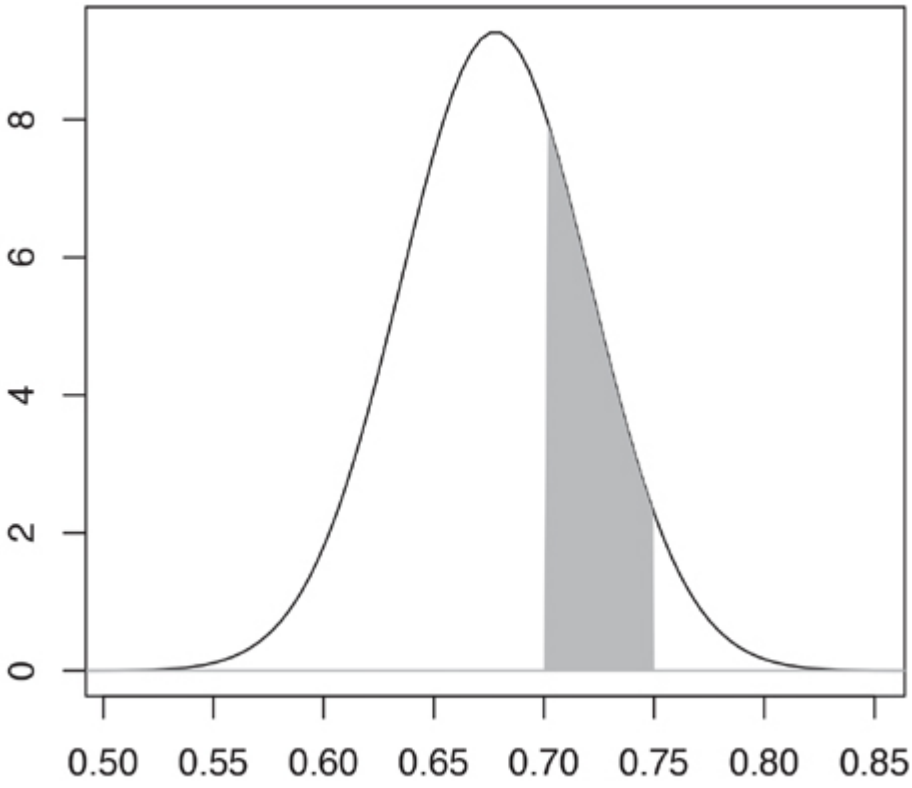


Figure 17-2: Visualizing the sampling distribution for the voting example as per Equation (17.1). The shaded area represents $Pr\left(0.7 < \widehat{\mathrm{P}} < 0.75\right)$*, which is the probability that the true sample proportion for samples of size* n *= 118 lies between 0.7 and 0.75.*

This distribution is assumed to describe the variability in the sample proportion of voters who already know how they will vote, for other samples of this size. For example, the shaded area in Figure 17-2 highlights the probability that in another sample of size n = 118, the sample proportion of voters in the given city who already know how they're going to vote is somewhere between 0.7 and 0.75. You can add this shaded area to the curve to complete Figure 17-2 with the following code:

```
R> pvals.sub <- pvals[pvals>=0.7 & pvals<=0.75]
R> p.samp.sub <- p.samp[pvals>=0.7 & pvals<=0.75]
R> polygon(cbind(c(0.7,pvals.sub,0.75),c(0,p.samp.sub,0)),
           border=NA,col="gray")
```

And with knowledge of `pnorm`, introduced in Section 16.2.2, you can use the following code to calculate the probability of interest:

```
R> pnorm(0.75,mean=p.hat,sd=p.se) - pnorm(0.7,mean=p.hat,sd=p.se)
[1] 0.257238
```

To put this result into words, there's roughly a 25.7 percent chance we'll encounter a proportion of people who know how they'll vote being somewhere between 0.7 and 0.75 if we were to draw another sample of the same size from the same population.

Exercise 17.1

A teacher wants to test all of the 10th grade students at his school to gauge their basic mathematical understanding, but the photocopier breaks after making only six copies of the test. With no other choice, he chooses six students at random to take the test.

Their results, recorded as a score out of 65, have a sample mean of 41.1. Suppose we know the true population standard deviation of the marks of this test to be 11.3.

a. Calculate the standard error associated with the mean test score.

b. Assuming the scores are normally distributed, evaluate the probability that the mean score lies between 45 and 55 if the teacher took another sample of the same size.

c. A student who gets less than half the total marks receives a failing grade (F). Find the probability that the average score is an F based on another sample of the same size.

A marketing company wants to find out which of two energy drinks (drink A or drink B) teenagers prefer. It surveys 140 teens, and the results indicate that only 35 percent prefer drink A.

d. Use a quick check to decide whether it is valid to use the normal distribution to represent the sampling distribution of this proportion.

e. What is the probability that in another sample of the same size, the proportion of teenagers who prefer drink A is greater than 0.4?

f. Find the two values of this sampling distribution that identify the central 80 percent of values of the proportion of interest.

In Section 16.2.4, the time between cars passing an individual's location was modeled using an exponential distribution. Say that on the other side of town, her friend is curious about a similar problem. Standing outside her house, she records 63 individual times between cars passing. These sampled times have a mean of $\bar{x} = 37.8$ seconds with a standard deviation of $s = 34.51$ seconds.

g. The friend inspects a histogram of her raw measurements and notices that her raw data are heavily right-skewed. Briefly identify and describe the nature of the sampling distribution with respect to the sample mean and calculate the appropriate standard error.

h. Using the standard error from (g) and the appropriate probability distribution, calculate the probability that in another sample of the same size, the sample mean time between cars passing is as follows:

   i. More than 40 seconds

   ii. Less than half a minute

   iii. Between the given sample mean and 40 seconds

17.1.3 For Other Statistics

So far, you've looked at sampling distributions in cases dealing with a single sample mean or sample proportion, though it's important to note that in many problems we seek inference via more complicated statistics. Nevertheless, you can apply the concepts explored in this section to any statistic estimated from a finite-sized sample. The key is always to be able to understand the variability associated with your sample estimates.

In some settings, such as those covered so far, the sampling distribution is parametric, meaning that the functional (mathematical) form of the probability distribution itself is known and depends only on the provision of specific parameter values. This is sometimes contingent upon the satisfaction of certain conditions, as with the application of the normal distribution covered in this chapter. For other statistics, it may be the case that the mathematical form of the appropriate sampling distribution is not known; in these situations, you could use computer simulation or methods of approximation to obtain the required probabilities. In the remainder of this chapter and over the next few

chapters, you'll continue to explore statistics that are tied to parametric sampling distributions for common tests and models.

NOTE

The variability of an estimated quantity is only one side of the coin. Just as important is the issue of statistical bias. *Where "natural variability" should be associated with* random error, *bias is associated with* systematic error, *in the sense that a biased statistic does not settle on the corresponding true parameter value as the sample size increases. Bias can be caused by flaws in a study design or collection of data; it can also be the result of a poor estimator of the parameter of interest. Bias is an undesirable trait of any given estimator or statistical analysis unless it can be quantified and removed, which is often difficult if not impossible in practice. I've therefore dealt so far only with* unbiased *statistical estimators, many of which you may already be familiar with (for example, the arithmetic mean), and for the sake of simplicity I'll continue to assume unbiasedness moving forward.*

17.2 Confidence Intervals

A *confidence interval (CI)* is an interval defined by a lower limit l and an upper limit u, used to describe possible values of a corresponding population parameter in light of observed sample data. Interpretation of a confidence interval therefore allows you to state a "level of confidence" that a true parameter of interest falls between this upper and lower limit, often expressed as a percentage. It's a common and useful tool computed directly from the sampling distribution of the statistic of interest.

The following list summarizes the important points:

- We usually express the level of confidence as a percentage, such that you'd construct a $100 \times (1 - \alpha)$ percent confidence interval, where $0 < \alpha < 1$ is an "amount of tail probability."

- By convention, the three most common intervals are defined with either $\alpha = 0.1$ (a 90 percent interval), $\alpha = 0.05$ (a 95 percent interval), or $\alpha = 0.01$ (a 99 percent interval).
- Colloquially, we interpret a confidence interval (l, u) as "I am $100 \times (1 - \alpha)$ percent confident that the true parameter value lies somewhere between l and u."

Confidence intervals are constructed in different ways, depending on the type of statistic and therefore the shape of the corresponding sampling distribution. For symmetrically distributed sample statistics, like those involving means and proportions that will be used in this chapter, a general formula is

$$statistic \pm critical\ value \times standard\ error \qquad (17.2)$$

where *statistic* is the sample statistic under scrutiny, *critical value* is a value from the standardized version of the sampling distribution that corresponds to α, and *standard error* is the standard deviation of the sampling distribution. We refer to the product of the critical value and standard error as the *error component* of the interval; subtraction of the error component from the value of the statistic provides l, and addition provides u.

With reference to the appropriate sampling distribution, a CI yields the two values that mark off the central $100 \times (1 - \alpha)$ percent of the area under the density (precisely the process briefly mentioned in Exercise 17.1(f)). You then use the CI to make further interpretations concerning the true (typically unknown) parameter value that's being estimated by the statistic of interest.

17.2.1 An Interval for a Mean

You know from Section 17.1.1 that the sampling distribution of a single sample mean depends primarily on whether you know the true standard deviation of the raw measurements, σ_X. Then, provided the sample size

for this sample mean is roughly $n > 30$, the CLT ensures a symmetric sampling distribution, which will be normal if you know the true value of σ_X, or t-based with $v = n - 1$ df if you must use the sample standard deviation, s, to estimate σ_X (as is more common in practice). You've seen that the standard error is defined as the standard deviation divided by the square root of n. For a small n, you must also assume that the raw observations are normally distributed, since the CLT will not apply.

To construct an appropriate interval, you must first find the critical value corresponding to α. By definition the CI is symmetric, so this translates to a central probability of $(1 - \alpha)$ around the mean, which leaves exactly $\alpha/2$ in the lower tail and the same in the upper tail.

Recall the example from Section 17.1.1, which dealt with the mean daily maximum temperatures (degrees Celsius) in January in a New Zealand town. Suppose you know the observations are normally distributed but you don't know the true mean μ_X (which is given as 22) or the true standard deviation σ_X (which is given as 1.5). Setting it up in the same way as earlier, generate five random values as follows (remember, your values will be different to those shown here):

```
R> temp.sample <- rnorm(n=5,mean=22,sd=1.5)
R> temp.sample
[1] 20.46097 21.45658 21.06410 20.49367 24.92843
```

Pretend the five values you just generated are your observed dataset. As you're interested in the sample mean and its sampling distribution, you must calculate the sample mean x, the sample standard deviation s, and the appropriate standard error of the sample mean, $s/\sqrt{n}$:

```
R> temp.mean <- mean(temp.sample)
R> temp.mean
[1] 21.68075
R> temp.sd <- sd(temp.sample)
R> temp.sd
[1] 1.862456
R> temp.se <- temp.sd/sqrt(5)
```

```
R> temp.se
[1] 0.8329155
```

Now let's say the aim is to construct a 95 percent confidence interval for the true, unknown mean μ_X. This implies $\alpha = 0.05$ (the total amount of tail probability) for the relevant sampling distribution. Given you know the raw observations are normal and that you're using s (not σ_X), the appropriate distribution is the t-distribution with $n - 1 = 4$ degrees of freedom as per Section 17.1.1. For a central area of 0.95 under this curve, $\alpha/2 = 0.025$ must be in either tail. Knowing that R's `q` functions operate based on a total lower-tail area (refer back to Section 16.2.3 for the t-distribution), you therefore find the (positive) critical value by supplying a probability of $1 - \alpha/2 = 0.975$ to the appropriate function, `qt` in this case:

```
R> 1-0.05/2
[1] 0.975
R> critval <- qt(0.975,df=4)
R> critval
[1] 2.776445
```

Figure 17-3 shows why we use the `qt` function in this way (since I used similar code throughout Chapter 16, I haven't reproduced the code for Figure 17-3 here).

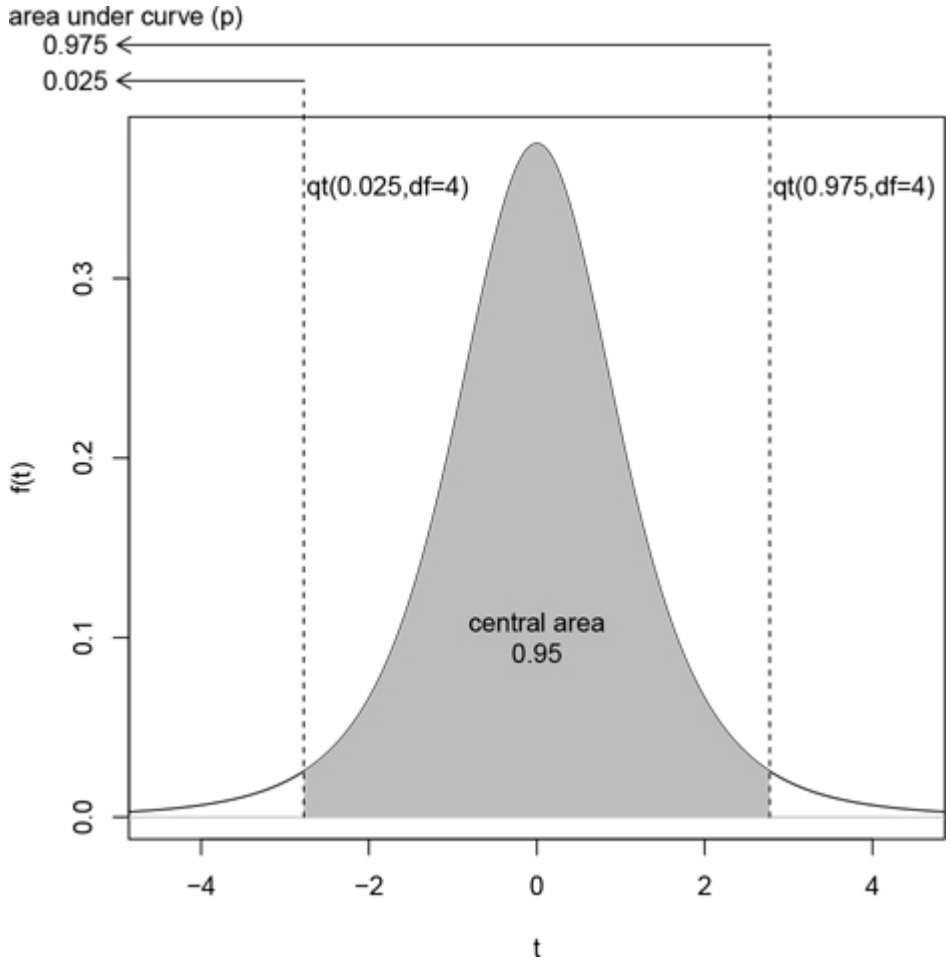


Figure 17-3: Illustrating the role of the critical value in a confidence interval for a sample mean, using the temperature example. The sampling distribution is t *with 4 df, and the use of* `qt` *with respect to symmetric tail probabilities related to* $\alpha/2 = 0.025$ *yields a central area of 0.95.*

Note that when viewed with respect to the negative version of the same critical value ("reflected" around the mean and obtained by using `qt(0.025,4)`), the central area under the curve must be 0.95. You can even confirm this using `pt`:

```
R> pt(critval,4)-pt(-critval,4)
[1] 0.95
```

So, all the ingredients are present. You find the 95 percent confidence interval for the true mean μ_X via Equation (17.2) with the following lines, which give l and u, respectively:

```
R> temp.mean-critval*temp.se
[1] 19.36821
R> temp.mean+critval*temp.se
[1] 23.99329
```

You therefore interpret the CI, given by approximately (19.37, 23.99), as follows: You are 95 percent "confident" that the true mean maximum temperature in Dunedin in January lies somewhere between 19.37 and 23.99 degrees Celsius. With this result, you've combined knowledge of the estimate of the mean itself with the inherent variability of a sample to define an interval of values in which you're fairly sure (albeit not completely sure) that the true mean will lie. As you know, the true mean in this case is 22, which is indeed encapsulated by the calculated CI.

It's easy to alter the intervals to change the confidence levels. You need to change only the critical value, which, as always, must define $\alpha/2$ in each tail. For example, you can find an 80 percent CI ($\alpha = 0.2$) and a 99 percent CI ($\alpha = 0.01$) for the same example value given here with these two lines, respectively:

```
R> temp.mean+c(-1,1)*qt(p=0.9,df=4)*temp.se
[1] 20.40372 22.95778
R> temp.mean+c(-1,1)*qt(p=0.995,df=4)*temp.se
[1] 17.84593 25.51557
```

Note the use of multiplication by the vector `c(-1,1)` so that the lower and upper limits can be obtained at once and the result returned as a vector of length 2. As usual, the `qt` function is used with respect to a complete lowertail area, so it's `p` argument is set at $1 - \alpha/2$.

These two intervals highlight the natural consequence of moving to a higher confidence level for a given CI. A higher probability in the central area translates directly to a more extreme critical value, resulting in a wider interval. This makes sense;in order to be "more confident" about the true parameter value, you'd need to take into account a larger range of possible values.

17.2.2 An Interval for a Proportion

Establishing a CI for a sample proportion follows the same rules as for the sample mean just discussed. With knowledge of the sampling distribution as per Section 17.1.2, you obtain critical values from the standard normal distribution, and for an estimate of $\widehat{p}$ from a sample of size n, the interval itself is constructed with the standard error:

$$\sqrt{\widehat{p}\left(1-\widehat{p}\right)/n}$$

Let's return to the example from Section 17.1.2, where 80 of 118 surveyed individuals said that they knew how they were going to vote in the next US presidential election. Recall you have the following:

```
R> p.hat <- 80/118
R> p.hat
[1] 0.6779661
R> p.se <- sqrt(p.hat*(1-p.hat)/118)
R> p.se
[1] 0.04301439
```

To construct a 90 percent CI ($\alpha = 0.1$), the appropriate critical value from the standardized sampling distribution of interest is as follows, implying $\Pr(-1.644854 < Z < 1.644854) = 0.9$ for $Z \sim N(0, 1)$:

```
R> qnorm(0.95)
[1] 1.644854
```

To obtain the interval, you again simply follow Equation (17.2):

```
R> p.hat+c(-1,1)*qnorm(0.95)*p.se
[1] 0.6072137 0.7487185
```

You thus conclude you're 90 percent confident that the true proportion of voters who know how they will vote in the next election lies somewhere between 0.61 and 0.75 (rounded to two decimal places).

17.2.3 Other Intervals

The two simple situations presented in Sections 17.2.1 and 17.2.2 highlight the importance of associating any estimate of a parameter (in other words, a sample statistic) with the idea of its variability. Confidence intervals can of course be constructed for other quantities, and over the following sections (as part of testing hypotheses), I'll expand on the discussion of confidence intervals to demonstrate how we formally establish statistical evidence of differences between two means and two proportions, as well as ratios of categorical counts. These more complicated statistics come with their own standard error formulas, though the corresponding sampling distributions are still symmetric via the normal and t curves (if, again, some standard assumptions are met), which means that the general formulation of Equation (17.2) still applies.

You can still calculate confidence intervals for quantities with asymmetric sampling distributions, however. These CIs similarly represent a central area of $1 - \alpha$ with respect to the sampling distribution of interest, but it doesn't make much sense to calculate them in a symmetric fashion via a single, standardized critical value like in Equation (17.2). Similarly, the mathematical form of the sampling distribution may not exist in a tractable or parametric fashion, and so you may not be willing to make any strong assumptions about its shape, such as symmetry. In such cases, you can use alternative methods for obtaining or approximating a CI. One of the simplest solutions is based on the raw quantiles (or estimated raw quantiles; see Section 13.2.4) of the supposed asymmetric sampling distribution. Using specific quantile values to mark off identical $\alpha/2$ upper- and lower-tail areas is a valid method that remains sensitive to the shape of the sampling distribution of interest, while still allowing you to construct a useful interval that reflects sampling variability and describes an appropriate range of potential values for the true, unknown parameter of interest.

17.2.4 Comments on the Interpretation of a CI

The typical statement of interpretation of a CI references a degree of confidence in where the true parameter value lies, but a more formal

interpretation considers and clarifies the probabilistic nature of the construction. Technically, given a $100(1 - \alpha)$ percent confidence level, the more accurate interpretation is as follows: Over many samples of the same size and from the same population where a CI, of the same confidence level, is constructed with respect to the same statistic from each sample, you would expect the true corresponding parameter value to fall within the limits of $100(1 - \alpha)$ percent of those intervals.

This comes from the fact that the theory of a sampling distribution describes the variability of a statistic across *multiple samples*, not just the sample that has been taken. At first glance it may be difficult to fully appreciate the difference between this and the colloquial "confidence statement," but it's important to remain aware of the technically correct definition, particularly given a CI is typically calculated from only one sample.

Exercise 17.2

A runner records the average time it takes him to sprint 100 meters. He completes the dash 34 times under identical conditions and finds that the mean of these is 19.22 seconds. Assume that he knows the standard deviation of his runs is $\sigma_X = 2.9$ seconds.

a. Construct and interpret a 90 percent confidence interval for the true mean time.

b. Repeat (a), but this time, assume that the standard deviation is not known and that $s = 2.9$ is estimated from the sample. How, if at all, does this change the interval?

In a particular country, the true proportion of citizens who are left-handed or ambidextrous is unknown. A random sample of 400 people is taken, and each individual is asked to identify with one of three options: right-handed only, left-handed only, or

ambidextrous. The results show that 37 selected left-handed and 11 selected ambidextrous.

c. Calculate and interpret a 99 percent CI for the true proportion of left-handed-only citizens.

d. Calculate and interpret a 99 percent CI for the true proportion of citizens who are either left-handed *or* ambidextrous.

Section 17.2.4 describes the technical interpretation of a CI with respect to its confidence level as the proportion of many similar intervals (that is, when calculated for samples of the same size from the same population) that contain the true value of the parameter of interest.

e. Your task is to write an example to demonstrate this behavior of confidence intervals using simulation. To do so, follow these instructions:

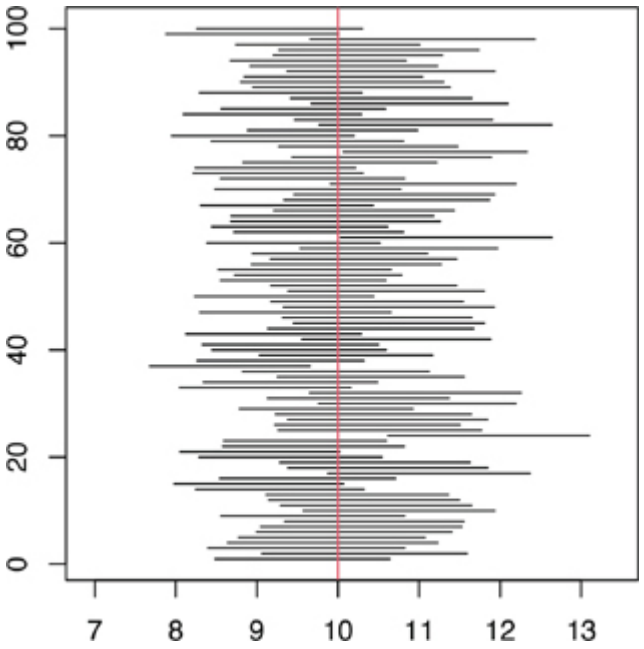
    1. Set up a matrix (see Chapter 3) filled with `NA`s (see Chapter 6) that has 5,000 rows and 3 columns.
    2. Use skills from Chapter 10 to write a `for` loop that, at each of 5,000 iterations, generates a random sample of size 300 from an exponential distribution with rate parameter $\lambda_e = 0.1$ (Section 16.2.4).
    3. Evaluate the sample mean and sample standard deviation of each sample, and use these quantities with the critical values from the appropriate sampling distribution to calculate a 95 percent CI for the true mean of the distribution.
    4. Within the `for` loop, the matrix should now be filled, row by row, with your results. The first column will contain the lower limit, the second will contain the upper limit, and the third will be a

logical value that is `TRUE` if the corresponding interval contains the true mean of $1/\lambda_e$ and that is `FALSE` otherwise.

5. When the loop is completed, compute the proportion of `TRUE`s in the third column of the filled matrix. You should find that this proportion is close to 0.95; this will vary randomly each time you rerun the loop.

f. Create a plot that draws the first 100 of your estimated confidence intervals as separate horizontal lines drawn from their lower limit l to the corresponding upper limit u, one on top of another. One way to do this is to use skills from Sections 7.2 and 7.3 to first create an empty plot with preset x and y-limits (the latter as `c(1,100)`) and then progressively add each line using `lines` (or `segments`) with appropriate coordinates (this could be done using another `for` loop). As a final touch, use `abline` to add a red vertical line at the value of the true mean. Confidence intervals that do not include the true mean will not intersect that vertical line.

The following image provides an example of what this plot should look like:


0
20
40
60
80
100
7
8
9
10
11
12
13

18
HYPOTHESIS TESTING



In this chapter, you'll build on your experience with confidence intervals and sampling distributions to make more formal statements about the value of a true, unknown parameter of interest. For this, you'll learn about frequentist *hypothesis testing*, whereby a probability is drawn from a relevant sampling distribution and used as evidence against some claim about the true value. When using a probability in this way, we refer to it as a *p*-value. In this chapter, you'll interpret results for relatively basic statistics, but you can apply the same concepts to statistics arising from more complicated methods (such as regression modeling in Chapter 19).

18.1 Components of a Hypothesis Test

As an example of hypothesis testing, suppose I told you that 7 percent of a certain population was allergic to peanuts. You then randomly selected 20 individuals from that population and found that 18 of them were allergic to peanuts. Assuming your sample was unbiased and truly

reflective of the population, what would you then think about my claim that the true proportion of allergic individuals is 7 percent?

Naturally, you would doubt the correctness of my claim. In other words, there is such a small probability of observing 18 or more successes out of 20 trials for a set success rate of 0.07 that you can state that you have statistical evidence against the claim that the true rate is 0.07. Recall the definition of the binomial distribution from Section 16.1.2: When defining X as the number of allergic individuals out of 20 by assuming $X \sim \text{BIN}(20, 0.07)$, evaluating $\Pr(X \geq 18)$ gives you the precise p-value, which is tiny:

```
R> dbinom(18,size=20,prob=0.07) + dbinom(19,size=20,prob=0.07) +
   dbinom(20,size=20,prob=0.07)
[1] 2.69727e-19
```

This p-value represents the probability of observing the results in your sample, $X = 18$, or a more extreme outcome ($X = 19$ or $X = 20$), if the chance of success was truly 7 percent.

Before looking at specific hypothesis tests and their implementation in R, the remainder of this section will introduce terminology that you'll come across in the reporting of such tests.

18.1.1 Hypotheses

In hypothesis testing, we formally state a *null* and an *alternative* hypothesis. The null hypothesis is interpreted as the *baseline* or *no-change* hypothesis and, at the outset of the test, is the claim that is assumed to be true. The alternative hypothesis is the conjecture that you're testing for, against the null hypothesis.

In general, we denote the null and alternative hypotheses H_0 and H_A, respectively, and write them as follows:

$$H_0 : \ldots$$

$$H_A : \ldots$$

The null hypothesis is often defined as an equality, =, to a null value. Conversely, the alternative hypothesis (the situation you're testing for) is often defined in terms of an *inequality* to the null value:

- When H_A is defined in terms of a "less-than" statement, with <, it is *one-sided*; this is also called a *lower-tailed test*.
- When H_A is defined in terms of a "greater-than" statement (>), it is *one-sided* or an *upper-tailed test*.
- When H_A is merely defined in terms of a "different-to" statement (≠), it is *two-sided* or a *two-tailed test*.

These test variants are entirely situation specific and depend upon the problem at hand.

18.1.2 Test Statistic

Once the hypotheses are formed, sample data are collected, and statistics are calculated according to the parameters detailed in the hypotheses. You compare the *test statistic* to the appropriate standardized sampling distribution to yield the *p*-value.

A test statistic is typically a standardized or rescaled version of the sample statistic of interest. The distribution and extremity (that is, distance from zero) of the test statistic are the sole drivers of the smallness of the *p*-value (which indicates the strength of the evidence against the null hypothesis; see Section 18.1.3). Specifically, the test statistic is determined both by the difference between the original sample statistic and the null value from H_0 and by the standard error of the sample statistic.

18.1.3 p-Value

The *p*-value is the probability value you use to quantify the amount of evidence against the null hypothesis the sample provides. More formally, the *p*-value is found to be the probability of observing the test

statistic, or something more extreme, assuming the null hypothesis is true.

The exact nature of calculating a *p*-value is dictated by the type of statistic being tested and the nature of H_A. In reference to this, you'll see the following terms:

- A lower-tailed test implies the *p*-value is a left-hand tail probability from the sampling distribution of interest.
- For an upper-tailed test, the *p*-value is a right-hand tail probability.
- For a two-sided test, the *p*-value is the sum of a left-hand tail probability and right-hand tail probability. When the sampling distribution is symmetric (for example, normal or *t*, as in all examples coming up in Sections 18.2 and 18.3), this is equivalent to two times the area in one of those tails.

Put simply, the more extreme the test statistic, the smaller the *p*-value. The smaller the *p*-value, the greater the statistical evidence against H_0 being true.

18.1.4 Significance Level

For every hypothesis test, a *significance level*, denoted α, is assumed. You use this to qualify the result of the test. The significance level defines a cutoff point, at which you decide whether there is sufficient evidence to view H_0 as untrue/incorrect and favor H_A instead:

- If the *p*-value is greater than or equal to α, then you conclude there is insufficient evidence against the null hypothesis, and therefore you *retain* H_0 when compared to H_A.
- If the *p*-value is less than α, then the result of the test is *statistically significant*. This implies there is sufficient evidence against the null hypothesis, and therefore you *reject* H_0 in favor of H_A.

Conventional values of α mirror those used in the construction of confidence intervals (Section 17.2). The most common are $\alpha = 0.1$, $\alpha = 0.05$, and $\alpha = 0.01$.

18.1.5 Criticisms of Hypothesis Testing

The terminology just presented will become easier to understand after looking at some examples in the upcoming sections. However, even at this early stage, it's important to recognize that hypothesis testing is susceptible to justifiable criticism. The end result of any hypothesis test is to either retain or reject the null hypothesis, a decision that is solely dependent upon the rather arbitrary choice of significance level α; this is most often set at one of the conventionally used values.

It is also important to note that a p-value doesn't provide "proof" of either H_0 or H_A being truly correct. It can only ever quantify evidence against the null hypothesis, which one rejects given a sufficiently small p-value $< \alpha$. In other words, rejecting a null hypothesis is not the same as disproving it. Rejecting H_0 merely implies that the sample data suggest H_A ought to be preferred, and the p-value merely indicates the strength of this preference.

In recent years, there has been a push against emphasizing these aspects of statistical inference in some introductory statistics courses owing at least in part to the overuse, and even misuse, of p-values in some areas of applied research. A particularly good article by Sterne & Smith (2001) discusses the role of, and problems surrounding, hypothesis testing from the point of view of medical research. Another good reference is Reinhart (2015), which discusses common misinterpretations of p-values in statistics.

That being said, probabilistic inference with respect to sampling distributions is a cornerstone of frequentist statistical practice. The best way to improve the use and interpretation of statistical tests and modeling is with a sound introduction to the relevant ideas and methods so that, from the outset, you understand the concept of statistical significance and what a hypothesis test can and cannot tell you.

18.2 Testing Means

The validity of hypothesis tests involving sample means is dependent upon the same assumptions and conditions mentioned in Section 17.1.1. In particular, throughout this section you should assume that the central limit theorem holds and, if the sample sizes are small (less than around 30), that the raw data are normally distributed. You'll also focus on examples that use the sample standard deviation s to estimate the true standard deviation, σ_X, because this is the most common situation you'll encounter in practice. Again, mirroring Section 17.1.1, this means you need to use the t-distribution instead of the normal distribution when calculating the critical values and p-values.

18.2.1 Single Mean

As you've already met the standard error formula, $s/\sqrt{n}$, and the R functionality needed to obtain quantiles and probabilities from the t-distribution (`qt` and `pt` as per Section 16.2.3), the only new concepts I'll introduce here relate to the definition of the hypotheses themselves and the interpretation of the result.

Calculation: One-Sample t-Test

Let's dive straight into an example: a one-sample t-test. Recall the problem in Section 16.2.2 where a manufacturer of a snack was interested in the mean net weight of contents in an advertised 80-gram pack. Say that a consumer calls in with a complaint; over time, they have bought and precisely weighed the contents of 44 randomly selected 80-gram packs from different stores and recorded the weights as follows:

```
R> snacks <- c(87.7,80.01,77.28,78.76,81.52,74.2,80.71,79.5,77.87,81.94,80.7,
               82.32,75.78,80.19,83.91,79.4,77.52,77.62,81.4,74.89,82.95,
               73.59,77.92,77.18,79.83,81.23,79.28,78.44,79.01,80.47,76.23,
               78.89,77.14,69.94,78.54,79.7,82.45,77.29,75.52,77.21,75.99,
               81.94,80.41,77.7)
```

The customer claims that they've been shortchanged because their data cannot have arisen from a distribution with mean $\mu = 80$, so the true mean weight must be less than 80. To investigate this claim, the manufacturer conducts a hypothesis test using a significance level of $\alpha = 0.05$.

First, the hypotheses must be defined, with a null value of 80 grams. Remember, the alternative hypothesis is "what you're testing for"; in this case, $\mathrm{H_A}$ is that μ is smaller than 80. The "no change" null hypothesis is given as $\mu = 80$; this directly states that the true mean is in fact 80 grams. You formalize these hypotheses like so:

$$\begin{aligned} \mathrm{H}_0&: \quad \mu = 80 \\ \mathrm{H}_\mathrm{A}&: \quad \mu < 80 \end{aligned} \qquad (18.1)$$

Second, you must estimate the mean and standard deviation from the sample:

```
R> n <- length(snacks)
R> snack.mean <- mean(snacks)
R> snack.mean
[1] 78.91068
R> snack.sd <- sd(snacks)
R> snack.sd
[1] 3.056023
```

Your hypothesis test will be based on the following interpretation: Given the estimated standard deviation, what's the probability of observing a sample mean (when $n = 44$) of 78.91 grams or less if the true mean is 80 grams? To answer this question, you need to calculate the relevant test statistic.

Formally, the test statistic T in a hypothesis test for a single mean with respect to a null value of μ_0 is given as

$$T = \frac{x - \mu_0}{(s/\sqrt{n})} \qquad (18.2)$$

based on a sample of size n, a sample mean of $\bar{x}$, and a sample standard deviation of s (the denominator is the estimated standard error of the mean as per Section 17.1.1). Under the aforementioned assumptions, T follows a t-distribution with $v = n - 1$ degrees of freedom.

In R, the following code provides you with the standard error of the sample mean for the snacks data:

```
R> snack.se <- snack.sd/sqrt(n)
R> snack.se
[1] 0.4607128
```

Then, you can calculate T as follows:

```
R> snack.T <- (snack.mean-80)/snack.se
R> snack.T
[1] -2.364419
```

Finally, use the test statistic to obtain the p-value. Recall that the p-value is the probability that you observe T or something more extreme. The nature of "more extreme" is determined by the alternative hypothesis H_A, which, as a less-than statement in this case, directs you to find a left-hand, lower-tail probability as the p-value. In other words, the p-value is provided as the area under the sampling distribution (here, a t-distribution with 43 df) to the left of a vertical line at T. You can do this with `pt` (see Section 16.2.3):

```
R> pt(snack.T,df=n-1)
[1] 0.01132175
```

Your result states that if the H_0 were true, there would be only a little more than a 1 percent chance that you'd observe the customer's sample mean of $\bar{x} = 78.91$, or less, as a random phenomenon. Since this p-value is smaller than the predefined significance level of $\alpha = 0.05$, the manufacturer concludes that there is sufficient evidence to reject the null hypothesis in favor of the alternative, lending weight to the conjecture that the true value of μ is in fact less than 80 grams.

In examining the result, with a p-value of around 0.011, remember to be careful when interpreting hypothesis tests. Given that the significance level, α, was chosen to be 0.05 for this particular test, H_0 is rejected. But what if the test had been carried out with $\alpha = 0.01$? The p-value is greater than 0.01, so in that case, H_0 would be retained, for no reason other than the arbitrary choice of α. In these situations, it's helpful to comment on the perceived strength of the evidence against the null hypothesis. For the current example, you could reasonably state that there exists some evidence to support H_A, but this evidence is not especially strong.

R Function: t.test

You can perform the one-sample t-test directly in R with the built-in `t.test` function. The following output confirms the result of the snacks example:

```
R> t.test(x=snacks,mu=80,alternative="less")

   One Sample t-test

data:  snacks
t = -2.3644, df = 43, p-value = 0.01132
alternative hypothesis: true mean is less than 80
95 percent confidence interval:
     -Inf 79.68517
sample estimates:
mean of x
 78.91068
```

The function takes the raw data vector as `x`, the null value for the mean as `mu`, and the direction of the test (in other words, how to find the p-value under the appropriate t-curve) as `alternative`. The `alternative` argument has three available options: `"less"` for H_A with $<$; `"greater"` for H_A with $>$; and `"two.sided"` for H_A with $\neq$. The default value of α is 0.05.

If you want a different significance level than 0.05, you must provide this to `t.test` as $1 - \alpha$, passed to the argument `conf.level`.

Note that the output of `t.test` reports the value of T, as well as the degrees of freedom and the p-value. You also get a 95 percent "interval," but its values of `-Inf` and `79.68517` might look a little strange: The lower limit is negatively infinite! It turns out that when you do a one-sided hypothesis test in this way, the reported CI in the `t.test` is actually a *one-sided confidence bound*, the specific nature of which is dictated by the `alternative` argument. For a lower-tailed test, it simply provides an upper bound on the statistic such that the entire lower-tail area of the sampling distribution of interest is 0.95, as opposed to a *central* area as the traditional CI you learned about in Section 17.2.4.

One-sided bounds are less common than the fully bounded two-sided interval; to obtain the latter (as the component `conf.int`) from a relevant call to `t.test`, set `alternative="two.sided"`:

```
R> t.test(x=snacks,mu=80,alternative="two.sided")$conf.int
[1] 77.98157 79.83980
attr(,"conf.level")
[1] 0.95
```

Note that the corresponding confidence level, $1 - \alpha$, is stored alongside this component as an attribute (refer to Section 6.2.1). To check the result, calculate the CI "manually" by following the recipe outlined in Section 17.2.1:

```
R> snack.mean+c(-1,1)*qt(0.975,n-1)*snack.se
[1] 77.98157 79.83980
```

Notice that it does *not* include the null value of 80; this will always reflect the result of the corresponding two-sided hypothesis test performed on the same sample with the same value of α. In this case it's also consistent with the conclusion reached by our one-sided test.

Exercise 18.1

a. Adult domestic cats of a certain breed are said to have an average weight of 3.5 kilograms. A feline enthusiast disagrees and collects a sample of 73 weights of cats of this breed. From her sample, she calculates a mean of 3.97 kilograms and a standard deviation of 2.21 kilograms. Perform a hypothesis test of her claim that the true mean weight μ is *not* 3.5 kilograms by setting up the appropriate hypothesis, carrying out the analysis, and interpreting the p-value (assume the significance level is $\alpha = 0.05$).

b. Suppose it was previously believed that the mean magnitude of seismic events off the coast of Fiji is 4.3 on the Richter scale. Use the data in the `mag` variable of the ready-to-use `quakes` dataset, providing 1,000 sampled seismic events in that area, to test the claim that the true mean magnitude is in fact *greater* than 4.3. Set up appropriate hypotheses, use `t.test` (conduct the test at a significance level of $\alpha = 0.01$), and draw a conclusion.

c. Manually compute a two-sided 99 percent confidence interval for the true mean of (b).

18.2.2 Two Means

Testing a single sample mean often isn't enough to answer the question you're interested in. In many settings, a researcher wants to directly compare the means of two distinct groups of measurements, which boils down to a hypothesis test for the true difference between two means; call them μ_1 and μ_2.

The way in which two groups of data relate to each other affects the specific form of standard error for the difference between two sample

means and therefore the test statistic itself. The actual comparison of the two means, however, is often of the same nature: The typical null hypothesis is usually defined as μ_1 and μ_2 being equal. In other words, the null value of the difference between the two means is often simply zero.

Unpaired/Independent Samples: Unpooled Variances

In the most general case, the two sets of measurements are based on two independent, separate groups (also referred to as *unpaired* samples). You compute the sample means and sample standard deviations of both datasets, define the hypotheses of interest, and then calculate the test statistic.

When you cannot assume the variances of the two populations are equal, then you perform the *unpooled* version of the two-sample *t*-test; I'll discuss this first. If, however, you can safely assume equal variances, then you can perform a *pooled* two-sample *t*-test, which improves the precision of the results. You'll look at the pooled version of the test in a moment.

For an unpooled example, return to the 80-gram snack packet example from Section 18.2.1. After collecting a sample of 44 packs from the original manufacturer (label this sample size n_1), the disgruntled consumer goes out and collects $n_2 = 31$ randomly selected 80-gram packs from a rival snack manufacturer. This second set of measurements is stored as `snacks2`:

```
R> snacks2 <- c(80.22,79.73,81.1,78.76,82.03,81.66,80.97,81.32,80.12,78.98,
                79.21,81.48,79.86,81.06,77.96,80.73,80.34,80.01,81.82,79.3,
                79.08,79.47,78.98,80.87,82.24,77.22,80.03,79.2,80.95,79.17,81)
```

You already found the mean and standard deviation of the first sample of size $n_1 = 44$ (computed in Section 18.2.1). These are stored as `snack.mean` (around 78.91) and `snack.sd` (around 3.06), respectively; think of these as x_1 and s_1. Compute the same quantities, x_2 and s_2, respectively, for the new data:

```
R> snack2.mean <- mean(snacks2)
R> snack2.mean
[1] 80.1571
R> snack2.sd <- sd(snacks2)
R> snack2.sd
[1] 1.213695
```

Denote the true mean of the original sample with μ_1 and the true mean of the new sample from the rival company's packs with μ_2. You're now interested in testing whether there is statistical evidence to support the claim that μ_2 is greater than μ_1. This suggests the hypotheses of $H_0 : \mu_1 = \mu_2$ and $H_A : \mu_1 < \mu_2$, which you can write as follows:

$$H_0 : \mu_2 - \mu_1 = 0$$

$$H_A : \mu_2 - \mu_1 > 0$$

That is, the difference between the true mean of the rival company packs and the original manufacturer's packs, when the original is subtracted from the rival, is bigger than zero. The "no-change" scenario, the null hypothesis, is that the two means are the same, or equivalently, that their difference is zero.

Now that you've constructed the hypotheses, let's look at how to actually test them. The difference between two means is the quantity of interest. For two independent samples arising from populations with true means μ_1 and μ_2, sample means x_1 and x_2, and sample standard deviations s_1 and s_2, respectively (and that meet the relevant conditions for the validity of the *t*-distribution), the standardized test statistic T for testing the difference between μ_2 and μ_1, in that order, is given as:

$$T = \frac{x_2 - x_1 - \mu_0}{\sqrt{s_1^2/n_1 + s_2^2/n_2}} \qquad (18.3)$$

Here, μ_0 is the null value of interest. For the current snacks example (and as is usually the case for tests of these kinds of "difference" statistics), $\mu_0 = 0$, so this term would disappear from the numerator of Equation (18.3). The denominator of T is the standard error of the difference between two means for independent groups.

The sampling distribution of T is approximated by a t-distribution with ν degrees of freedom, where:

$$\nu = \frac{\left(s_1^2/n_1 + s_2^2/n_2\right)^2}{\left(s_1^2/n_1\right)^2/(n_1-1) + \left(s_2^2/n_2\right)^2/(n_2-1)} \qquad (18.4)$$

Sometimes you may see ν rounded down to the nearest integer, but this isn't necessary in most modern software implementations (R included).

NOTE

This two-sample t*-test, conducted using Equation (18.3), is also called* Welch's t*-test. This refers to use of Equation (18.4), called the* Welch–Satterthwaite equation. *Crucially, it assumes that the two samples have different true variances, which is why it's referred to as the* unpooled variance *version of the test.*

It's important to be consistent when defining your two sets of parameters and when constructing the hypotheses. In this example, since the test aims to find evidence for μ_2 being greater than μ_1, a difference of $\mu_2 - \mu_1 > 0$ forms H_A (a greater-than, upper-tailed test), and this order of subtraction is mirrored when calculating T. The same test could be carried out if you defined the difference the other way around. In that case, your alternative hypothesis would suggest a lower-tailed test because if you're testing for μ_2 being bigger than μ_1, H_A would correctly be written as $\mu_1 - \mu_2 < 0$. Again, this would modify the order of subtraction in the numerator of Equation (18.3) accordingly.

The same care must apply to the use of `t.test` for two-sample comparisons. You supply the two samples as the arguments `x` and `y`, but the function interprets `x` as greater than `y` when doing an upper-tailed test and interprets `x` as less than `y` when doing a lower-tailed test. Therefore, when performing the test with `alternative="greater"` for the snack pack example, you must supply `snacks2` to `x`:

```
R> t.test(x=snacks2,y=snacks,alternative="greater",conf.level=0.9)

        Welch Two Sample t-test

data:  snacks2 and snacks
t = 2.4455, df = 60.091, p-value = 0.008706
alternative hypothesis: true difference in means is greater than 0
90 percent confidence interval:
 0.5859714       Inf
sample estimates:
mean of x mean of y
 80.15710  78.91068
```

With a small *p*-value of 0.008706, you'd conclude that there is sufficient evidence to reject H_0 in favor of H_A (indeed, the *p*-value is certainly smaller than the stipulated $\alpha = 0.1$ significance level as implied by `conf.level=0.9`). The evidence suggests that the mean net weight of snacks from the rival manufacturer's 80-gram packs is greater than the mean net weight for the original manufacturer.

Note that the output from `t.test` has reported a df value of 60.091, which is the result of Equation (18.4). You also receive a one-sided confidence bound (based on the aforementioned confidence level), triggered by the one-sided nature of this test. Again, the more common two-sided 90 percent interval is also useful; knowing that $v = 60.091$ and using the statistic and the standard error of interest (numerator and denominator of Equation (18.3), respectively), you can calculate it as follows:

```
R> (snack2.mean-snack.mean) +
      c(-1,1)*qt(0.95,df=60.091)*sqrt(snack.sd^2/44+snack2.sd^2/31)
[1] 0.3949381 2.0978918
```

Here, you've used the previously stored sample statistics `snack.mean`, `snack.sd` (the mean and standard deviation of the 44 raw measurements from the original manufacturer's sample), `snack2.mean`, and `snack2.sd` (the same quantities for the 31 observations corresponding to the rival manufacturer). Note that the CI takes the same form as detailed by Equation (17.2) on page 446 and that to provide the correct $1 - \alpha$ central area, the `q`-function for the appropriate *t*-distribution requires $1 - \alpha/2$ as its supplied probability value. You can interpret this as being "90 percent confident that the true difference in mean net weight between the rival and the original manufacturer (in that order) is somewhere between 0.395 and 2.098 grams." The fact that zero isn't included in the interval, and that the interval is wholly positive, supports the conclusion from the hypothesis test.

Unpaired/Independent Samples: Pooled Variance

In the unpooled variance example just passed, there was no assumption that the variances of the two populations whose means were being compared were equal. This is an important note to make because it leads to the use of Equation (18.3) for the test statistic calculation and Equation (18.4) for the associated degrees of freedom in the corresponding *t*-distribution. However, if you *can* assume equivalence of variances, the precision of the test will improve: You use a different formula for the standard error of the difference and for calculating the associated df.

Again, the quantity of interest is the difference between two means, written as $\mu_2 - \mu_1$. Assume you have two independent samples of sizes n_1 and n_2 arising from populations with true means μ_1 and μ_2, sample means x_1 and x_2, and sample standard deviations s_1 and s_2, respectively, and assume that the relevant conditions for the validity of the *t*-

distribution have been met. Additionally, assume that the true variances of the samples, σ_1^2 and σ_2^2, are equal such that $\sigma_p^2 = \sigma_1^2 = \sigma_2^2$.

NOTE

There's a simple rule of thumb to check the validity of the "equal variance" assumption. If the ratio of the larger sample standard deviation to the smaller sample standard deviation is less than 2, you can assume equal variances. For example, if $s_1 > s_2$, *then if* $\frac{s_1}{s_2} < 2$, *you can use the pooled variance test statistic that follows.*

The standardized test statistic T for this scenario is given as

$$T = \frac{x_2 - x_1 - \mu_0}{\sqrt{s_p^2(1/n_1 + 1/n_2)}} \qquad (18.5)$$

whose distribution is a t-distribution with $v = n_1 + n_2 - 2$ degrees of freedom, where

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2} \qquad (18.6)$$

is the *pooled estimate of the variance* of all the raw measurements. This is substituted in place of s_1 and s_2 in the denominator of Equation (18.3), resulting in Equation (18.5).

All other aspects of the two-sample t-test remain as earlier, including the construction of appropriate hypotheses, the typical null value of μ_0, and the calculation and interpretation of the p-value.

For the comparison of the two means in the snack pack example, you'd find it difficult to justify using the pooled version of the t-test. Applying the rule of thumb, the two estimated standard deviations ($s_1 \approxeq 3.06$ and $s_2 \approxeq 1.21$ for the original and rival manufacturer's samples, respectively) have a large-to-small ratio that is greater than 2:

```
R> snack.sd/snack2.sd
[1] 2.51795
```

Although this is a pretty informal check, if the assumption cannot reasonably be made, it's best to stick with the unpooled version of the test.

Let's therefore consider a new example to illustrate the pooled two-sample *t*-test. The intelligence quotient (IQ) is a quantity commonly used to measure how clever a person is. IQ scores are reasonably assumed to be normally distributed, and the average IQ of the adult population is said to be 100. Suppose a sociologist is interested in assessing whether there is a difference in mean IQ scores between men and women of a certain country, suggesting the following hypotheses:

$$H_0 : \mu_{\text{men}} - \mu_{\text{women}} = 0$$

$$H_A : \mu_{\text{men}} - \mu_{\text{women}} \neq 0$$

The researcher randomly samples the population and obtains the following data, with sub-sample sizes $n_{\text{men}} = 12$ and $n_{\text{women}} = 20$:

```
R> men <- c(102,87,101,96,107,101,91,85,108,67,85,82)
R> women <- c(73,81,111,109,143,95,92,120,93,89,119,79,90,126,62,92,77,106,
              105,111)
```

To test the hypotheses, let's first calculate the basic statistics required:

```
R> mean(men)
[1] 92.66667
R> sd(men)
[1] 12.0705
R> mean(women)
[1] 98.65
R> sd(women)
[1] 19.94802
```

These give the sample averages x_{men} and x_{women}, as well as their respective sample standard deviations s_{men} and s_{women}. Use the aforementioned rule-of-thumb to quickly check the ratio of the standard deviations:

```
R> sd(women)/sd(men)
[1] 1.652626
```

You can see that the ratio of the larger sample standard deviation to the smaller is less than 2, so the sociologist could assume equal variances in carrying out the hypothesis test.

As per Equations (18.5) and (18.6), the `t.test` command enables the pooled two-sample *t*-test when you set the optional argument `var.equal=TRUE` (as opposed to the default `var.equal=FALSE`, which triggers Welch's *t*-test as we saw with the snack pack example):

```
R> t.test(x=men,y=women,alternative="two.sided",conf.level=0.95,var.equal=TRUE)

   Two Sample t-test

data:  men and women
t = -0.9376, df = 30, p-value = 0.3559
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -19.016393   7.049727
sample estimates:
mean of x mean of y
 92.66667  98.65000
```

Note also that H_A for this example implies a two-tailed test, hence the provision of `alternative="two.sided"`. (This is the default value for `alternative`; I've written it out in calling `t.test` here only for the sake of exposition.) For these tests, you find the *p*-value by multiplying the area in the extreme of the *t*-distribution (as demarcated by the test statistic *T*) by 2, as mentioned in Section 18.1.3. This works because the sampling distribution is symmetric and is why you'll often see a two-tailed *p*-value calculated as twice the upper-tail area of the absolute

value (see Section 13.2.3) of the test statistic. Thereby, you may confirm the resulting *p*-value shown in the output, 0.3559, with the following line:

```
R> 2*(1-pt(0.9376,df=30))
[1] 0.3559375
```

To explain this further, note that 0.9376 is the absolute value of the reported test statistic –0.9376; that the df is reported as 30; that you use a "one-minus" operation with `pt` (see Section 16.2.3) to obtain a right-hand, upper-tail probability; and finally, that multiplication of this upper tail by 2 gives you the correct total two-tailed *p*-value reflective of the ≠ in H_A.

The resulting *p*-value of this test is certainly larger than the conventional cutoff level of 0.05. The sociologist's conclusion here is that there is no evidence to reject H_0: There is insufficient evidence to support a true difference in the mean IQ scores of men compared to women.

Paired/Dependent Samples

Lastly, we'll look comparing groups where the data are *paired*. This setting is often confused with—but is different from—the setting for the unpaired *t*-test. It is used when the observations making up the two groups are inherently related, accounting for assumed *dependence* between pairs of observations across the two groups. This notion has important consequences.

Paired data occur if the measurements forming the two sets of observations are recorded on the same individual, or if they are related in some other important or obvious way. A classic example of this is "before" and "after" observations, such as two measurements made on each person before and after some kind of intervention. Rather than working with the two datasets separately, a paired *t*-test works with a single mean: the true mean of the individual paired differences μ_d. As such, we must think of this test as a test examining the *mean difference*

value (as opposed to the *difference between means* explicitly tested in the unpaired *t*-tests considered thus far).

As an example, consider a company interested in the efficacy of a drug designed to reduce resting heart rates in beats per minute (bpm). The resting heart rates of 16 individuals are measured. The individuals are then administered a course of the treatment, and their resting heart rates are again measured. The data are provided in the two vectors `rate.before` and `rate.after` as follows:

```
R> rate.before <- c(52,66,89,87,89,72,66,65,49,62,70,52,75,63,65,61)
R> rate.after <- c(51,66,71,73,70,68,60,51,40,57,65,53,64,56,60,59)
```

It quickly becomes clear why any test comparing these two groups of observations must take some kind of individual-level dependence into account. Heart rate is affected by an individual's age, build, and level of physical fitness, among other factors. An unfit individual older than 60 years of age is likely to have a higher baseline resting heart rate than a fit 20-year-old, and if both are given the same drug to lower their heart rate, their final heart rates are still likely to reflect the baselines. Any true effect of the drug therefore has the potential to be hidden if you approached the analysis using either of the unpaired *t*-tests.

To overcome this problem, the paired two-sample *t*-test uses the raw pairwise differences between each pair of values. Labeling one set of n measurements as $x_1, \ldots, x_n$ and the other set of n observations as $y_1, \ldots, y_n$, the differences are defined as $d_i = y_i - x_i$; $i = 1, \ldots, n$. In R, you can easily compute them using the language's natural element-wise behavior:

```
R> rate.d <- rate.after-rate.before
R> rate.d
[1]  -1   0 -18 -14 -19  -4  -6 -14  -9  -5  -5   1 -11  -7  -5  -2
```

The following code calculates the sample mean $\bar{d}$ and standard deviation s_d of these differences:

```
R> rate.dbar <- mean(rate.d)
R> rate.dbar
[1] -7.4375
R> rate.sd <- sd(rate.d)
R> rate.sd
[1] 6.196437
```

You want to see how much the heart rate is reduced by, so the test at hand will focus on the following hypotheses:

$$\mathrm{H}_0 : \mu_d = 0$$

$$\mathrm{H}_\mathrm{A} : \mu_d < 0$$

Given the order or subtraction used to obtain the differences, detection of a successful reduction in heart rate will be represented by an "after" mean that is smaller than the "before" mean. This would be represented by a negative μ_d, which is reflected in the expression of H_A.

Expressing all this mathematically, note again there are two sets of n measurements, $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, with pairwise differences $d_1, \ldots, d_n$. The relevant conditions for the validity of the t-distribution must be met; in this case, if the number of pairs n is less than 30, then you must be able to assume the raw data are normally distributed. The test statistic T is given as

$$T = \frac{d - \mu_0}{s_d / \sqrt{n}} \qquad (18.7)$$

where d is the mean of the pairwise differences, s_d is the sample standard deviation of the pairwise differences, and μ_0 is the null value (usually zero). The statistic T follows a t-distribution with $n - 1$ df.

The form of Equation (18.7) is actually the same as the form of the test statistic in (18.2), once the sample statistics for the individual paired differences have been calculated. Furthermore, it's important to note

that *n* represents the total number of *pairs*, not the total number of individual observations.

For the current set of hypotheses, you can find the test statistic and *p*-value using `rate.dbar` and `rate.sd`:

```
R> rate.T <- rate.dbar/(rate.sd/sqrt(16))
R> rate.T
[1] -4.801146
R> pt(rate.T,df=15)
[1] 0.000116681
```

These results suggest evidence to reject H_0.

Confirm these results via `t.test`, wherein you must set the optional logical argument `paired` to `TRUE`:

```
R> t.test(x=rate.after,y=rate.before,alternative="less",conf.level=0.95,
          paired=TRUE)

    Paired t-test

data:  rate.after and rate.before
t = -4.8011, df = 15, p-value = 0.0001167
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -4.721833
sample estimates:
mean of the differences
                -7.4375
```

Note that the order in which you supply your data vectors to the `x` and `y` arguments follows the same rules as for the unpaired tests, given the desired value of `alternative`. The same *p*-value as was calculated manually is confirmed through the use of `t.test`, and since this is less than an assumed conventional significance level of $\alpha = 0.05$, a valid conclusion would be to state that there is statistical evidence that the medication does reduce the mean resting heart rate. You could go on to say you're

95 percent confident that the true mean difference in heart rate after taking the course of medication lies somewhere between

```
R> rate.dbar-qt(0.975,df=15)*(rate.sd/sqrt(16))
[1] -10.73935
```

and

```
R> rate.dbar+qt(0.975,df=15)*(rate.sd/sqrt(16)),
[1] -4.135652
```

which, as noted previously, would reflect the results of a two-tailed test (had we tested the more general $H_A : \mu_d \neq 0$).

NOTE

On some occasions, such as when your data strongly indicate non-normality, you may not be comfortable assuming the validity of the CLT (see Section 17.1.1). An alternative approach to the tests discussed here is to employ a nonparametric *technique that relaxes these distributional requirements. In the two-sample case, you could employ the* Mann–Whitney U test *(also known as the* Wilcoxon rank-sum test*). This is a hypothesis test that can be interpreted as comparing two medians, as opposed to two means. You can use the R function* `wilcox.test` *for this purpose; its help page provides useful commentary and references on the particulars of the technique. A good introductory text on the methodology of various nonparametric tests is that of Corder & Foreman (2014).*

Exercise 18.2

In the package `MASS` you'll find the dataset `anorexia`, which contains data on pre- and post-treatment weights (in pounds) of 72 young women suffering from the disease, obtained from Hand et al. (1994). One

group of women is the control group (in other words, no intervention), and the other two groups are the cognitive behavioral program and family support intervention program groups. Load the `MASS` package and ensure you can access the data frame and understand its contents. Let μ_d denote the true mean difference in weight, computed as (*post-weight* – *pre-weight*).

a. Regardless of which group the participants fall into, conduct and conclude an appropriate hypothesis test with $\alpha = 0.05$ for the entire set of weights for the following hypotheses:

$$H_0 : \mu_d = 0$$

$$H_A : \mu_d > 0$$

b. Next, conduct three separate hypothesis tests using the same defined hypotheses, based on which treatment group the participants fall into. What do you notice?

Another ready-to-use dataset in R is `PlantGrowth` (Dobson 1983), which records a continuous measure of the yields of a certain plant, looking at the potential effect of two supplements administered during growth to increase the yield when compared to a control group with no supplement.

c. Create two different objects for the yield weights: those corresponding to plants in the control group and those corresponding to plants that were treated with either supplement. Set up hypotheses to test whether the mean yield for the control group is less than the mean yield from a plant given either of the treatments. Determine whether this test should proceed using a pooled estimate of the variance or whether Welch's *t*-test would be more appropriate.

d. Conduct the test and make a conclusion (assuming normality of the raw observations).

As discussed in the text, there is a rule of thumb for deciding whether to use a pooled estimate of the variance in an unpaired *t*-test. This will be the focus of the remaining task.

e. Your job is to write a *wrapper* function that calls `t.test` after deciding whether it should be executed with `var.equal=FALSE` according to the rule of thumb. Use the following guidelines:

    - Your function should take four named arguments: `x` and `y` with no defaults, to be treated in the same way as the same arguments in `t.test`; and `var.equal` and `paired`, with defaults that are the same as the defaults of `t.test`.
    - You should include an ellipsis (Section 9.2.5) to represent any additional arguments to be directly passed to the internal call to `t.test`.
    - Upon execution, the function should assess whether `paired=FALSE`:
        - If `paired` is `TRUE`, then there is no need to proceed with the check of a pooled variance; your function can proceed to directly calling `t.test`.
        - If `paired` is `FALSE`, then the function should determine the value for `var.equal` automatically by using the rule of thumb.
    - If the value of `var.equal` was set automatically, you can assume it will override any value of this argument initially supplied by the user.

- After these checks and operations, then call `t.test` appropriately.

f. Try your new function on all three examples in the text of Section 18.2.2, ensuring you reach identical results.

18.3 Testing Proportions

A focus on means is especially common in statistical modeling and hypothesis testing, and therefore you must also consider sample proportions, interpreted as the mean of a series of n binary trials, in which the results are success (1) or failure (0). This section focuses on the parametric tests of proportions, which assume normality of the target sampling distributions and are sometimes referred to as Z-tests.

The general rules regarding the setup and interpretation of hypothesis tests for sample proportions remain the same as for sample means. In this introduction to Z-tests, you can consider these as tests regarding the true value of a single proportion or the difference between two proportions.

18.3.1 Single Proportion

Section 17.1.2 introduced the sampling distribution of a single sample proportion to be normally distributed, with a mean centered on the true proportion π and with a standard error of $\sqrt{\pi(1-\pi)/n}$. Provided the trials are independent, that n isn't "too small" and that π isn't "too close" to 0 or 1, those formulas are applicable here.

NOTE

A rule of thumb to check the latter condition on n *and* π *simply involves checking that* $\mathrm{n}\hat{\mathrm{p}}$ *and* $\mathrm{n}(1-\hat{\mathrm{p}})$ *are both greater than 5, where* $\hat{\mathrm{p}}$ *is the sample estimate of* π.

It's worth noting that the standard error in the case of hypothesis tests involving proportions is itself dependent upon π. This is important; remember that any hypothesis test assumes satisfaction of H_0 in the relevant calculations. In dealing with proportions, that means when computing the test statistic, the standard error ought to make use of the null value π_0 rather than the estimated sample proportion $\widehat{p}$.

I'll clarify this in an example. Suppose an individual fond of a particular fast-food chain notices that he tends to have an upset stomach within a certain amount of time after having his usual lunch. He comes across the website of a blogger who believes that the chance of getting an upset stomach shortly after eating that particular food is 20 percent. The individual is curious to determine whether his true rate of stomach upset π is any different from the blogger's quoted value and, over time, visits these fast-food outlets for lunch on $n = 29$ separate occasions, recording the success (TRUE) or failure (FALSE) of experiencing an upset stomach. This suggests the following pair of hypotheses, which will be used to illustrate calculations in the remainder of this section:

$$H_0 : \pi = 0.2$$

$$H_A : \pi \neq 0.2$$

Calculation: One-Sample Z-Test

In testing for the true value of some proportion of success, π, let $\widehat{p}$ be the sample proportion observed over n trials, and let the null value be denoted with π_0. You find the test statistic with the following

$$Z = \frac{\widehat{p} - \pi_0}{\sqrt{\frac{\pi_0(1-\pi_0)}{n}}} \qquad (18.8)$$

provided the aforementioned conditions on the size of n and the value of π can be assumed, $Z \sim N(0, 1)$.

The denominator of Equation (18.8), the standard error of the proportion, is calculated with respect to the null value π_0, not $\widehat{p}$. As mentioned, this is to satisfy the assumption of "truth" of H_0 as the test is carried out, so it allows interpretation of the resulting p-value as usual. The standard normal distribution is used to find the p-value with respect to Z; the direction underneath this curve given Z is governed by the nature of H_A just as before.

Getting back to the fast-food example, suppose these are the observed data, where `1` is recorded for an upset stomach and is `0` otherwise:

```
sick <- c(0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0,0,1)
```

The number of successes and probability of success in this sample are therefore as follows:

```
R> sum(sick)
[1] 8
R> p.hat <- mean(sick)
R> p.hat
[1] 0.2758621
```

A quick check indicates that as per the rule of thumb, the test is reasonable to carry out:

```
R> 29*0.2
[1] 5.8
R> 29*0.8
[1] 23.2
```

After Equation (18.8), the test statistic Z for this example is as follows:

```
R> Z <- (p.hat-0.2)/sqrt(0.2*0.8/29)
R> Z
[1] 1.021324
```

The alternative hypothesis is two-sided, so you compute the corresponding *p*-value as a two-tailed area under the standard normal curve. With a positive test statistic, you can evaluate this by doubling the upper-tailed area from *Z*:

```
R> 2*(1-pnorm(Z))
[1] 0.3071008
```

Assume a conventional α level of 0.05. The high *p*-value given as 0.307 suggests the results in the sample of size 29 are not unusual enough, under the assumption that the null hypothesis is true, to reject H_0. There is insufficient evidence to suggest that the proportion of instances of an upset stomach that this individual experiences is any different from 0.2 as noted by the blogger.

You can support this conclusion with a confidence interval. At the level of 95 percent, you can calculate the CI with the following line (done so here with the raw sample proportion used to compute the standard error, not the null value), which we see includes the null value of 0.2:

```
R> p.hat+c(-1,1)*qnorm(0.975)*sqrt(p.hat*(1-p.hat)/29)
[1] 0.1131927 0.4385314
```

R Function: prop.test

Once more, R rescues you from tedious step-by-step calculations. The ready-to-use `prop.test` function allows you to perform, among other things, a single sample proportion test. The function actually performs the test in a slightly different way under the hood, using the chi-squared distribution (see Section 16.2.5). However, the test is equivalent, and the resulting *p*-value from `prop.test` is identical to the one reached using the *Z*-based test.

To the `prop.test` function, as used for a single sample test of a proportion, you provide the number of successes observed as `x`, the total number of trials as `n`, and the null value as `p`. The two further

arguments, `alternative` (defining the nature of H_A) and `conf.level` (defining $1 - \alpha$), are identical to the same arguments in `t.test` and have defaults of `"two.sided"` and `0.95`, respectively. Lastly, it's sensible to explicitly set the optional argument `correct=FALSE` if your data satisfy the $n\widehat{p}$ and $n\left(1-\widehat{p}\right)$ rule of thumb.

For our running example, you perform the test with this code:

```
R> prop.test(x=sum(sick),n=length(sick),p=0.2,correct=FALSE)

        1-sample proportions test without continuity correction

data:  sum(sick) out of length(sick), null probability 0.2
X-squared = 1.0431, df = 1, p-value = 0.3071
alternative hypothesis: true p is not equal to 0.2
95 percent confidence interval:
 0.1469876 0.4571713
sample estimates:
        p
0.2758621
```

The *p*-value is the same as you got earlier. Note, however, that the reported CI is not quite the same (the normal-based interval, dependent upon the CLT). The CI produced by `prop.test` is referred to as the *Wilson score interval*, which takes into account the direct association that a "probability of success" has with the binomial distribution (see Section 16.1.2). For simplicity, you'll continue to work with normal-based intervals when performing hypothesis tests involving proportions in this chapter.

Also note that, just like `t.test`, any one-sided test performed with `prop.test` will provide only a single-limit confidence bound; you'll see this in the following example.

18.3.2 Two Proportions

With a basic extension to the previous procedure, by way of a modification to the standard error, you can compare *two* estimated

proportions from independent populations. As with the difference between two means, you're often testing a null hypothesis under which the two proportions are the same, that is, have a difference (null value) of zero.

For an example, consider a group of students taking a statistics exam. In this group are $n_1 = 233$ students majoring in psychology, of whom $x_1 = 180$ pass, and $n_2 = 197$ students majoring in geography, of whom 175 pass. Suppose it is claimed that geography students have a higher pass rate in statistics than the psychology students.

Representing the true pass rates for psychology students as π_1 and geography students as π_2, this claim can be statistically tested using a pair of hypotheses defined as follows:

$$H_0 : \pi_2 - \pi_1 = 0$$

$$H_A : \pi_2 - \pi_1 > 0$$

Just as with a comparison between two means, it's important to keep the order of differencing consistent throughout the test calculations. This example shows an upper-tailed test.

Calculation: Two-Sample Z-Test

In testing for the true difference between two proportions mathematically, π_1 and π_2, let $\widehat{p}_1 = x_1/n_1$ be the sample proportion for x_1 successes in n_1 trials corresponding to π_1, and the same quantities as $\widehat{p}_2 = x_2/n_2$ for π_2. With a null value of the difference denoted π_0, the test statistic is given by the following:

$$Z = \frac{\widehat{p}_2 - \widehat{p}_1 - \pi_0}{\sqrt{p^*(1-p^*)\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}} \qquad (18.9)$$

Provided you can apply the aforementioned conditions for a proportion with respect to n_1, n_2 and π_1, π_2, you can treat $Z \sim N(0, 1)$.

As noted, in this kind of test it's common for the null value, the true difference in proportions, to be set to zero (in other words, $\pi_0 = 0$).

A new quantity, p^*, is present in the denominator of Equation (18.9). This is a *pooled* proportion, given as follows:

$$p^* = \frac{x_1 + x_2}{n_1 + n_2} \qquad (18.10)$$

The denominator of Equation (18.9) is itself the standard error of the difference between two proportions as used in a hypothesis test. The need to use p^* lies once more in the fact that H_0 is assumed to be true. Using $\widehat{p}_1$ and $\widehat{p}_2$ separately in the denominator of (18.9), in the form of

$$\sqrt{\widehat{p}_1\left(1-\widehat{p}_1\right)/n_1 + \widehat{p}_2\left(1-\widehat{p}_2\right)/n_2}$$

(the standard error of the difference between two proportions outside the confines of a hypothesis test), would violate the assumed "truth" of H_0.

So, returning to the statistics exam taken by the psychology and geography students, you can calculate the required quantities as such:

```
R> x1 <- 180
R> n1 <- 233
R> p.hat1 <- x1/n1
R> p.hat1
[1] 0.7725322
R> x2 <- 175
R> n2 <- 197
R> p.hat2 <- x2/n2
R> p.hat2
[1] 0.8883249
```

The results indicate sample pass rates of around 77.2 percent for the psychology students and 88.8 percent for the geography students; this is

a difference of roughly 11.6 percent. From examining the values of $\widehat{p}_1$, n_1 and $\widehat{p}_2$, n_2, you can see that the rule of thumb is satisfied for this test.

The pooled proportion p^*, following Equation (18.10), is as follows:

```
R> p.star <- (x1+x2)/(n1+n2)
R> p.star
[1] 0.8255814
```

You calculate the test statistic Z as per Equation (18.9) with the following:

```
R> Z <- (p.hat2-p.hat1)/sqrt(p.star*(1-p.star)*(1/n1+1/n2))
R> Z
[1] 3.152693
```

In light of the hypotheses, you find the corresponding p-value as a right-hand, upper-tail area from Z underneath the standard normal curve as follows:

```
R> 1-pnorm(Z)
[1] 0.0008088606
```

You observe a p-value that's substantially smaller than α, so the formal decision is to reject the null hypothesis in favor of the alternative. The sample data provide sufficient evidence against H_0 such that you can conclude that evidence exists to support the pass rate for geography students being higher than the pass rate for psychology students.

R Function: prop.test

Once more, R allows you to perform the test with one line of code using `prop.test`. For comparisons of two proportions, you pass the number of successes in each group as a vector of length 2 to `x` and the respective sample sizes as another vector of length 2 to `n`. Note that the order of the entries must reflect the order of `alternative` if this is one-sided (in other words, here, the proportion that is to be tested as

"greater" corresponds to the first elements of `x` and `n`). Once more, the `correct` argument is set to `FALSE`:

```
R> prop.test(x=c(x2,x1),n=c(n2,n1),alternative="greater",correct=FALSE)

   2-sample test for equality of proportions without continuity correction

data:  c(x2, x1) out of c(n2, n1)
X-squared = 9.9395, df = 1, p-value = 0.0008089
alternative hypothesis: greater
95 percent confidence interval:
 0.05745804 1.00000000
sample estimates:
   prop 1    prop 2
0.8883249 0.7725322
```

The p-value is identical to the one generated by the previous series of calculations, suggesting a rejection of H_0 under a conventional significance level of $\alpha = 0.05$. Since `prop.test` was called as a one-sided test, the confidence interval returned provides a single bound. To provide a two-sided CI for the true difference, it makes sense, considering the outcome of the test, to construct this using the separate $\widehat{p}_1$ and $\widehat{p}_2$ instead of using the denominator of Equation (18.9) specifically (which assumes truth of H_0). The "separate-estimate" version of the standard error of the difference between two proportions was given in the text beneath Equation (18.10), and a 95 percent CI is therefore calculated with the following:

```
R> (p.hat2-p.hat1) +
   c(-1,1)*qnorm(0.975)*sqrt(p.hat1*(1-p.hat1)/n1+p.hat2*(1-p.hat2)/n2)
[1] 0.04628267 0.18530270
```

With that, you're 95 percent confident that the true difference between the proportion of geography students passing the exam and the proportion of psychology students passing the exam lies somewhere between 0.046 and 0.185. In this case, the two-sided interval also

reflects the result of the one-sided hypothesis test: It doesn't include the null value of zero and is wholly positive.

Exercise 18.3

An advertisement for a skin cream claims 9 out of 10 women who use it would recommend it to a friend. A skeptical salesperson in a department store believes the true proportion of women users who'd recommend it, π, is much smaller than 0.9. She follows up with 89 random customers who had purchased the skin cream and asks if they would recommend it to others, to which 71 answer yes.

a. Set up an appropriate pair of hypotheses for this test and determine whether it will be valid to carry out using the normal distribution.

b. Compute the test statistic and the p-value and state your conclusion for the test using a significance level of $\alpha = 0.1$.

c. Using your estimated sample proportion, construct a two-sided 90 percent confidence interval for the true proportion of women who would recommend the skin cream.

The political leaders of a particular country are curious as to the proportion of citizens in two of its states that support the decriminalization of marijuana. A small pilot survey taken by officials reveals that 97 out of 445 randomly sampled voting-age citizens residing in state 1 support the decriminalization and that 90 out of 419 votingage citizens residing in state 2 support the same notion.

d. Letting π_1 denote the true proportion of citizens in support of decriminalization in state 1 and π_2 the same measure in state 2, conduct and conclude a hypothesis test under a

significance level of $\alpha = 0.05$ with reference to the following hypotheses:

$$H_0 : \pi_2 - \pi_1 = 0$$

$$H_A : \pi_2 - \pi_1 \neq 0$$

e. Compute and interpret a corresponding CI.

Your next task is to write your own relatively simple R function to perform a one- or two-sample normal-based hypothesis test for proportions using the formulas presented in Sections 18.3.1 and 18.3.2.

f. Use the following guidelines to write your function, to be named `z.test`, in the R editor:

- The function should take the following arguments: `p1` and `n1` (no default) to pose as the estimated proportion and sample size; `p2` and `n2` (both defaulting to `NULL`) that contain the second sample proportion and sample size in the event of a two-sample test; `p0` (no default) as the null value; and `alternative` (default `"two.sided"`) and `conf.level` (default `0.95`), to be used in the same way as in `t.test`.
- When conducting a two-sample test, it should be `p1` that is tested as being smaller or larger than `p2` when `alternative="less"` or `alternative="greater"`, the same as in the use of `x` and `y` in `t.test`.
- The function should perform a one-sample *Z*-test using `p1`, `n1`, and `p0` if either `p2` or `n2` (or both) is `NULL`.
- The function should contain a check for the rule-of-thumb to ensure the validity of the normal distribution in both one- and two-sample settings. If

this is violated, the function should still complete but should issue an appropriate warning message (see Section 12.1.1).

- Calculation of the confidence interval should make use of the same standard error used in the hypothesis test evaluated with either the null proportion value (one-sample test) or the pooled proportion value (two-sample test), not the raw sample proportions.
- All that need be returned is a list containing the members `Z` (test statistic), `P` (appropriate *p*-value, which can be determined by `alternative`; for a two-sided test, determining whether `Z` is positive can help), and `CI` (two-sided CI with respect to `conf.level`).

g. Replicate the two examples in the text of Sections 18.3.1 and 18.3.2 using `Z.test`; ensure you reach identical *p*-values (your confidence intervals will be slightly different to those illustrated in the text).

h. Call `Z.test(p1=0.11,n1=10,p0=0.1)` to try your warning message in the one-sample setting.

18.4 Testing Categorical Variables

The normal-based *Z*-test is particular to data that are binary in nature. To statistically test claims regarding more general categorical variables, with more than two distinct levels, you use the ubiquitous *chi-squared test*. As the name suggests, the chi-squared distribution (Section 16.2.5) is the sampling distribution of interest.

There are two common variants of the chi-squared test. You use a chi-squared test of *distribution*, or a *goodness of fit (GOF)* test, when assessing the frequencies in the levels of a single categorical variable. A chi-squared test of *independence* is employed when investigating the

relationship between frequencies in the levels of two such variables. Let's start by discussing chi-squared tests of distribution.

18.4.1 Single Categorical Variable

Like the *Z*-test, the one-dimensional chi-squared test is also concerned with comparing proportions, but in a setting where there are more than two proportions. You can use this test when you have *k* levels (or categories) of a categorical variable and want to hypothesize about their relative frequencies. The categories must be *mutually exclusive* (in other words, an observation cannot take more than one of the possible categories) and *exhaustive* (in other words, the *k* categories cover all possible outcomes).

Let's walk through an example that illustrates how hypotheses are constructed and introduces the relevant ideas and methods. Suppose a researcher in sociology is interested in the dispersion of rates of facial hair in men of his local city and whether they are uniformly represented in the male population. He defines a categorical variable with three levels: clean shaven (1), beard only *or* mustache only (2), and beard *and* mustache (3). He collects data on 53 randomly selected men and finds the following outcomes:

```
R> hairy <- c(2,3,2,3,2,1,3,3,2,2,3,2,2,2,3,3,3,2,3,2,2,2,1,3,2,2,2,1,2,2,3,
              2,2,2,2,1,2,1,1,1,2,2,2,3,1,2,1,2,1,2,1,3,3)
```

Now the research question asks whether the proportions in each category are equally represented. Let π_1, π_2, and π_3 represent the true proportion of men in the city who fall into groups 1, 2, and 3, respectively. You therefore seek to test these hypotheses:

$$\mathrm{H}_0 \ : \ \pi_1 = \pi_2 = \pi_3 = \frac{1}{3}$$
$$\mathrm{H}_\mathrm{A} \ : \ \mathrm{H}_0 \text{ is incorrect.}$$

The appearance of the alternative hypothesis is a little different from what you've seen so far but is an accurate reflection of the interpretation

of a chi-squared GOF test. In these types of problems, H_0 is always posed in terms of the proportions for each group being equal to stated values, and H_A that the data, as a whole, do not match these null proportions. As with any hypothesis test, we conduct this test assuming the null hypothesis is true, and evidence against the no-change, baseline setting will be represented as a small *p*-value.

Calculation: Chi-Squared Test of Distribution

The quantities of interest are the proportion of n observations in each of k categories, $\pi_1, \ldots, \pi_k$, for a single mutually exclusive and exhaustive categorical variable. The null hypothesis defines hypothesized null values for each proportion; label these respectively as $\pi_{0(1)}, \ldots, \pi_{0(k)}$. The test statistic χ^2 is given as

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \qquad (18.11)$$

where O_i is the *observed* count and E_i is the *expected* count in the ith category; $i = 1, \ldots, k$. The O_i are obtained directly from the raw data, and the expected counts, $E_i = n\pi_{0(i)}$, are merely the product of the overall sample size n with the respective null proportion for each category. The result, χ^2, follows a chi-squared distribution with $v = k - 1$ degrees of freedom; refer to Section 16.2.5. We usually consider the test to be valid based on an informal rule-of-thumb stating that at least 80 percent of the expected counts E_i should be at least 5.

In this type of chi-squared test, it is important to note the following:

- The term *goodness-of-fit* refers to the proximity of the observed data to the distribution hypothesized in H_0.
- Positive extremity of the result of Equation (18.11) provides evidence against H_0. As such, *the corresponding* p*-value is always computed as an upper-tail area*.

- As in the current example, a test for uniformity simplifies the null hypothesis slightly by having equivalent null proportions $\pi_0 = \pi_{0(1)} = \ldots = \pi_{0(k)}$.
- A rejected H_0 doesn't tell you about the true values of π_i. It merely suggests that they do not follow H_0 specifically.

The current facial hair example is a test for the uniformity of the distribution of frequencies in the three categories. You can obtain the observed counts and corresponding proportions with `table` (see Section 13.2.1):

```
R> n <- length(hairy)
R> n
[1] 53
R> hairy.tab <- table(hairy)
R> hairy.tab
hairy
 1  2  3
11 28 14
R> hairy.tab/n
hairy
        1         2         3
0.2075472 0.5283019 0.2641509
```

For computation of the test statistic χ^2, you have the observed counts O_i in `hairy.tab`. The expected count E_i is a straightforward arithmetical calculation of the total number of observations multiplied by the null proportion 1/3, yielding the same value for each category, and in turn reflecting the aforementioned uniformity of the null hypothesis in this example.

We can present these quantities, as well as the contribution of each category to the test statistic, in a matrix with `cbind` (Section 3.1.1) as follows:

```
R> expected <- 1/3*n
R> expected
```

```
[1] 17.66667
R> hairy.matrix <- cbind(1:3,hairy.tab,expected,
                         (hairy.tab-expected)^2/expected)
R> dimnames(hairy.matrix) <- list(c("clean","beard OR mous.",
                                    "beard AND mous."),
                                  c("i","Oi","Ei","(Oi-Ei)^2/Ei"))
R> hairy.matrix
                i Oi       Ei (Oi-Ei)^2/Ei
clean           1 11 17.66667    2.5157233
beard OR mous.  2 28 17.66667    6.0440252
beard AND mous. 3 14 17.66667    0.7610063
```

Note that the expected counts are comfortably greater than 5, which satisfies the informal rule-of-thumb mentioned earlier. In terms of R coding, note also that the single number `expected` is implicitly recycled to match the length of the other vectors supplied to `cbind` in creating `hairy.matrix`, and that the `dimnames` attribute (refer to Section 6.2.1) has been accessed to annotate the rows and columns.

The test statistic, as per Equation (18.11), is given as the sum of the $(O_i - E_i)^2/E_i$ contributions in the fourth column of `hairy.matrix`:

```
R> X2 <- sum(hairy.matrix[,4])
R> X2
[1] 9.320755
```

The corresponding p-value is the appropriate upper-tail area from the chi-squared distribution with $v = 3 - 1 = 2$ degrees of freedom:

```
R> 1-pchisq(X2,df=2)
[1] 0.009462891
```

This small p-value provides evidence to suggest that the true frequencies in the defined categories of male facial hair for this population are not uniformly distributed in a 1/3, 1/3, 1/3 fashion. Remember that the test result doesn't give you the true proportions; it only suggests that they do not follow those in H_0.

R Function: chisq.test

Like `t.test` and `prop.test`, R provides a quick-use function for performing a chi-squared GOF test. The `chisq.test` function takes the vector of observed frequencies as its first argument `x`. For the facial hair example, this line provides the same results as found previously:

```
R> chisq.test(x=hairy.tab)

    Chi-squared test for given probabilities

data:  hairy.tab
X-squared = 9.3208, df = 2, p-value = 0.009463
```

By default, the function performs a test for uniformity, taking the number of categories as the length of the vector supplied to `x`. However, suppose that the researcher collecting the facial hair data realizes that he was doing so in November, a month during which many men grow mustaches in support of "Mo-vember" to raise awareness of men's health. This changes the researcher's thoughts on the true rates in terms of his clean-shaven (1), beard-only *or* mustache-only (2), and beard *and* mustache (3) categories. He now wants to test the following:

$$\mathrm{H}_0 : \pi_{0(1)} = 0.25; \pi_{0(2)} = 0.5; \pi_{0(3)} = 0.25$$
$$\mathrm{H}_\mathrm{A} : \mathrm{H}_0 \text{ is incorrect.}$$

If a GOF test of uniformity is not desired—that is, when the "true" rates across the categories being tested in H_0 are not all the same—the `chisq.test` function requires you to supply the null proportions as a vector of the same length as `x` to the `p` argument. Naturally, each entry in `p` must correspond to the categories tabulated in `x`:

```
R> chisq.test(x=hairy.tab,p=c(0.25,0.5,0.25))

    Chi-squared test for given probabilities
```

```
data:  hairy.tab
X-squared = 0.5094, df = 2, p-value = 0.7751
```

In these results, the very high *p*-value means there's no evidence to reject H_0 in this scenario. There is no evidence to suggest that the proportions hypothesized in H_0 are incorrect.

18.4.2 Two Categorical Variables

You can also apply the chi-squared test in situations where you have *two* mutually exclusive and exhaustive categorical variables at hand; call them variable *A* and variable *B*. In this context, it's used to detect whether there might be some influential relationship (in other words, *dependence*) between *A* and *B* by looking at the way in which the distribution of frequencies change together with respect to their categories. If there is no relationship, the distribution of frequencies in variable *A* will have nothing to do with the distribution of frequencies in variable *B*. As such, this particular variant of the chi-squared test is called a *test of independence* and is performed with the following hypotheses:

H_0 : Variables *A* and *B* are independent.

(There is no relationship between *A* and *B*.)

H_A : Variables *A* and *B* are *not* independent.

(There *is* a relationship between *A* and *B*.)

To carry out the test, you compare the observed data to the counts you'd expect to see if the distributions were completely unrelated (satisfying the assumption that H_0 is true). An overall large departure from the expected frequencies will result in a small *p*-value, thus providing evidence against the null.

To best present data with two categorical variables in R, use a two-dimensional structure, such as a standard matrix. For example, suppose some dermatologists at a certain clinical practice are interested in their successes in treating a common skin affliction. Their records show $N = 355$ patients have been treated at their clinic using one of four possible treatments: a course of tablets, a series of injections, a laser treatment, and an herbal-based remedy. The level of success in curing the affliction is also recorded: none, partial success, and full success. The data are given here with the constructed matrix `skin`:

```
R> skin <- matrix(c(20,32,8,52,9,72,8,32,16,64,30,12),4,3,
                  dimnames=list(c("Injection","Tablet","Laser","Herbal"),
                  c("None","Partial","Full")))
R> skin
          None Partial Full
Injection   20       9   16
Tablet      32      72   64
Laser        8       8   30
Herbal      52      32   12
```

A two-dimensional table presenting frequencies in this fashion is called a *contingency table*.

Calculation: Chi-Squared Test of Independence

To compute the test statistic, presume data are presented as a $k_r \times k_c$ contingency table; in other words, a matrix of counts, based on two categorical variables (both mutually exclusive and exhaustive). The focus of the test is the way in which the frequencies of N observations between the k_r levels of the "row" variable and the k_c levels of the "column" variable are jointly distributed. The test statistic χ^2 is given with

$$\chi^2 = \sum_{i=1}^{k_r} \sum_{j=1}^{k_c} \frac{\left(O_{[i,j]} - E_{[i,j]}\right)^2}{E_{[i,j]}} \qquad (18.12)$$

where $O_{[i,j]}$ is the observed count and $E_{[i,j]}$ is the expected count at row position i and column position j. Each $E_{[i,j]}$ is found as the sum total of row i multiplied by the sum total of column j, all divided by N.

$$E_{[i,j]} = \frac{\left(\sum_{u=1}^{kr} O_{[u,j]}\right) \times \left(\sum_{v=1}^{kc} O_{[i,v]}\right)}{N} \qquad (18.13)$$

The result, χ^2, follows a chi-squared distribution with $v = (k_r - 1) \times (k_c - 1)$ degrees of freedom. Again, the p-value is always an upper-tailed area, and you can consider the test valid with the satisfaction of the condition that at least 80 percent of the $E_{[i,j]}$ are at least 5.

For this calculation, it's important to note the following:

- It's not necessary to assume that $k_r = k_c$.
- The functionality of Equation (18.12) is the same as that of (18.11): an overall sum involving the squared differences between the observed and expected values of each cell.
- The double-sum in Equation (18.12) just represents the total sum over all the cells, in the sense that you can compute the total sample size N with $\sum_{i=1}^{k_r} \sum_{j=1}^{k_c} O_{[i,j]}$.
- A rejected H_0 doesn't tell you about the nature of *how* the frequencies depend on one another, just that there is evidence to suggest that some kind of dependency between the two categorical variables exists.

Continuing with the example, the dermatologists want to determine whether their records suggest there is statistical evidence to indicate some relationship between the type of treatment and the level of success in curing the skin affliction. For convenience, store the total number of categories k_r and k_c for the row and column variables, respectively:

```
R> kr <- nrow(skin)
R> kc <- ncol(skin)
```

You have the $O_{[i,j]}$ in `skin`, so now you must now compute the $E_{[i,j]}$. In light of Equation (18.13), which deals with row and column sums of array-like structures, you can evaluate these using the built-in `rowSums` and `colSums` functions (first used in Section 12.2.3):

```
R> rowSums(skin)
Injection    Tablet     Laser    Herbal
       45       168        46        96
R> colSums(skin)
   None Partial    Full
    112     121     122
```

These results indicate the totals in each group, regardless of the other variable. To get the expected counts for all matrix cells, Equation (18.13) requires each row sum to be multiplied by each column sum once. You could write a `for` loop, but this would be inefficient and rather inelegant. It is better to use `rep` with the optional `each` argument (refer to Section 2.3.2). By repeating each element of the column totals (level of success) four times, you can then use vector-oriented behavior to multiply that repeated vector by the shorter vector produced by `rowSums`. You can then call `sum(skin)` to divide this by N and rearrange it into a matrix. The following lines show how this example works step-by-step:

```
R> rep(colSums(skin),each=kr)
   None    None    None    None Partial Partial Partial Partial    Full
    112     112     112     112     121     121     121     121     122
   Full    Full    Full
    122     122     122
R> rep(colSums(skin),each=kr)*rowSums(skin)
   None    None    None    None Partial Partial Partial Partial    Full
   5040   18816    5152   10752    5445   20328    5566   11616    5490
   Full    Full    Full
  20496    5612   11712
R> rep(colSums(skin),each=kr)*rowSums(skin)/sum(skin)
    None     None     None     None  Partial  Partial  Partial  Partial
14.19718 53.00282 14.51268 30.28732 15.33803 57.26197 15.67887 32.72113
    Full     Full     Full     Full
```

```
15.46479 57.73521 15.80845 32.99155
R> skin.expected <- matrix(rep(colSums(skin),each=kr)*rowSums(skin)/sum(skin),
                           nrow=kr,ncol=kc,dimnames=dimnames(skin))
R> skin.expected
             None  Partial     Full
Injection 14.19718 15.33803 15.46479
Tablet    53.00282 57.26197 57.73521
Laser     14.51268 15.67887 15.80845
Herbal    30.28732 32.72113 32.99155
```

With this output, note that all the expected values are greater than 5, as preferred.

While this next step isn't essential, it may be desirable to construct a single object to hold the results of the different stages of calculations leading to the test statistic, as you did for the one-dimensional example. Since each stage is a matrix, you can bind the relevant matrices together with `cbind` and produce an array of the appropriate dimensions (refer to Section 3.2 for a refresher):

```
R> skin.array <- array(data=cbind(skin,skin.expected,
                                  (skin-skin.expected)^2/skin.expected),
                       dim=c(kr,kc,3),
                       dimnames=list(dimnames(skin)[[1]],dimnames(skin)[[2]],
                                     c("O[i,j]","E[i,j]",
                                       "(O[i,j]-E[i,j])^2/E[i,j]")))
R> skin.array
, , O[i,j]

          None Partial Full
Injection   20       9   16
Tablet      32      72   64
Laser        8       8   30
Herbal      52      32   12

, , E[i,j]

             None  Partial     Full
Injection 14.19718 15.33803 15.46479
```

```
Tablet    53.00282 57.26197 57.73521
Laser     14.51268 15.67887 15.80845
Herbal    30.28732 32.72113 32.99155

, , (O[i,j]-E[i,j])^2/E[i,j]

              None   Partial        Full
Injection  2.371786 2.6190199  0.01852279
Tablet     8.322545 3.7932587  0.67978582
Laser      2.922614 3.7607992 12.74002590
Herbal    15.565598 0.0158926 13.35630339
```

The test statistic given by Equation (18.12) is just the grand total of all elements of the matrix represented by the third layer of `skin.array`:

```
R> X2 <- sum(skin.array[,,3])
R> X2
[1] 66.16615
```

The corresponding *p*-value for this test of independence is as follows; recall that the relevant degrees of freedom are defined as $v = (k_r - 1) \times (k_c - 1)$:

```
R> 1-pchisq(X2,df=(kr-1)*(kc-1))
[1] 2.492451e-12
```

The extremely small *p*-value provides strong evidence against the null hypothesis. The appropriate conclusion would be to reject H_0 and state that there does appear to be a relationship between the type of treatment for the skin affliction and the level of success in curing it.

R Function: chisq.test

Yet once more, no section in this chapter would be complete without showcasing the built-in functionality R possesses for these fundamental procedures. The default behavior of `chisq.test`, when supplied a matrix as `x`, is to perform a chi-squared test of independence with respect to the row and column frequencies, just as performed manually here for the

skin affliction example. The following result easily confirms your previous calculations:

```
R> chisq.test(x=skin)

    Pearson's Chi-squared test

data:  skin
X-squared = 66.1662, df = 6, p-value = 2.492e-12
```

Exercise 18.4

`HairEyeColor` is a ready-to-use dataset in R. This 4×4×2 array provides frequencies of hair and eye colors of 592 statistics students, split by sex (Snee 1974).

a. Perform and interpret, at a significance level of $\alpha = 0.01$, a chi-squared test of independence for hair against eye color for all students, regardless of their sex.

In Exercise 8.1 on page 183, you used the `Duncan` dataset accessible via the contributed package `car`; the dataset contains markers of job prestige collected in 1950. Make sure the package is installed (see Section 1.3.5) and load the data frame.

b. The first column of `Duncan` is the variable `type`, recording the type of job as a factor with three levels: `prof` (professional or managerial), `bc` (blue collar), and `wc` (white collar). Construct appropriate hypotheses and perform a chi-squared GOF test to determine whether the three job types are equally represented in the dataset.

    i. Interpret the resulting p-value with respect to a significance level of $\alpha = 0.05$.

ii. What conclusion would you reach if you used a significance level of $\alpha = 0.01$?

18.5 Errors and Power

In discussing all these forms of statistical hypothesis testing, there's been one common thread: the interpretation of a *p*-value and what it tells you about your problem with respect to the assumed null hypothesis. Frequentist hypothesis testing is ubiquitous in many fields of research; hence it is important to understand concepts relating to formal *errors* in statistical tests.

18.5.1 Hypothesis Test Errors

Hypothesis testing is performed with the objective of obtaining a *p*-value in order to quantify evidence against the null statement H_0. This is rejected in favor of the alternative, H_A, if the *p*-value is itself less than a predefined significance level α (which is conventionally 0.05 or 0.01). As touched upon in Section 18.1.5, this approach is justifiably criticized since the choice of α is essentially arbitrary; a decision to reject or retain H_0 can change depending solely upon the α value.

Consider for the moment, given a specific test, what the *correct* outcome is. If H_0 is really true, then you'd want to retain it. If H_A is really true, you'd want to reject the null. This "truth," one way or another, is impossible to know in practice. That being said, it's useful to consider in a theoretical sense just how good (or bad) a given hypothesis test is at yielding a result that leads to the correct conclusion. To that end, you must be able to identify two kinds of errors:

- A *Type I error* occurs when you incorrectly reject a true H_0. In any given hypothesis test, the probability of a Type I error is equivalent to the significance level α.

- A *Type II error* occurs when you incorrectly retain a false H_0 (in other words, fail to accept a true H_A). Since this depends upon what the true H_A actually is, the probability of committing such an error, labeled β, is not usually known in practice.

18.5.2 Type I Errors

In performing any hypothesis test, if your p-value is less than α, the appropriate course of action is to reject H_0. If the null is really true, though, the α directly defines the probability that you *incorrectly* reject it, committing a Type I error (because we always perform a hypothesis test under the assumption that H_0 is true).

Figure 18-1 provides a conceptual illustration of a Type I error probability for a supposed hypothesis test of a sample mean, where the hypotheses are set up as $H_0 : \mu = \mu_0$ and $H_A : \mu > \mu_0$.

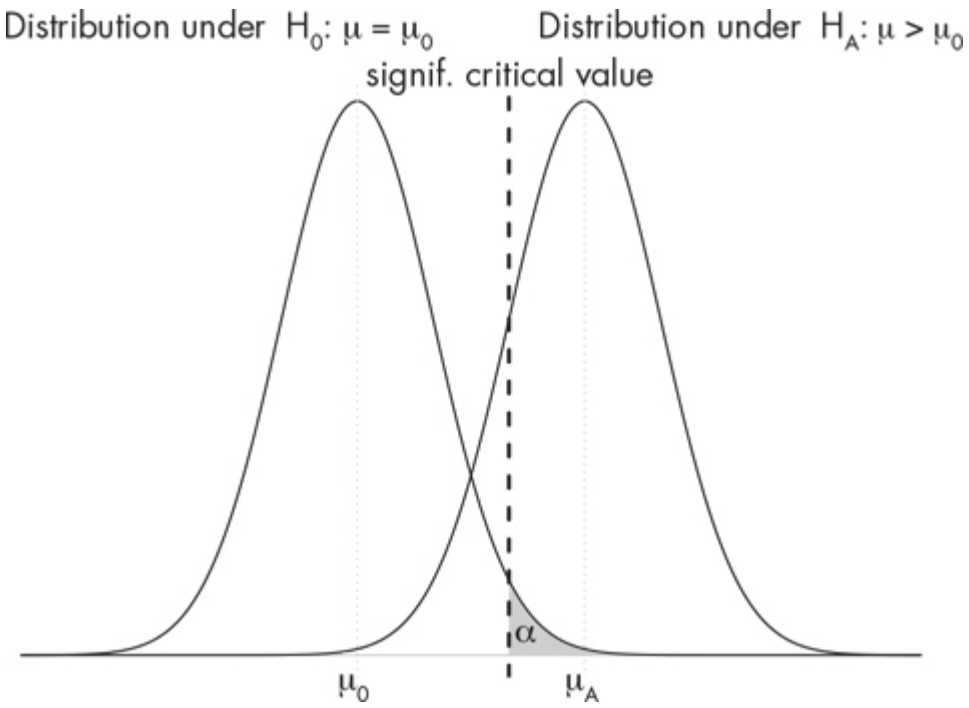


Figure 18-1: A conceptual diagram of the Type I error probability α (shaded region)

The null hypothesis distribution is centered on the null value μ_0; the alternative hypothesis distribution is centered to its right at some mean

μ_A in Figure 18-1. As you can see, if the null hypothesis is really true, then you can ignore the alternative distribution and the probability the null hypothesis is incorrectly rejected for this test will be equal to the significance level α, located in the upper tail of the null distribution.

Simulation

To demonstrate the Type I error rate via *numerical simulation* (here, this refers to randomly generating hypothetical data samples), you can write code that does the equivalent of repeating a hypothesis test under known conditions. Define the following function in the R script editor so you can use this code multiple times:

```
typeI.tester <- function(mu0,sigma,n,alpha,ITERATIONS=10000){
    pvals <- rep(NA,ITERATIONS)
    for(i in 1:ITERATIONS){
        temporary.sample <- rnorm(n=n,mean=mu0,sd=sigma)
        temporary.mean <- mean(temporary.sample)
        temporary.sd <- sd(temporary.sample)
        pvals[i] <- 1-pt((temporary.mean-mu0)/(temporary.sd/sqrt(n)),df=n-1)
    }
    return(mean(pvals<alpha))
}
```

The `typeI.tester` function is designed to generate `ITERATIONS` samples from a particular normal distribution. With each sample, you'll perform an upper-tailed test of the mean (refer to Section 18.2.1) in the spirit of Figure 18-1, assuming the hypotheses of $H_0 : \mu = \mu_0$ and $H_A : \mu > \mu_0$.

You can decrease `ITERATIONS` to generate fewer entire samples, which speeds up computation time but results in more variable simulated rates. Each entire sample of size `n` of hypothetical raw measurements is generated using `rnorm` with the mean equal to the `mu0` argument (and standard deviation equal to the `sigma` argument). The desired significance level is set by `alpha`.

In the `for` loop, the sample mean and sample standard deviation are calculated for each generated sample. Were each sample subjected to a

"real" hypothesis test, the p-value would be taken from the right-hand area of the t-distribution with `n-1` degrees of freedom (using `pt`), with respect to the standardized test statistic given earlier in Equation (18.2).

The calculated p-value at each iteration is stored in a predefined vector `pvals`. The logical vector `pvals<alpha` therefore contains corresponding `TRUE`/`FALSE` values; the former logical value flags rejection of the null hypothesis, and the latter flags retention. The Type I error rate is determined by calling `mean` on that logical vector, which yields the proportion of `TRUE`s (in other words, the overall proportion of "null hypothesis rejections") arising from the simulated samples. Remember, the samples are generated randomly, so your results are liable to change slightly each time you run the function.

This function works because, by definition of the problem, the generated samples come from a distribution that truly has the mean set at the null value, in other words, $\mu_A = \mu_0$. Therefore, any statistical rejection of this statement, obtained with a p-value less than the significance level α, is clearly incorrect and is purely a result of random variation.

To try this, import the function and execute it generating the default `ITERATIONS=10000` samples. Use the standard normal as the null (and "true" in this case!) distribution; make each sample of size 40 and set the significance level at the conventional $\alpha = 0.05$. Here's an example:

```
R> typeI.tester(mu0=0,sigma=1,n=40,alpha=0.05)
[1] 0.0489
```

This indicates that $10,000 \times 0.0489 = 489$ of the samples taken yielded a corresponding test statistic that provided a p-value, which would incorrectly result in rejection of H_0. This simulated Type I error rate lies close to the preset `alpha=0.05`.

Here's another example, this time for non-standard normal data samples with $\alpha = 0.01$:

```
R> typeI.tester(mu0=-4,sigma=0.3,n=60,alpha=0.01)
[1] 0.0108
```

Again, the numerically simulated rate of Type I error reflects the significance level.

These results are theoretically straightforward: If the true distribution is indeed the null distribution, you'll naturally observe those "extreme" test statistic values in practice at a rate equal to α. The catch, of course, is that in practice the true distribution is unknown, highlighting once more the fact that a rejection of any H_0 can never be interpreted as proof of the truth of H_A. It might simply be that the sample you observed followed the null hypothesis perfectly, but it produced an extreme test statistic value purely by chance, however small that chance might be.

Bonferroni Correction

The fact that Type I errors naturally occur because of random variation leads us to consider the *multiple testing problem*. If you're conducting many hypothesis tests, you should be cautious in simply reporting the "number of statistically significant outcomes"; as you increase the number of hypothesis tests, you increase the chance of encountering or observing an erroneous result. In, say, 20 tests conducted under $\alpha = 0.05$, on average one will be a so-called "false positive;" if you conduct 40 or 60 tests, the expected number of false positives to be encountered increases.

When conducting several hypothesis tests, you can curb this multiple testing problem by using the *Bonferroni correction*. The Bonferroni correction suggests that when performing a total of N independent hypothesis tests, each under a significance level of α, you should instead use $\alpha_B = \alpha/N$ for any interpretation of statistical significance. Be aware, however, that this correction to the level of significance represents the simplest solution to the multiple testing

problem and is critiqued for its conservative nature, particularly when N is large.

The Bonferroni and other corrective measures were developed in an attempt to mitigate the issue of committing Type I errors in multiple tests. In general, though, when performing even one hypothesis test, it suffices to be aware of the possibility that H_0 may still be true, even if the p-value you end up with is very small.

18.5.3 Type II Errors

The issues with Type I errors might suggest that it's desirable to perform a hypothesis test with a smaller α value because doing so would commensurately reduce your chances of committing a Type I error. Unfortunately, it isn't that simple, as reducing the significance level for any given test leads directly to an increase in the chance of committing a Type II error.

A Type II error refers to incorrect retention of the null hypothesis—in other words, obtaining a p-value greater than the significance level when the alternative hypothesis is actually true. For the scenario you've been looking at so far (an upper-tailed test for a single sample mean), Figure 18-2 illustrates the probability of a Type II error, shaded and denoted β.

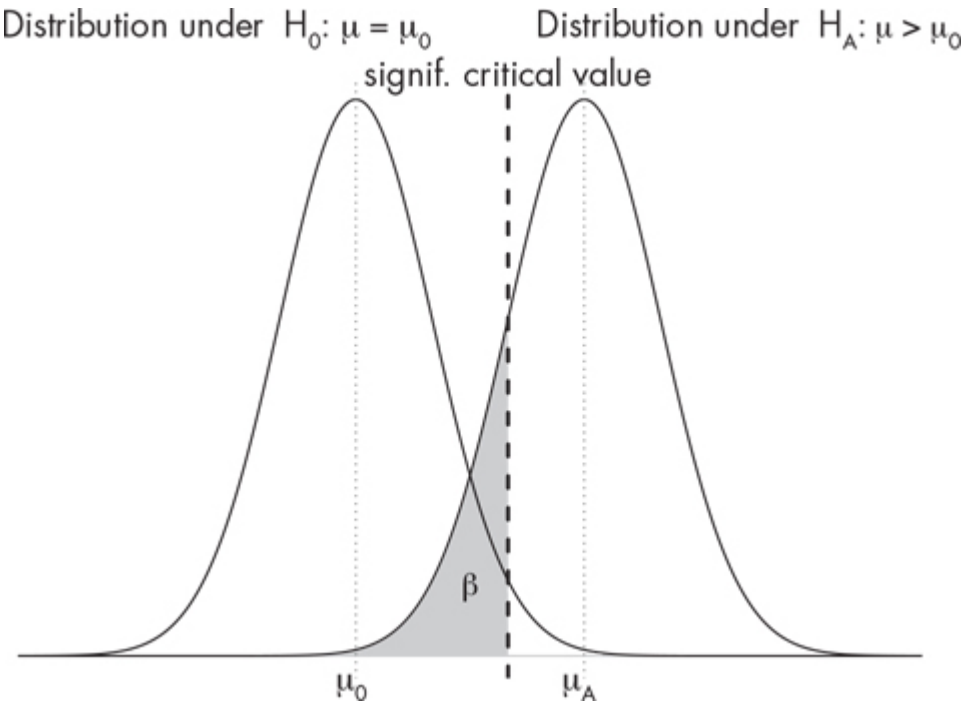


Figure 18-2: A conceptual diagram of the Type II error probability β *(shaded region)*

It's not as easy to find β as it is to find the probability of making a Type I error because β depends, among other things, on what the true value of μ_A actually is (which, in general, you won't know). If μ_A is closer to the hypothesized null value of μ_0, you can imagine the alternative distribution curve in Figure 18-2 translating (shifting) to the left and the vertical dashed line staying put, resulting in an increase in the shaded region representing β. Similarly, staying with Figure 18-2, imagine decreasing the significance level α. Doing so means the curves stay put, but now the vertical dashed line (denoting the corresponding critical value) moves to the right, also increasing the shaded area of β. Intuitively, this makes sense: The closer the true alternative value is to the null or the smaller the significance level, the harder H_A is to detect by rejection of H_0.

As noted, you usually can't calculate β in practice because you need to know what the true distribution actually is. This quantity is, however, useful in providing an idea of how prone a test is to incorrect retention of a null hypothesis under particular conditions. Suppose, for example,

you're performing a one-sample *t*-test for $H_0 : \mu = \mu_0$ and $H_A : \mu > \mu_0$ with $\mu_0 = 0$ but that the (true) alternative distribution of the raw measurements has mean $\mu_A = 0.5$ and standard deviation $\sigma = 1$. Given a random sample of size $n = 30$ and using $\alpha = 0.05$, what is the probability of committing a Type II error in any given hypothesis test (using the same standard deviation for the null distribution)? To answer this, look again at Figure 18-2; you need the critical value marked off by the significance level (the dashed vertical line). If you assume σ is known, then the sampling distribution of interest will be normal with mean $\mu_0 = 0$ and a standard error of $1/\sqrt{30}$ (see Section 17.1.1). Therefore, with an upper-tail area of 0.05, you can find the critical value with the following:

```
R> critval <- qnorm(1-0.05,mean=0,sd=1/sqrt(30))
R> critval
[1] 0.3003078
```

This value represents the position of the vertical dashed line in this specific setting (see Section 16.2.2 for a refresher on use of `qnorm`). The Type II error in this example is found as the left-hand tail area under the alternative, "true" distribution, from that critical value derived from the null:

```
R> pnorm(critval,mean=0.5,sd=1/sqrt(30))
[1] 0.1370303
```

From this, you can see that a hypothesis test under these conditions has roughly a 13.7 percent chance of incorrect retention of the null hypothesis.

Simulation

Simulation is especially useful here. In the editor, consider the function `typeII.tester` defined as follows:

```
typeII.tester <- function(mu0,muA,sigma,n,alpha,ITERATIONS=10000){
    pvals <- rep(NA,ITERATIONS)
```

```
    for(i in 1:ITERATIONS){
        temporary.sample <- rnorm(n=n,mean=muA,sd=sigma)
        temporary.mean <- mean(temporary.sample)
        temporary.sd <- sd(temporary.sample)
        pvals[i] <- 1-pt((temporary.mean-mu0)/(temporary.sd/sqrt(n)),df=n-1)
    }
    return(mean(pvals>=alpha))
}
```

This function is similar to `typeI.tester`. The null value, standard deviation of raw measurements, sample size, significance level, and number of iterations are all as before. Additionally, you now have `muA`, providing the "true" mean μ_A under which to generate the samples. Again, at each iteration, a random sample of size `n` is generated, its mean and standard deviation are calculated, and the appropriate *p*-value for the test is computed using `pt` from the usual standardized test statistic with `df=n-1`. (Remember, since you're estimating the true standard deviation of the measurements σ with the sample standard deviation s, it's technically correct to use the *t*-distribution.) Following completion of the `for` loop, the proportion of *p*-values that were greater than or equal to the significance level `alpha` is returned.

After importing the function into the workspace, you can simulate β for the test conditions mooted earlier:

```
R> typeII.tester(mu0=0,muA=0.5,sigma=1,n=30,alpha=0.05)
[1] 0.1471
```

My result indicates something close to the theoretical β evaluated previously, albeit slightly larger because of the additional uncertainty that is naturally present when using a *t*-based sampling distribution instead of a normal. Again, each time you run `typeII.tester`, the results will vary slightly since everything is based on randomly generated hypothetical data samples.

Turning your attention once more to Figure 18-2, you can see (as noted earlier) that if, in an effort to decrease the chance of a Type I error, you use $\alpha = 0.01$ instead of 0.05, the vertical line moves to the

right, thereby increasing the probability of a Type II error, with all other conditions being held constant. This is confirmed for the running example by another execution of `typeII.tester`:

```
R> typeII.tester(mu0=0,muA=0.5,sigma=1,n=30,alpha=0.01)
[1] 0.3891
```

Other Influences on the Type II Error Rate

The significance level isn't the only contributing factor in driving β. Keeping α at 0.01, see what happens if you increase the standard deviation of the raw measurements from $\sigma = 1$ to $\sigma = 1.1$ and then $\sigma = 1.2$:

```
R> typeII.tester(mu0=0,muA=0.5,sigma=1.1,n=30,alpha=0.01)
[1] 0.4815
R> typeII.tester(mu0=0,muA=0.5,sigma=1.2,n=30,alpha=0.01)
[1] 0.5501
```

Increasing the variability of the measurements, without touching anything else in the scenario, also increases the chance of a Type II error. You can imagine the curves in Figure 18-2 becoming flatter and more widely dispersed owing to a larger standard error of the mean, which would result in more probability weight in the left-hand tail marked off by the critical value. Conversely, if the variability of the raw measurements is smaller, then the sampling distributions of the sample mean will be taller and skinnier, meaning a reduction in β.

A smaller or larger sample size will have a similar impact. Located in the denominator of the standard error formula, a smaller n will result in a larger standard error, hence a flatter curve and an increased β; conversely, a larger sample size will have the opposite effect. If you remain with the latest values—$\mu_0 = 0$, $\mu_A = 0.5$, $\sigma = 1.2$, and $\alpha = 0.01$—reducing the sample size to 20 (from 30) results in an increased simulated Type II error rate compared with the most recent result of 0.5501, but increasing the sample size to 40 improves upon this rate:

```
R> typeII.tester(mu0=0,muA=0.5,sigma=1.2,n=20,alpha=0.01)
[1] 0.7319
R> typeII.tester(mu0=0,muA=0.5,sigma=1.2,n=40,alpha=0.01)
[1] 0.4219
```

Finally, as noted at the beginning of the discussion, the specific value of μ_A affects β just as you'd expect. Again, keeping the latest values for all other components, which resulted in my case in $\beta = 0.4219$, note that shifting the "true" mean closer to μ_0 by changing from $\mu_A = 0.5$ to $\mu_A = 0.4$ means the probability of committing a Type II error is increased; the opposite is true if the difference is increased to $\mu_A = 0.6$:

```
R> typeII.tester(mu0=0,muA=0.4,sigma=1.2,n=40,alpha=0.01)
[1] 0.6147
R> typeII.tester(mu0=0,muA=0.6,sigma=1.2,n=40,alpha=0.01)
[1] 0.2287
```

To summarize, although these simulated rates have been applied to the specific situation in which the hypothesis test is an upper-tailed test for a single mean, the general concepts and ideas discussed here hold for any hypothesis test. It's easy to establish that the Type I error rate matches the predefined significance level and so can be decreased by reducing α. In contrast, controlling the Type II error rate is a complex balancing act that can involve sample size, significance level, observation variability, and magnitude of the difference between the true value and the null. The Type II error rate's direct relationship to *statistical power* often plays a critical role in preparing for data collection, especially when you're considering sample size requirements, as you'll see in the next section.

Exercise 18.5

a. Write a new version of `typeI.tester` called `typeI.mean`. The new function should be able to simulate the Type I error rate for tests of a single mean in any direction (in other words, one- or two-sided). The new function should take an additional argument, `test`, which takes a character string `"less"`, `"greater"`, or `"two.sided"` depending on the type of desired test. You can achieve this by modifying `typeI.tester` as follows:

   1. Instead of calculating and storing *p*-values directly in the `for` loop, simply store the test statistic.
   2. When the loop is complete, set up stacked `if...else` statements that cater to each of the three types of test, calculating the *p*-value as appropriate.
   3. For the two-sided test, remember that the *p*-value is defined as twice the area "more extreme" than the null. Computationally, the easiest way to achieve this is to multiply by 2 the area in the upper tail demarcated by the absolute value of the test statistic (see the `abs` function in Section 13.2.3, as well as the IQ example in Section 18.2.2).
   4. If the value of `test` is not one of the three possibilities, the function should throw an appropriate error using `stop` (Section 12.1.1).

   i. Experiment with your function using the first example setting in the text with $\mu_0 = 0$, $\sigma = 1$, $n = 40$, and $\alpha = 0.05$. Call `typeI.mean` three times, using each of the three possible options for `test`. You should find that all simulated results sit close to 0.05.

   ii. Repeat (i) using the second example setting in the text with $\mu_0 = -4$, $\sigma = 0.3$, $n = 60$, and $\alpha = 0.01$.

Again, all simulated results should sit close to the value of α.

b. Modify `typeII.tester` in the same way as you did `typeI.tester`; call the new function `typeII.mean`. Simulate the Type II error rates for the following hypothesis tests. As per the text, assume μ_A, σ, α, and n denote the true mean, standard deviation of raw observations, significance level, and sample size, respectively.

   i. $H_0 : \mu = -3.2$; $H_A : \mu \neq -3.2$
   with $\mu_A = -3.3$, $\sigma = 0.1$, $\alpha = 0.05$, and $n = 25$

   ii. $H_0 : \mu = 8994$; $H_A : \mu < 8994$
   with $\mu_A = 5600$, $\sigma = 3888$, $\alpha = 0.01$, and $n = 9$

   iii. $H_0 : \mu = 0.44$; $H_A : \mu > 0.44$
   with $\mu_A = 0.4$, $\sigma = 2.4$, $\alpha = 0.05$, and $n = 68$

18.5.4 Statistical Power

For any hypothesis test, it is useful to consider its potential statistical power. *Power* is the probability of *correctly* rejecting a null hypothesis that is untrue. For a test that has a Type II error rate of β, the statistical power is its complement, that is, $1 - \beta$. For obvious reasons, it's desirable for a test to have a power that's as high as possible. The simple relationship with the Type II error probability means that all factors impacting the value of β also directly affect power.

For the same one-sided $H_0 : \mu = \mu_0$ and $H_A : \mu > \mu_0$ scenario discussed thus far, Figure 18-3 shades the power of the test, which is the complement to the Type II error rate. By convention, a hypothesis test that has a power greater than 0.8 is considered *statistically powerful*.

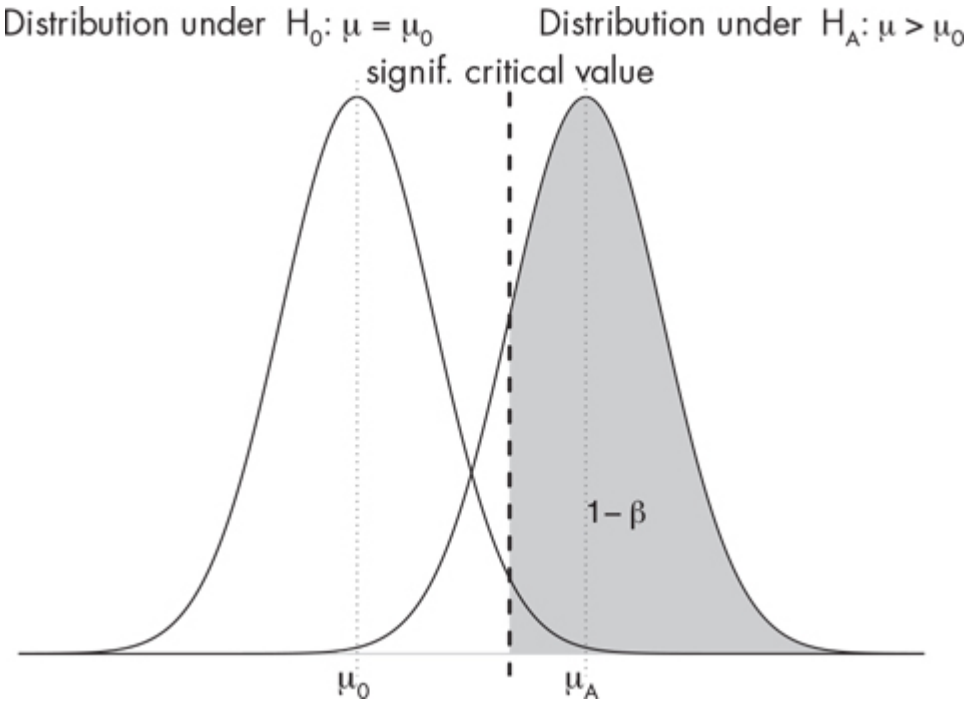


Figure 18-3: A conceptual diagram of statistical power 1 – β (shaded region)

You can numerically evaluate power under specific testing conditions using simulation. For the previous discussion on Type II errors, you're able to subtract all simulated results of β from 1 to evaluate the power of that particular test. For example, the power of detection of $\mu_A = 0.6$ when $\mu_0 = 0$, taking samples of size $n = 40$ and with $\sigma = 1.2$ and $\alpha = 0.01$, is simulated as $1 - 0.2287 = 0.7713$ (using my most recent result of β from the previous section). This means there's roughly a 77 percent chance of correctly rejecting H_0 in a hypothesis test based on a sample of measurements generated under those conditions.

Researchers are often interested in the relationship between power and sample size (though bear in mind that this is only one of the influential ingredients in the determination of power). Before you begin collecting data to examine a particular hypothesis, you might have an idea of the potential true value of the parameter of interest from past research or pilot studies. This can guide your sample size by helping to answer questions like "How big does my sample need to be in order to

be able to conduct a statistically powerful test to correctly reject H_0, if the true mean is actually μ_A?"

Simulation

For the most recent testing conditions, with a sample size of $n = 40$, you've seen that there's a power of around 0.77 of detecting $\mu_A = 0.6$. For the purposes of this example, let's say you want to find how much you should increase n by to conduct a statistically powerful test. To answer this, define the following function `power.tester` in the editor:

```
power.tester <- function(nvec,...){
    nlen <- length(nvec)
    result <- rep(NA,nlen)
    pbar <- txtProgressBar(min=0,max=nlen,style=3)
    for(i in 1:nlen){
        result[i] <- 1-typeII.tester(n=nvec[i],...)
        setTxtProgressBar(pbar,i)
    }
    close(pbar)
    return(result)
}
```

The `power.tester` function uses the `typeII.tester` function defined in Section 18.5.3 to evaluate the power of a given upper-tailed hypothesis test of a single sample mean. It takes a vector of sample sizes supplied as the `nvec` argument (you pass all other arguments to `typeII.tester` using an ellipsis; refer to Section 11.2.4). A `for` loop defined in `power.tester` cycles through the entries of `nvec` one at a time, simulates the power for each sample size, and stores them in a corresponding vector that's returned to the user. Remember, through `typeII.tester`, this function is using random generation of hypothetical data samples, so there may be some fluctuation in the results you observe each time you run `power.tester`.

There can be a slight delay when evaluating the power for many individual sample sizes, so this function also provides a good

opportunity to showcase a progress bar in a practical implementation (refer to Section 12.2.1 for details).

Set up the following vector, which uses the colon operator (Section 2.3.2) to construct a sequence of integers between 5 and 100 inclusive for the sample sizes to be examined:

```
R> sample.sizes <- 5:100
```

Importing the `power.tester` function, you can then simulate the power for each of these sample sizes for this particular test (`ITERATIONS` is halved to `5000` to reduce the overall completion time):

```
R> pow <- power.tester(nvec=sample.sizes,
                       mu0=0,muA=0.6,sigma=1.2,alpha=0.01,ITERATIONS=5000)
  |======================================================================| 100%
R> pow
 [1] 0.0630 0.0752 0.1018 0.1226 0.1432 0.1588 0.1834 0.2162 0.2440 0.2638
[11] 0.2904 0.3122 0.3278 0.3504 0.3664 0.3976 0.4232 0.4478 0.4680 0.4920
[21] 0.5258 0.5452 0.5552 0.5616 0.5916 0.6174 0.6326 0.6438 0.6638 0.6844
[31] 0.6910 0.7058 0.7288 0.7412 0.7552 0.7718 0.7792 0.7950 0.8050 0.8078
[41] 0.8148 0.8316 0.8480 0.8524 0.8600 0.8702 0.8724 0.8800 0.8968 0.8942
[51] 0.8976 0.9086 0.9116 0.9234 0.9188 0.9288 0.9320 0.9378 0.9370 0.9448
[61] 0.9436 0.9510 0.9534 0.9580 0.9552 0.9648 0.9656 0.9658 0.9684 0.9756
[71] 0.9742 0.9770 0.9774 0.9804 0.9806 0.9804 0.9806 0.9854 0.9848 0.9844
[81] 0.9864 0.9886 0.9890 0.9884 0.9910 0.9894 0.9906 0.9930 0.9926 0.9938
[91] 0.9930 0.9946 0.9948 0.9942 0.9942 0.9956
```

As expected, the power of detection rises steadily as n increases; the conventional cutoff of 80 percent is visible in these results as lying between 0.7950 and 0.8050. If you don't want to identify the value visually, you can find which entry of `sample.sizes` corresponds to the 80 percent cutoff by first using `which` to identify the indexes of `pow` that are at least 0.8 and then returning the lowest value in that category with `min`. The identified index may then be specified in square brackets to `sample.sizes` to give you the value of n that corresponds to that simulated power (0.8050 in this case). You nest these commands as follows:

```
R> minimum.n <- sample.sizes[min(which(pow>=0.8))]
R> minimum.n
[1] 43
```

The result indicates that if your sample size is at least 43, a hypothesis test under these particular conditions should be statistically powerful (based on the randomly simulated output in `pow` in this instance).

What if the significance level for this test were relaxed? Say you wanted to conduct the test (still upper-tailed under the condition of μ_0 = 0, μ_A = 0.6, and σ = 1.2) using a significance level of α = 0.05 rather than 0.01. If you look again at Figure 18-3, this alteration means the vertical line (critical value) moves to the left, decreasing β and so increasing power. That therefore suggests you'd require a smaller sample size than earlier (in other words, n < 43) to perform a statistically powerful test when α is increased.

To simulate this situation for the same range of sample sizes and store the resulting powers in `pow2`, examine the following:

```
R> pow2 <- power.tester(nvec=sample.sizes,
                        mu0=0,muA=0.6,sigma=1.2,alpha=0.05,ITERATIONS=5000)
  |======================================================================| 100%
R> minimum.n2 <- sample.sizes[min(which(pow2>0.8))]
R> minimum.n2
[1] 26
```

This result indicates a sample size of at least 26 is required, which is a noticeable reduction from the 43 noted if α = 0.01. However, relaxing α means an increased risk of committing a Type I error!

Power Curves

For comparison, you can plot your simulated powers as a kind of "power curve" using both `pow` and `pow2` with the following code:

```
R> plot(sample.sizes,pow,xlab="sample size n",ylab="simulated power")
R> points(sample.sizes,pow2,col="darkgray")
```

```
R> abline(h=0.8,lty=2)
R> abline(v=c(minimum.n,minimum.n2),lty=3,col=c("black","darkgray"))
R> legend("bottomright",legend=c("alpha=0.01","alpha=0.05"),
          col=c("black","darkgray"),pch=1)
```

Figure 18-4 illustrates my particular result. A horizontal line marks off the power of 0.8, and the vertical line marks the minimum sample size values identified and stored in `minimum.n` and `minimum.n2`. As a final touch, a legend is added to reference the α values of each curve.

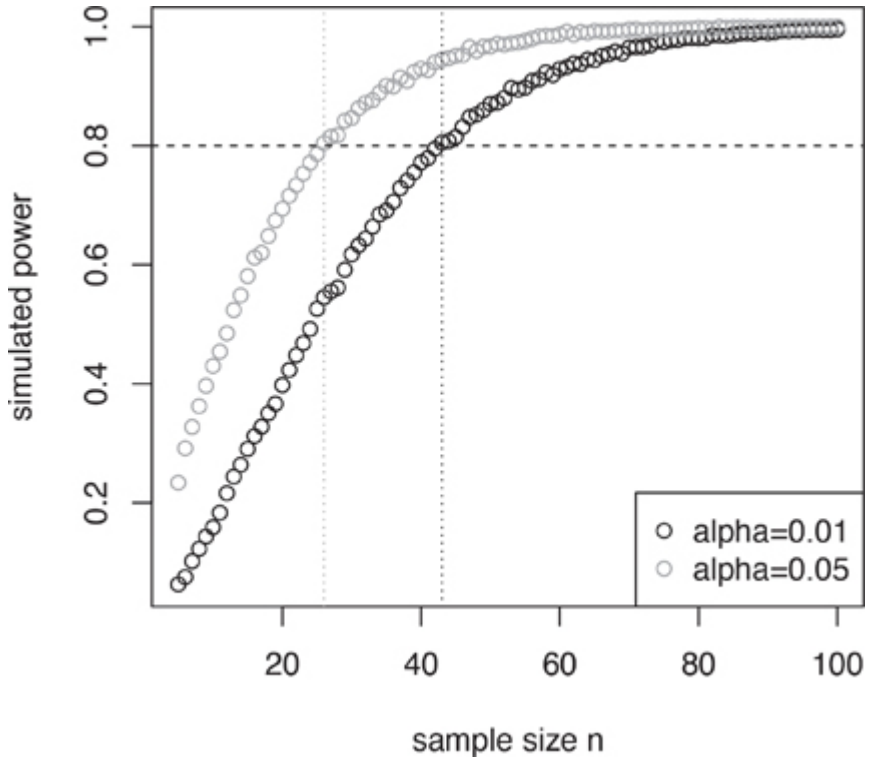


Figure 18-4: Simulated power curves for the upper-tailed hypothesis test of a single sample mean

The curves themselves indicate that the power of detection increases as the sample size is incremented. You can also note the "flattening off" occurring as the power rises ever closer to the "perfect" rate of 1, which is typical of a power curve. For $\alpha = 0.05$, the curve sits almost consistently above the curve for $\alpha = 0.01$, though the difference looks negligible as n rises above 70 or so.

The preceding discussion of errors and power highlights the need for care in interpreting the results of even the most basic of statistical tests. A *p*-value is merely a probability, and as such, no matter how small it may be in any circumstance, it can never prove or disprove a claim on its own. You should consider issues surrounding the quality of a hypothesis test (parametric or otherwise), though this is arguably difficult in practice. Nevertheless, an awareness of Type I and Type II errors, as well as the concept of statistical power, is extremely useful in the implementation and appraisal of any formal statistical testing procedure.

Exercise 18.6

a. For this exercise, you'll need to have written `typeII.mean` from Exercise 18.5 (b). Using this function, modify `power.tester` so that a new function, `power.mean`, calls `typeII.mean` instead of calling `typeII.tester`.

   i. Confirm that the power of the test given by $H_0 : \mu = 10$; $H_A : \mu \neq 10$, with $\mu_A = 10.5$, $\sigma = 0.9$, $\alpha = 0.01$, and $n = 50$, is roughly 88 percent.

   ii. Review the hypothesis test example in Section 18.2.1 for the mean net weight of an 80-gram pack of snacks, based on the $n = 44$ observations provided in the `snack` vector. The hypotheses are as follows:

$$H_0 : \mu = 80$$

$$H_A : \mu < 80$$

If the true mean is $\mu_A = 78.5$ g and the true standard deviation of the weights is $\sigma = 3.1$ g, use `power.mean` to determine whether the test is statistically powerful, assuming $\alpha = 0.05$. Does your answer to this change if $\alpha = 0.01$?

b. Staying with the snacks hypothesis test, using the `sample.sizes` vector from the text (the integer sequence given by `5:100`), determine the minimum sample size required for a statistically powerful test using both $\alpha = 0.05$ and $\alpha = 0.01$. Produce a plot showing the two power curves.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`t.test`	One- and two-sample *t*-test	Section 18.2.1, p. 461
`prop.test`	One- and two-sample *Z*-test	Section 18.3.1, p. 475
`chisq.test`	χ^2 test of distribution/independence	Section 18.4.1, p. 483

19
ANALYSIS OF VARIANCE



Analysis of variance (ANOVA), in its simplest form, is used to compare multiple means in a test for equivalence. In that sense, it's a straightforward extension of the hypothesis test comparing two means. There's a continuous variable from which the means of interest are calculated, and there's at least one categorical variable that tells you how to define the groups for those means. In this chapter, you'll explore the ideas surrounding ANOVA and look at comparing means first split by one categorical variable (one-way ANOVA) and then split by multiple categorical variables (multiple-factor ANOVA).

19.1 One-Way ANOVA

We use the simplest version of ANOVA, a *one-way* or *one-factor* analysis, to test two or more means for equality. Those means are split by a categorical *group* or *factor* variable. ANOVA is often used to analyze experimental data to assess the impact of an intervention. You might, for example, be interested in comparing the mean weights of the chicks in the built-in `chickwts` dataset, split according to the different food types they were fed.

19.1.1 Hypotheses and Diagnostic Checking

Say you have a categorical-nominal variable that splits a total of N numeric observations into k distinct groups, where $k \geq 2$. You're looking to statistically compare the k groups' means, $\mu_1, \ldots, \mu_k$, to see whether they can be claimed to be equal. The standard hypotheses are as follows:

$$H_0: \ \mu_1 = \mu_2 = \ldots = \mu_k$$

$$H_A \ : \ \mu_1, \mu_2, \ldots, \mu_k \text{ are not all equal}$$

(alternatively, "at least one mean differs").

In fact, when $k = 2$, the two-sample t-test is equivalent to ANOVA; for that reason, ANOVA is most frequently employed when $k > 2$.

The following assumptions need to be satisfied in order for the results of the basic one-way ANOVA test to be considered reliable:

Independence The samples making up the k groups must be independent of one another, and the observations in each group must be independent and identically distributed (iid).

Normality The observations in each group should be normally distributed, or at least approximately so.

Equality of variances The variance of the observations in each group should be equal, or at least approximately so.

If the assumptions of equality of variances or normality are violated, it doesn't necessarily mean your results will be completely worthless, but it will impact the overall effectiveness of detecting a true difference in the means (refer to the discussion of statistical power in Section 18.5.4). It's always a good idea to assess the validity of these assumptions before using ANOVA; I'll do this informally for the upcoming example.

It's also worth noting that you don't need to have an equal number of observations in each group to perform the test (in which case, we

refer to it as *unbalanced*). However, having unbalanced groups renders the test more sensitive to detrimental effects if your assumptions of equality of variances and normality are not sound.

Let's return to the `chickwts` data for the example: the weights of chicks based on $k = 6$ different feeds. You're interested in comparing the mean weights according to feed type to see whether they're all equal. Use `table` to summarize the six sample sizes and use `tapply` (see Section 13.2.1) to get each group mean, as follows:

```
R> table(chickwts$feed)

   casein horsebean   linseed  meatmeal   soybean sunflower
       12        10        12        11        14        12
R> chick.means <- tapply(chickwts$weight,INDEX=chickwts$feed,FUN=mean)
R> chick.means
   casein horsebean   linseed  meatmeal   soybean sunflower
 323.5833  160.2000  218.7500  276.9091  246.4286  328.9167
```

Your skills from Section 14.4.2 allow you to produce side-by-side box plots of the distributions of weights. The next two lines

```
R> boxplot(chickwts$weight~chickwts$feed)
R> points(1:6,chick.means,pch=4,cex=1.5)
```

result in the plot on the left of Figure 19-1.

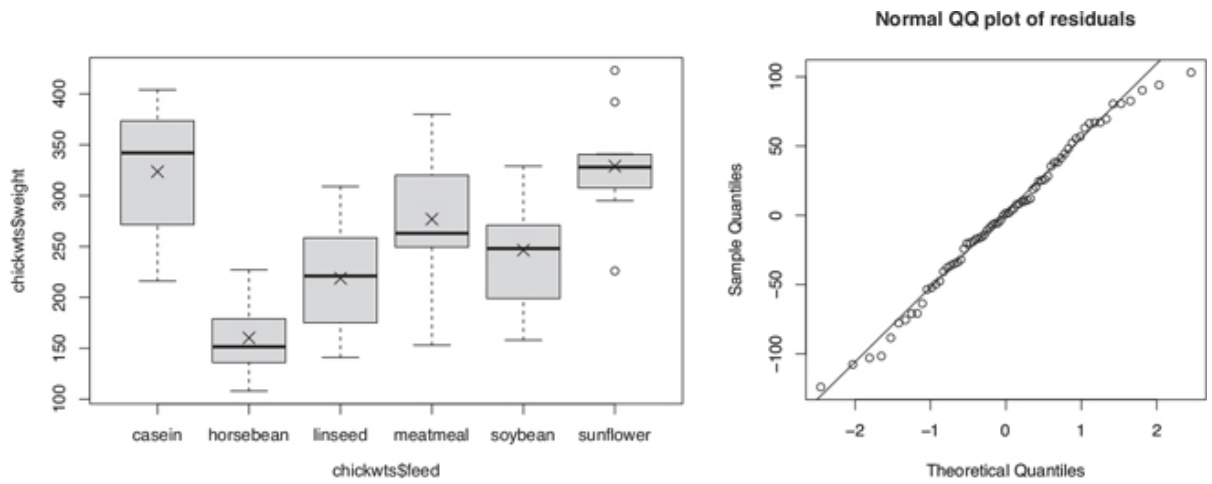


Figure 19-1: Exploring the `chickwts` data: side-by-side box plots of chick weight split by feed type, with each group mean marked by a × (left), and a normal QQ plot of the mean-centered data of each feed group (right)

Because box plots display the median, not the mean, the second line of code adds the feed-specific means (stored in the `chick.means` object you just created) to each box using `points`.

The left plot of Figure 19-1 certainly appears to indicate a difference in the mean weights. Is any appArent difference statistically significant, though? To find out, the ANOVA test for this example concerns the following hypotheses:

$$H_0 : \mu_{\text{casein}} = \mu_{\text{horsebean}} = \mu_{\text{linseed}} = \mu_{\text{meatmeal}} = \mu_{\text{soybean}} = \mu_{\text{sunflower}}$$

$$H_A : \text{The means are not all equal.}$$

Assuming independence of the data, you must check that the other assumptions are valid before implementing the test. To examine equality of variances, you can use the same informal rule of thumb as used in the two-sample *t*-test. That is, you can assume equality of variances if the ratio of the largest sample standard deviation to the smallest is less than 2. For the chick weights data, the following code determines this:

```
R> chick.sds <- tapply(chickwts$weight,INDEX=chickwts$feed,FUN=sd)
R> chick.sds
   casein horsebean   linseed  meatmeal   soybean sunflower
 64.43384  38.62584  52.23570  64.90062  54.12907  48.83638
R> max(chick.sds)/min(chick.sds)
[1] 1.680238
```

This informal result indicates that it's reasonable to make the assumption.

Next, consider the assumption of normality of the raw observations. This can be difficult to determine in many real-data examples. At the least, though, it's worthwhile to inspect histograms and QQ plots for signs of non-normality (see Section 16.2.2). For an ANOVA, you need

to do this with respect to the grouping of the observations (that is, not just "overall" for the whole set of weights regardless of their individual groupings).

To achieve this for the `chickwts` data, first *mean-center* each weight by its respective sample mean. You do this by taking the original vector of weights and subtracting from it the `chick.means` vector, but first you must rearrange and replicate the latter elements to correspond to the elements in the former. This is done by using `as.numeric` on the factor vector that represents feed type, giving the numeric value of the levels of `chickwts$feed` for each record in the original data frame. When that numeric vector is passed via the square brackets to `chick.means`, you get the correct group mean matched to each observation. As an exercise, inspect all the ingredients that go into creating the following `chick.meancen` object to understand what's going on:

```
R> chick.meancen <- chickwts$weight-chick.means[as.numeric(chickwts$feed)]
```

In the context of the current analysis, these group-wise, mean-centered values are also referred to as *residuals*, a term you'll come across frequently when you study regression methods in the next few chapters.

You can now assess normality of the observations as a whole using the residuals. To inspect a normal QQ plot, the relevant functions are `qqnorm` and `qqline` (see Section 16.2.2). The following two lines produce the right-hand image of Figure 19-1:

```
R> qqnorm(chick.meancen,main="Normal QQ plot of residuals")
R> qqline(chick.meancen)
```

Based on this plot (the proximity of the plotted points to the perfect straight line), it doesn't seem unreasonable to assume normality for these data, particularly when compared to QQ plots of generated normal data of the same sample size (an example was given on the left of Figure 16-9 on page 417).

Investigating the validity of any required assumptions is referred to as *diagnostic checking*. If you wanted to perform a more rigorous

diagnostic check for an ANOVA, other visual diagnostics may involve inspecting QQ plots split by group (you'll do this in an example in Section 19.3) or plotting the sample standard deviation for each group against the corresponding sample means. Indeed, there are also general hypothesis tests for normality (such as the Shapiro–Wilk test or Anderson–Darling test, the former of which you'll see used in Section 22.3.2), as well as tests for equality of variances (such as Levene's test), but I'll stick with the basic rule of thumb and visual checks for now.

19.1.2 Table Construction

Turning your attention back to the left of Figure 19-1, remember that the goal is to statistically evaluate the equality of the means marked by ×. This task will therefore require you to consider not only the variability *within* each of the k samples but also the variability *between* the samples; this is why we refer to the test as an "analysis of variance."

The test proceeds by first calculating various metrics associated with the overall variability and then calculating the within- and between-group variability. These figures involve sums of squared quantities and associated degrees of freedom values. All this culminates in a single test statistic and p-value targeting the aforementioned hypotheses. These ingredients are typically presented in a table, which is defined as follows.

Let $x_1, \ldots, x_N$ represent all N observations, regardless of group; let $x_{1(j)}, \ldots, x_{nj}(j)$ denote the specific group observations in group $j = 1, \ldots, k$ such that $n_1 + \ldots + n_k = N$. Let the "grand mean" of all observations be defined as $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$. The ANOVA table is then constructed, as demonstrated in Table 19-1, where SS stands for "sum-of-squares," df stands for "degrees of freedom," MS stands for "mean square," F refers to the F test statistic, and p refers to the p-value.

Table 19-1: The ANOVA Table

	df	SS	MS	*F*	*p*

	df	SS	MS	*F*	*p*
Overall	1	**(1)**			
Group (or "Factor")	$k-1$	**(2)**	**(5)**	**(5)÷(6)**	p-value
Error (or "Residual")	$N-k$	**(3)**	**(6)**		
TOTAL	N	**(4)**			

You calculate the values with these formulae:

(1): Nx^2

(2): $\sum_{j=1}^{k} \frac{\left(\sum_{i=1}^{n_j} x_{i(j)}\right)^2}{n_j}$

(3): **(4) – (2) – (1)**

(4): $\sum_{i=1}^{N} x_i^2$

(5): **(2)** ÷ $(k-1)$

(6): **(3)** ÷ $(N-k)$

There are three input sources of variation that are assumed to make up the total variation in the observed data, which, when added together, thus result in the TOTAL row. Let's think about these in a little more detail:

Overall row This relates to the scale on which the data as a whole sit. It doesn't affect the outcome of the hypothesis test (since you're interested only in the relative differences between means) and is sometimes removed from the table, affecting the TOTAL values accordingly.

Group row / Factor row This relates to the data in the individual groups of interest, thereby accounting for the *between-group variability*.

Error row / Residual row This accounts for the random deviation from the estimated means of each group, thereby accounting for the *within-group variability*.

TOTAL row This represents the raw data, based on the previous three ingredients. It is used to find the Error SS by differencing.

The three input sources each have a corresponding degrees of freedom (df) value in the first column and a sum-of-squares (SS) value attached to the df in the second column. Between- and within-group variability is averaged by dividing the SS by the df, giving the mean squared (MS) component for these two items. The test statistic, F, is found by dividing the mean squared group (MSG) effect by the mean squared error (MSE) effect. This test statistic follows the F-distribution (refer to Section 16.2.6), which itself requires a pair of degrees-of-freedom values ordered as df_1 (which represents the Group df, $k - 1$) and df_2 (which represents the Error df, $N - k$). Also recall from Chapter 16 that, like the chi-squared distribution, the F-distribution is unidirectional in nature and the p-value is obtained as the upper-tail area from the test statistic.

19.1.3 ANOVA Table Construction with the aov Function

R allows you to easily construct an ANOVA table for the chick weight test using the built-in `aov` function as follows:

```
R> chick.anova <- aov(weight~feed,data=chickwts)
```

Then, print the table to the console screen using `summary`:

```
R> summary(chick.anova)
            Df Sum Sq Mean Sq F value   Pr(>F)
feed         5 231129   46226   15.37 5.94e-10 ***
Residuals   65 195556    3009
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

There are several comments to make here. Note that you employ formula notation `weight~feed` to specify the measurement variable of interest, weight, as modeled by the categorical-nominal variable of interest, feed type. In this case, the variable names `weight` and `feed` are not required to be prefaced by `chickwts$` since the optional `data` argument has been passed the data frame of interest.

Remember from Section 14.4.2 that for the notation in the expression `weight~feed`, the "outcome" of interest must always appear on the left of the ~ (this notation will become particularly relevant in Chapters 20 to 22).

To actually view the table, you must apply the `summary` command to the object resulting from the call to `aov`. R omits the first and last rows (Overall and TOTAL) since these are not directly involved in calculating the *p*-value. Other than that, it's easy to identify that the `feed` row refers to the Group row and the `Residuals` row refers to the Error row.

NOTE

By default, R annotates model-based `summary` *output like this with* significance stars. *These show intervals of significance, and the number of stars increases as the* p*-value decreases beyond a cutoff mark of 0.1. This can be useful when you're examining more complicated analyses where multiple* p*-values are summarized, though not everyone likes this feature. You can turn off this feature in a given R session by entering* `options(show.signif.stars=FALSE)` *at the prompt. Alternatively, you can turn off the feature directly in the call to* `summary` *by setting the additional argument* `signif.stars=FALSE`. *In this book, I'll leave it be.*

From the contents of the ANOVA for this example, you can quickly confirm the calculations. Note that the MSE, found as 3,009, is defined as the Error SS divided by the Error df. Indeed, in R, the same result is

achieved manually (the table output has been rounded to the nearest integer):

```
R> 195556/65
[1] 3008.554
```

You can confirm all the other results in the table output using the relevant equations from earlier.

Interpreting a hypothesis test based on ANOVA follows the same rules as any other test. With the understanding of a *p*-value as "the probability that you observe the sample statistics at hand or something more extreme, if H_0 is true," a small *p*-value indicates evidence against the null hypothesis. In the current example, a tiny *p*-value provides strong evidence against the null that the mean chick weights are the same for the different diets. In other words, you reject H_0 in favor of H_A; the latter states that there is a difference.

In a similar fashion as in the chi-squared tests, rejection of the null in one-way ANOVA doesn't tell you exactly where a difference lies, merely that there's evidence one exists. Further scrutiny of the data in the individual groups is necessary to identify the offending means. At the simplest level, you could turn back to pairwise two-sample *t*-tests, in which case you could also use the MSE from the ANOVA table as an estimate of the pooled variance. The substitution is valid if the assumption of equal variance holds, and such a step is beneficial because the corresponding *t*-based sampling distribution will utilize the Error df (this is naturally higher than would otherwise be the case if the df was based on just the sample sizes of the two groups being tested).

Exercise 19.1

Consider the following data:

Site I	Site II	Site III	Site IV
93	85	100	96
120	45	75	58
65	80	65	95
105	28	40	90
115	75	73	65
82	70	65	80
99	65	50	85
87	55	30	95
100	50	45	82
90	40	50	
78		45	
95		55	
93			
88			
110			

These figures provide the depths (in centimeters) at which important archaeological finds were made at four sites in New Mexico (see Woosley & McIntyre 1996). Store these data in your R workspace as a two-column data frame: one column containing depth and the other containing the site of each observation as a factor.

a. Produce side-by-side box plots of the depths split by group, and use additional points to mark the locations of the sample means in the same way as Figure 19-1.

b. Assuming independence, perform diagnostic checks for normality and equality of variances.

c. Perform and conclude a one-way ANOVA test for evidence of a difference between the means.

In Section 14.6, you looked at the dataset providing measurements on petal and sepal sizes for three species of iris flowers. This is available in R as the built-in `iris` object.

d. Based on diagnostic checks for normality and equality of variances, decide which of the four outcome measurements (sepal length/width and petal length/width) would be suitable for ANOVA (using the species as the group variable).

e. Carry out one-way ANOVA for any suitable measurement variables.

19.2 Two-Way ANOVA

In many studies, the numeric outcome variable you're interested in will be categorized by more than just one grouping variable. In these cases, you use the *multiple-factor* ANOVA rather than the one-way ANOVA. This technique is directly referred to by the number of grouping variables used, with two- and three-way ANOVA being the next and most common extensions.

Increasing the number of grouping variables complicates matters somewhat: Performing just a one-way ANOVA for each variable separately is inadequate. In dealing with more than one categorical grouping factor, you must consider the *main effects* of each factor on the numeric outcome, while simultaneously accounting for the presence of the other grouping factor(s). That's not all, though. It's just as important to additionally investigate the idea of an *interactive effect*: If an interactive effect exists, then it suggests that the impact one of the grouping

variables has on the outcome of interest, specified by its main effect, varies according to the levels of the other grouping variable(s).

19.2.1 A Suite of Hypotheses

For this explanation, denote your numeric outcome variable with O and your two grouping variables as G_1 and G_2. In two-way ANOVA, the hypotheses should be set along the following lines:

H_0 : G_1 has no main (marginal) effect on the mean of O.

G_2 has no main (marginal) effect on the mean of O.

There is no interactive effect of G_1 with G_2 on the mean of O.

H_A : Separately, each statement in H_0 is incorrect.

This suite of hypotheses imply that you'll now have to obtain a p-value for each of the three statements.

For the example, let's use the built-in `warpbreaks` data frame (Tippett 1950), which provides the number of "warp break" imperfections (variable `breaks`) observed in 54 pieces of yarn of equal length. Each piece of yarn is classified according to two categorical variables: `wool` (the type of yarn, with levels `A` and `B`) and `tension` (the level of tension applied to that piece: `L`, `M`, or `H` for low, medium, or high). Using `tapply`, you can inspect the mean number of warp breaks for each classification:

```
R> tapply(warpbreaks$breaks,INDEX=list(warpbreaks$wool,warpbreaks$tension),
          FUN=mean)
        L        M        H
A 44.55556 24.00000 24.55556
B 28.22222 28.77778 18.77778
```

You can supply more than one grouping variable to the INDEX argument as separate members of a list (any factor vectors given to this argument should be the same length as the first argument that specifies the data of interest). The results are returned as a matrix for two grouping variables, a 3D array for three grouping variables, and so on.

For some analyses, however, you might need the same information provided earlier in a different format. The aggregate function is similar to tapply, but it returns a data frame with the results in *long* or *stacked* format according to the specified grouping variables (as opposed to the array returned by tapply). It's called in much the same way. The first argument takes the data vector of interest. The second argument, by, should be a list of the desired grouping variables, and in FUN, you specify the function to operate on each subset. In the following code, I've stored the result of the call to aggregate as the object wb.means for later use:

```
R> wb.means <- aggregate(warpbreaks$breaks,
                         by=list(warpbreaks$wool,warpbreaks$tension),FUN=mean)
R> wb.means
  Group.1 Group.2        x
1       A       L 44.55556
2       B       L 28.22222
3       A       M 24.00000
4       B       M 28.77778
5       A       H 24.55556
6       B       H 18.77778
```

19.2.2 Main Effects and Interactions

I mentioned earlier that you could perform just a one-way ANOVA on each grouping variable separately, but this generally isn't a good idea. I'll demonstrate this now with the warpbreaks data (quickly inspecting the relevant diagnostics—refer back to Section 19.1.1—shows no obvious cause for concern):

```
R> summary(aov(breaks~wool,data=warpbreaks))
            Df Sum Sq Mean Sq F value Pr(>F)
```

```
wool         1    451   450.7   2.668  0.108
Residuals   52   8782   168.9
R> summary(aov(breaks~tension,data=warpbreaks))
            Df Sum Sq Mean Sq F value  Pr(>F)
tension      2   2034  1017.1   7.206 0.00175 **
Residuals   51   7199   141.1
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This output tells you that if you ignore `tension`, there is no evidence to suggest that there is any difference in the mean number of imperfections based on the type of `wool` alone (*p*-value 0.108). If you ignore `wool`, however, there *is* evidence to suggest a difference in warp breaks according to `tension` only.

The problem here is that by ignoring one of the variables in either of the one-variable ANOVAs, you lose the ability to detect differences (or, more generally, statistical relationships) that may occur at a finer level. For example, though the `wool` type alone seems to have no remarkable impact on the mean number of warp breaks, you cannot tell whether this would be the case if you just looked at `wool` types at one particular level of `tension`.

Instead, you investigate this kind of question using two-way ANOVA. The following executes a two-way ANOVA for the warp breaks data based only on the main effects of the two grouping variables:

```
R> summary(aov(breaks~wool+tension,data=warpbreaks))
            Df Sum Sq Mean Sq F value  Pr(>F)
wool         1    451   450.7   3.339 0.07361 .
tension      2   2034  1017.1   7.537 0.00138 **
Residuals   50   6748   135.0
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Take a look at the formula in this call to `aov`. Specifying `wool+tension` to the right of the outcome variable and the `~` allows you to take both grouping variables into account at the same time. The results reveal a

small drop in the size of the *p*-values now attached to each grouping variable; indeed, the *p*-value for `wool` is around 0.074, approaching the conventional cutoff significance level of $\alpha = 0.05$. To interpret the results, you hold one grouping variable constant: If you focus on just one type of wool, there is still statistically significant evidence to suggest a difference in the mean number of warp breaks between the different `tension` levels. If you focus on just one level of `tension`, the evidence of a difference considering the two `wool` types has increased a little but is still not statistically significant (assuming the aforementioned $\alpha = 0.05$).

There's still a limitation with considering only main effects. While the previous analysis shows that there's variation in the outcome between the different levels of the two categorical variables, it doesn't address the possibility that a difference in the mean number of warp breaks might vary further according to precisely *which* level of either `tension` or `wool` is being used when holding the other variable constant. This relatively subtle yet important consideration is known as an *interaction*. Specifically, if there is an interactive effect present between `tension` and `wool` with respect to warp breaks, then this would imply that the magnitude or direction of the difference in the mean number of warp breaks *is not the same* at different levels of the two grouping factors.

To account for interactions, make a slight adjustment to the two-way ANOVA model code as follows:

```
R> summary(aov(breaks~wool+tension+wool:tension,data=warpbreaks))
             Df Sum Sq Mean Sq F value   Pr(>F)
wool          1    451   450.7   3.765 0.058213 .
tension       2   2034  1017.1   8.498 0.000693 ***
wool:tension  2   1003   501.4   4.189 0.021044 *
Residuals    48   5745   119.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

You can explicitly specify the interaction as the main effects model formula *plus* the notation `wool:tension`, where a `:` separates the two grouping variables. (Note, in this setting, the `:` operator has nothing to

do with the shortcut for creating an integer sequence that was first discussed in Section 2.3.2.)

This most recent ANOVA table shows that, statistically, there *is* evidence of an interactive effect; that is, the very nature of the difference in the means is dependent upon the factor levels themselves, even though that evidence is relatively weak. Of course, the *p*-value of around 0.021 tells you only that, overall, there might be an interaction but not the precise features of the interaction.

To help with this, you can interpret such a two-way interaction effect in more detail with an *interaction plot*, provided in R with `interaction.plot`:

```
R> interaction.plot(x.factor=wb.means[,2],trace.factor=wb.means[,1],
                    response=wb.means$x,trace.label="wool",
                    xlab="tension",ylab="mean warp breaks")
```

When `interaction.plot` is called, the outcome means should be supplied to the argument `response`, and the vectors providing the corresponding levels of each of the two factors should be supplied to the arguments `x.factor` (for the variable on the horizontal axis that refers to moving between levels from the left to the right) and `trace.factor` (each level of which will produce a different line, referenced in an automatically produced legend; the title of this legend is passed to `trace.label`). It doesn't matter which grouping variable is which; the appearance of the plot will change accordingly, but your interpretations will be the same (usually, the grouping variable with the fewest individual categories is chosen for the `trace.factor`). Figure 19-2 shows the result of the previous line of code.

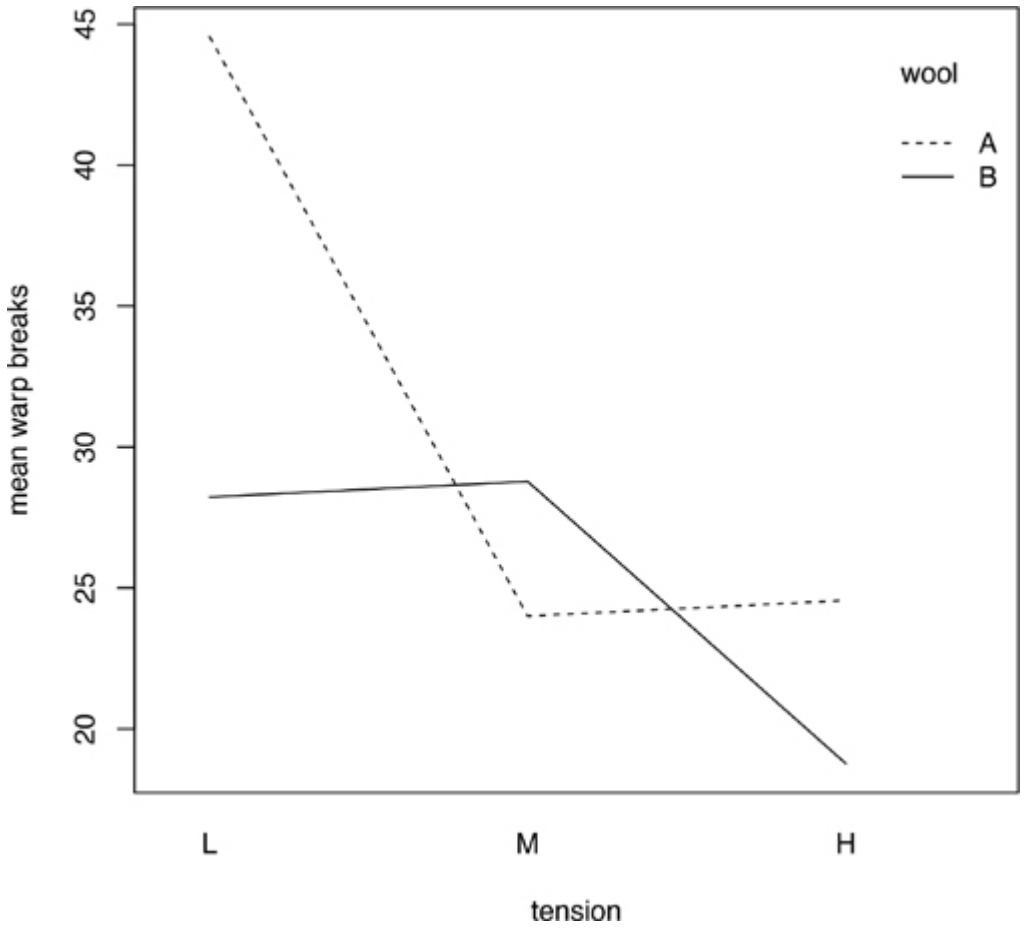


Figure 19-2: An interaction plot for the full two-way ANOVA model of the `warpbreaks` dataset

The two-way interaction plot displays the outcome variable on the vertical axis and splits the recorded means by the levels of the two grouping variables. This allows you to inspect the potential effect that varying the levels of the grouping variables has on the outcome. In general, when the lines (or segments thereof) are not parallel, it suggests an interaction could be present. Vertical separations between the plotted locations indicate the individual main effects of the grouping variables.

It turns out that the columns returned by a call to `aggregate` are actually perfectly suited to `interaction.plot`. As usual, you can specify the common graphical parameters, like those you initially encountered in Section 7.2, to control specific features of the plot and axis annotation. For Figure 19-2, you've specified that `x.factor` should be the second column of the `wb.means` matrix, meaning that the `tension` levels vary

horizontally. The `trace.factor` here is the type of `wool`, so there are only two distinct lines corresponding to the two levels `A` and `B`. The `response` is that third column of `wb.means`, extracted using `$x` (take another look at the `wb.means` object; you'll see the column containing the results of interest is labeled `x` by default after a call to `aggregate`).

Considering the actual appearance of the plot in Figure 19-2, it appears that the mean number of warp breaks for wool type `A` is higher if `tension` is low, but the nature of the difference changes if you move to a medium tension, where `B` has a higher point estimate than `A`. Moving to a high tension, type `A` again has a higher estimate of the mean number of breaks than `B`, though here the difference between `A` and `B` is nowhere near as big as it is at a low tension. (Note, however, that the interaction plot does not display any kind of standard error measurements, so you must remember that all point estimates of the means are subject to variability.)

Interactions are not a concept unique to multiple-factor ANOVA; they form an important consideration in many different types of statistical models. For the moment, it suffices to gain a basic appreciation of interactions.

19.3 Kruskal–Wallis Test

When comparing multiple means, there may be situations when you're unwilling to assume normality or have even found the assumption of normality invalid in diagnostic checks. In this case, you can use the *Kruskal–Wallis test*, an alternative to the one-way ANOVA that relaxes the dependence the normality assumption. This method tests for "equality of distributions" of the measurements in each level of the grouping factor. If you make the usual assumption of equal variances across these groups, you can therefore think of this test as one that compares multiple *medians* rather than means.

19.3.1 Hypotheses

Still assuming equality of variances, the hypotheses governing the Kruskal–Wallis test are altered from those for basic one-way ANOVA accordingly:

H_0 : Group medians are all equal.

H_A : Group medians are not all equal

(alternatively, at least one group median differs).

This test is a *nonparametric* approach since it does not rely on quantiles of a standardized parametric distribution (in other words, the normal distribution) or any of its functions. In the same way that the basic ANOVA is a generalization of the two-sample *t*-test, the Kruskal–Wallis ANOVA is a generalization of the Mann–Whitney test for two medians. It's also referred to as the Kruskal–Wallis *rank sum* test, and you use the chi-squared distribution to calculate the *p*-value.

19.3.2 Example and Interpretation

Turn your attention to the data frame `survey`, located in the `MASS` package (`MASS` is included with every standard R installation). These data record particular characteristics of 237 first-year undergraduate statistics students collected from a class at the University of Adelaide, South Australia. Load the required package first with a call to `library("MASS")` and then enter `?survey` at the prompt. You can read the help file to understand which variables are present in the data frame.

Suppose you're interested to see whether the age of the students, `Age`, tends to differ with respect to four smoking categories reported in `Smoke`. An inspection of the relevant side-by-side box plots (refer to Section 14.4.2) and a normal QQ plot (Section 16.2.2) of the mean-centered residuals suggests a straightforward one-way ANOVA isn't necessarily a good idea. The following code mimics the steps in Section 19.1.1:

```
R> boxplot(Age~Smoke,data=survey)
R> age.means <- tapply(survey$Age,survey$Smoke,mean)
R> age.meancen <- survey$Age-age.means[as.numeric(survey$Smoke)]
R> qqnorm(age.meancen,main="Normal QQ plot of residuals")
R> qqline(age.meancen)
```

This produces the two images in Figure 19-3, which show the assumption of normality is questionable.

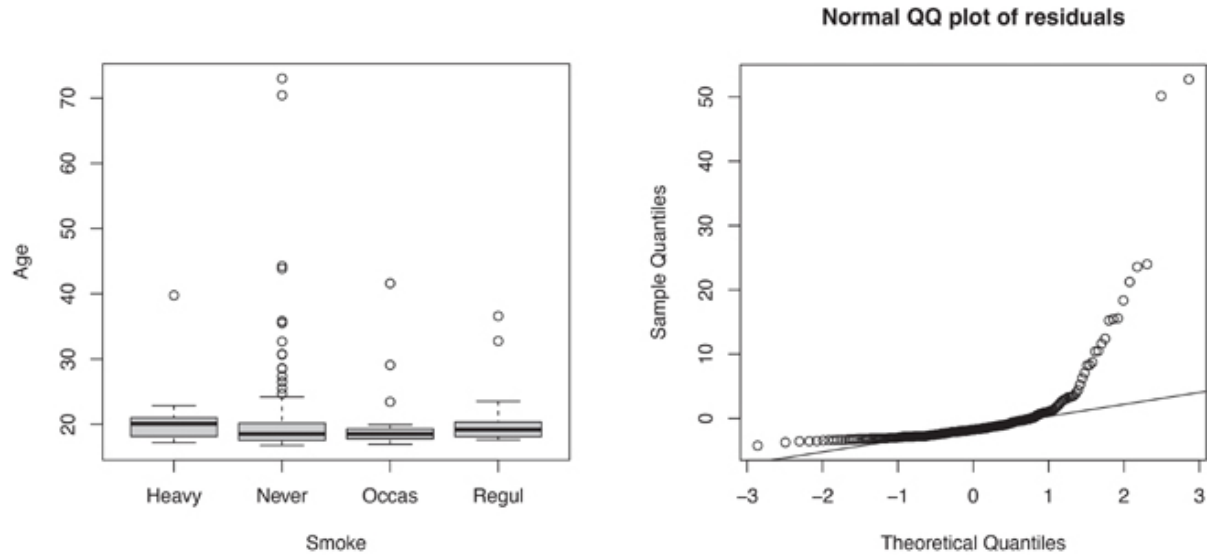


Figure 19-3: Side-by-side box plots (left) and a normal QQ plot of the residuals (right) for the student age observations split by smoking status

With this possible violation of normality, you could therefore apply the Kruskal–Wallis test instead of the parametric ANOVA. A quick check for equality of variances further supports this, with the ratio of the largest to the smallest sample standard deviations clearly being less than 2:

```
R> tapply(survey$Age,survey$Smoke,sd)
   Heavy    Never    Occas    Regul
6.332628 6.675257 5.861992 5.408822
```

In R, use `kruskal.test` to perform a Kruskal–Wallis test:

```
R> kruskal.test(Age~Smoke,data=survey)
```

```
        Kruskal-Wallis rank sum test

data:  Age by Smoke
Kruskal-Wallis chi-squared = 3.9262, df = 3, p-value = 0.2695
```

The syntax for this test is the same as for `aov`. As you might suspect from Figure 19-3, the large *p*-value suggests there's no evidence against the null hypothesis that states that the medians are all equal. In other words, there doesn't seem to be an overall age difference between the students in the four smoking categories.

Exercise 19.2

Recall the built-in `quakes` data frame, which describes the locations, magnitudes, depths, and number of observation stations that detected 1,000 seismic events off the coast of Fiji; this was first seen in Exercise 8.1 (page 183).

a. Use `cut` (see Section 4.3.7) to create a new factor vector defining the depths of each event according to the following three categories: (0, 200], (200, 400], and (400, 680].

b. Decide whether a one-way ANOVA or a Kruskal–Wallis test is more appropriate to use to compare the distributions of the number of detecting stations, split according to the three categories in (a).

c. Perform your choice of test in (b) (assume an $\alpha = 0.01$ level of significance) and conclude.

Load the `MASS` package with a call to `library("MASS")` if you haven't already done so in the current R session. This package includes the ready-to-use `Cars93` data frame, which contains detailed data on

93 cars for sale in the United States in 1993 (Lock 1993, Venables & Ripley 2002).

d. Use `aggregate` as illustrated in Section 19.2.1 to compute the mean length of the 93 cars, split by two categorical variables: `AirBags` (type of airbags available; levels are `Driver & Passenger`, `Driver only`, and `None`) and `Man.trans.avail` (whether the car comes in a manual transmission; levels are `Yes` and `No`).

e. Produce an interaction plot using the results in (d). Does there appear to be an interactive effect of `AirBags` with `Man.trans.avail` on the mean length of these cars (if you consider only these variables)?

f. Fit a full two-way ANOVA model for the mean lengths according to the two grouping variables (assume satisfaction of all relevant assumptions). Is the interactive effect statistically significant? Is there evidence of any main effects?

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`aov`	Produce ANOVA table	Section 19.1.3, p. 508
`aggregate`	Stacked statistics by factor	Section 19.2.1, p. 512
`interaction.plot`	Two-factor interaction plot	Section 19.2.2, p. 514
`kruskal.test`	Kruskal–Wallis test	Section 19.3.2, p. 517

20
SIMPLE LINEAR REGRESSION



Though straightforward comparative tests of individual statistics are useful in their own right, you'll often want to learn more from your data. In this chapter and the next, you'll look at *linear regression* models, which encompass methodology used to quantify relationships between variables.

Simple linear regression models describe the effect that a particular variable, called the *explanatory variable*, might have on the value of a continuous outcome variable, called the *response variable*. In general, explanatory variables may be continuous, discrete, or categorical, but the phrasing "simple linear regression" typically implies a single, continuous predictor variable. This will be the predominant focus of this chapter to introduce key concepts. In the final section, I'll cover how the representation of the model changes if the explanatory variable is categorical, in preparation for the chapter that follows.

20.1 An Example of a Linear Relationship

Let's start with an example using the student survey data from Section 19.3.2 and look at the data (the `survey` data frame in the package `MASS`) a little more closely. If you haven't already done so, with the required package loaded (call `library("MASS")`), you can read the help file `?survey` for details on the variables present.

Plot the student heights on the y-axis and their handspans (of their writing hand) on the x-axis with the following line:

```
R> plot(survey$Height~survey$Wr.Hnd,xlab="Writing handspan (cm)",
       ylab="Height (cm)")
```

Figure 20-1 shows the result.

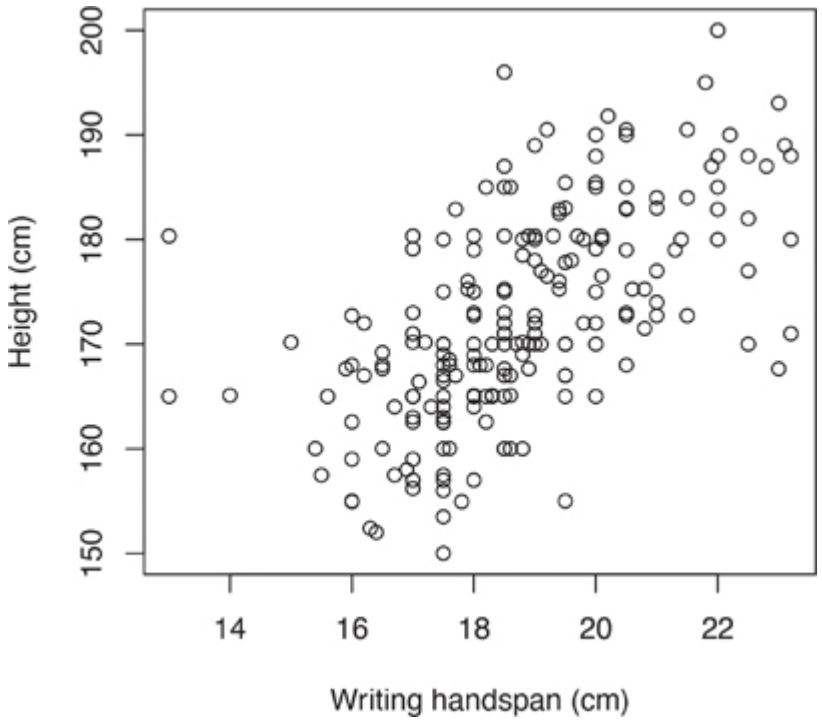


Figure 20-1: A scatterplot of height against writing handspan for a sample of first-year statistics students

Note that the call to `plot` uses formula notation (also referred to as *symbolic notation*) to specify "height *on* handspan." You can produce the same scatterplot by using the coordinate vector form of (*x*, *y*)—that is, `plot(survey$Wr.Hnd,survey$Height,...)`—but I'm using the symbolic notation

here because it nicely reflects how you'll fit the linear model in a moment.

Turning to the data themselves, as you might expect, there's a positive association between a student's handspan and their height. That relationship appears, at least visually, to be linear in nature. To assess the strength of the linear relationship, find the estimated correlation coefficient (see Section 13.2.6):

```
R> cor(survey$Wr.Hnd,survey$Height,use="complete.obs")
[1] 0.6009909
```

Though there are 237 records in the data frame, this particular plot doesn't actually show 237 points. This is because there are missing observations (coded `NA`; see Section 6.1.3) amongst the two variables of `survey` we've plotted. By default, R removes any "incomplete" data pairs when producing a plot like this, and other common commands often need to be explicitly instructed as to how `NA`s should be handled. In the previous use of `cor`, for example, you must also specify the optional argument `use="complete.obs"` so that the calculated statistic takes into account only those observation pairs in the `Wr.Hnd` and `Height` vectors where *neither* element is `NA`. You can think of this argument as behaving similarly to `na.rm=TRUE` in univariate summary statistic functions such as `mean` and `sd` (see examples of the R implementation of summary statistics in Section 13.2.1).

To find out how many offending observations have been deleted, use the short-form logical operator `|` (Section 4.1.3) in conjunction with `is.na` (Section 6.1.3) and `which` (Section 4.1.5). You then use `length` to discover there are 29 observation pairs where at least one of the two values of interest is missing:

```
R> incomplete.obs <- which(is.na(survey$Height)|is.na(survey$Wr.Hnd))
R> length(incomplete.obs)
[1] 29
```

20.2 General Concepts

The purpose of a linear regression model is to come up with a function that estimates the *mean* of one variable given a particular value of another variable. These variables are known as the *response variable* (the "outcome" variable whose mean you are attempting to find) and the *explanatory variable*(the "predictor" variable whose value you already have).

In terms of the student survey example, you might ask something like "What's the expected height of a student if their handspan is 14.5 cm?" Here, the response variable is the height, and the explanatory variable is the handspan.

20.2.1 Defining the Model

Assume you're looking to determine the value of response variable Y given the value of an explanatory variable X. The *simple linear regression model* states that the value of a response is expressed as the following equation:

$$Y \mid X = \beta_0 + \beta_1 X + \epsilon \qquad (20.1)$$

We read the notation $Y|X$ as "the value of Y conditional upon the value of X."

Residual Assumptions

The validity of the conclusions you can draw based on the model in Equation (20.1) is critically dependent on the assumptions made about ϵ, which are defined as follows:

- The value of ϵ is assumed to be normally distributed such that $\epsilon \sim N(0, \sigma)$.
- ϵ is centered (that is, has a mean of zero).
- The variance of ϵ, σ^2, is constant.

The ϵ term represents random error. In other words, you assume that any raw value of the response is owed to a linear change in a given value of X, plus or minus some random, *residual* variation or normally distributed *noise*.

Parameters

The value denoted by β_0 is called the *intercept*, and that of β_1 is called the *slope*. We refer to them as the *regression coefficients* and interpret them as follows:

- We interpret the intercept, β_0, as the expected value of the response variable when the predictor is zero.
- Generally, the slope, β_1, is the focus of interest. Interpret this as the change in the mean response for each one-unit increase in the predictor. When the slope is positive, the regression line increases from left to right (the mean response is higher when the predictor is higher); when the slope is negative, the line decreases from left to right (the mean response is lower when the predictor is higher). A slope of 0 implies that the predictor has no effect on the value of the response. The more extreme the value of β_1 (that is, away from 0), the steeper the increasing or decreasing line becomes.

20.2.2 Estimating the Intercept and Slope Parameters

The goal is to use your data to estimate the regression parameters, yielding the estimates $\widehat{\beta}_0$ and $\widehat{\beta}_1$; this is referred to as *fitting* the linear model. In this case, the data comprise n pairs of observations for each individual. The fitted model of interest concerns the mean response value, denoted $\hat{y}$, for a specific value of the predictor, x. We write it as follows:

$$\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x \qquad (20.2)$$

Sometimes, alternative notation such as $\mathbb{E}[Y]$ or $\mathbb{E}[Y|X = x]$ is used on the left side of Equation (20.2) to emphasize the fact that the model gives the mean (that is, the expected value) of the response. For compactness, many simply use something like $\hat{y}$, as shown here.

Let your n observed data pairs be denoted x_i and y_i for the predictor and response variables, respectively; $i = 1, \ldots, n$. Then, the parameter estimates for the simple linear regression function are

$$\widehat{\beta}_1 = \rho_{xy}\frac{s_y}{s_x} \quad \text{and} \quad \widehat{\beta}_0 = y - \widehat{\beta}_1 x \qquad (20.3)$$

where:

- x and y are the sample means of the x_is and y_is.
- s_x and s_y are the sample standard deviations of the x_is and y_is.
- ρ_{xy} is the estimate of correlation between X and Y based on the data (see Section 13.2.6).

Estimating the model parameters in this way is referred to as *least-squares regression*; the reason for this will become clear in a moment.

20.2.3 Fitting Linear Models with lm

In R, the command `lm` performs the estimation for you. For example, the following line creates a fitted linear model object of the mean student height by handspan and stores it in your global environment as `survfit`:

```
R> survfit <- lm(Height~Wr.Hnd,data=survey)
```

The first argument is the familiar `response ~ predictor` formula, which specifies the desired model. You don't have to use the `survey$` prefix to extract the vectors from the data frame because you specifically instruct `lm` to look in the object supplied to the `data` argument.

The fitted linear model object itself, `survfit`, has a particular class in R: one of `"lm"`. You can think of an object of class `"lm"` as a special list

containing several components that describe the model. You'll look at these in a moment.

If you simply enter the name of the `"lm"` object at the prompt, it will provide the most basic output: a repeat of your call and the estimates of the intercept ($\widehat{\beta}_0$) and slope ($\widehat{\beta}_1$):

```
R> survfit

Call:
lm(formula = Height ~ Wr.Hnd, data = survey)

Coefficients:
(Intercept)       Wr.Hnd
    113.954        3.117
```

This reveals that the linear model for this example is estimated as follows:

$$\hat{y} = 113.954 + 3.117x \qquad (20.4)$$

If you evaluate the mathematical function for $\hat{y}$, seen in Equation (20.2), at a range of different values for x, you end up with a straight line when you plot the results. Recall the definition of intercept given earlier as the expected value of the response variable when the predictor is zero. In the current example, this would imply that the mean height of a student with a handspan of 0 cm is 113.954 cm (an arguably less-than-useful statement since a value of 0 for the explanatory variable doesn't make sense in this context; you'll consider these and related issues in Section 20.4). The slope, or the change in the mean response for each one-unit increase in the predictor, is 3.117. This states that, on average, for every 1 cm increase in handspan, a student's height is estimated to increase by 3.117 cm.

Given a plot of the data like that in Figure 20-1, you can add the fitted regression line using `abline`. So far, you've only used the `abline` command to add perfectly horizontal and vertical lines to an existing plot (Section 7.3), but when passed an object of class `"lm"` that represents

a simple linear model, the fitted regression line will be added instead. The following code replicates the scatterplot of Figure 20-1 and adds the slightly thickened diagonally increasing line that represents the fitted model:

```
R> plot(survey$Height~survey$Wr.Hnd,xlab="Writing handspan (cm)",
        ylab="Height (cm)")
R> abline(survfit,lwd=2)
```

Figure 20-2 shows the result (you will add the vertical dashed lines with additional code in Section 20.2.4).

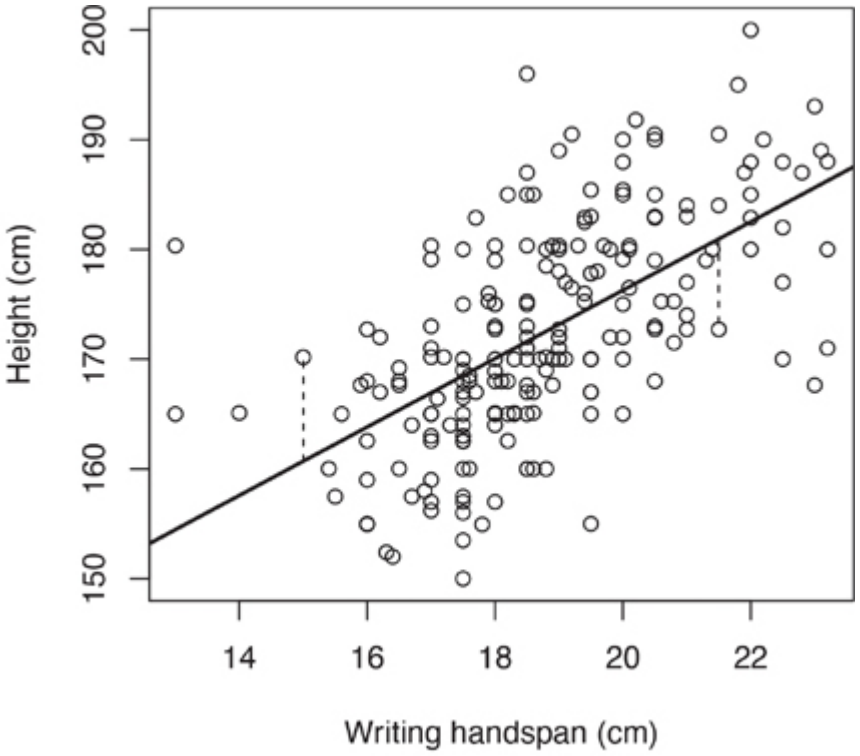


Figure 20-2: The simple linear regression line (solid, bold) fitted to the observed data for the student height vs. handspan example. Two dashed vertical line segments provide examples of a positive (leftmost) and negative (rightmost) residual.

20.2.4 Illustrating Residuals

When the parameters are estimated using Equation (20.3), the fitted line is referred to as an implementation of *least-squares regression* because the resulting line effectively minimizes the average squared difference

between the observed data and itself. This concept is easier to understand by drawing the distances between the observations and the fitted line, formally called *residuals*, for a couple of individual observations within Figure 20-2.

First, let's extract two specific records from the `Wr.Hnd` and `Height` data vectors and call the resulting vectors `obsA` and `obsB`:

```
R> obsA <- c(survey$Wr.Hnd[197],survey$Height[197])
R> obsA
[1]  15.00 170.18
R> obsB <- c(survey$Wr.Hnd[154],survey$Height[154])
R> obsB
[1]  21.50 172.72
```

Next, briefly inspect the names of the members of the `survfit` object:

```
R> names(survfit)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "na.action"     "xlevels"       "call"          "terms"
[13] "model"
```

These members are the components that automatically make up a fitted model object of class `"lm"`, mentioned earlier. Note the `"coefficients"` component, which contains a numeric vector of the estimates of the intercept and slope. You can extract this component (and any of the other ones listed here) in the same way you would perform a member reference on a named list (refer to Section 5.1.2): by entering `survfit$coefficients` at the prompt. Where possible, though, it's technically preferable for programming purposes to extract such components using a "direct-access" function. For the `coefficients` component of an `"lm"` object, you use the `coef` function:

```
R> mycoefs <- coef(survfit)
R> mycoefs
(Intercept)      Wr.Hnd
 113.953623    3.116617
```

```
R> beta0.hat <- mycoefs[1]
R> beta1.hat <- mycoefs[2]
```

Here, you've extracted the regression coefficients from the fitted model object and then separately assigned them to the `beta0.hat` and `beta1.hat` objects. Other common direct-access functions include `resid` and `fitted`; these two pertain to the `"residuals"` and `"fitted.values"` components, respectively.

Let's add the two vertical dashed lines to Figure 20-2; these represent the residuals of data points 197 and 154 as stored in `obsA` and `obsB`, respectively. First, ensure you have executed the code to initialize Figure 20-2, the scatterplot of the data with the fitted regression line superimposed. With the plot open, you can then use `segments` (Section 7.3) to draw the two specific residuals:

```
R> segments(x0=c(obsA[1],obsB[1]),y0=c(obsA[2],obsB[2]),
            x1=c(obsA[1],obsB[1]),y1=beta0.hat+beta1.hat*c(obsA[1],obsB[1]),
            lty=2)
```

Note that in use of `segments`, the dashed lines meet the fitted line at the vertical axis locations passed to `y1`, which, with the use of the regression coefficients `beta0.hat` and `beta1.hat`, reflects the line defined by Equation (20.4).

Now imagine a collection of alternative regression lines drawn through the data (achieved by altering the value of the intercept and slope). Then, for each of the alternative regression lines, imagine you calculate the residuals (vertical distances) between the response value of every observation and the *fitted value* of that line. The simple linear regression line estimated as per Equation (20.3) is the line that lies "closest to all observations." This means that the fitted regression model is represented by the line that passes through the coordinate provided by the variable means ($\bar{x}$, $\bar{y}$), and is the line that yields the smallest overall measure of the squared residual distances. For this reason, another name for a least-squares-estimated regression equation like this is the *line of best fit*.

20.3 Statistical Inference

Estimating a regression equation is relatively straightforward, but this is merely the beginning. You should now think about what you can infer from your result. Simple linear regression usually seeks to answer the question of whether there is statistical evidence to support the presence of a relationship between the predictor and the response. To put it another way, is there evidence that a change in the explanatory variable affects the mean outcome? You'll investigate this by following the ideas introduced in Chapters 17 and 18, when you began thinking about the variability present in estimated statistics, constructed confidence intervals and executed hypothesis tests.

20.3.1 The summary Command

Model-based inference is automatically carried out by R when we examine the details of an `lm` object. For the moment, you'll focus on just two aspects of this information: the significance tests associated with the regression coefficients and the interpretation of the so-called *coefficient of determination* (labeled `R-squared` in the output), which I'll explain shortly.

In brief, use the `summary` command on a fitted model object of class `"lm"` to obtain a detailed report of the fit. Do so to `survfit`, our running example from the previous sections, and you'll see the following:

```
R> summary(survfit)

Call:
lm(formula = Height ~ Wr.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-19.7276  -5.0706  -0.8269   4.9473  25.8704

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9536     5.4416   20.94   <2e-16 ***
```

```
Wr.Hnd        3.1166     0.2888   10.79   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.909 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3612,   Adjusted R-squared:  0.3581
F-statistic: 116.5 on 1 and 206 DF,  p-value: < 2.2e-16
```

I'll refer to this output in the following discussion.

20.3.2 Regression Coefficient Significance Tests

Let's begin by focusing on the way the estimated regression coefficients are reported. The first column of the coefficients table contains the point estimates of the intercept and slope (the intercept is labeled as such, and the slope is labeled after the name of the predictor variable in the data frame); the table also includes estimates of the standard errors of these statistics. It can be shown that simple linear regression coefficients, when estimated using least-squares, follow a t-distribution with $n - 2$ degrees of freedom (when given the number of observations, n, used in the model fit). The standardized t value and a p-value are reported for each parameter. These represent the results of a two-tailed hypothesis test formally defined as

$$\mathrm{H}_0 : \beta_j = 0$$

$$\mathrm{H}_\mathrm{A} : \beta_j \neq 0$$

where $j = 0$ for the intercept and $j = 1$ for the slope, using the notation in Equation (20.1).

Focus on the row of results for the predictor `Wr.Hnd`. With a null value of zero, truth of H_0 implies that the predictor has no effect on the response. The claim here is interested in whether there is *any effect* of the covariate, not the direction of this effect, so H_A is two-sided (via $\neq$). As with any hypothesis test, the smaller the p-value, the stronger the

evidence against H_0. With a small p-value ($< 2 \times 10^{-16}$) attached to this particular test statistic (which you can confirm using the formula given in Equation (18.2): $T = (3.116 - 0) / 0.2888 = 10.79$), you'd therefore conclude there is strong evidence *against* the claim that the predictor has no effect on the mean level of the response.

The same test is carried out for the intercept, but the test for the slope parameter β_1 is typically more interesting (since rejection of the null hypothesis for β_0 simply indicates evidence that the regression line does not strike the vertical axis at 0), especially when the observed data don't include $x = 0$, as is the case here.

From this, you can conclude that the fitted model suggests there is evidence that an increase in handspan is associated with an increase in height among the population being studied. For each additional centimeter of handspan, the average increase in height is approximately 3.12 cm.

You could also produce confidence intervals for your estimates using Equation (17.2) on page 446 and knowledge of the sampling distributions of the regression parameters; however, yet again, R provides a convenient function for an object of class `"lm"` to do this for you:

```
R> confint(survfit,level=0.95)
                2.5 %     97.5 %
(Intercept) 103.225178 124.682069
Wr.Hnd        2.547273   3.685961
```

To the `confint` function you pass your model object as the first argument and your desired level of confidence as `level`. This indicates that you should be 95 percent confident that the true value of β_1 lies somewhere between 2.55 and 3.69 (to 2 d.p.). As usual, the exclusion of the null value of zero (the interval is wholly positive) reflects the statistically significant result from earlier.

20.3.3 Coefficient of Determination

The output of `summary` also provides you with the values of `Multiple R-squared` and `Adjusted R-squared`, which are particularly interesting. Both of these are referred to as the *coefficient of determination*; they describe the proportion of the variation in the response that can be attributed to the predictor.

For simple linear regression, the first (unadjusted) measure is simply obtained as the square of the estimated correlation coefficient (refer to Section 13.2.6). For the student height example, first store the estimated correlation between `Wr.Hnd` and `Height` as `rho.xy`, and then square it:

```
R> rho.xy <- cor(survey$Wr.Hnd,survey$Height,use="complete.obs")
R> rho.xy^2
[1] 0.3611901
```

You get the same result as the `Multiple R-squared` value (usually written mathematically as R^2). This tells you that about 36.1 percent of the variation in the student heights can be attributed to writing handspan.

The adjusted measure is an alternative estimate that takes into account the number of parameters that require estimation. The adjusted measure is generally important only if you're using the coefficient of determination to assess the overall "quality" of the fitted model in terms of a balance between goodness of fit and complexity. I cover this concept in greater detail in Chapter 22.

20.3.4 Other summary Output

The `summary` of the model object provides you with various other pieces of useful information. The "residual standard error" is the estimated standard error of the ϵ term (in other words, the square root of the estimated variance of ϵ, namely, σ^2—refer to Equation (20.1)); below that, it also reports any missing values. The 29 observation pairs "deleted due to missingness" match the number of incomplete observations determined in Section 20.1.

The output also provides a five-number summary (Section 13.2.4) for the residual distances; I cover this further in Section 22.3. As the

final result, you're provided with a certain hypothesis test performed using the *F*-distribution (Section 16.2.6). This is a global test of the impact of your predictor(s) on the response; you'll explore this alongside multiple linear regression in Section 21.3.5.

You can access all the output provided by `summary` directly as individual R objects, rather than having to read them off the screen from the entire printed summary. Just as `names(survfit)` provides you with an indication of the contents of the stand-alone `survfit` object, the following code gives you the names of all the components accessible *after* using `summary` to process `survfit`:

```
R> names(summary(survfit))
[1] "call"          "terms"         "residuals"     "coefficients"
[5] "aliased"       "sigma"         "df"            "r.squared"
[9] "adj.r.squared" "fstatistic"    "cov.unscaled"  "na.action"
```

It's fairly easy to match most of the components with the printed `summary` output, and they can be extracted using the `$` operator as usual. For example, you can retrieve the residual standard error directly with this:

```
R> summary(survfit)$sigma
[1] 7.90878
```

For more details on this behavior, execute `?summary.lm` at the R prompt.

20.4 Prediction

To wrap up these preliminary details of linear regression, you'll now look at using your fitted model for predictive purposes. The ability to fit a statistical model means that you not only can understand and quantify the nature of relationships in your data (like the estimated 3.1166 cm increase in mean height per 1 cm increase in handspan for the student example) but also can *predict* values of the outcome of interest for any predictor value(s). As with any statistic, though, you should always accompany any point estimates or predictions with a measure of spread.

20.4.1 Confidence Interval or Prediction Interval?

A fitted simple linear model allows you to calculate a point estimate of the *mean response* value, conditional upon the value of an explanatory variable. To do so, simply plug in (to the fitted model equation) the value of x you're interested in. As statistics like this are always subject to variation, you use a *confidence interval (CI)* for the mean response to gauge this uncertainty, just like you did with sample statistics in earlier chapters.

Assume a simple linear regression line has been fitted to n observations such that $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x$. You calculate a $100(1 - \alpha)$ percent confidence interval for the mean response given a value of x with

$$\hat{y} \pm t_{(1-\alpha/2,n-2)} s_{\epsilon} \sqrt{\frac{1}{n} + \frac{(x-x)^2}{(n-1)s_x^2}} \qquad (20.5)$$

where, as usual, you obtain the lower limit by subtraction and the upper limit by addition.

Here, $\hat{y}$ is the fitted value (from the regression line) at x; $t_{(1-\alpha/2,n-2)}$ is the appropriate critical value from a t-distribution with $n - 2$ degrees of freedom (in other words, resulting in an uppertail area of exactly $\alpha/2$); s_{ϵ} is the estimated residual standard error; and $\overline{x}$ and s_x^2 represent the sample mean and the variance of the observations of the predictor, respectively.

A *prediction interval (PI)* for an observed response is different from the confidence interval in terms of context. While we use CIs to describe the variability of the *mean* response, we use a PI to provide the possible range of values that an *individual realization* of the response variable might take, given x. This distinction is subtle but important: The CI corresponds to a mean, and the PI corresponds to an individual observation.

Let's remain with the previous notation. It can be shown that a $100(1 - \alpha)$ percent prediction interval for an individual response given a value of x is calculated with the following:

$$\hat{y} \pm t_{(1-\alpha/2,n-2)} s_\epsilon \sqrt{1 + \frac{1}{n} + \frac{(x-x)^2}{(n-1)s_x^2}} \qquad (20.6)$$

It transpires that the only difference from Equation (20.5) is the extra 1+ that appears in the square root. As such, a PI at x is always wider than a CI at x.

20.4.2 Interval Interpretation

Continuing with our running example, let's say you want to determine the mean height for students with a handspan of 14.5 cm and for students with a handspan of 24 cm. To calculate the point estimates, just plug the desired x values into the regression Equation (20.4) like so:

```
R> as.numeric(beta0.hat+beta1.hat*14.5)
[1] 159.1446
R> as.numeric(beta0.hat+beta1.hat*24)
[1] 188.7524
```

According to the model, you can expect mean heights to be around 159.14 and 188.75 cm for handspans of 14.5 and 24 cm, respectively. The `as.numeric` coercion function (first encountered in Section 6.2.4) simply strips the result of the annotative names that are otherwise present from the `beta0.hat` and `beta1.hat` objects.

Confidence Intervals for Mean Heights

While you could calculate confidence intervals for these estimates manually using Equation (20.5), R's built-in `predict` command can do it for you. To use `predict`, you first need to store your x values as a column in a new data frame. The name of the column *must match that of the predictor* used in the original call to create the fitted model object. In this example, I'll create a new data frame, `xvals`, with the column named `Wr.Hnd`, which contains only two values of interest, the handspans of 14.5 and 24 cm:

```
R> xvals <- data.frame(Wr.Hnd=c(14.5,24))
R> xvals
```

```
  Wr.Hnd
1   14.5
2   24.0
```

Now, when `predict` is called, the first argument must be the fitted model object of interest, `survfit` for this example. Next, in the argument `newdata`, you pass the specially constructed data frame containing the specified predictor values. To the `interval` argument you must specify `"confidence"` as a character string value. The confidence level, here set for 95 percent, is passed (on the scale of a probability) to `level`:

```
R> mypred.ci <- predict(survfit,newdata=xvals,interval="confidence",level=0.95)
R> mypred.ci
       fit      lwr      upr
1 159.1446 156.4956 161.7936
2 188.7524 185.5726 191.9323
```

This call will return a matrix with three columns, whose number (and order) of rows correspond to the predictor values you supplied in the `newdata` data frame. The first column, with a heading of `fit`, is the point estimate on the regression line; these numbers match the values you worked out earlier. The other columns provide the lower and upper CI limits as the `lwr` and `upr` columns, respectively. In this case, you'd interpret this as 95 percent confidence that the mean height of a student with a handspan of 14.5 cm lies somewhere between 156.5 cm and 161.8 cm and lies between 185.6 cm and 191.9 cm for a handspan of 24 cm (when rounded to 1 d.p.). Remember, these CIs, calculated as per Equation (20.5) through `predict`, are for the *mean* response value.

Prediction Intervals for Individual Observations

The `predict` function will also provide your prediction intervals. To find the prediction interval for possible individual observations with a certain probability, you simply need to change the `interval` argument to `"prediction"`:

```
R> mypred.pi <- predict(survfit,newdata=xvals,interval="prediction",level=0.95)
R> mypred.pi
       fit      lwr      upr
1 159.1446 143.3286 174.9605
2 188.7524 172.8390 204.6659
```

Notice the fitted values remain the same, as Equations (20.5) and (20.6) indicate. The widths of the PIs, however, are significantly larger than those of the corresponding CIs; this is because raw observations themselves, at a specific x value, will naturally be more variable than their mean.

Interpretation changes accordingly. The intervals describe where raw student heights are predicted to lie "95 percent of the time." For a handspan of 14.5 cm, the model predicts individual observations to lie somewhere between 143.3 cm and 175.0 cm with a probability of 0.95; for a handspan of 24 cm, the same level of PI is estimated at 172.8 cm and 204.7 cm (when rounded to 1 d.p.).

20.4.3 Interval Plots

Both CIs and PIs are well suited to visualization for simple linear regression models. With the following code, you can start off Figure 20-3 by plotting the data and estimated regression line just as for Figure 20-2, but this time using `xlim` and `ylim` in `plot` to widen the x and y-limits a little in order to accommodate the full length and breadth of the CI and PI:

```
R> plot(survey$Height~survey$Wr.Hnd,xlim=c(13,24),ylim=c(140,205),
        xlab="Writing handspan (cm)",ylab="Height (cm)")
R> abline(survfit,lwd=2)
```

To this you add the locations of the fitted values for x = 14.5 and x = 24, as well as two sets of vertical lines showing the CIs and PIs:

```
R> points(xvals[,1],mypred.ci[,1],pch=8)
R> segments(x0=c(14.5,24),y0=c(mypred.pi[1,2],mypred.pi[2,2]),
           x1=c(14.5,24),y1=c(mypred.pi[1,3],mypred.pi[2,3]),col="gray",lwd=3)
```

```
R> segments(x0=c(14.5,24),y0=c(mypred.ci[1,2],mypred.ci[2,2]),
            x1=c(14.5,24),y1=c(mypred.ci[1,3],mypred.ci[2,3]),lwd=2)
```

The call to `points` marks the fitted values for these two particular values of *x*. The first call to `segments` lays down the PIs as thickened vertical gray lines, and the second lays down the CIs as the shorter vertical black lines. The coordinates for these plotted line segments are taken directly from the `mypred.pi` and `mypred.ci` objects, respectively.

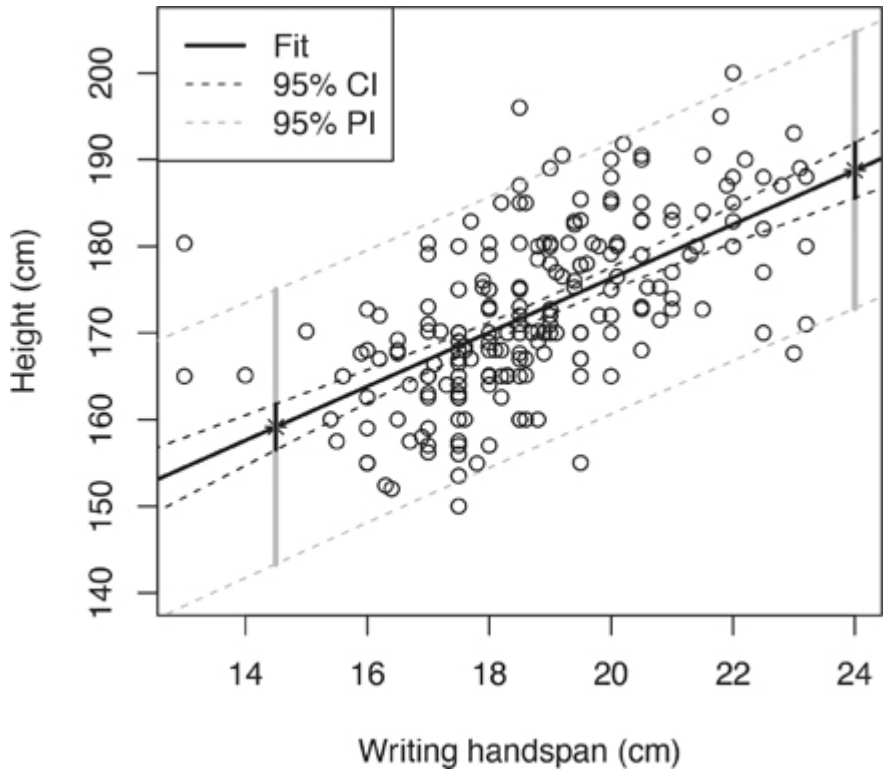


Figure 20-3: The student height regression example, showing the fitted regression line, point estimates at x *= 14.5 and* x *= 24, and 95 percent CIs and PIs*

You can also produce "bands" around the fitted regression line that mark one or both of these intervals over *all* values of the predictor. From a programming standpoint, this isn't technically possible for a continuous variable, but you can easily approximate it by defining a fine sequence of values along the x-axis (using `seq` with a high `length` value) and evaluating the CI and PI at every point in this fine sequence. Then you just plot this sequence of coordinates as lines.

In R, this type of operation requires you to rerun the `predict` command as follows:

```
R> xseq <- data.frame(Wr.Hnd=seq(12,25,length=100))
R> ci.band <- predict(survfit,newdata=xseq,interval="confidence",level=0.95)
R> pi.band <- predict(survfit,newdata=xseq,interval="prediction",level=0.95)
```

The first line in this code creates the fine sequence of predictor values and stores it in the format required by the `newdata` argument. The y-axis coordinates for CI and PI bands are stored as the second and third columns of the matrix objects `ci.band` and `pi.band`. Finally, `lines` adds each of the four dashed lines corresponding to the upper and lower limits of the two intervals, and a legend adds a final touch:

```
R> lines(xseq[,1],ci.band[,2],lty=2)
R> lines(xseq[,1],ci.band[,3],lty=2)
R> lines(xseq[,1],pi.band[,2],lty=2,col="gray")
R> lines(xseq[,1],pi.band[,3],lty=2,col="gray")
R> legend("topleft",legend=c("Fit","95% CI","95% PI"),lty=c(1,2,2),
          col=c("black","black","gray"),lwd=c(2,1,1))
```

Figure 20-3 shows the end result of all these additions to the plot. Note that the black dashed CI bands meet the vertical black lines and the gray dashed PI bands meet the vertical gray lines for the two individual *x* values from earlier, just as you'd expect. The "bowing inward" curvature of the intervals is characteristic of this kind of plot and is especially visible in the CI. This curve occurs because there is naturally less variation if you're predicting where there are more data. For more information on `predict` for linear model objects, take a look at the `?predict.lm` help file.

20.4.4 Interpolation vs. Extrapolation

Before finishing this introduction to prediction, it's important to clarify the definitions of two key terms: interpolation and extrapolation. These terms describe the nature of a given prediction. A prediction is referred to as *interpolation* if the *x* value you specify falls within the range of your

observed data; *extrapolation* is when the x value of interest lies outside this range. For the point predictions you just made, the location x = 14.5 is an example of interpolation, and x = 24 is an example of extrapolation.

In general, interpolation is preferable to extrapolation: It makes more sense to use a fitted model for prediction in the vicinity of data that have already been observed. Extrapolation that isn't too far out of that vicinity may still be considered reliable, though. The extrapolation for the student height example at x = 24 is a case in point. This is outside the range of the observed data, but not by much in terms of scale, and the estimated intervals for the expected value of $\hat{y} = 188.75$ cm appear, at least visually, not unreasonable given the distribution of the other observations. In contrast, it would make less sense to use the fitted model to predict student height at a handspan of, say, 50 cm:

```
R> predict(survfit,newdata=data.frame(Wr.Hnd=50),interval="confidence",
           level=0.95)
       fit      lwr      upr
1 269.7845 251.9583 287.6106
```

Such extreme extrapolation suggests that the mean height of an individual with a handspan of 50 cm is almost 270 cm, both being fairly unrealistic measurements. The same is true in the other direction; the intercept $\widehat{\beta}_0$ doesn't have a particularly useful practical interpretation, indicating that the mean height of a student with a handspan of 0 cm is around 114 cm.

The main message here is to use common sense when making any prediction from a linear model fit. In terms of the reliability of the results, predictions made at values within an appropriate proximity of the observed data are preferable.

Exercise 20.1

Continue to use the `survey` data frame from the package `MASS` for the next few tasks.

a. Using your fitted model of student height on writing handspan, (the `survfit` object from the text), provide point estimates and 99 percent confidence intervals for the mean student height for handspans of 12, 15.2, 17, and 19.9 cm.

b. At the end of Section 20.1 you defined the `incomplete.obs` object, a numeric vector that provides the records of `survey` that were automatically removed from consideration when estimating the model parameters. Now use the `incomplete.obs` vector along with `survey` and Equation (20.3) to calculate $\widehat{\beta}_0$ and $\widehat{\beta}_1$ (remember the functions `mean`, `sd`, and `cor`; ensure your answers match the output from `survfit`).

c. The `survey` data frame has a number of other variables present aside from `Height` and `Wr.Hnd`. For this exercise, the end aim is to fit a simple linear model to predict the mean student height, but this time from their pulse rate, given in `Pulse` (continue to assume the conditions listed in Section 20.2 are satisfied).

    i. Fit the regression model and produce a scatterplot with the fitted line superimposed upon the data. Make sure you can write down the fitted model equation. Keep the plot open.

    ii. Identify and interpret the point estimate of the slope, as well as the outcome of the test associated with the hypotheses $H_0 : \beta_1 = 0$; $H_A : \beta_1 \neq 0$. In addition, find a 90 percent CI for the slope parameter.

    iii. Based on the example in Section 20.4.3, use your model to add lines for 90 percent confidence and prediction intervals, shown as "bands," to the plot

from (i). Add a legend to differentiate between the lines.

iv. Based on the code from the end of Section 20.1, create an analogous `incomplete.obs` vector for the current "height on pulse" data. Use that vector to calculate the sample mean of the height observations that were used for the model fitted in (i). Then add a perfectly horizontal line to the plot at this mean (use color or line type options to avoid confusion with the other lines present). What do you notice? Does the plot support your conclusions from (ii)?

Next, examine the help file for the `mtcars` dataset, which you first saw in Section 9.3.1. For this exercise, the goal is to model fuel efficiency, measured in miles per gallon (MPG), in terms of the overall weight of the vehicle (in thousands of pounds).

d. Plot the data: `mpg` on the y-axis and `wt` on the x-axis.

e. Fit the simple linear regression model as described. Add the fitted line to the plot from (d).

f. Write down the regression equation and interpret the point estimate of the slope. Is the effect of `wt` on mean `mpg` statistically significant?

g. Produce a point estimate and associated 95 percent PI for a car that weighs 6,000 lbs. Do you trust the model to predict observations accurately for this value of the explanatory variable? Why or why not?

20.5 Understanding Categorical Predictors

So far, you've looked at simple linear regression models, in which the continuous response variable Y is modelled as a function of a single continuous predictor variable. In linear regression it's also quite possible to use a discrete or categorical explanatory variable—a variable defined as having k distinct groups or levels—as a predictor for the continuous response. While this scenario moves beyond what people typically refer to by "simple" linear regression, we still make the same assumptions noted in Section 20.2: that observations are all independent of one another and residuals are normally distributed with an equal variance.

To begin with, you'll look at the simplest case in which you're dealing with a binary-valued categorical predictor ($k = 2$). I'll then demonstrate how this forms the basis of the more complicated situation in which the categorical predictor has more than two levels (a *multilevel* predictor: $k > 2$), which also serves to prime the discussion on multiple linear regression coming up in the following chapter.

20.5.1 Working with Binary Variables: k = 2

Turn your attention back to Equation (20.1), where the regression model is specified as $Y|X = \beta_0 + \beta_1 X + \epsilon$ for a response variable Y and predictor X, and $\epsilon \sim \text{N}(0, \sigma^2)$. Now suppose your predictor variable is categorical, with only two possible levels (binary; $k = 2$) and observations coded either 0 or 1. For this case, Equation (20.1) still holds, but the interpretation of the model parameters, β_0 and β_1, isn't really one of an "intercept" and a "slope" anymore. Instead, it's better to think of them as being something like two intercepts, where β_0 provides the *baseline* or *reference* value of the response when $X = 0$ and β_1 represents the *additive effect* on the mean response if $X = 1$. In other words, when $X = 0$, then $Y = \beta_0 + \epsilon$; when $X = 1$, then $Y = \beta_0 + \beta_1 + \epsilon$. As usual, estimation is in terms of the *mean* response $\hat{y} \equiv \mathbb{E}[Y|X = x]$ as per Equation (20.2), so the equation for the fitted model becomes $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x$.

Go back to the `survey` data frame and note the variable `Sex`, where the students recorded their biological sex. Look at the documentation on the help page `?survey`, execute `str(survey)`, or enter something like the following in order to understand the form of the `Sex` variable:

```
R> class(survey$Sex)
[1] "factor"
R> table(survey$Sex)

Female   Male
   118    118
```

You'll see that the `Sex` column is a factor vector with two levels, `Female` and `Male`, and that there happens to be an equal number of the two (1 of the 237 records has a missing value for this variable).

You're going to determine whether there is statistical evidence that the height of a student is affected by sex. This means that you're again interested in modeling height as the response variable, but this time, you'll use the categorical variable `Sex` as the predictor.

Visualize the data by making the following call to `plot`, which results in a pair of box plots:

```
R> plot(survey$Height~survey$Sex)
```

This results in box plots because the response variable specified to the left of the `~` is numeric and the explanatory variable to the right is a factor, and the default behavior of R in that situation is to produce side-by-side box plots (recall the examples in Section 14.4.2).

To further emphasize the categorical nature of the explanatory variable, you can superimpose the raw height and sex observations on top of the box plots. To do this, just convert the factor vector to numeric with a call to `as.numeric`; this can be done directly in a call to `points`:

```
R> points(survey$Height~as.numeric(survey$Sex),cex=0.5)
```

Remember that box plots mark off the median as the central bold line but that least-squares linear regressions are defined by the mean outcome, so it's useful to also display the mean heights according to sex:

```
R> means.sex <- tapply(survey$Height,INDEX=survey$Sex,FUN=mean,na.rm=TRUE)
R> means.sex
  Female     Male
165.6867 178.8260
R> points(1:2,means.sex,pch=4,cex=2)
```

You were introduced to `tapply` in Section 10.4.2; in this call, the argument `na.rm=TRUE` is matched to the ellipsis in the definition of `tapply` and is passed to `mean` (you need it to ensure the missing values present in the data do not end up producing `NA`s as the results). A further call to `points` adds those coordinates (as × symbols) to the image; Figure 20-4 gives the final result.

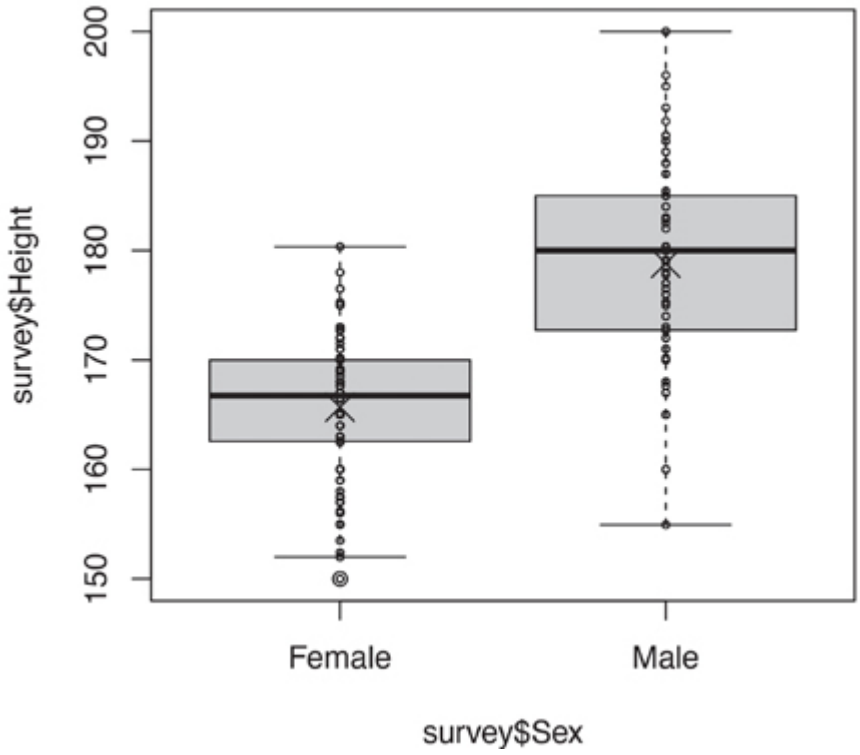


Figure 20-4: Box plots of the student heights split by sex, with the raw observations and sample means (small ∘ and large × symbols, respectively) superimposed

Overall, the plot indicates that males tend to be taller than females, but is there statistical evidence of a difference to back this up?

Linear Regression Model of Binary Variables

To investigate a relationship between these two variables with a simple linear regression model, you can use `lm` to produce least-squares estimates just like every other linear model you've fitted so far:

```
R> survfit2 <- lm(Height~Sex,data=survey)
R> summary(survfit2)

Call:
lm(formula = Height ~ Sex, data = survey)

Residuals:
    Min      1Q  Median      3Q     Max
-23.886  -5.667   1.174   4.358  21.174

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  165.687      0.730  226.98   <2e-16 ***
SexMale       13.139      1.022   12.85   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 7.372 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.4449, Adjusted R-squared:  0.4422
F-statistic: 165.1 on 1 and 206 DF,  p-value: < 2.2e-16
```

However, because the predictor is a factor object (instead of a numeric vector), the labeling of the coefficients is slightly different. The estimate of β_0 is again reported as (`Intercept`); this is the estimate of the mean height if a student is female. The estimate of β_1 is reported as `SexMale`. The corresponding regression coefficient of 13.139 is the estimated difference that is imparted upon the mean height of a student if that student is male as opposed to female. If you look at the corresponding regression equation

$$\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x = 165.687 + 13.139x \qquad (20.7)$$

you can see that the model has been fitted assuming the variable x is defined as "the individual is male," taking on the value 0 for no/false and 1 for yes/true. In other words, the level of "female" for the sex variable is taken to be a reference point of sorts, and it's the offsetting effect of "being male" on mean height that's explicitly estimated. The hypothesis test for β_0 and β_1 is performed with the same hypotheses defined in Section 20.3.2:

$$\mathrm{H}_0 : \beta_j = 0$$

$$\mathrm{H}_\mathrm{A} : \beta_j \neq 0$$

Again, it's the test for β_1 that's generally of the most interest since it's this value that tells you whether there is statistical evidence that the mean response variable is affected by the explanatory variable—that is, if β_1 is significantly different from zero.

Predictions from a Binary Categorical Variable

Because there are only two possible values for x, prediction is straightforward here. When you evaluate the equation, the only decision that needs to be made is whether to use $\widehat{\beta}_1$ (in other words, if an individual is male) or not (if an individual is female). For example, you can enter the following code to create a factor of five extra observations with the same level names as the original data and store the new data in `extra.obs`:

```
R> extra.obs <- factor(c("Female","Male","Male","Male","Female"))
R> extra.obs
[1] Female Male   Male   Male   Female
Levels: Female Male
```

Then, use `predict` in the now-familiar fashion to find the mean heights at those extra values of the predictor:

```
R> predict(survfit2,newdata=data.frame(Sex=extra.obs),interval="confidence",
           level=0.9)
       fit      lwr      upr
1 165.6867 164.4806 166.8928
2 178.8260 177.6429 180.0092
3 178.8260 177.6429 180.0092
4 178.8260 177.6429 180.0092
5 165.6867 164.4806 166.8928
```

Remember that when you pass in new data to `predict` using the `newdata` argument, the predictors must be in exactly the same form as the data that were used to fit the model in the first place.

You can see from the output that the predictions are different only between the two sets of values: The point estimates of the two instances of `Female` are identical, simply $\widehat{\beta}_0$ with 90 percent CIs. The point estimates and CIs for the instances of `Male` are also all the same as each other, based on a point estimate of $\widehat{\beta}_0 + \widehat{\beta}_1$.

On its own, this example isn't super exciting. However, it's critical to understand how R presents regression results when using categorical predictors, especially when considering multiple regression.

20.5.2 Working with Multilevel Variables: k > 2

It's common to work with data where the categorical predictor variables have more than two levels—that is, when $k > 2$. We refer to these as *multilevel* categorical variables. To deal with this more complicated situation while retaining interpretability of your parameters, we typically first "dummy code" a multilevel predictor into $k - 1$ binary variables.

Dummy Coding Multilevel Variables

To see how dummy coding works, assume that you want to find the value of response variable Y when given the value of a categorical variable X, where X has $k > 2$ levels (also assume for the purposes of this

discussion that the conditions for validity of the linear regression model —Section 20.2—are satisfied).

In regression modeling, *dummy coding* is the procedure used to create several binary variables from a categorical variable like X. Instead of the single categorical variable with possible realizations

$$X = 1, 2, 3, \ldots, k$$

you recode it into several yes/no variables (one for each level) with possible realizations:

$$X_{(1)} = 0, 1; X_{(2)} = 0, 1; X_{(3)} = 0, 1; \ldots; X_{(k)} = 0, 1$$

As you can see, $X_{(i)}$ represents a binary variable for the ith level of the original X. For example, if an individual has $X = 2$ in the original categorical variable, then $X_{(2)} = 1$ ("yes") and all of the others ($X_{(1)}$, $X_{(3)}$, . . . , $X_{(k)}$) will be zero ("no").

Suppose X is a variable that can take any one of the $k = 4$ values 1, 2, 3, or 4, and you've made six observations of this variable: 1, 2, 2, 1, 4, 3. Table 20-1 shows these observations and their dummy-coded equivalents $X_{(1)}$, $X_{(2)}$, $X_{(3)}$, and $X_{(4)}$.

Table 20-1: Dummy Coding for Six Observations of a Categorical Variable with $k = 4$ Groups

X	$X_{(1)}$	$X_{(2)}$	$X_{(3)}$	$X_{(4)}$
1	1	0	0	0
2	0	1	0	0
2	0	1	0	0
1	1	0	0	0
4	0	0	0	1

X	$X_{(1)}$	$X_{(2)}$	$X_{(3)}$	$X_{(4)}$
3	0	0	1	0

In fitting the subsequent model, you use only $k - 1$ of the dummy binary variables; the remaining one acts as a *reference* or *baseline* level, and it's incorporated into the overall intercept of the model. In practice, you would end up with an estimated model like

$$\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 X_{(2)} + \widehat{\beta}_2 X_{(3)} + \ldots + \widehat{\beta}_{k-1} X_{(k)} \qquad (20.8)$$

assuming category 1 is the reference level. As you can see, in addition to the overall intercept term $\widehat{\beta}_0$, you have $k - 1$ other estimated intercept terms that modify the baseline coefficient $\widehat{\beta}_0$, depending on which of the original categories an observation takes on. For example, in light of the coding we have imposed, if a particular observation is $X = 3$, this means $X_{(3)} = 1$ and all other binary values will be 0. In turn, looking at Equation (20.8), the predicted mean value of the response for the observation $X = 3$ would be $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_2$. On the other hand, because the reference level is defined as 1, if an observation falls into this category, we'll have values of 0 for all the binary variables $X_{(2)}$, $X_{(3)}$, and $X_{(4)}$. In this case, the prediction of the response variable would simply be $\hat{y} = \widehat{\beta}_0$.

It's necessary to dummy code for multilevel categorical variables because, in general, categories cannot be related to each other in the same numeric sense as continuous variables. It's often not appropriate, for example, to think that an observation in category 4 is "twice as much" as one in category 2, which is what a fitted model using a numeric predictor would imply. Binary yes/no variables *are* valid, however, and can be easily incorporated into this framework as you saw in Section 20.5.1. Choosing the reference level is generally of secondary importance: The specific values of the estimated coefficients will change

accordingly, but any overall interpretations you make based on the fitted model will be the same regardless.

NOTE

Implementing this dummy-coding approach is technically a form of multiple regression since you're now including several binary variables in the model. It's important, however, to be aware of the somewhat artificial nature of dummy coding; you should still think of the multiple coefficients as representing a single categorical variable since the binary variables $X_{(1)}$, . . . , $X_{(k)}$ *are not independent of one another. This is why I've chosen to define these models in this chapter; Chapter 21 will formally discuss multiple regression.*

Linear Regression Model of Multilevel Variables

R makes working with categorical predictors in this way quite simple since it automatically dummy codes for any such explanatory variable when you call `lm`. There are two things you should check before fitting your model, though:

- The categorical variable of interest should be stored as a (formally unordered) factor (Section 4.3).
- Confirm that you're happy with the category assigned as the reference level (for interpretative purposes; this will be discussed in Section 20.5.3).

In general, you must also of course be happy with the validity of the familiar assumptions of normality and independence of ϵ (refer back to Section 20.2.1).

To demonstrate all these definitions and ideas, let's return to the student survey data from the `MASS` package and keep "student height" as the response variable of interest. Among the data is the variable `Smoke`. This variable describes the kind of smoker each student reports

themselves as, defined by frequency and split into four categories: "heavy," "never," "occasional," and "regular," as shown here:

```
R> is.factor(survey$Smoke)
[1] TRUE
R> table(survey$Smoke)

Heavy Never Occas Regul
   11   189    19    17
R> levels(survey$Smoke)
[1] "Heavy" "Never" "Occas" "Regul"
```

Here, the result from `is.factor(survey$Smoke)` indicates that you do indeed have a factor at hand, the call to `table` yields the number of students in each of the four categories, and `levels` reports the level names as character strings (see Section 4.3.1).

Let's investigate whether there's statistical evidence to support a difference in mean student height according to smoking frequency. You can create a set of box plots of these data with the following two lines:

```
R> boxplot(Height~Smoke,data=survey)
R> points(1:4,tapply(survey$Height,survey$Smoke,mean,na.rm=TRUE),pch=4)
```

Figure 20-5 shows the result.

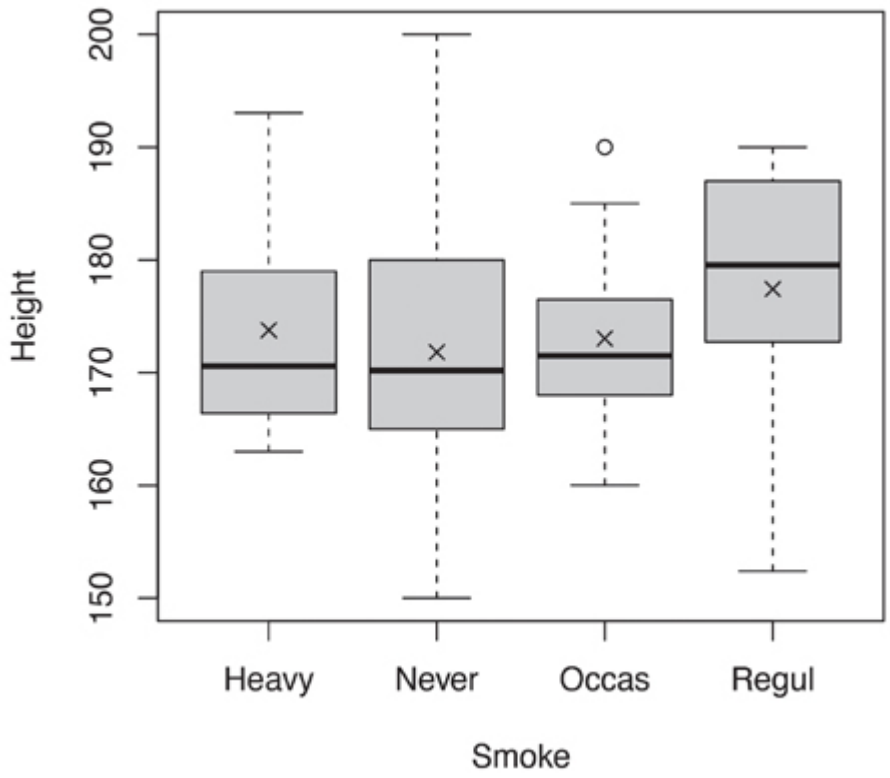


Figure 20-5: Box plots of the observed student heights split by smoking frequency; respective sample means marked with ×

Note from earlier R output that unless explicitly defined at creation, the levels of a factor appear in alphabetical order by default, as is the case for `Smoke`. By default, R will automatically set the first one (as shown in the output of a call to `levels`) to be the reference level when that factor is used as a predictor in subsequent model fitting. Fitting the linear model in mind using `lm`, you can see from a subsequent call to `summary` that the first level of `Smoke`, the category of heavy smokers, has indeed been used as the reference level:

```
R> survfit3 <- lm(Height~Smoke,data=survey)
R> summary(survfit3)

Call:
lm(formula = Height ~ Smoke, data = survey)

Residuals:
   Min     1Q Median     3Q    Max
```

```
-25.02  -6.82  -1.64   8.18  28.18

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 173.7720     3.1028  56.005   <2e-16 ***
SmokeNever   -1.9520     3.1933  -0.611    0.542
SmokeOccas   -0.7433     3.9553  -0.188    0.851
SmokeRegul    3.6451     4.0625   0.897    0.371
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 9.812 on 205 degrees of freedom
  (28 observations deleted due to missingness)
Multiple R-squared:  0.02153, Adjusted R-squared:  0.007214
F-statistic: 1.504 on 3 and 205 DF,  p-value: 0.2147
```

As outlined in Equation (20.8), you get estimates of coefficients corresponding to the dummy binary variables for three of the four possible categories in this example: the three non-reference levels. The observation in the reference category `Heavy` is represented solely by $\widehat{\beta}_0$, designated first as the overall `(Intercept)`, with the other coefficients providing the effects associated with an observation in one of the other categories.

Predictions from a Multilevel Categorical Variable

You find point predictions and any associated confidence or prediction intervals through `predict` command in the same way as in earlier examples:

```
R> one.of.each <- factor(c("Heavy","Never","Occas","Regul"))
R> one.of.each
[1] Heavy Never Occas Regul
Levels: Heavy Never Occas Regul
R> predict(survfit3,newdata=data.frame(Smoke=one.of.each),
           interval="confidence",level=0.95)
       fit      lwr      upr
1 173.7720 167.6545 179.8895
2 171.8200 170.3319 173.3081
```

```
3 173.0287 168.1924 177.8651
4 177.4171 172.2469 182.5874
```

Here, I've created the object `one.of.each` for illustrative purposes; it represents one observation in each of the four categories, stored as an object matching the class (and levels) of the original `Smoke` data exactly. A student in the `Occas` category, for example, is predicted to have a mean height of 173.772 – 0.7433 = 173.0287.

The output from the model summary earlier, however, shows that none of the binary dummy variable coefficients are considered statistically significant from zero (because all the *p*-values are too large). The results indicate, as you might have suspected, that there's no evidence that smoking frequency (or more specifically, having a smoking frequency that's different from the reference level) affects mean student heights based on this sample of individuals. As is common, the baseline coefficient $\widehat{\beta}_0$ is highly statistically significant, but that suggests only that the overall intercept probably isn't zero. (Because your response variable is a measurement of height and will clearly not be centered anywhere near 0 cm, that result makes sense.) The confidence intervals supplied are calculated in the usual *t*-based fashion.

The small `R-Squared` value reinforces this conclusion, indicating that barely any of the variation in the response (only a little over 2 percent) can be explained by accounting for the smoking frequency categories. Furthermore, the overall *F*-test *p*-value is rather large at around 0.215, suggesting an overall nonsignificant effect of the predictor on the response; this test was first mentioned in Section 20.3.4 (you'll look at it in more detail in Section 20.5.5 and later on in Section 21.3.5).

It's important that you interpret such results—indeed any results based on a *k*-level categorical variable in regression—in a *collective* fashion. You can claim only that there is *no* discernible effect of smoking on height because *all* the *p*-values for the binary dummy coefficients are non-significant. If one of the levels was in fact highly significant (through a small *p*-value), it would imply that the smoking factor as defined here, as a whole, *does* have a statistically detectable effect on the

response (even if the other two levels were still associated with very high *p*-values). I discuss this further in several more examples in Chapter 21.

As a brief aside, in Section 4.3.2 it was noted that a stricter, more formal style of ordering can be imposed when a factor is created via `factor` with the optional `ordered` argument set to `TRUE`. The default treatment of a factor variable in regression models fitted by commands such as `lm` is different depending on whether the factor has been formally stored as unordered or ordered. The former version (the more common form illustrated in this book) is represented with dummy coding, which leads to something known as *orthogonal contrasts*, a property of how the regression coefficients relate to one another. A formally ordered factor object, on the other hand, will by default describe the effect of the categorical variable on the response in a more complicated functional form, yielding *polynomial contrasts*. That discussion is beyond the scope of this text, but it suffices to say that this approach can be beneficial when your interest lies in the specific functional nature of "moving up" through an ordered set of categories. For more on the technical details, see Kuhn & Johnson (2013). In this book, we'll work exclusively with factor objects that are not formally declared as ordered.

20.5.3 Changing the Reference Level

Sometimes you might decide to change the default reference level, compared to which the effects of taking on any of the other levels are estimated. Changing the baseline will result in the estimation of different coefficients, meaning that individual *p*-values are subject to change, but the overall result (in terms of global significance of the factor) will not be affected. Because of this, altering the reference level is done only for interpretative purposes; sometimes there's an intuitively natural baseline of the predictor (for example, "Placebo" versus "Drug A" and "Drug B" as a treatment variable in the analysis of some clinical trial data) from which you want to estimate deviation in the mean response with respect to the other possible categories.

To quickly redefine the reference level, use the built-in `relevel` function in R (refer back to Section 4.3.2). Recall this function allows you to choose which level "comes first" in the definition of a given factor object and will therefore be designated as the reference level in subsequent model fitting. In the current example, let's say you'd rather have the nonsmokers as the reference level:

```
R> SmokeReordered <- relevel(survey$Smoke,ref="Never")
R> levels(SmokeReordered)
[1] "Never" "Heavy" "Occas" "Regul"
```

With this, the `relevel` function has moved the `Never` category into the first position in the new factor. If you fit the model again using the `SmokeReordered` vector instead of the original `Smoke` column of `survey`, it'll provide estimates of coefficients associated with the three different levels of smokers.

20.5.4 Treating Categorical Variables as Numeric

The way in which `lm` decides to define the parameters of the fitted model depends primarily on the kind of data you pass to the function. As discussed in Section 20.5.2, `lm` imposes the defined dummy coding only if the explanatory variable is a (formally unordered) factor vector.

Sometimes the categorical data you want to analyze haven't been stored as a factor in your data object. If the categorical variable is a character vector, `lm` will implicitly coerce it into a factor. If, however, the intended categorical variable is numeric, then `lm` performs linear regression exactly as if it were a continuous numeric predictor; it estimates a single regression coefficient, which is interpreted as a "per-one-unit-change" in the mean response.

This may seem inappropriate if the original explanatory variable is supposed to be made up of distinct groups. In some settings, however, especially when the variable can be naturally treated as numeric-discrete, this treatment not only is valid statistically but also helps with interpretation.

Let's take a break from the `survey` data and go back to the ready-to-use `mtcars` dataset (first seen in Section 9.3.1). Say you're interested in the variables mileage, `mpg` (continuous), and number of cylinders, `cyl` (discrete; the dataset contains cars with either four, six, or eight cylinders). Now it's perfectly sensible to automatically think of `cyl` as a categorical variable. Taking `mpg` to be the response variable, box plots are-well suited to reflect the grouped nature of `cyl` as a predictor; the result of the following line is given on the left of Figure 20-6:

```
R> boxplot(mtcars$mpg~mtcars$cyl,xlab="Cylinders",ylab="MPG")
```

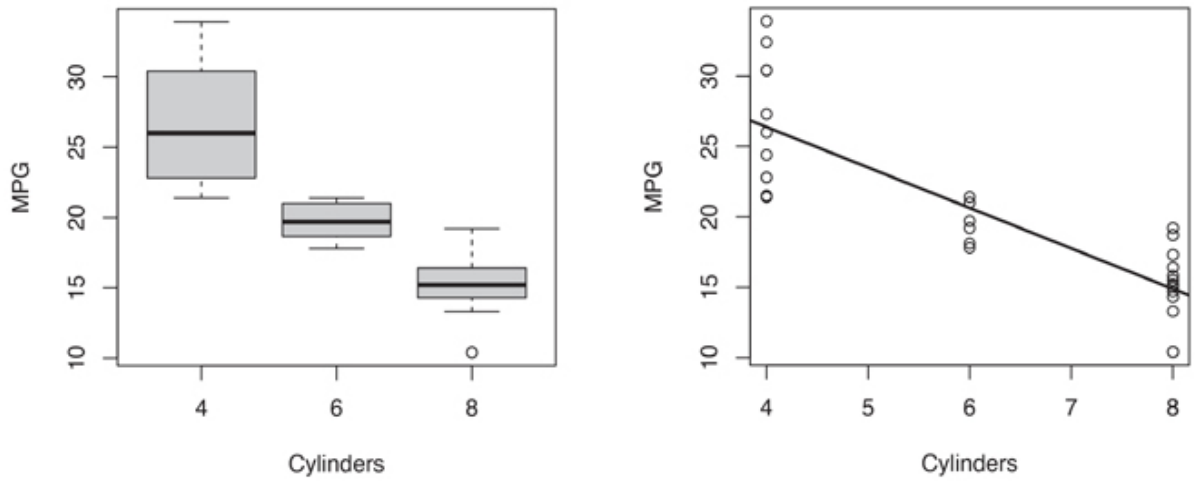


Figure 20-6: Box plots of mileage split by cylinders for the `mtcars` dataset (left), and a scatterplot of the same data with fitted regression line (treating `cyl` as numeric-continuous) superimposed (right)

When fitting the associated regression model, you must be aware of what you're instructing R to do. Since the `cyl` column of `mtcars` is numeric, and not a factor vector per se, `lm` will treat it as continuous if you just directly access the data frame:

```
R> class(mtcars$cyl)
[1] "numeric"
R> carfit <- lm(mpg~cyl,data=mtcars)
R> summary(carfit)

Call:
```

```
lm(formula = mpg ~ cyl, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.9814 -2.1185  0.2217  1.0717  7.5186

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  37.8846     2.0738   18.27  < 2e-16 ***
cyl          -2.8758     0.3224   -8.92 6.11e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 3.206 on 30 degrees of freedom
Multiple R-squared:  0.7262, Adjusted R-squared:  0.7171
F-statistic: 79.56 on 1 and 30 DF,  p-value: 6.113e-10
```

As in earlier examples that use a continuous predictor, here you've received an intercept and a slope estimate; the latter is highly statistically significant, indicating that there is evidence against the true value of the slope being zero. Your fitted regression line is

$$\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x = 37.88 - 2.88x$$

where $\hat{y}$ is the average mileage and x is numeric (the number of cylinders). For each single additional cylinder, the model says your mileage will decrease by 2.88 MPG, on average.

It's important to recognize the fact that you've fitted a continuous line to what is effectively categorical data. The right panel of Figure 20-6, created with the following lines, highlights this fact:

```
R> plot(mtcars$mpg~mtcars$cyl,xlab="Cylinders",ylab="MPG")
R> abline(carfit,lwd=2)
```

You might want to purposefully fit certain categorical or discrete predictors as continuous variables, if it makes contextual sense to do so, for a variety of reasons. First, it allows interpolation; for example, you could use this model to evaluate the average MPG for a fivecylinder car.

Second, it means there are fewer parameters that require estimation; in addition to the intercept term, instead of $k - 1$ coefficients for a categorical variable with k groups, you need only one parameter for a slope in the case of a continuous variable. Finally, it can be a convenient way to control for so-called "nuisance" variables; this will become clearer in Chapter 21. The downside is that you no longer get group-specific information. It can be misleading to proceed in this way if any differences in the mean response according to the predictor category of an observation are not well represented linearly, in which case detection of significant effects can be lost altogether.

At the very least, be aware of this distinction when fitting models. If you had only just now recognized that R had fitted the `cyl` variable as continuous and wanted to actually fit the model with `cyl` as categorical, you'd have to explicitly convert it into a factor vector beforehand or in the actual call to `lm`:

```
R> carfit <- lm(mpg~factor(cyl),data=mtcars)
R> summary(carfit)

Call:
lm(formula = mpg ~ factor(cyl), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-5.2636 -1.8357  0.0286  1.3893  7.2364
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   26.6636     0.9718  27.437  < 2e-16 ***
factor(cyl)6  -6.9208     1.5583  -4.441 0.000119 ***
factor(cyl)8 -11.5636     1.2986  -8.905 8.57e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.223 on 29 degrees of freedom
Multiple R-squared:  0.7325, Adjusted R-squared:  0.714
F-statistic:  39.7 on 2 and 29 DF,  p-value: 4.979e-09
```

Here, by wrapping `cyl` in a call to `factor` when specifying the formula for `lm`, you've obtained parameter estimates for the levels of `cyl` corresponding to six- and eight-cylinder cars (with the reference level automatically set to four-cylinder cars).

20.5.5 Checking for Equivalence with One-Way ANOVA

There's one final observation to make about regression models with a single nominal categorical predictor. Think about the fact that these models describe a mean response value for the k different groups. Does this remind you of anything? In this particular setting, you're actually doing the same thing as in one-way ANOVA (Section 19.1): comparing more than two means and determining whether there is statistical evidence that at least one mean is different from the others. You need to be able to make the same key assumptions of independence and normality for both techniques.

In fact, simple linear regression with a single categorical predictor, implemented using least-squares estimation, is just another way to perform one-way ANOVA. Or, perhaps more concisely, you can think of ANOVA as a special case of least-squares regression. The outcome of a one-way ANOVA test is a single p-value quantifying a level of statistical evidence against the null hypothesis that states that group means are equal. When you have one categorical predictor in a regression, it's exactly that p-value that's reported at the end of the `summary` of an `lm` object, something I've referred to a couple of times now as the "overall" or "global" significance test (for example, in Section 20.3.4).

In the final result of that global significance test for the student height modeled by smoking status example (Section 20.5.2), you had a p-value of 0.2147. This came from an F test statistic of 1.504 with $df_1 = 3$ and $df_2 = 205$. Now suppose you were just handed the data on these two variables and asked to perform a one-way ANOVA of height on smoking. Using the `aov` function as introduced in Section 19.1, you'd call something like this:

```
R> summary(aov(Height~Smoke,data=survey))
             Df Sum Sq Mean Sq F value Pr(>F)
Smoke         3    434  144.78   1.504  0.215
Residuals   205  19736   96.27
28 observations deleted due to missingness
```

Those same values seen in the linear model output are returned here.

You can also look at the ANOVA table and find the square root of the MSE:

```
R> sqrt(96.27)
[1] 9.811728
```

This is in fact the same "residual standard error" given in the `lm` summary in Section 20.5.2. The two conclusions you'd draw about the impact of smoking status on height (one for the `lm` output, the other for the ANOVA test) are equivalent.

The global test that `lm` provides isn't just there for the benefit of confirming ANOVA results. As a generalization of ANOVA, least-squares regression models provide more than just coefficient-specific tests. That global test is formally referred to as the *omnibus* F-*test*, and while it is indeed equivalent to one-way ANOVA in the "single categorical predictor" setting, it's also a useful overall, stand-alone test of the statistical contribution of several predictors to the outcome value. You'll explore this further in Section 21.3.5 after you've begun modeling your response variable using multiple explanatory variables.

Exercise 20.2

Continue using the `survey` data frame from the package `MASS` for these tasks.

a. The `survey` dataset has a variable named `Exer`, a factor with $k = 3$ levels characterizing the frequency that a student

chooses to spend on physical exercise: “none,” “some,” or “frequent.” Obtain a count of the number of students in each category and produce side-by-side box plots of student height split by exercise.

b. Assuming independence of the observations and normality as usual, fit a linear regression model with height as the response variable and exercise as the explanatory variable (dummy coding). What’s the default reference level of the predictor? Produce a model summary.

c. Draw a conclusion based on the fitted model from (b). Does it appear that exercise frequency has any impact on mean height? What is the nature of the estimated effect?

d. Predict the mean heights of one individual in each of the three exercise categories, accompanied by 95 percent prediction intervals.

e. Do you arrive at the same result and interpretation for the height-by-exercise model if you construct an ANOVA table using `aov`?

f. Is there any change to the outcome of (e) if you alter the model so that the reference level of the exercise variable is “none”? Would you expect there to be?

Now turn back to the ready-to-use `mtcars` dataset. One of the variables in this data frame is `qsec`, described as the time in seconds it takes to race a quarter mile; another is `gear`, the number of forward gears (cars in this dataset have either three, four, or five gears).

g. Using the vectors straight from the data frame, fit a simple linear regression model with `qsec` as the response variable and `gear` as the explanatory variable and interpret the model summary.

h. Explicitly convert `gear` to a factor vector and refit the model. Compare the model summary with that from (g). What do you find?

i. Explain, with the aid of a relevant plot in the same style as the right image of Figure 20-6, why you think there is a difference between the two models (g) and (h).

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`lm`	Fit linear model	Section 20.2.3, p. 525
`coef`	Get estimated coefficients	Section 20.2.4, p. 527
`summary`	Summarize linear model	Section 20.3.1, p. 528
`confint`	Get CIs for estimated coefficients	Section 20.3.2, p. 530
`predict`	Predict from linear model	Section 20.4.2, p. 533

21
MULTIPLE LINEAR REGRESSION



Multiple linear regression is a straightforward generalization of the singlepredictor models discussed in the previous chapter. It allows you to model your continuous response variable in terms of more than one predictor so you can measure the joint effect of several
explanatory variables on the response variable. In this chapter, you'll see how to model your response variable in this way, and you'll use R to fit the model using the method of least squares. You'll also explore other key statistical aspects of linear modeling in the R environment, such as transforming variables and including interactive effects.

Multiple linear regression represents an important part of the practice of statistics. It lets you control or adjust for multiple sources of influence on the value of the response, rather than just measuring the effect of one explanatory variable (in most situations, there is more than one contributor to the outcome measurements). At the heart of this class of methods is the intention to uncover potentially causal relationships between your response variable and the (joint) effect of any explanatory variables. In reality, causality itself is extremely difficult to establish, but you can strengthen any evidence of causality by using a well-designed study supported by sound data collection and by fitting models that might realistically gauge the relationships present in your data.

21.1 Terminology

Before you look at the theory behind multiple regression models, it's important to have a clear understanding of some terminology associated with the relevant variables:

- A *lurking variable* influences the response, another predictor, or both, but it goes unmeasured (or is not included) in a predictive model. For example, say a researcher establishes a positive link between the volume of trash thrown out by a household and whether the household owns a trampoline. The potential lurking variable here would be the number of children in the household; this variable is more likely to be positively associated with an increase in trash *and* chances of owning a trampoline. An interpretation that suggests owning a trampoline is itself a cause of increased waste would be erroneous.
- The presence of a lurking variable can lead to spurious conclusions about causal relationships between the response and the other predictors, or it can mask a true cause-and-effect association; this kind of error is referred to as *confounding*. You can think of confounding as the entanglement of the effects of one or more predictors on the response.
- A *nuisance* or *extraneous variable* is a predictor of secondary or no interest that has the potential to confound relationships between other variables, thereby adversely affecting the reliability of your estimates of the other regression coefficients. Nuisance variables are included in the modeling as a matter of necessity, but the specific nature of their influence on the response is not the primary interest of the analysis.

These definitions will become clearer once you begin fitting and interpreting the regression models in Section 21.3. The main message I want to emphasize here is that correlation does not imply causation. If a fitted model finds a statistically significant association between a predictor (or predictors) and a response, it's important to consider the possibility that lurking variables are contributing to the results and attempt to control any confounding before you draw conclusions. Multiple regression models allow you to do this.

21.2 Theory

Before you start using R to fit regression models, you'll examine the technical definitions of a linear regression model with multiple predictors. Here, you'll look at how the models work in a mathematical sense and get a glimpse of the

calculations that happen “behind the scenes” when estimating the model parameters in R.

21.2.1 Extending the Simple Model to a Multiple Model

Rather than having just one predictor, you want to determine the value of a continuous response variable Y given the values of $p > 1$ independent explanatory variables $X_1, X_2, \ldots, X_p$. The overarching model is defined as the following:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_2 X_p + \in \quad (21.1)$$

Here, $\beta_0, \ldots, \beta_p$ are the regression coefficients and, as before, you assume independent, normally distributed residuals $\in \sim N(0, \sigma)$ around the mean.

In practice, you have n data records; each record provides values for each of the predictors $X_j; j = \{1, \ldots, p\}$. The model to be fitted is given in terms of the mean response, conditional upon a particular realization of the set of explanatory variables:

$$\hat{y} = \mathbb{E}[Y|X_1 = x_1,\ X_2 = x_2,\ \ldots,\ X_p = x_p] = \widehat{\beta}_0 + \widehat{\beta}_1 x_1 + \widehat{\beta}_2 x_2 + \ldots + \widehat{\beta}_p x_p$$

Note that the $\widehat{\beta}_j$s represent estimates of the regression coefficients.

In simple linear regression, where you have only one predictor variable, recall that the goal is to find the "line of best fit." The idea of least-squares estimation for linear models with multiple independent predictors follows much the same motivation. Now, however, in an abstract sense you can think of the relationship between response and predictors as a multidimensional plane or surface. You want to find the "linear surface" that best fits your multivariate data in terms of minimizing the overall squared distance between itself and the raw response data.

More formally, for your n data records, the $\widehat{\beta}_j$s are found as the values that jointly minimize the sum of squared errors, also called the *residual sum of squares (RSS)*, given by the following expression:

$$\mathrm{RSS}(\beta_0, \ldots, \beta_p) = \sum_{i=1}^{n} \{ y_i - (\beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i} + \ldots + \beta_p x_{p,i}) \}^2 \quad (21.2)$$

Here, $x_{j,i}$ is the observed value of individual i for explanatory variable X_j and y_i is the response value of individual i.

21.2.2 Representing in Matrix Notation

Matrix notation makes the calculations involved in minimizing the RSS of Equation (21.2) much easier to read. When dealing with n multivariate observations, you can write Equation (21.1) as follows:

$$\boldsymbol{Y = X \cdot \beta + \epsilon}$$

Here, $\mathbf{Y}$ and ϵ denote $n \times 1$ column matrices such that:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \text{and} \quad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Here, y_i and ϵ_i refer to the response observation and random error term for the ith individual. The quantity β is a $(p + 1) \times 1$ column matrix of the regression coefficients, and then the observed predictor data for all individuals and explanatory variables are stored in an $n \times (p + 1)$ matrix $\boldsymbol{X}$, called the *design matrix*:

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix} \quad \text{and} \quad \boldsymbol{X} = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{p,1} \\ 1 & \mathrm{x}_{1,2} & \dots & x_{p,2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{x}_{1,n} & \dots & x_{p,n} \end{bmatrix}$$

The value of β that minimizes Equation (21.2), providing the estimated regression coefficient values, is found with the following calculation:

$$\widehat{\beta} = \begin{bmatrix} \widehat{\beta}_0 \\ \widehat{\beta}_1 \\ \vdots \\ \widehat{\beta}_p \end{bmatrix} = \left(X^{\mathrm{T}} \cdot X\right)^{-1} \cdot X^{\mathrm{T}} \cdot Y \quad (21.3)$$

Note the following:

- The $\cdot$ symbol represents matrix multiplication, the superscript$^{\mathrm{T}}$ represents the transpose, and $^{-1}$ represents the matrix inverse (as per Section 3.1.3).

- Extending the size of β and $\boldsymbol{X}$ (note the leading column of 1s in $\boldsymbol{X}$) to create structures of size $p + 1$ (as opposed to just the number of predictors p) allows for the estimation of the overall intercept β_0.
- As well as Equation (21.3), the design matrix plays a key role in the estimation of other quantities, such as the standard errors of the coefficients.

21.2.3 Reviewing a Basic Example

You can "manually" estimate the β_j ($j = 0, 1, \ldots, p$) in R using the functions covered in Chapter 3: `%*%` (matrix multiplication), `t` (matrix transposition), and `solve` (matrix inversion). As a quick demonstration, let's say you have two predictor variables: X_1 as continuous and X_2 as binary. Your target regression equation is therefore $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_{1x1} + \widehat{\beta}_{2x2}$. Suppose you collect the following data, where the response data, data for X_1, and data for X_2, for $n = 8$ individuals, are given in the columns `y`, `x1`, and `x2`, respectively:

```
R> demo.data <- data.frame(y=c(1.55,0.42,1.29,0.73,0.76,-1.09,1.41,-0.32),
                           x1=c(1.13,-0.73,0.12,0.52,-0.54,-1.15,0.20,-1.09),
                           x2=c(1,0,1,1,0,1,0,1))
R> demo.data
      y    x1 x2
1  1.55  1.13  1
2  0.42 -0.73  0
3  1.29  0.12  1
4  0.73  0.52  1
5  0.76 -0.54  0
6 -1.09 -1.15  1
7  1.41  0.20  0
8 -0.32 -1.09  1
```

To get your three point estimates for $\beta = [\beta_0, \beta_1, \beta_2]^\top$ for the linear model, you first have to construct $\boldsymbol{X}$ and $\boldsymbol{Y}$ as required by Equation (21.3):

```
R> Y <- matrix(demo.data$y)
R> Y
      [,1]
[1,]  1.55
[2,]  0.42
[3,]  1.29
[4,]  0.73
[5,]  0.76
[6,] -1.09
```

```
[7,]  1.41
[8,] -0.32
R> n <- nrow(demo.data)
R> X <- matrix(c(rep(1,n),demo.data$x1,demo.data$x2),nrow=n,ncol=3)
R> X
     [,1]  [,2] [,3]
[1,]    1  1.13    1
[2,]    1 -0.73    0
[3,]    1  0.12    1
[4,]    1  0.52    1
[5,]    1 -0.54    0
[6,]    1 -1.15    1
[7,]    1  0.20    0
[8,]    1 -1.09    1
```

Then, execute the line corresponding to Equation (21.3):

```
R> BETA.HAT <- solve(t(X)%*%X)%*%t(X)%*%Y
R> BETA.HAT
           [,1]
[1,]  1.2254572
[2,]  1.0153004
[3,] -0.6980189
```

You've just used least-squares to fit your model based on the observed data in `demo.data`, which results in the estimates $\widehat{\beta}_0 = 1.225$, $\widehat{\beta}_1 = 1.015$, and $\widehat{\beta}_2 = -0.698$.

21.3 R Implementation and Interpretation

Ever helpful, R automatically builds the matrices and carries out all the necessary calculations when you instruct it to fit a multiple linear regression model. As with simple regression models, you use `lm`; simply specify any additional predictors in the formula argument. I'll focus only on predictor *main effects* for the moment; you'll explore more complex relationships later in the chapter.

When it comes to output and interpretation, working with multiple explanatory variables follows the same rules as you've seen in Chapter 20. Any numeric-continuous variables (or a categorical variable being treated as such) have a slope coefficient that provides a "per-unit-change" quantity. Any k-level categorical variables (formally unordered factors; see the discussion at the end of Section 20.5.2) are dummy coded and are associated with $k - 1$ additional parameters. The one important thing you must remember when examining the

results for any single predictor variable in a multiple regression scenario is that its coefficients are deemed to represent the "partial" impact it has on the (mean) response after the other present predictors have been accounted for. This will be emphasized in Section 21.3.2.

21.3.1 Adding Predictors

Let's first confirm the manual matrix calculations from a moment ago. Using the `demo.data` object, fit the multiple linear model and examine the coefficients from that object as follows:

```
R> demo.fit <- lm(y~x1+x2,data=demo.data)
R> coef(demo.fit)
(Intercept)          x1          x2
  1.2254572   1.0153004  -0.6980189
```

Indeed, you're obtaining exactly the same point estimates stored earlier in `BETA.HAT`.

With the response variable on the left as usual, you specify the multiple predictors on the right side of the `~` symbol; altogether this represents the formula argument. To fit a model with several main effects, use `+` to separate any variables you want to include. You've already seen this notation in Section 19.2.2, when investigating two-way ANOVA.

To study the interpretation of the parameter estimates of a multiple linear regression model, let's return to the `survey` dataset in the `MASS` package. In Chapter 20, you explored several simple linear regression models based on a response variable of student height, as well as stand-alone predictors of handspan (continuous) and sex (categorical, $k = 2$). You found that handspan was highly statistically significant, with the estimated coefficient suggesting an average increase of about 3.12 cm for each 1 cm increase in handspan. When you looked at the same t-test using sex as the explanatory variable, the model also suggested evidence against the null hypothesis, with "being male" adding around 13.14 cm to the mean height when compared to the mean for females (the category used as the reference level).

What those models can't tell you is the *joint effect* of sex and handspan on predicting height. If you include both predictors in a multiple linear model, you can (to some extent) reduce any confounding that might otherwise occur in the isolated fits of the effect of either single predictor on height.

Here is the output from the multiple linear regression model fitted to the height data with both handspan and sex simultaneously as the two predictor variables:

```
R> survmult <- lm(Height~Wr.Hnd+Sex,data=survey)
R> summary(survmult)
Call:
lm(formula = Height ~ Wr.Hnd + Sex, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-17.7479  -4.1830   0.7749   4.6665  21.9253

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 137.6870     5.7131  24.100  < 2e-16 ***
Wr.Hnd        1.5944     0.3229   4.937 1.64e-06 ***
SexMale       9.4898     1.2287   7.724 5.00e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.987 on 204 degrees of freedom
  (30 observations deleted due to missingness)
Multiple R-squared:  0.5062, Adjusted R-squared:  0.5014
F-statistic: 104.6 on 2 and 204 DF,  p-value: < 2.2e-16
```

The coefficient for handspan is about 1.59, almost half of its corresponding value (3.12 cm) when it is used as the only predictor in a simple linear regression model for height. Despite this, it's still highly statistically significant in the presence of sex. The coefficient for sex has also reduced in magnitude when compared with its simple linear model and is also still significant in the presence of handspan. You'll interpret these new figures in a moment.

As for the rest of the output, `Residual standard error` still provides you with an estimate of the standard error of the random noise term ϵ, and you're also provided with an `R-squared` value. When associated with more than one predictor, the latter is formally referred to as the coefficient of *multiple* determination. The calculation of this coefficient, as in the single predictor setting, comes from the correlations between the variables in the model. I'll leave the theoretical intricacies to more advanced texts; note that `R-squared` still represents the proportion of variability in the response that's explained by the regression (in this example, it sits at around 0.5).

You can continue to add explanatory variables in the same way if you need to do so. In Section 20.5.2, you examined smoking frequency as a stand-alone categorical predictor for height and found that this explanatory variable provided no statistical evidence of an impact on the mean response. But could the smoking

variable contribute in a statistically significant way if you control for handspan and sex? Let's take a look:

```
R> survmult2 <- lm(Height~Wr.Hnd+Sex+Smoke,data=survey)
R> summary(survmult2)

Call:
lm(formula = Height ~ Wr.Hnd + Sex + Smoke, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-17.4869  -4.7617   0.7604   4.3691  22.1237

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 137.4056     6.5444  20.996  < 2e-16 ***
Wr.Hnd        1.6042     0.3301   4.860 2.36e-06 ***
SexMale       9.3979     1.2452   7.547 1.51e-12 ***
SmokeNever   -0.0442     2.3135  -0.019    0.985
SmokeOccas    1.5267     2.8694   0.532    0.595
SmokeRegul    0.9211     2.9290   0.314    0.753
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.023 on 201 degrees of freedom
  (30 observations deleted due to missingness)
Multiple R-squared:  0.5085, Adjusted R-squared:  0.4962
F-statistic: 41.59 on 5 and 201 DF,  p-value: < 2.2e-16
```

Since it's a categorical variable with $k > 2$ levels, `Smoke` is dummy coded (with heavy smokers as the default reference level), giving you three extra intercepts for the three non-reference levels of the variable; the fourth is incorporated into the overall intercept.

In the `summary` of the latest fit, you can see that while handspan and sex continue to yield very small p-values, smoking frequency suggests no such evidence against the hypotheses of zero coefficients. The smoking variable has had little effect on the values of the other coefficients compared with the previous model in `survmult`, and the `R-squared` coefficient of multiple determination has barely increased.

One question you might now ask is, if smoking frequency doesn't benefit your ability to predict mean height in any substantial way, should you remove that variable from the model altogether? This is the primary goal of *model selection*: to find the "best" model for predicting the outcome, without fitting one that is

unnecessarily complex (by including more explanatory variables than is required). You'll look at some common ways researchers attempt to achieve this in Section 22.2.

21.3.2 Interpreting Partial Effects

In multiple regression, the estimation of each predictor takes into account the effect of all other predictors present in the model. A coefficient for a specific predictor Z should therefore be considered "partial" in the sense that we interpret it as the change in the mean response for a one-unit increase in Z, *while holding all other predictors constant*.

As you've determined that smoking frequency still appears to have no discernible impact on mean height when taking sex and handspan into consideration, return your focus to `survmult`, the model fitted in Section 21.3.1 that includes only sex and handspan as predictors. From that output, note the following:

- For students of the same sex (that is, focusing on either males or females), a 1 cm increase in handspan leads to an estimated increase of 1.5944 cm in mean height.
- For students of the same handspan, males on average will be 9.4898 cm taller than females.
- The difference in the values of the two estimated predictor coefficients when compared with their respective simple linear model fits examined in the previous chapter, plus the fact that both continue to indicate evidence against the null hypothesis of "being zero" in the multivariate fit, suggests that confounding (in terms of the effect of both handspan and sex on the response variable of height) is present in the single-predictor models.

The final point highlights the general usefulness of multiple regression. It shows that, in this example, if you use only single predictor models, the determination of the "true" impact that each explanatory variable has in predicting the mean response is misleading since some of the change in height is determined by sex, but some is also attributed to handspan. It's worth noting that the coefficient of determination (refer to Section 20.3.3) for the `survmult` model is noticeably higher than the same quantity in either of the single-variate models, so you're actually accounting for more variation in the response by using multiple regression.

Think of the fitted model as

$$\text{“Mean height”} = 137.687 + 1.594 \times \text{“handspan”} + 9.49 \times \text{“sex”} \quad (21.4)$$

where “handspan” is the writing handspan supplied in centimeters and “sex” is supplied as either 1 (if male) or 0 (if female).

NOTE

The baseline (overall) intercept of around 137.687 cm represents the mean height of a female with a handspan of 0 cm; again, this is clearly not directly interpretable in the context of the application. For this kind of situation, some researchers center the offending continuous predictor (or predictors) on 0 by subtracting the sample mean of all the observations on that predictor from each observation prior to fitting the model. They then use the centered predictor data in place of the original (untranslated) data. The resulting fitted model allows you to use the mean value of the untranslated predictor (in this case, handspan) rather than a zero value in order to directly interpret the intercept estimate $\widehat{\beta}_0$.

21.3.3 Visualizing the Multiple Linear Model

As shown here, “being male” changes the overall intercept by around 9.49 cm:

```
R> survcoefs <- coef(survmult)
R> survcoefs
(Intercept)      Wr.Hnd     SexMale
 137.686951    1.594446    9.489814
R> as.numeric(survcoefs[1]+survcoefs[3])
[1] 147.1768
```

Because of this, you could also write Equation (21.4) as two equations. Here’s the equation for female students:

$$\text{“Mean height”} = 137.687 + 1.594 \times \text{“handspan”}$$

Here’s the equation for male students:

$$\begin{aligned} \text{“Mean height”} &= (137.687 + 9.4898) + 1.594 \times \text{“handspan”} \\ &= 147.177 + 1.594 \times \text{“handspan”} \end{aligned}$$

This is handy because it allows you to visualize the multivariate model in much the same way you might visualize the corresponding simple linear models. The following code produces Figure 21-1:

```
R> plot(survey$Height~survey$Wr.Hnd,
        col=c("gray","black")[as.numeric(survey$Sex)],
        pch=16,xlab="Writing handspan (cm)",ylab="Height (cm)")
R> abline(a=survcoefs[1],b=survcoefs[2],col="gray",lwd=2)
R> abline(a=survcoefs[1]+survcoefs[3],b=survcoefs[2],col="black",lwd=2)
R> legend("topleft",legend=levels(survey$Sex),col=c("gray","black"),pch=16)
```

First, a scatterplot of the height and handspan observations, split by sex, is drawn. Then, `abline` adds the line corresponding to females and adds a second one corresponding to males, based on those two equations.

Although this plot might look like two separate simple linear model fits, one for each level of sex, it's important to recognize that isn't the case. You're effectively looking at a representation of a multivariate model on a two-dimensional canvas, where the statistics that determine the fit of the two visible lines have been estimated "jointly"—in other words, when considering the combined effect of both predictors.

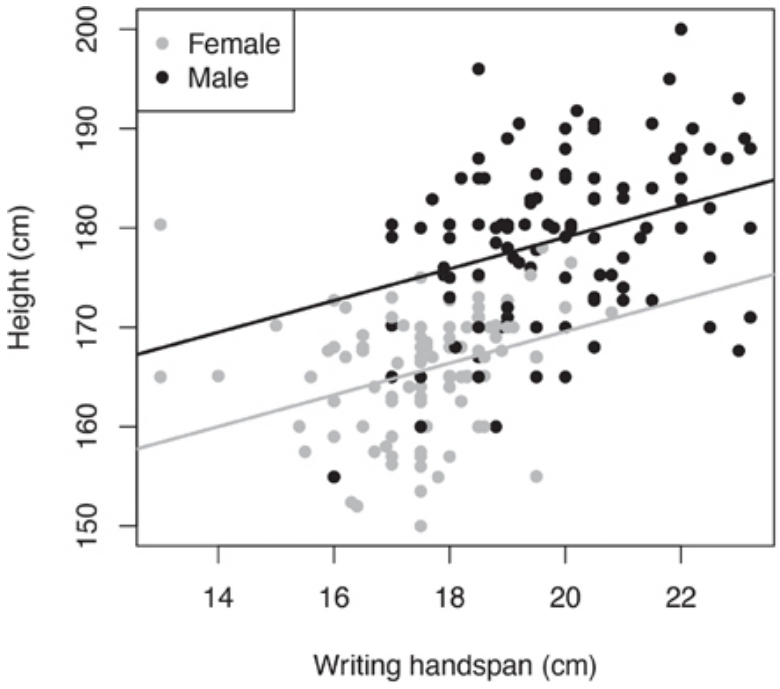


Figure 21-1: Visualizing the observed data and fitted multiple linear model of student height modeled by handspan and sex

21.3.4 Finding Confidence Intervals

As in Chapter 20, you can easily find confidence intervals for any of the regression parameters in multiple regression models with `confint`. Using `survmult2` from Section 21.3.1 (the object of the fitted model for student height including all three

predictors: sex, handspan, and smoking frequency), the output of a call to `confint` looks like this:

```
R> confint(survmult2)
                 2.5 %     97.5 %
(Intercept) 124.5010442 150.310074
Wr.Hnd        0.9534078   2.255053
SexMale       6.9426040  11.853129
SmokeNever   -4.6061148   4.517705
SmokeOccas   -4.1312384   7.184710
SmokeRegul   -4.8543683   6.696525
```

Note that the `Wr.Hnd` and `SexMale` variables were shown to be statistically significant at the 5 percent level in the earlier model summary and that the 95 percent confidence intervals for the corresponding regression parameters do not include the null value of zero. On the other hand, all the coefficients for the dummy variables associated with the smoking frequency predictor are nonsignificant, and their confidence intervals clearly include zero. This reflects the fact that the smoking variable isn't, as a whole, considered statistically significant in this particular model.

21.3.5 Computing the Omnibus F-Test

You can think of the omnibus *F*-test (first encountered in Section 20.5.2 in the context of multilevel predictors) more generally for multiple regression models as a test with the following hypotheses:

$$\begin{aligned} &\mathrm{H}_0 : \beta_1 = \beta_2 = \ldots = \beta_p = 0 \\ &\mathrm{H}_\mathrm{A}: \text{At least one of the } \beta_j \neq 0 \quad (\text{for } j = 1, \ldots, p) \end{aligned} \tag{21.5}$$

The test is effectively comparing the amount of error attributed to the "null" model (that which only has an intercept) with the amount of error attributed to the predictors when all the predictors are present. In other words, the more the predictors are able to model the response, the more error they explain, giving you a more extreme *F* statistic and therefore a smaller *p*-value. The single result makes the test especially useful when you have many explanatory variables. The test works the same regardless of the mix of predictors you have in a given model: One or more might be continuous, discrete, binary, or categorical with $k > 2$ levels. When multiple regression models are fitted, the amount of output alone can take time to digest and interpret, and care must be taken to avoid Type I errors (incorrect rejection of a true null hypothesis—refer to Section 18.5).

The *F*-test helps boil all that down, allowing you to conclude either of the following:

1. Evidence against H_0 if the associated *p*-value is smaller than your chosen significance level α, which suggests that your regression—your combination of the explanatory variables—does a significantly better job of predicting the response than if you removed *all* those predictors.
2. No evidence against H_0 if the associated *p*-value is larger than α, which suggests that using the predictors has no tangible benefit over having an intercept alone.

The downside is that the test doesn't tell you which of the predictors (or which subset thereof) is having a beneficial impact on the fit of the model, nor does it tell you anything about their coefficients or respective standard errors.

You can compute the *F*-test statistic using the coefficient of determination, R^2, from the fitted regression model. Let p be the number of regression parameters requiring estimation, excluding the intercept β_0. Then

$$\mathscr{F} = \frac{R^2(n-p-1)}{(1-R^2)p} \quad (21.6)$$

where n is the number of observations used in fitting the model (after records with missing values have been deleted). Then, under H_0 in Equation (21.5), $\mathscr{F}$ follows an *F* distribution (see Sections 16.2.6 and 19.1.2) with $df_1 = p$, $df_2 = n - p - 1$ degrees of freedom. Equation (21.6)'s associated *p*-value is yielded as the upper-tail area of that *F* distribution.

As a quick exercise to confirm this, turn your attention back to the fitted multiple regression model `survmult2` in Section 21.3.1, which is the model for student height by handspan, sex, and smoking status from `survey`. You can extract the coefficient of multiple determination from the `summary` report (using the technique noted in Section 20.3.4):

```
R> R2 <- summary(survmult2)$r.squared
R> R2
[1] 0.508469
```

This matches the multiple R-squared value seen in the `survmult2` output in Section 21.3.1. Now get n as the original size of the dataset in `survey` minus any missing values (reported as 30 in the earlier `summary` output):

```
R> n <- nrow(survey)-30
R> n
[1] 207
```

Then, get p as the number of estimated regression parameters (minus 1 for the intercept):

```
R> p <- length(coef(survmult2))-1
R> p
[1] 5
```

You can then confirm the value of $n - p - 1$, which matches the `summary` output (`201 degrees of freedom`):

```
R> n-p-1
[1] 201
```

Finally, find the test statistic $\mathscr{F}$ as dictated by Equation (21.6), and you can use the `pf` function as follows to obtain the corresponding p-value for the test:

```
R> Fstat <- (R2*(n-p-1))/((1-R2)*p)
R> Fstat
[1] 41.58529
R> 1-pf(Fstat,df1=p,df2=n-p-1)
[1] 0
```

The omnibus F-test for this example gives a p-value that's so small, it's effectively zero. These calculations match the relevant results reported in the output of `summary(survmult2)` completely.

Looking back at the student height multiple regression fit based on handspan, sex, and smoking in `survmult2` in Section 21.3.1, it's little surprise that with two of the predictors yielding small p-values, the omnibus F-test suggests strong evidence against H_0 based on Equation (21.5). This highlights the "umbrella" nature of the omnibus test: Although the smoking frequency variable itself doesn't appear to contribute anything statistically important, the F-test for that model still suggests `survmult2` should be preferred over a "no-predictor" model because both handspan and sex are important.

21.3.6 Predicting from a Multiple Linear Model

Prediction (or *forecasting*) for multiple regression follows the same rules as for simple regression. It's important to remember that point predictions found for a particular *covariate profile*, or the collection of predictor values for a given

individual, are associated with the mean (or *expected value*) of the response; that confidence intervals provide measures for mean responses; and that prediction intervals provide measures for raw observations. You also have to consider the issue of interpolation (predictions based on x values that fall within the range of the originally observed covariate data) versus extrapolation (prediction from x values that fall outside the range of said data). Other than that, the R syntax for `predict` is identical to that used in Section 20.4.

As an example, using the model fitted on student height as a linear function of handspan and sex (in `survmult`), you can estimate the mean height of a male student with a writing handspan of 16.5 cm and accompany the prediction with a confidence interval:

```
R> predict(survmult,newdata=data.frame(Wr.Hnd=16.5,Sex="Male"),
           interval="confidence",level=0.95)
       fit      lwr      upr
1 173.4851 170.9419 176.0283
```

The result indicates that you have an expected value of about 173.48 cm and that you can be 95 percent confident the true value lies somewhere between 170.94 and 176.03 (rounded to 2 d.p.). In the same way, the mean height of a female with a handspan of 13 cm is estimated at 158.42 cm, with a 99 percent prediction interval of 139.76 to 177.07:

```
R> predict(survmult,newdata=data.frame(Wr.Hnd=13,Sex="Female"),
           interval="prediction",level=0.99)
       fit      lwr      upr
1 158.4147 139.7611 177.0684
```

There are in fact two female students in the dataset with writing handspans of 13 cm, as you can see in Figure 21-1. Using your knowledge of subsetting data frames (see Section 5.2.4), you can inspect these two records and examine the three variables of interest:

```
R> survey[survey$Sex=="Female" & survey$Wr.Hnd==13,c("Sex","Wr.Hnd","Height")]
       Sex Wr.Hnd Height
45  Female     13 180.34
152 Female     13 165.00
```

The output shows the second female's height falls well inside the prediction interval, but the first female's height is notably higher than the upper limit. It's important to realize that, technically, nothing has gone wrong here in terms of the model fitting and interpretation; it's still possible that an observation can fall

outside a prediction interval, even a wide 99 percent interval, though it's perhaps improbable. There could be any number of reasons for this occurring. First, the model could be inadequate. For example, you might be excluding important predictors in the fitted model and therefore have less predictive power. Second, although the prediction is within the range of the observed data, it has occurred at one extreme end of the range, where it's less reliable because your data are relatively sparse. Third, the observation itself may be tainted in some way: Perhaps the individual recorded her handspan incorrectly, in which case her invalid observation should be removed prior to model fitting. A good statistician will appraise data and models with a critical eye; this is a skill that I'll emphasize further as this chapter unfolds.

Exercise 21.1

In the `MASS` package, you'll find the data frame `cats`, which provides data on sex, body weight (in kilograms), and heart weight (in grams) for 144 household cats (see Venables & Ripley 2002, for further details); you can read the documentation with a call to `?cats`. Load the `MASS` package with a call to `library("MASS")`, and access the object directly by entering `cats` at the console prompt.

a. Plot heart weight on the vertical axis and body weight on the horizontal axis, using different colors or point characters to distinguish between male and female cats. Use skills from Chapters 7 and 14 to annotate your plot with a legend and appropriate axis labels.

b. Fit a least-squares multiple linear regression model using heart weight as the response variable and the other two variables as predictors, and view a model summary.

    i. Write down the equation for the fitted model and interpret the estimated regression coefficients for body weight and sex. Are both statistically significant? What does this say about the relationship between the response and predictors?

    ii. Report and interpret the coefficient of determination and the outcome of the omnibus *F*-test.

c. Tilman's cat, Sigma, is a 3.3 kg female. Use your model to estimate her mean heart weight and provide a 95 percent prediction interval.

d. Use `predict` in the same way as in Section 21.3.3 to superimpose continuous lines based on the fitted linear model on your plot from (a), one for male cats and one for female. What do you notice? Does this reflect the statistical significance (or lack thereof) of the parameter estimates?

The `boot` package (Davison & Hinkley 1997, Canty & Ripley 2024) is another library of R code that's included with the standard installation but isn't automatically loaded. Load `boot` with a call to `library("boot")`. You'll find a data frame called `nuclear`, which contains data on the construction of nuclear power plants in the United States in the late 1960s (Cox & Snell 1981).

e. Enter `?nuclear` at the prompt to access the documentation and examine the details of the variables. Use `pairs` (Section 14.6.2) to produce a quick scatterplot matrix of the data.

f. One of the original objectives was to predict the cost of further construction of these power plants. Create a fit and summary of a linear regression model that aims to model `cost` by `t1` and `t2`, two variables that describe different elapsed times associated with the application for and issuance of various permits. Take note of the estimated regression coefficients and their significance in the fitted model.

g. Refit the model, but this time also include an effect for the date the construction permit was issued. Contrast the output for this new model against the previous one. What do you notice, and what does this information suggest about the relationships in the data with respect to these predictors?

h. Fit a third model for power plant cost, using the predictors for "date of permit issue," "power plant capacity," and the binary variable describing whether the plant was sited in the northeastern United States. Write down the fitted model equation and provide 95 percent confidence intervals for each estimated coefficient.

The following table gives an excerpt of a historical dataset compiled between 1961 and 1973. It concerns the annual murder rate in Detroit, Michigan; the data were originally presented and analyzed by Fisher (1976) and are reproduced here from Harraway (1995). In the dataset you'll find the number of murders, police officers, and gun licenses issued per 100,000

population, as well as the overall unemployment rate as a percentage of the overall population.

Murders	Police	Unemployment	Guns
8.60	260.35	11.0	178.15
8.90	269.80	7.0	156.41
8.52	272.04	5.2	198.02
8.89	272.96	4.3	222.10
13.07	272.51	3.5	301.92
14.57	261.34	3.2	391.22
21.36	268.89	4.1	665.56
28.03	295.99	3.9	1131.21
31.49	319.87	3.6	837.60
37.39	341.43	7.1	794.90
46.26	356.59	8.4	817.74
47.24	376.69	7.7	583.17
52.33	390.19	6.3	709.59

i. Create your own data frame object of these data, and produce a scatterplot matrix with `pairs`. Which of the variables appears to be most strongly related to the murder rate?

j. Fit a multiple linear regression model using the number of murders as the response and all other variables as predictors. Write down the model equation and interpret the coefficients. Is it reasonable to state that all relationships between the response and the predictors are causal?

k. Identify the amount of variation in the response attributed to the joint effect of the three explanatory variables. Then refit the model excluding the predictor associated with the largest (in other words, "most nonsignificant") p-value. Compare the new coefficient of determination with that of the previous model. Is there much difference?

l. Use your model from (k) to predict the mean number of murders per 100,000 residents, with 300 police officers per 100,000 and 500 issued gun licenses per 100,000. Compare this to the mean response if there were no gun licenses issued and provide 99 percent confidence intervals for both predictions.

21.4 Transforming Numeric Variables

Sometimes, the linear function as strictly defined by the standard regression equation, (21.1), can be inadequate when it comes to capturing relationships between a response and selected covariates. You might, for example, observe curvature in a scatterplot between two numeric variables to which a perfectly straight line isn't necessarily best suited. To a certain extent, you can relax the requirement that your data exhibit such linear behavior by simply transforming (typically in a nonlinear fashion) certain variables before any estimation or model fitting takes place. In general, transformation doesn't represent a universal solution to solving problems of nonlinearity in the trends in your data, but it can at least improve how faithfully a linear model is able to represent those trends.

More specifically, *numeric transformation* refers to the application of a mathematical function to your numeric observations in order to rescale them. Finding the square root of a number or converting a temperature from Fahrenheit to Celsius are both examples of a numeric transformation. In the context of regression, transformation is most frequently applied to continuous variables and can be done in any number of ways. In this section, you'll use two of the most common approaches: *polynomial* and *logarithmic* transformations. However, note that you must consider the appropriateness of any given transformation and any modeling benefits that might occur on a case-by-case basis.

21.4.1 Polynomial

Let's say you observe a curved relationship in a plot of your data such that a straight line isn't a sensible choice for modeling it. In an effort to fit your data more closely, you can apply a polynomial or *power* transformation to a specific predictor variable in your regression model. This is a straightforward technique that, by allowing *polynomial curvature* in the relationships, allows changes in that predictor to influence the response in more complex ways than otherwise possible. You achieve this by including additional terms in the model definition that

represent the impact of progressively higher powers of the variable of interest on the response.

To clarify the concept of polynomial curvature, consider the following sequence between –4 and 4, and the simple vectors computed from it:

```
R> x <- seq(-4,4,length=50)
R> y <- x
R> y2 <- x + x^2
R> y3 <- x + x^2 + x^3
```

Here, you're taking the original value of `x` and calculating specific functionals of it. The vector `y`, as a copy of `x`, is clearly linear (in technical terms, this is a "polynomial of order 1"). You assign `y2` to take on an additionally squared value of x, providing *quadratic* behavior: a polynomial of order 2. Lastly, the vector y3 represents the results of a *cubic* function of the values of x, with the inclusion of x raised to the power of 3: a polynomial of order 3. The following three lines of code produce, separately, the plots from left to right in Figure 21-2:

```
R> plot(x,y,type="l")
R> plot(x,y2,type="l")
R> plot(x,y3,type="l")
```

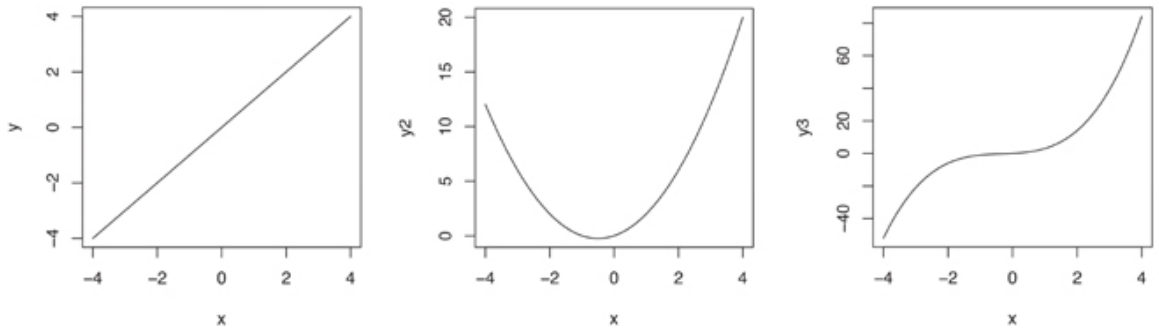


Figure 21-2: Illustrating linear (left), quadratic (middle), and cubic functions (right) of x

Perhaps a bit more generally, let's say you have data for a continuous predictor, X, that you want to use to model your response, *Y*. The simple model is $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x$; you can model a quadratic trend in X via the multiple regression $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x + \widehat{\beta}_2 x^2$; you can capture a cubic relationship by $\hat{y} = \widehat{\beta}_0 + \widehat{\beta}_1 x + \widehat{\beta}_2 x^2 + \widehat{\beta}_3 x^3$; and so on. From the plots in Figure 21-2, note that including extra terms increases the complexity of the curves that can be captured. At order 1, the linear relationship allows no curvature. At order 2, a quadratic function of any given variable allows one "bend." At order 3, the model can cope

with two bends in the relationship, and this continues if you keep adding terms corresponding to increasing powers of the covariate. The regression coefficients associated with these terms (all implied to be 1 in the code that produced the previous plots) are able to control the specific appearance (in other words, the *strength* and *direction)* of the curvature.

Fitting a Polynomial Transformation

Return your attention to the built-in mtcars dataset. Consider the disp variable, which describes engine displacement volume in cubic inches, against a response variable of miles per gallon. If you examine a plot of the data in Figure 21-3, you can see that there does appear to be a slight yet noticeable curve in the relationship between displacement and mileage:

```
R> plot(mtcars$disp,mtcars$mpg,xlab="Displacement (cu. in.)",ylab="MPG")
```

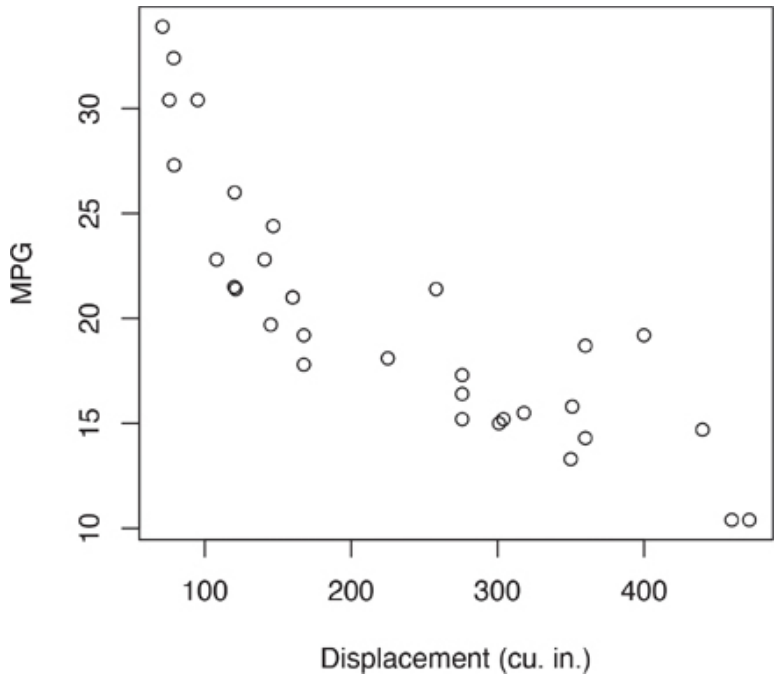


Figure 21-3: A scatterplot of miles per gallon and engine displacement for the `mtcars` data

To investigate whether the straight line that a simple linear regression model would provide is the best way to represent this relationship, start by fitting that basic linear setup:

```
R> car.order1 <- lm(mpg~disp,data=mtcars)
R> summary(car.order1)
```

```
Call:
lm(formula = mpg ~ disp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.8922 -2.2022 -0.9631  1.6272  7.2305

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
disp        -0.041215   0.004712  -8.747 9.38e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.251 on 30 degrees of freedom
Multiple R-squared:  0.7183,    Adjusted R-squared:  0.709
F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

This clearly indicates statistical evidence of a negative linear impact of displacement on mileage: For each additional cubic inch of displacement, the mean response decreases by about 0.041 miles per gallon.

Now try to capture the appArent curve in the data by adding a quadratic term in `disp` to the model. You can do this in two ways. First, you could create a new vector in the workspace by simply squaring the `mtcars$disp` vector and then supplying the result to the formula in `lm`. Second, you could specify `disp^2` directly as an additive term in the formula. If you do it this way, it's essential to wrap that particular expression in a call to `I` as follows:

```
R> car.order2 <- lm(mpg~disp+I(disp^2),data=mtcars)
R> summary(car.order2)

Call:
lm(formula = mpg ~ disp + I(disp^2), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9112 -1.5269 -0.3124  1.3489  5.3946

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.583e+01  2.209e+00  16.221 4.39e-16 ***
disp        -1.053e-01  2.028e-02  -5.192 1.49e-05 ***
I(disp^2)    1.255e-04  3.891e-05   3.226   0.0031 **
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.837 on 29 degrees of freedom
Multiple R-squared:  0.7927,    Adjusted R-squared:  0.7784
F-statistic: 55.46 on 2 and 29 DF,  p-value: 1.229e-10
```

Use of the `I` function wrapped around a given term in the formula is necessary when said term requires an arithmetical calculation—in this case, `disp^2`—to ensure validity of the formula argument. If this isn't done, the intended model will not be fitted.

Turning to the fitted model itself, you can see that the contribution of the squared component is statistically significant: The output corresponding to `I(disp^2)` shows a *p*-value of 0.0031. This implies that even if a linear trend is taken into account, the model that includes a quadratic component (which introduces a curve) is a better-fitting model. This conclusion is supported by a noticeably higher coefficient of determination compared to the first fit (0.7927 against 0.7183). You can see the fit of this quadratic curve in Figure 21-4 (the code for which follows shortly).

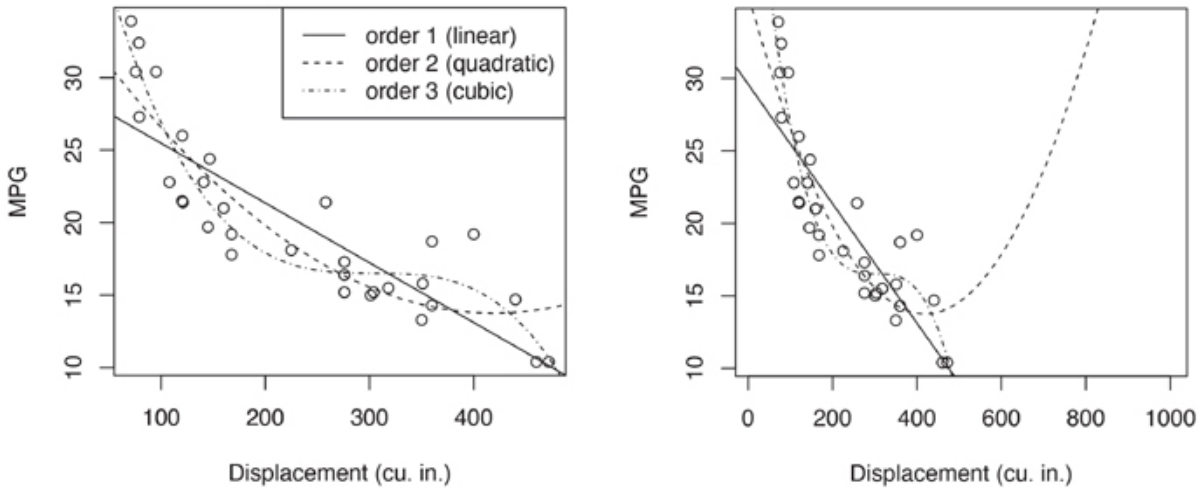


Figure 21-4: Three different models, polynomials of orders 1, 2, and 3, fitted to the "mileage per displacement" relationship from the `mtcars` dataset. Left: Visible plot limits constrained to the data. Right: Visible plot limits widened considerably to illustrate unreliability in extrapolation (right).

Here, you might reasonably wonder whether you can improve the ability of the model to capture the relationship further by adding yet another higher-order term in the covariate of interest. Let's add a cubic term and take a look:

```
R> car.order3 <- lm(mpg~disp+I(disp^2)+I(disp^3),data=mtcars)
R> summary(car.order3)
```

```
Call:
lm(formula = mpg ~ disp + I(disp^2) + I(disp^3), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0896 -1.5653 -0.3619  1.4368  4.7617

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.070e+01  3.809e+00  13.310 1.25e-13 ***
disp        -3.372e-01  5.526e-02  -6.102 1.39e-06 ***
I(disp^2)    1.109e-03  2.265e-04   4.897 3.68e-05 ***
I(disp^3)   -1.217e-06  2.776e-07  -4.382  0.00015 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.224 on 28 degrees of freedom
Multiple R-squared:  0.8771,    Adjusted R-squared:  0.8639
F-statistic: 66.58 on 3 and 28 DF,  p-value: 7.347e-13
```

The output shows that a cubic component also offers a statistically significant contribution. However, if you were to continue adding higherorder terms, you'd find that fitting a polynomial of order 4 to these data isn't able to improve the fit at all, with several coefficients being rendered nonsignificant (the order 4 fit isn't shown).

So, letting $\hat{y}$ be miles per gallon and x be displacement in cubic inches, and expanding the e-notation from the previous output, the fitted multiple regression model is

$$\hat{y} = 50.7 - 0.3372x + 0.0011x^2 - 0.000001x^3$$

which is precisely what the "order 3" line in the left panel of Figure 21-4 reflects.

Plotting the Polynomial Fit

To address the plot itself, you visualize the data and the first (simple linear) model in `car.order1` in the usual way. To begin Figure 21-4, execute the following code:

```
R> plot(mtcars$disp,mtcars$mpg,xlab="Displacement (cu. in.)",ylab="MPG")
R> abline(car.order1)
```

It's a little more difficult to add the line corresponding to either of the polynomial-termed models since abline is equipped to handle only straight-line

trends. One way to do this is to make use of `predict` for each value in a sequence that represents the desired values of the explanatory variable. (I favor this approach because it also allows you to simultaneously calculate confidence and prediction bands ifyou want.) To add the line for the order 2 model only, first create the required sequence (see Section 2.3.2) over the observed range of disp:

```
R> disp.seq <- seq(min(mtcars$disp)-50,max(mtcars$disp)+50,length=30)
```

Here, the sequence has been widened a little by minus and plus 50 to predict a small amount on either side of the scope of the original covariate data, so the curve meets the edges of the graph. Then you make the prediction itself and superimpose the fitted line with `lines` (Section 8):

```
R> car.order2.pred <- predict(car.order2,newdata=data.frame(disp=disp.seq))
R> lines(disp.seq,car.order2.pred,lty=2)
```

You use the same technique, followed by the final addition of the legend, for the order 3 polynomial:

```
R> car.order3.pred <- predict(car.order3,newdata=data.frame(disp=disp.seq))
R> lines(disp.seq,car.order3.pred,lty=4)
R> legend("topright",lty=c(1,2,4),
          legend=c("order 1 (linear)","order 2 (quadratic)","order 3 (cubic)"))
```

The result of all this is in the left panel of Figure 21-4. Even though you've used raw data from only one covariate, `disp`, the example illustrated here is considered multiple regression because more than one parameter (in addition to the universal intercept β_0) required estimation in the order 2 and 3 models.

The different trend lines fitted to the mileage-on-displacement data clearly show different interpretations of the relationship. Visually, you could reasonably argue that the simple linear fit is inadequate at modeling the relationship between response and predictor, but it's harder to come to a clear conclusion when choosing between the order 2 and order 3 versions. The order 2 fit captures the curve that tapers off as `disp` increases; the order 3 fit additionally allows for a bump (in technical terms a *saddle* or *inflection*), followed by a steeper downward trend in the same domain.

So, which model is "best"? In this case, the statistical significance of the parameters suggests that the order 3 model should be preferred. Having said that, there are other things to consider when choosing between different models, which you'll also think about more carefully in Section 22.2.

Knowing the Pitfalls of Polynomials

One particular drawback associated with polynomial terms in linear regression models is the instability of the fitted trend when trying to perform any kind of extrapolation. The right plot in Figure 21-4 shows the same three fitted models (MPG by displacement), but this time with a much wider scale for displacement. To create this plot, you can use the same code that created the left plot; simply use `xlim` in the call to `plot` to widen the x-axis range and define the `disp.seq` object to a correspondingly wider sequence (in this case, I just set `xlim=c(10,1000)` with matching `from` and `to` limits in the creation of `disp.seq`).

As you can see, the validity of these models anywhere beyond the observed scope of the predictor is questionable. Though the order 2 and 3 models fit MPG acceptably within the range of the observed data, if you move even slightly outside the maximum threshold of observed displacement values, the predictions of the mean mileage go wildly off course. The order 2 model in particular becomes completely nonsensical, suggesting a rapid improvement in MPG once the engine displacement rises over 500 cubic inches. You must keep this natural mathematical behavior of polynomial functions in mind if you're considering using higher-order terms in your regression models.

NOTE

We still refer to models like this as linear *regression models, which might seem a bit confusing since the fitted trends for higher-order polynomials are clearly nonlinear. This is because "linear regression" refers to the fact that the function defining the mean response is linear in terms of the regression parameters* $\beta_0, \beta_1, \ldots, \beta_p$. *As such, any transformation applied to individual variables doesn't affect the linearity of the function with respect to the coefficients themselves.*

21.4.2 Logarithmic

In statistical modeling situations where you have positive numeric observations, it's common to perform a log transformation when there is need to dramatically reduce the overall range of the data and bring extreme observations closer to a measure of centrality. In that sense, transformation to a logarithmic scale can help reduce the severity of heavily skewed data (see Section 15.2.4). In the context of regression modeling, you can use log transformations to capture trends where appArent curves "flatten off," without the same kind of instability outside the range of the observed data that you saw with some of the polynomials.

To refresh your memory on logarithms, turn back to Section 2.1.2; it suffices here to note that the logarithm is the power to which you must raise a base value in order to obtain an x value. For example, in $3^5 = 243$, the logarithm is 5 and 3 is the base, expressed as $\log_3 243 = 5$. Because of the ubiquity of the exponential function in common probability distributions, statisticians often work with the natural log (logarithm to the base e). From here, assume all mentions of the log transformation refer to the natural log.

To briefly illustrate the typical behavior of the log transformation, take a look at Figure 21-5, achieved with the following:

```
R> plot(1:1000,log(1:1000),type="l",xlab="x",ylab="",ylim=c(-8,8))
R> lines(1:1000,-log(1:1000),lty=2)
R> legend("topleft",legend=c("log(x)","-log(x)"),lty=c(1,2))
```

This plots the log of the integers 1 to 1000 against the raw values, as well as plotting the negative log. You can see the way in which the logtransformed values taper off and flatten out as the raw values increase.

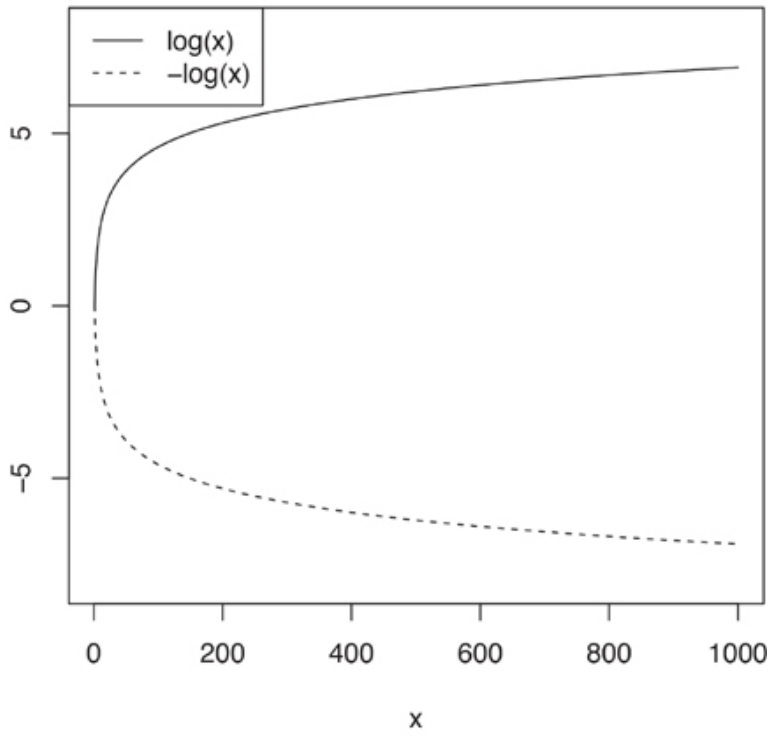


Figure 21-5: The log function applied to integers 1 to 1,000

Fitting the Log Transformation

As noted, one use of the log transformation in regression is to allow this kind of curvature in situations when a perfectly straight line doesn't suit the observed relationship. For an illustration, return to the `mtcars` examples and consider mileage as a function of both horsepower and transmission type (variables `hp` and `am`, respectively). Create a scatterplot of MPG against horsepower, with different colors distinguishing between automatic and manual cars:

```
R> plot(mtcars$hp,mtcars$mpg,pch=19,col=c("black","gray")[factor(mtcars$am)],
        xlab="Horsepower",ylab="MPG")
R> legend("topright",legend=c("auto","man"),col=c("black","gray"),pch=19)
```

The plotted points in Figure 21-6 suggest that curved trends in horsepower may be more appropriate than straight-line relationships. Note that you have to explicitly coerce the binary numeric `mtcars$am` vector to a factor here in order to use it as a selector for the vector of two colors. You'll add the lines in after fitting the linear model.

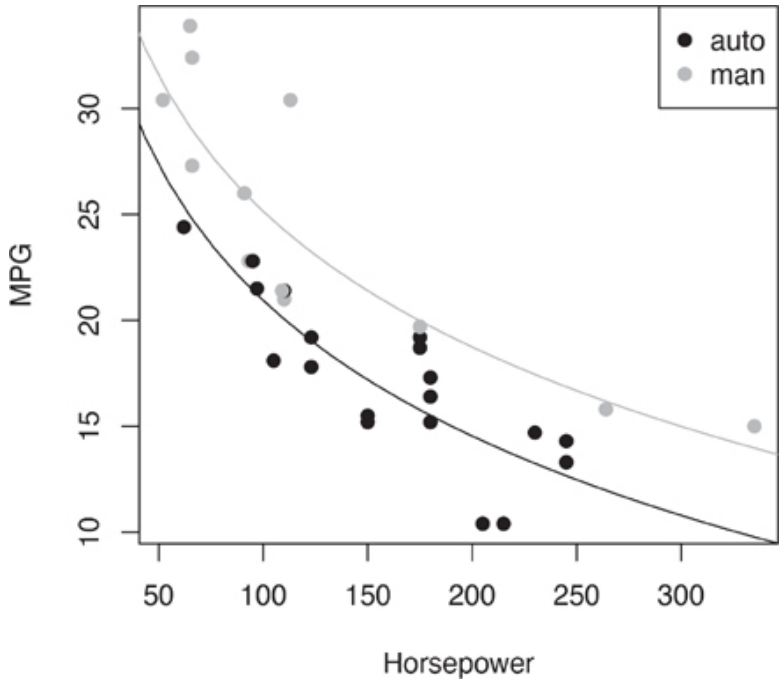


Figure 21-6: A scatterplot of MPG on horsepower, split by transmission type, with lines corresponding to a multiple linear regression using a log-scaled effect of horsepower superimposed

Let's fit a model using a log transformation of horsepower to try to capture the appArent curvature. Since, in this example, you also want to account for the potential of transmission type to affect the response, this is included as an additional predictor variable:

```
R> car.log <- lm(mpg~log(hp)+am,data=mtcars)
R> summary(car.log)

Call:
lm(formula = mpg ~ log(hp) + am, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9084 -1.7692 -0.1432  1.4032  6.3865

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  63.4842     5.2697  12.047 8.24e-13 ***
log(hp)      -9.2383     1.0439  -8.850 9.78e-10 ***
am            4.2025     0.9942   4.227 0.000215 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.592 on 29 degrees of freedom
Multiple R-squared:  0.827,     Adjusted R-squared:  0.8151
F-statistic: 69.31 on 2 and 29 DF,  p-value: 8.949e-12
```

The output indicates jointly statistically significant effects of both loghorsepower and transmission type on mileage. Keeping transmission constant, the mean MPG drops by around 9.24 for each additional unit of log-horsepower. Having a manual transmission increases the mean MPG by roughly 4.2 (estimated in this order owing to the coding of `am`: `0` for automatic, `1` for manual; see `?mtcars`). The coefficient of determination shows 82.7 percent of the variation in the response is explained by this regression, suggesting a satisfactory fit.

Plotting the Log Transformation Fit

To visualize the fitted model, you first need to calculate the fitted values for all desired predictor values. The following code creates a sequence of horsepower values (minus and plus 20 horsepower from the minimum and maximum observed values) and performs the required prediction for both transmission types:

```
R> hp.seq <- seq(min(mtcars$hp)-20,max(mtcars$hp)+20,length=30)
R> n <- length(hp.seq)
R> car.log.pred <- predict(car.log,newdata=data.frame(hp=rep(hp.seq,2),
                                                      am=rep(c(0,1),each=n)))
```

Since you want to plot predictions for both possible values of `am`, when using `newdata` you need to replicate `hp.seq` twice. Then, when you provide values for `am` to

`newdata`, one series of `hp.seq` is paired with an appropriately replicated `am` value of `0`, the other with `1`. The result of this is a vector of predictions of length twice that of `hp.seq`, `car.log.pred`, with the first `n` elements corresponding to automatic cars and the latter `n` to manuals.

Now, assuming you have the scatterplot of Figure 21-6 ready, you can add the fitted regression model lines with the following:

```
R> lines(hp.seq,car.log.pred[1:n])
R> lines(hp.seq,car.log.pred[(n+1):(2*n)],col="gray")
```

By examining the result, you can see that the fitted model appears to do a good job of estimating the joint relationship between horsepower/ transmission and MPG. The statistical significance of transmission type in this model directly affects the difference between the two added lines. If the `am` variable wasn't significant, the lines would be closer together (in that case, the model would be suggesting that one curve would be sufficient to capture the relationship). As usual, extrapolation too far outside the range of the observed predictor data isn't a great idea, though it's less unstable for log-transformed trends than for polynomial functions.

21.4.3 Other Transformations

Transformation can involve more than one variable of the dataset and isn't limited to just predictor variables either. In their original investigation into the `mtcars` data, Henderson & Velleman (1981) also noted the presence of the same curved relationships you've uncovered between the response and variables such as horsepower and displacement. They argued that it's preferable to use gallons per mile (GPM) instead of MPG as the response variable to improve linearity. This would involve modeling a transformation of MPG, namely, that GPM = 1/MPG.

The authors also commented on the limited influence that both horsepower and displacement have on GPM if the weight of the car is included in a fitted model because of the relatively high correlations present among these three predictors (a phenomenon known as *multicollinearity*). To address this, the authors created a new predictor variable calculated as horsepower divided by weight. This measures, in their words, how "overpowered" a car is, and they proceeded to use that new predictor instead of horsepower or displacement alone. This is just some of the experimentation that took place in the search for an appropriate way to model this particular dataset.

To this end, however you choose to model your own data, the objective of transforming numeric variables should always be to fit a valid model that

represents the data and the relationships therein more realistically and accurately. When reaching for this goal, there's plenty of freedom in how you can transform numeric observations in applications of regression methods. For a further discussion on transformations in linear regression, Chapters 7 and 9 of Faraway (2014) provide informative discussions.

Exercise 21.2

The following table presents data collected in one of Galileo's famous "ball" experiments, in which he rolled a ball down a ramp of different heights and measured how far it traveled from the base of the ramp. For more on this and other interesting examples, see "Teaching Statistics with Data of Historic Significance" by Dickey & Arnold (1995).

Initial height	Distance
1,000	573
800	534
600	495
450	451
300	395
200	337
100	253

a. Create a data frame in R based on this table and plot the data points with distance on the y-axis.

b. Galileo hypothesized there was a quadratic relationship between initial height and the distance traveled.

   i. Use skills from Section 21.4.1 to fit a linear model with an order 2 polynomial in height (set distance travelled as the response).

   ii. Fit also a cubic (order 3) and a quartic (order 4) model for these data and inspect the summarized output. What do

these models tell you about the nature of the relationship?

c. Based on your models from (b), choose the one that you think best represents the data and plot the fitted line on the raw data. Add 90 percent confidence bands for mean distance traveled to the plot.

The contributed R package `faraway` contains a large number of datasets that accompany a linear regression textbook by Faraway (2014). Install the package from CRAN (see Section 1.3.5) and then call `library("faraway")` to load it. One of the datasets is `trees`, which provides data on the dimensions of felled trees of a certain type (for example, see Atkinson 1985).

d. Access the data object at the prompt and plot volume against girth (the latter along the x-axis).

e. Fit two models with `Volume` as the response: one quadratic model in `Girth` and the other based on log transformations of both `Volume` and `Girth`. Write down the model equations for each and comment on the similarity (or difference) of the fits in terms of the coefficient of determination and the omnibus *F*-test.

f. Use `predict` to add lines to the plot from (d) for each of the two models from (e). Use different line types; add a corresponding legend. Also include 95 percent prediction intervals, with line types matching those of the fitted values (note that for the model that involves log transformation of the response and the predictor, any returned values from `predict` will themselves be on the log scale; you have to backtransform these to the original scale using `exp` before the lines for that model can be superimposed). Comment on the respective fits and their estimated prediction intervals.

Lastly, turn your attention back to the built-in `mtcars` data frame used in several examples and exercises in this chapter.

g. Fit and summarize a multiple linear regression model in order to determine mean MPG from horsepower, weight, and displacement.

h. In the spirit of Henderson & Velleman (1981), use `I` (see Section 21.4.1) in order to refit the model in (g) in terms of GPM = 1/MPG. Which of the two models, this one or the one from (g), explains a greater amount of variation in the response?

21.5 Interactive Terms

So far, you've looked only at the "main effects" of how predictors affect the outcome variable (and one-to-one transformations thereof). Now you'll look at interactions between covariates. An *interactive effect* between predictors is an additional change to the response that occurs at particular combinations of the predictor values. In other words, an interactive effect is present if, for a given covariate profile, the values of the predictors are such that they produce an effect that "augments" the otherwise stand-alone main effects associated with those predictors.

21.5.1 Concept and Motivation

We often use diagrams such as those found in Figure 21-7 to help explain the concept of interactive effects. These diagrams show your mean response value, $\hat{y}$, on the vertical axis, as usual, and a predictor value for the variable x_1 on the horizontal axis. They also show a binary categorical variable x_2, which can be either zero or one. These hypothetical variables are labeled as such in the images.

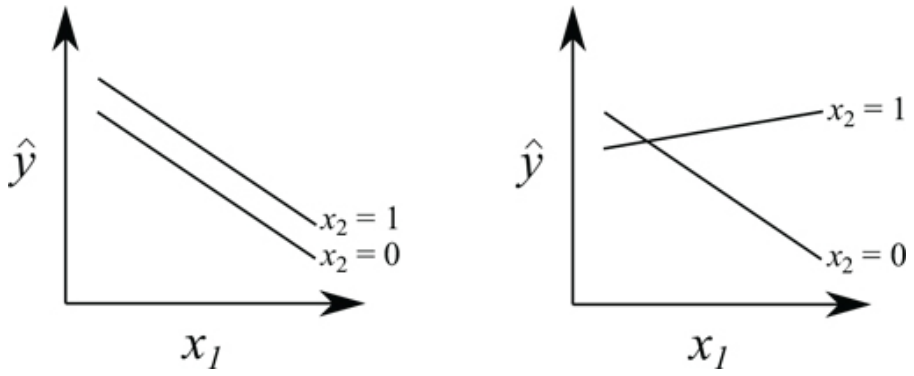


Figure 21-7: The concept of an interactive effect between two predictors, x_1 and x_2, on the mean response value $\hat{y}$. Only main effects of x_1 and x_2 influence $\hat{y}$ (left). An interaction between x_1 and x_2 is needed in addition to their main effects in order to model $\hat{y}$ (right).

The left diagram shows the limit of the models you've considered so far in this chapter: that both x_1 and x_2 affect $\hat{y}$ independently of each other. The right diagram, however, clearly shows that the effect of x_1 on $\hat{y}$ changes completely depending on the value of x_2. On the left, only main effects of x_1 and x_2 are needed to determine $\hat{y}$; on the right, main effects *and* an interactive effect between x_1 and x_2 are present.

NOTE

When estimating regression models, you ought always to accompany interactions with the main effects of the relevant predictors, for reasons of interpretability. Since interactions are themselves best understood as an augmentation of the main effects, it makes no sense to remove the latter and leave in the former.

Pharmacology provides good examples of interactions. Interactive effects between medicines are relatively common, which is why health care professionals often ask about other medicines you might be taking. Users of statins (drugs commonly used to reduce cholesterol) are told to avoid grapefruit juice because it contains natural chemical compounds that inhibit the efficacy of the enzyme responsible for the correct metabolization of the drug. If an individual is taking statins and not consuming grapefruit, you would expect a negative relationship between cholesterol level and statin use (think about "statin use" either as a continuous or a categorical dosage variable); that is, as statin use increases or is affirmative, the cholesterol level decreases. On the other hand, for an individual on statins who *is* consuming grapefruit, the nature of the relationship between cholesterol level and statin use could be weakened negative, neutral, or even positive. Since the effect of the statins on cholesterol changes according to the value of another variable (whether grapefruit is consumed) this would be considered an interaction between those two predictors.

Interactions can occur between categorical variables, numeric variables, or both. As it's most common to encounter *two-way* interactions, which occur between exactly two predictors, you'll focus on this in Sections 21.5.2 through 21.5.4. Three-way and higher-order interactive effects are technically possible but less common, partly because they are difficult to interpret in a real-world context. Nevertheless, you'll also consider an example of this in Section 21.5.5.

21.5.2 One Categorical, One Continuous

You can interpret a two-way interaction between a categorical and a continuous predictor as effecting a change in the slope of the continuous predictor with respect to the non-reference levels of the categorical predictor. In the presence of a term for the continuous variable, a categorical variable with k levels will have $k - 1$ main effect terms, so there will be a further $k - 1$ interactive terms between all the alternative levels of the categorical variable and the continuous variable.

The right-hand image in Figure 21-7 clearly shows the different slopes for x_1 by category of x_2 for $\hat{y}$. In such a situation, in addition to the main effects for x_1 and x_2, there would be one interactive term in the fitted model corresponding to

$x_2 = 1$. This defines the additive term needed to change the slope in x_1 for $x_2 = 0$ to the new slope in x_1 for $x_2 = 1$.

For an example, let's access a new dataset. In Exercise 21.2, you looked at the `faraway` package (Faraway 2014) to access the `trees` data. In this package, you'll also find the `diabetes` object: a dataset detailing various cardiovascular risk factors and other characteristics of 403 African Americans (originally investigated and reported in Schorling et al. 1997, Willems et al. 1997). Install `faraway` if you haven't already and load it with `library("faraway")`. Restrict your attention to the total cholesterol level (`chol`: continuous), age of the individual (`age`: continuous), and body frame type (`frame`: categorical with $k = 3$ levels: `"small"` as the reference level, `"medium"`, and `"large"`). The data are plotted in Figure 21-8 on page 589.

You'll look at modeling total cholesterol by age and body frame. It seems logical to expect that cholesterol is related to both age and body type, so it makes sense to also consider the possibility that the effect of age on cholesterol is different for individuals of different body frames. To investigate, let's fit the multiple linear regression and include a two-way interaction between the two variables. In the call to `lm`, you specify the main effects first, using `+` as usual, and then specify an interactive effect of two predictors by using a colon (`:`) between them:

```
R> dia.fit <- lm(chol~age+frame+age:frame,data=diabetes)
R> summary(dia.fit)

Call:
lm(formula = chol ~ age + frame + age:frame, data = diabetes)

Residuals:
    Min      1Q  Median      3Q     Max
-131.90  -26.24   -5.33   22.17  226.11

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)     155.9636    12.0697  12.922  < 2e-16 ***
age               0.9852     0.2687   3.667  0.00028 ***
framemedium      28.6051    15.5503   1.840  0.06661 .
framelarge       44.9474    18.9842   2.368  0.01840 *
age:framemedium  -0.3514     0.3370  -1.043  0.29768
age:framelarge   -0.8511     0.3779  -2.252  0.02490 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 42.34 on 384 degrees of freedom
  (13 observations deleted due to missingness)
Multiple R-squared:  0.07891,   Adjusted R-squared:  0.06692
F-statistic:  6.58 on 5 and 384 DF,  p-value: 6.849e-06
```

Inspecting the estimated model parameters in the output, you can see a main effect coefficient for `age`, main effect coefficients for the two levels of `frame` (that aren't the reference level), and two further terms for the interactive effect of `age` with those same non-reference levels.

NOTE

R's cross-factor *notation provides a shortcut to doing this. The same model shown previously could have been fitted by using* `chol~age*frame` *in* `lm`*; you can interpret the* `*` *symbol as "include an intercept, all main effects, and the interaction." I'll use this shortcut from now on.*

The output shows the significance of `age` and some evidence to support the presence of a main effect of `frame`. Assuming a significance level of $\alpha = 0.05$, there's also a slight indication of significance of the interaction with a p-value of around 0.025 for the `age:framelarge` coefficient. Assessing significance in this case, where one predictor is categorical with $k > 2$ levels, follows the same rule as noted in the discussion of multilevel variables in Section 20.5.2: If at least one of the coefficients is associated with a significant p-value, the entire effect should be deemed significant.

You can write the general equation for the fitted model directly from the output as follows:

$$\begin{aligned}\text{\textbackslash\textbackslash"Mean total cholesterol"} = \; & 155.9636 + 0.9852 \times \text{\textbackslash\textbackslash"age"} \\ & + 28.6051 \times \text{\textbackslash\textbackslash"medium frame"} \\ & + 44.9474 \times \text{\textbackslash\textbackslash"large frame"} \\ & - 0.3514 \times \text{\textbackslash\textbackslash"age : medium frame"} \\ & - 0.8511 \times \text{\textbackslash\textbackslash"age : large frame"} \end{aligned} \tag{21.7}$$

Note I've used a colon (:) to denote the interactive terms to mirror the R output.

For the reference level of the categorical predictor, body type "small," the fitted model can be expressed as the following equation:

"Mean total cholesterol" = 155.9636 + 0.9852 × "age"

For a model with the main effects only, changing body type to "medium" or "large" would affect only the intercept; you know from Section 20.5 that the relevant effect is simply added to the outcome, effectively offsetting $\widehat{\beta}_0$. The presence of the interaction, however, means that *in addition* to the change in the intercept, the slope of `age` must now also be changed according to the relevant interactive term. For an individual with a "medium" frame, the model is

$$\begin{aligned}
\text{"Mean total cholesterol"} &= 155.9636 + 0.9852 \times \text{"age"} + 28.6051 \\
&\quad - 0.3514 \times \text{"age"} \\
&= 184.5687 + (0.9852 - 0.3514) \times \text{"age"} \\
&= 184.5687 + 0.6338 \times \text{"age"}
\end{aligned}$$

and for an individual with a "large" frame, the model is:

$$\begin{aligned}
\text{"Mean total cholesterol"} &= 155.9636 + 0.9852 \times \text{"age"} + 44.9474 \\
&\quad - 0.8511 \times \text{"age"} \\
&= 200.911 + (0.9852 - 0.8511) \times \text{"age"} \\
&= 200.911 + 0.1341 \times \text{"age"}
\end{aligned}$$

You can easily calculate these in R by accessing the coefficients of the fitted model object:

```
R> dia.coef <- coef(dia.fit)
R> dia.coef
    (Intercept)             age     framemedium      framelarge
    155.9635868       0.9852028      28.6051035      44.9474105
age:framemedium  age:framelarge
     -0.3513906      -0.8510549
```

Next, let's sum the relevant components of this vector. With these results, you'll be able to visualize the fitted model:

```
R> dia.small <- c(dia.coef[1],dia.coef[2])
R> dia.small
(Intercept)         age
155.9635868   0.9852028
R> dia.medium <- c(dia.coef[1]+dia.coef[3],dia.coef[2]+dia.coef[5])
R> dia.medium
(Intercept)         age
184.5686904   0.6338122
R> dia.large <- c(dia.coef[1]+dia.coef[4],dia.coef[2]+dia.coef[6])
R> dia.large
```

```
(Intercept)         age
200.9109973   0.1341479
```

The three lines are stored as numeric vectors of length 2, with the intercept first and the slope second. This is the form required by the optional `coef` argument of `abline`, which allows you to superimpose these straight lines on a plot of the raw data. The following code produces Figure 21-8:

```
R> cols <- c("black","darkgray","lightgray")
R> plot(diabetes$chol~diabetes$age,col=cols[diabetes$frame],
       cex=0.5,xlab="age",ylab="cholesterol")
R> abline(coef=dia.small,lwd=2)
R> abline(coef=dia.medium,lwd=2,col="darkgray")
R> abline(coef=dia.large,lwd=2,col="lightgray")
R> legend("topright",legend=c("small frame","medium frame","large frame"),
         lty=1,lwd=2,col=cols)
```

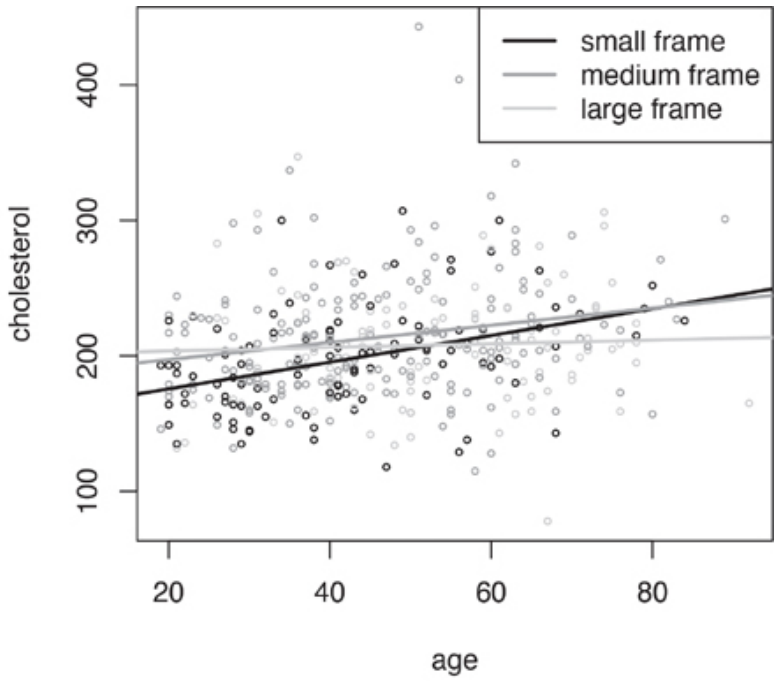


Figure 21-8: The fitted linear model, main effects, and interaction for mean total cholesterol by age and body frame

If you examine the fitted model in Figure 21-8, it's clear that inclusion of an interaction between age and body frame has allowed more flexibility in the way that mean total cholesterol relates to the two predictors. The nonparallel nature of the three plotted lines reflects the concept illustrated in Figure 21-7.

I walked through this to illustrate how the concept works, but in practice you don't need to go through all of these steps to find the point estimates (and any associated confidence intervals). You can predict from a fitted linear model with interactions in the same way as for main-effect-only models through the use of `predict`.

21.5.3 Two Categorical

You met the concept of interactions between two categorical explanatory variables in the introduction to two-way ANOVA in Section 19.2. There, you uncovered evidence of an interactive effect of wool type and tension on the mean number of warp breaks in lengths of yarn (based on the readyto-use `warpbreaks` data frame). You then visualized the interaction with an interaction plot (Figure 19-2 on page 515) that, unsurprisingly, resembles the diagrams in Figure 21-7.

Let's implement the same model as the last `warpbreaks` example in Section 19.2.2 in an explicit linear regression format:

```
R> warp.fit <- lm(breaks~wool*tension,data=warpbreaks)
R> summary(warp.fit)

Call:
lm(formula = breaks ~ wool * tension, data = warpbreaks)

Residuals:
     Min       1Q   Median       3Q      Max
-19.5556  -6.8889  -0.6667   7.1944  25.4444

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   44.556      3.647  12.218 2.43e-16 ***
woolB        -16.333      5.157  -3.167 0.002677 **
tensionM     -20.556      5.157  -3.986 0.000228 ***
tensionH     -20.000      5.157  -3.878 0.000320 ***
woolB:tensionM  21.111      7.294   2.895 0.005698 **
woolB:tensionH  10.556      7.294   1.447 0.154327
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.94 on 48 degrees of freedom
Multiple R-squared:  0.3778,    Adjusted R-squared:  0.3129
F-statistic: 5.828 on 5 and 48 DF,  p-value: 0.0002772
```

Here, I've used the cross-factor `*` symbol, rather than `wool + tension + wool:tension`. When both predictors in a two-way interaction are categorical, there will be a term for each nonreference level of the first predictor combined with all nonreference levels of the second predictor. In this example, `wool` is binary with only $k = 2$ levels and `tension` has $k = 3$; therefore, the only interaction terms present are the "medium" (`M`) and "high" (`H`) tension levels ("low", `L`, is the reference level) with wool type `B` (`A` is the reference level). Therefore, altogether in the fitted model, there are terms for `B`, `M`, `H`, `B:M`, and `B:H`.

These results provide the same conclusion as the ANOVA analysis: There is indeed statistical evidence of an interactive effect between wool type and tension on mean breaks, on top of the contributing main effects of those predictors.

You can understand the fitted model as:

$$\begin{aligned}\text{"Mean warp breaks"} = \; & 44.556 - 16.333 \times \text{"wool type B"} \\ & - 20.556 \times \text{"medium tension"} \\ & - 20.000 \times \text{"high tension"} \\ & + 21.111 \times \text{"wool type B : medium tension"} \\ & + 10.556 \times \text{"wool type B : high tension"}\end{aligned}$$

The additional interaction terms work the same way as the main effects: When only categorical predictors are involved, the model can be seen as a series of potential additive terms to the baseline intercept. Exactly which of these additive terms is used in any given prediction depends on the specific values of the two predictors.

Let's have a quick series of examples: For wool A at low tension, the mean number of warp breaks is predicted as simply the overall intercept; none of the other coefficients are relevant. For wool A at high tension, you need the overall intercept and the main effect term for high tension; for wool B at low tension, you have the overall intercept and the main effect for wool type B only; and for wool B at medium tension, you need the overall intercept, the main effect for wool type B, the main effect for medium tension, *and* the interactive term for wool B with medium tension.

You can use `predict` to estimate the mean warp breaks for these four scenarios; they're accompanied here with 90 percent confidence intervals:

```
R> nd <- data.frame(wool=c("A","A","B","B"),tension=c("L","H","L","M"))
R> predict(warp.fit,newdata=nd,interval="confidence",level=0.9)
       fit      lwr      upr
1 44.55556 38.43912 50.67199
```

```
2 24.55556 18.43912 30.67199
3 28.22222 22.10579 34.33866
4 28.77778 22.66134 34.89421
```

21.5.4 Two Continuous

Finally, you'll look at the situation when the two predictors that interact are both continuous. In this case, an interaction term operates as a modifier on the continuous plane that's fitted using the main effects only. In a similar way to an interaction between a continuous and a categorical predictor, an interaction between two continuous explanatory variables allows the slope associated with one variable to be affected, but this time, that modification is made in a continuous way (that is, according to the value of the other continuous variable).

Returning to the `mtcars` data frame, consider MPG once more as a function of horsepower and weight. The fitted model, shown next, includes the interaction in addition to the main effects of the two continuous predictors. As you can see, there is a single estimated interactive term, and it is deemed significantly different from zero:

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> summary(car.fit)

Call:
lm(formula = mpg ~ hp * wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.0632 -1.6491 -0.7362  1.4211  4.5513

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 49.80842    3.60516  13.816 5.01e-14 ***
hp          -0.12010    0.02470  -4.863 4.04e-05 ***
wt          -8.21662    1.26971  -6.471 5.20e-07 ***
hp:wt        0.02785    0.00742   3.753 0.000811 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.153 on 28 degrees of freedom
Multiple R-squared:  0.8848,    Adjusted R-squared:  0.8724
F-statistic: 71.66 on 3 and 28 DF,  p-value: 2.981e-13
```

You write the model as follows:

$$
\begin{aligned}
\text{\\Mean MPG//} &= 49.80842 - 0.12010 \times \text{\\horsepower//} \\
&\quad - 8.21662 \times \text{\\weight//} \\
&\quad + 0.02785 \times \text{\\horsepower : weight//} \\
&= 49.80842 - 0.12010 \times \text{\\horsepower//} \\
&\quad - 8.21662 \times \text{\\weight//} \\
&\quad + 0.02785 \times \text{\\horsepower//} \times \text{\\weight//}
\end{aligned}
$$

The second version of the model equation provided here reveals for the first time an interaction expressed as the *product* of the values of the two predictors, which is exactly how the fitted model is used to predict the response. (Technically, this is the same as when at least one of the predictors is categorical, but the dummy coding simply results in zeros and ones for the respective terms, so multiplication just amounts to the presence or absence of a given term, as you've seen.)

You can interpret an interaction between two continuous predictors by considering the sign (+ or –) of the coefficient. Negativity suggests that as the values of the predictors increase, the response is reduced after computing the result of the main effects. Positivity, as is the case here, suggests that as the values of the predictors increase, the effect is an additional increase, an amplification, on the mean response.

Contextually, the negative main effects of `hp` and `wt` indicate that mileage is naturally reduced for heavier, more powerful cars. However, positivity of the interactive effect suggests that this impact on the response is "softened" as horsepower or weight is increased. To put it another way, the negative relationship imparted by the main effects is rendered "less extreme" as the values of the predictors get bigger and bigger.

Figure 21-9 contrasts the main-effects-only version of the model (obtained using `lm` with the formula `mpg~hp+wt`; not explicitly fitted in this section) with the interaction version of the model fitted just above as the object `car.fit`.

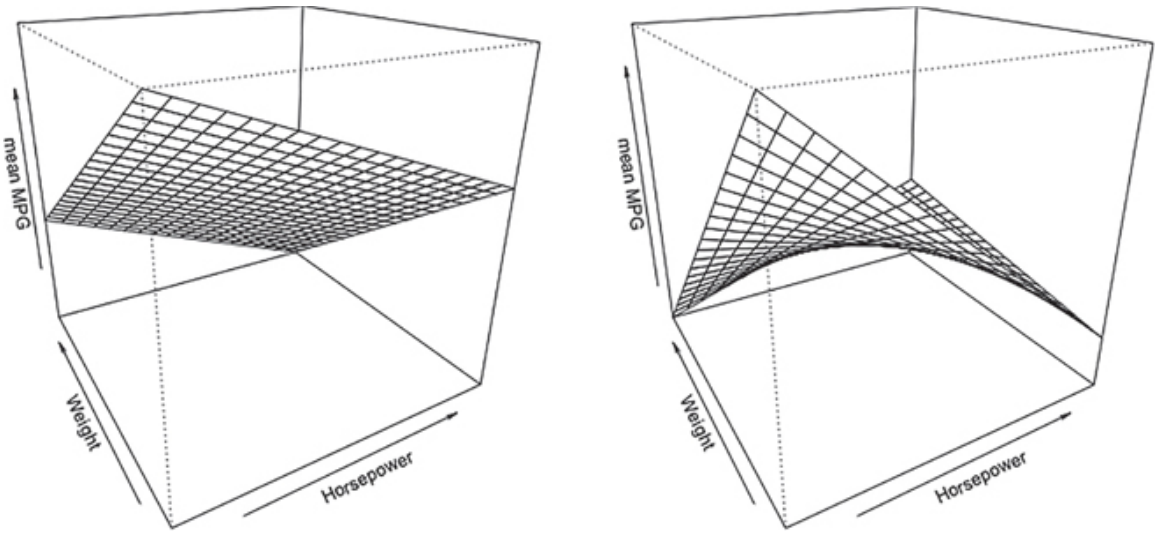


Figure 21-9: The response surfaces for mean MPG by horsepower and weight for a main-effects-only model (left) and one that includes the two-way interaction between the continuous predictors (right)

The plotted *response surfaces* show the mean MPG on the vertical z-axis and the two predictor variables on the horizontal axes as marked. You can interpret the predicted mean MPG, based on a given horsepower and weight value, as a point on the surface. Note that both surfaces decrease in MPG (vertically along the z-axis) as you move to larger values of either predictor along the respective horizontal axes.

You'll see how to create these plots in Chapter 25. For now, they serve simply to highlight the aforementioned "softening" impact of the interaction in `car.fit`. On the left, the main-effects-only model shows a flat plane decreasing according to the negative linear slopes in each predictor. On the right, however, the presence of the positive interactive term flattens this plane out, meaning the rate of decrease is slowed as the values of the predictor variables increase together.

21.5.5 Higher-Order Interactions

As mentioned, two-way interactions are the most common kind of interactions you'll encounter in applications of regression methods. This is because for three-way or higher-order terms, you need a lot more data for a reliable estimation of interactive effects, and there are a number of interpretative complexities to overcome. Three-way interactions are far rarer than two-way effects, and four-way and above are rarer still.

In Exercise 21.1, you used the `nuclear` dataset found in the `boot` package (provided with the standard R installation), which includes data on the constructions of nuclear power plants in the United States. In the exercises, you

focused mainly on date and time predictors related to construction permits to model the mean cost of construction for the nuclear power plants. For the sake of this example, assume you don't have the data on these predictors. Can the cost of construction be adequately modeled using only the variables that describe characteristics of the plant itself?

Load the `boot` package and access the `?nuclear` help page to find details on the variables: `cap` (continuous variable describing the capacity of the plant); `cum.n` (treated as continuous, describing the number of similar constructions the engineers had previously worked on); `ne` (binary, describing whether the plant was in the northeastern United States); and `ct` (binary, describing whether the plant had a cooling tower).

The following model is fitted with the final construction cost of the plant as the response. We include a main effect for capacity, as well as main effects of, and all two-way interactions and the three-way interaction among, `cum.n`, `ne`, and `ct`:

```
R> nuc.fit <- lm(cost~cap+cum.n*ne*ct,data=nuclear)
R> summary(nuc.fit)

Call:
lm(formula = cost ~ cap + cum.n * ne * ct, data = nuclear)

Residuals:
     Min       1Q   Median       3Q      Max
-162.475  -50.368   -8.833   43.370  213.131

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  138.0336    99.9599   1.381 0.180585
cap            0.5085     0.1127   4.513 0.000157 ***
cum.n        -24.2433     6.7874  -3.572 0.001618 **
ne          -260.1036   164.7650  -1.579 0.128076
ct          -187.4904    76.6316  -2.447 0.022480 *
cum.n:ne      44.0196    12.2880   3.582 0.001577 **
cum.n:ct      35.1687     8.0660   4.360 0.000229 ***
ne:ct        524.1194   200.9567   2.608 0.015721 *
cum.n:ne:ct  -64.4444    18.0213  -3.576 0.001601 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 107.3 on 23 degrees of freedom
Multiple R-squared:  0.705,     Adjusted R-squared:  0.6024
F-statistic: 6.872 on 8 and 23 DF,  p-value: 0.0001264
```

In this code, you specify the higher-order interactions by extending the number of variables connected with a `*` instead of `:`, since you want to include all the lower-order effects of those three predictors as well.

In the results, the estimated main effect for `cap` is positive, showing that an increased power capacity is tied to an increased construction cost. All other main effects are negative, which at face value seems to imply that a reduced construction cost is associated with more experienced engineers, plants constructed in the Northeast, and plants with a cooling tower. However, this isn't an accurate statement since you haven't yet considered the interactive terms in those predictors. All estimated two-way interactive effects are positive: Having more experienced engineers means a higher construction cost in the Northeast regardless of whether there's a cooling tower, and having more experienced engineers also means higher costs for plants *with* a cooling tower, regardless of region.

Cost is also dramatically increased for plants in the Northeast with a cooling tower, regardless of the experience of the engineer. All that being said, the negative three-way interaction suggests that the increased cost associated with more experienced engineers working in the Northeast *and* on a plant with a cooling tower is lessened somewhat after the main effects and two-way interactive effects are calculated.

At the least, this example highlights the complexities associated with interpreting model coefficients for higher-order interactions. It's also possible that statistically significant high-order interactions crop up due to lurking variables that have gone unaccounted for—that is, that the significant interactions are a spurious manifestation of patterns in the data that simpler terms involving those missing predictors could explain just as well (if not better). In part, this motivates the importance of adequate model selection, which I discuss next.

Exercise 21.3

Return your attention to the `cats` data frame in package `MASS`. In the first few tasks of Exercise 21.1, you fitted the main-effect-only model to predict the heart weights of domestic cats by total body weight and sex.

a. Fit the model again, and this time include an interaction between the two predictors. Inspect the model summary. What do you notice in

terms of the parameter estimates and their significance when compared to the earlier main-effectonly version?

b. Produce a scatterplot of heart weight on body weight using different point characters or colors to distinguish the observations according to sex. Use `abline` to add two lines denoting the fitted model. How does this plot differ from the one in Exercise 21.1 (d)?

c. Predict the heart weight of Tilman's cat using the new model (remember that Sigma is a 3.3 kg female) accompanied by a 95 percent prediction interval. Compare it to the maineffects-only model from the earlier exercise.

In Exercise 21.2, you accessed the `trees` data frame in the contributed `faraway` package. After loading the package, access the `?trees` help file; you'll find the volume and girth measurements you used earlier, as well as data on the height of each tree.

d. Without using any transformations of the data, fit and inspect a main-effects-only model for predicting tree volume from girth and height. Then, fit and inspect a second version of this model including an interaction.

e. Repeat (d), but this time use the log transformation of all variables. What do you notice about the significance of the interaction between the untransformed and transformed models? What does this suggest about the relationships in the data?

Turn back to the `mtcars` dataset and remind yourself of the variables present by accessing the documentation with `?mtcars`.

f. Fit a linear model for `mpg` based on a two-way interaction between `hp` and `factor(cyl)` and their main effects, as well as a main effect for `wt`. Produce a summary of the fit.

g. Interpret the estimated coefficients for the interaction between horsepower and the (categorical) number of cylinders.

h. Suppose you're keen on purchasing a 1970s performance car. Your mother advises you to purchase a "practical and economical" car that's capable of an average MPG value of at least 25. You see three vehicles advertised: Car 1 is a fourcylinder, 100 horsepower car that weighs 2,100 lbs; car 2 is an eight-cylinder, 210 horsepower car that weighs

3,900 lbs; and car 3 is a six-cylinder, 200 horsepower car that weighs 2,900 lbs.

i. Use your model fitted in part (f) to predict the mean MPG for each of the three cars; provide 95 percent confidence intervals. Based on your point estimates only, which car would you propose to your mother?

ii. You still want the most gas-guzzling car you can own with your mother's blessing, so you decide to be sneaky and base your decision on what the confidence intervals tell you instead. Could this change your choice of vehicle?

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`I`	Include formula arithmetic	Section 21.4.1, p. 575
`:`	Interaction term (formula)	Section 21.5.2, p. 586
`*`	Cross-factor operator (formula)	Section 21.5.3, p. 590

22
LINEAR MODEL SELECTION AND DIAGNOSTICS



You've now spent a fair amount of time on many aspects of linear regression models. In this chapter, you'll use formal R tools and statistical techniques to investigate two other (equally important) aspects of regression: choosing an appropriate model for your analysis and assessing the validity of the assumptions that underpin the fit.

22.1 Goodness-of-Fit vs. Complexity

The overarching goal of fitting any statistical model is to faithfully represent the data and the relationships held within them. In general, fitting statistical models boils down to a balancing act between two things: goodness-of-fit and complexity. *Goodness-of-fit* refers to the goal of obtaining a model that best represents the relationships between the response and the predictor (or predictors). *Complexity* describes how complicated a model is; this is always tied to the number of terms in the model that require estimation: The inclusion of more predictors and

additional functions (such as polynomial transformations and interactions) leads to a more complex model.

22.1.1 Principle of Parsimony

Statisticians refer to the balancing act between goodness-of-fit and complexity as the *principle of parsimony*, where the goal of the associated *model selection* is to find a model that's as simple as possible (in other words, with relatively low complexity) without sacrificing too much goodness-of-fit. We'd say that a model that satisfies this notion is a *parsimonious* fit. When researchers talk about choosing the "best" model, they're referring to parsimony.

So, how do you decide where to draw the line on such a balance? Naturally, statistical significance plays a role here, and arriving at your final model often comes down to simply assessing the significance of the effect of predictors or functions of predictors on the response. To impart some degree of objectivity to such a process, you might opt to use specific model selection algorithms, such as those you'll learn about in Section 22.2, to decide upon inclusion or exclusion of multiple explanatory variables and any associated functionals thereof.

22.1.2 General Guidelines

Performing any kind of model selection or comparing several models against one another involves making decisions regarding the inclusion of available predictor variables. On this topic, there are several guidelines you must follow:

- First, it's important to remember that you can't remove individual levels of a categorical predictor in a given model; doing so makes no sense. In other words, if one of the non-reference levels is statistically significant but all others are non-significant, you should treat the categorical variable, as a whole, as making a statistically significant contribution to the determination of the mean response. You should consider *entire* removal of that categorical predictor only if *all* non-reference coefficients are

associated with a lack of evidence (against being zero). This also holds for interactive terms involving categorical predictors.

- If an interaction is present in the fitted model, all lower-order interactions and main effects of the relevant predictors must remain in the model. I touched on this in Section 21.5.1, when discussing the interpretation of interactive effects as augmentations of lower-order effects. As an example, you should consider removing the main effect of a predictor only if there are no interaction terms present in the fitted model involving that predictor (even if that main effect has a high *p*-value).
- In models where you've used a polynomial transformation of a certain explanatory variable (refer to Section 21.4.1), keep all lowerorder polynomial terms in the model if the highest is deemed significant. A model containing an order 3 polynomial transformation in a predictor, for example, must also include the order 1 and order 2 transformations of that variable. This is because of the mathematical behavior of polynomial functions: Only by explicitly separating out the linear, quadratic, and cubic (and so on) effects as distinct terms can you avoid confounding said effects with one another.

22.2 Model Selection Algorithms

A model selection algorithm is tasked with sifting through your available explanatory variables in some systematic fashion in order to establish which are best able to jointly describe the response, as opposed to fitting models by examining specific combinations of predictors in isolation, as you've done so far.

Model selection algorithms can be controversial. There are many techniques, and no single approach is universally appropriate. Different selection algorithms can result in different final models, as you'll see. In many cases, researchers will have additional information or knowledge about the problem that influences the decision—for example, that

certain predictors must always be included or that it makes no sense to ever include them. This must be considered at the same time as other complications, such as the possibility of interactions or unobserved lurking variables influencing significant relationships, as well as the need to ensure any fitted model is statistically valid (which you'll look at in Section 22.3).

It's helpful to keep in mind this famous quote from celebrated statistician George Box (1919-2013): "All models are wrong, but some are useful." In other words, any fitted model you produce can never be assumed to be the truth, but a model that's fitted and checked carefully and thoroughly can reveal interesting features of the data and thus have the potential to reveal associations and relationships previously unknown.

22.2.1 Nested Comparisons: The Partial F-Test

The *partial F-test* is probably the most direct way to compare several different models. It looks at two or more *nested* models, where the smaller, less complex model is a reduced version of the bigger, more complex one. Formally, let's say you've fitted two linear regression models as follows:

$$\hat{y}_{\text{redu}} = \widehat{\beta}_0 + \widehat{\beta}_1 x_1 + \widehat{\beta}_2 x_2 + \ldots + \widehat{\beta}_p x_p$$
$$\hat{y}_{\text{full}} = \widehat{\beta}_0 + \widehat{\beta}_1 x_1 + \widehat{\beta}_2 x_2 + \ldots + \widehat{\beta}_p x_p + \ldots + \widehat{\beta}_q x_q$$

Here, the reduced model, predicting $\hat{y}_{\text{redu}}$, has p predictors, plus one intercept. The full model, predicting $\hat{y}_{\text{full}}$, has q predictor terms. The notation implies that $q > p$ and that, along with the standard inclusion of an intercept $\widehat{\beta}_0$, the full model involves all p predictors of the reduced model defined by $\hat{y}_{\text{redu}}$, as well as $q - p$ additional terms. This emphasizes the fact that the model for $\hat{y}_{\text{redu}}$ is "nested within" $\hat{y}_{\text{full}}$.

It's important to note that increasing the number of predictors in a regression model will always improve R^2 (the coefficient of

determination; see Section 20.3.3) and any other measures of goodness-of-fit. The real question, however, is whether that improvement in goodness-of-fit is large enough to make the additional complexity involved with including any additional predictor terms "worth it." This is precisely what the partial F-test tries to answer in the context of nested regression models. Its goal is to test whether including those extra $q - p$ terms, which give the full model as opposed to the reduced model, provides a statistically significant improvement in goodness-of-fit. Formalizing this sentiment, the partial F-test addresses these hypotheses:

$$\begin{aligned} H_0 &: \beta_{p+1} = \beta_{p+2} = \ldots = \beta_q = 0 \\ H_A &: \text{At least one of the } \beta_j \neq 0 \text{ (for } j = p+1, \ldots, q) \end{aligned} \quad (22.1)$$

The calculation of the test statistic to address these hypotheses follows the same ideas behind the omnibus F-test automatically produced by R when summarizing a fitted linear model object (detailed in Section 21.3.5). Denote the coefficient of determination for the reduced and full models with R^2_{redu} and R^2_{full}, respectively. If n refers to the sample size of the data used to fit both models, the test statistic given by

$$F = \frac{\left(R^2_{\text{full}} - R^2_{\text{redu}}\right)(n-q-1)}{\left(1-R^2_{\text{full}}\right)(q-p)} \quad (22.2)$$

follows an F distribution (Section 16.2.6) with $\text{df}_1 = q - p$, $\text{df}_2 = n - q$ degrees of freedom under the assumption of H_0 in Equation (22.1). The p-value is found as the upper-tail area from $\mathscr{F}$ as usual; the smaller it is, the greater the evidence against the null hypothesis, which states that one or more of the additional parameters has no impact on the response variable.

Take the model objects `survmult` and `survmult2` from Section 21.3.1 (pages 561–562) as an example. The `survmult` model aims to predict mean student height from writing handspan and sex based on the `survey` data

frame from the MASS package; survmult2 adds smoking status to these predictors. If you need to, return to Section 21.3.1 to refit these two models. Printing the objects to the console previews the two fits and makes it easy to confirm that the smaller model is indeed nested within the larger model in terms of its explanatory variables:

```
R> survmult

Call:
lm(formula = Height ~ Wr.Hnd + Sex, data = survey)

Coefficients:
(Intercept)       Wr.Hnd      SexMale
    137.687        1.594        9.490

R> survmult2

Call:
lm(formula = Height ~ Wr.Hnd + Sex + Smoke, data = survey)

Coefficients:
(Intercept)       Wr.Hnd      SexMale   SmokeNever   SmokeOccas   SmokeRegul
   137.4056       1.6042       9.3979      -0.0442       1.5267       0.9211
```

Once you've fitted your nested models, R can carry out partial *F*-tests using the anova function (partial *F*-tests fall within the suite of analysis of variance—ANOVA—methodologies). To determine whether adding Smoke as a predictor provides any statistically significant improvement in fit, simply start with the reduced model object and follow it with the more complex model object supplied as arguments:

```
R> anova(survmult,survmult2)
Analysis of Variance Table

Model 1: Height ~ Wr.Hnd + Sex
Model 2: Height ~ Wr.Hnd + Sex + Smoke
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
```

```
1    204 9959.2
2    201 9914.3  3    44.876 0.3033  0.823
```

The output provides the quantities associated with calculation of R^2_{redu} and R^2_{full} and gives the test statistic $\mathscr{F}$ from Equation (22.2) in the column labeled `F`. Using the values of p and q from printing `survmult` and `survmult2`, you should be able to confirm, for example, the values of df_1 and df_2 appearing in the second row of the table in the columns `Df` and `Res.Df`, respectively. The result of this particular test, obtained from a test statistic of $\mathscr{F} = 0.3033$ associated with $\text{df}_1 = 3$, $\text{df}_2 = 201$, is a high p-value of 0.823, suggesting no evidence against H_0. This means that adding `Smoke` to the reduced model, which already includes the explanatory variables `Wr.Hnd` and `Sex`, offers no tangible improvement in fit when it comes to modeling student height. That conclusion isn't surprising, given the non-significant p-values of all non-reference levels of `Smoke`, seen previously in the summarized output of `survmul2` on page 562. In turn, this reveals how partial F-tests are used for model selection; in the current example, the reduced model would be the more parsimonious fit and preferred over the full model.

You can conduct comparisons among several nested models in a given call to `anova`, which can be useful for investigating things such as including interactive terms or polynomial transformations of predictors, since there's a natural hierarchy that requires you to retain any lower-order terms. For an example, let's use the `diabetes` data frame in the `faraway` package from Section 21.5.2 (page 586), with the model fitted to predict cholesterol level (`chol`) against age (`age`) and body frame (`frame`) and the interaction between those two predictors. Before using the partial F-tests to compare nested variants, ensure you're using the same records for each model and that there aren't missing values for any of the predictors that are subsequently going to be unavailable to the "fuller" models (so the sample size is the same for each comparison). To do this, you first need to define a version of `diabetes` that removes records with missing values in all the predictor variables you're considering.

Load the `faraway` package and use logical subsetting (see Section 5.2.4) to identify and delete any individuals with a missing value for `age` *or* for `frame`. Define this new version of the `diabetes` object:

```
R> diab <- diabetes[-which(is.na(diabetes$age) | is.na(diabetes$frame)),]
```

Now fit the following four models using your new `diab` object:

```
R> dia.model1 <- lm(chol~1,data=diab)
R> dia.model2 <- lm(chol~age,data=diab)
R> dia.model3 <- lm(chol~age+frame,data=diab)
R> dia.model4 <- lm(chol~age*frame,data=diab)
```

The first model is just an intercept, the second adds `age` as a predictor, the third has `age` and `frame`, and the fourth includes the interaction. Nesting is evident, and you can now compare the significance of the improvements in goodness-of-fit as you increase the complexity of the model at each step:

```
R> anova(dia.model1,dia.model2,dia.model3,dia.model4)
Analysis of Variance Table

Model 1: chol ~ 1
Model 2: chol ~ age
Model 3: chol ~ age + frame
Model 4: chol ~ age * frame
  Res.Df    RSS Df Sum of Sq       F    Pr(>F)
1    389 747265
2    388 712078  1     35187 19.6306 1.227e-05 ***
3    386 697527  2     14551  4.0589   0.01801 *
4    384 688295  2      9233  2.5755   0.07743 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If you hadn't deleted the records containing missing values in those predictors, you would've received an error telling you that the datasets for the four models were not equal sizes.

The results suggest that using `age` provides a significant improvement to modeling `chol` (as opposed to having no predictors at all); additionally including a main effect for `frame` provides a further mild improvement; and there's very weak evidence, if any, that subsequently including an interaction effect is beneficial to goodness-of-fit. All in all, it seems sensible to settle on `dia.mod3`, the main-effects-only model, as the most parsimonious model for mean cholesterol out of these four options.

22.2.2 Forward Selection

Partial *F*-tests are a natural way to investigate nested models but can be difficult to manage if you have many different possible models to try out. This can be the case, for example, by simply having a large number of predictor variables available in your dataset.

This is where *forward selection* (also referred to as *forward elimination*) comes in. The idea is to start with an intercept-only model and then perform a series of independent tests to determine which of your predictor variables "most significantly" improves the goodness-of-fit. Then you update your model object by adding that predictor in and execute the series of tests again for all remaining terms to determine which of those would *further* significantly improve the fit. The process repeats until there aren't any more terms that improve the fit in a statistically significant way. The ready-to-use R functions `add1` and `update` permit you to perform such a series of tests and update your fitted regression model.

You'll use the `nuclear` data frame in the `boot` library (seen in Exercise 21.1 on page 569 and Section 21.5.5 on page 594) as an example. The goal is to choose the most informative model for prediction of construction cost. Load `boot` and access the help file `?nuclear` to remember the variable definitions.

First, fit the model for construction cost with no predictors in it—that is, with a constant intercept term only:

```
R> nuc.0 <- lm(cost~1,data=nuclear)
R> summary(nuc.0)

Call:
lm(formula = cost ~ 1, data = nuclear)

Residuals:
    Min      1Q  Median      3Q     Max
-254.05 -151.24  -13.46  150.40  419.68

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   461.56      30.07   15.35 4.95e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 170.1 on 31 degrees of freedom
```

From earlier exploits we know that we can do better than this particular model when it comes to the reliable prediction of `cost`. So, consider the following line of code to start the forward selection (I've suppressed the output, which I'll show separately and discuss in a moment):

```
R> add1(nuc.0,scope=.~.+date+t1+t2+cap+pr+ne+ct+bw+cum.n+pt,test="F")
```

The first argument to `add1` is always the model you're aiming to update. The second argument, `scope`, is critical: You must supply a formula object defining the "fullest," most complex model you'd consider fitting. For this you would typically use the .~. notation, in which the dots refer to the definition of the model in the first argument. Specifically, the dots stand for "what is already there." In other words, through `scope` you're telling `add1` that the fullest model you'd consider has `cost` as the response, an intercept, and main effects of all other predictors in the `nuclear` data frame (I'll restrict the full model to main effects only for ease of demonstration). You don't need to supply the data frame as an argument since those data are contained within the

model object in the first argument. Lastly, you tell `add1` the test to perform. There are a handful of variants available (see the documentation by executing `?add1`), but here you'll stick with `test="F"` for partial *F*-tests.

Now focus on the output that's provided directly below the call to `add1`:

```
Single term additions

Model:
cost ~ 1
       Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>              897172 329.72
date    1    334335 562837 316.80 17.8205 0.0002071 ***
t1      1    186984 710189 324.24  7.8986 0.0086296 **
t2      1        27 897145 331.72  0.0009 0.9760597
cap     1    199673 697499 323.66  8.5881 0.0064137 **
pr      1      9037 888136 331.40  0.3052 0.5847053
ne      1    128641 768531 326.77  5.0216 0.0325885 *
ct      1     43042 854130 330.15  1.5118 0.2284221
bw      1     16205 880967 331.14  0.5519 0.4633402
cum.n   1     67938 829234 329.20  2.4579 0.1274266
pt      1    305334 591839 318.41 15.4772 0.0004575 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output comprises a series of rows, starting with `<none>` (doing nothing to the current model). You receive the `Sum of Sq` and `RSS` values, directly related to calculating the test statistic. The differences in degrees of freedom are also reported. Another measure of parsimony, `AIC`, is also provided (you'll look at that in more detail in Section 22.2.4).

Most relevant are the test outcomes; with `test="F"`, each row corresponds to an independent partial *F*-test comparing the model in the first argument (as $\hat{y}_{\text{redu}}$ if we use the notation from the start of Section 22.2.1), with the model that results from having added that row term only (as $\hat{y}_{\text{full}}$). Usually, therefore, you would update your model by

adding only the term with the largest (and "most significant") improvement. Here, you should be able to see that adding `date` as a predictor offers the largest significant improvement to modeling `cost` (if we go by the addition associated with the smallest *p*-value). So, let's update `nuc.0` to include that term with the code:

```
R> nuc.1 <- update(nuc.0,formula=.~.+date)
R> summary(nuc.1)

Call:
lm(formula = cost ~ date, data = nuclear)

Residuals:
    Min      1Q  Median      3Q     Max
-176.00 -105.27  -25.24   58.63  359.46

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -6553.57    1661.96  -3.943 0.000446 ***
date          102.29      24.23   4.221 0.000207 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 137 on 30 degrees of freedom
Multiple R-squared:  0.3727,    Adjusted R-squared:  0.3517
F-statistic: 17.82 on 1 and 30 DF,  p-value: 0.0002071
```

In `update`, you provide the model you want to update as the first argument, and the second argument, `formula`, tells `update` how to update the model. Again using the .~. notation, the instruction is to update `nuc.0` by adding `date` as a predictor, resulting in a fitted model object of the same class of the first argument. Call a `summary` of the new model, `nuc.1`, to see this. Let's keep going! Call `add1` again, but now use `nuc.1` as your first argument:

```
R> add1(nuc.1,scope=.~.+date+t1+t2+cap+pr+ne+ct+bw+cum.n+pt,test="F")
Single term additions
```

```
Model:
cost ~ date
       Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>              562837 316.80
t1      1     15322 547515 317.92  0.8115 0.3750843
t2      1     68161 494676 314.67  3.9959 0.0550606 .
cap     1    189732 373105 305.64 14.7471 0.0006163 ***
pr      1      4027 558810 318.57  0.2090 0.6509638
ne      1     92256 470581 313.07  5.6854 0.0238671 *
ct      1     54794 508043 315.52  3.1277 0.0874906 .
bw      1      1240 561597 318.73  0.0640 0.8020147
cum.n   1      4658 558179 318.53  0.2420 0.6264574
pt      1     90587 472250 313.18  5.5628 0.0252997 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that now there's no row for adding in `date`; it's already there in `nuc.1`. It seems the next most informative addition would be `cap`. Update `nuc.1` to that effect:

```
R> nuc.2 <- update(nuc.1,formula=.~.+cap)
```

Continue testing and updating in the same manner. By calling `add1` on `nuc.2` (output not shown here), you'll find that the next most significant addition is `pt` (by a small margin). Update to a new object named `nuc.3`, which includes the following term:

```
R> nuc.3 <- update(nuc.2,formula=.~.+pt)
```

Then test again, using `add1` on `nuc.3`. You'll find weak evidence to additionally include a main effect for `ne`, so update with that inclusion to create `nuc.4`:

```
R> nuc.4 <- update(nuc.3,formula=.~.+ne)
```

At this point, you may be reasonably certain there won't be any more useful additions, but check with one final call to `add1` on the latest fit to be thorough:

```
R> add1(nuc.4,scope=.~.+date+t1+t2+cap+pr+ne+ct+bw+cum.n+pt,test="F")
Single term additions

Model:
cost ~ date + cap + pt + ne
       Df Sum of Sq    RSS    AIC F value Pr(>F)
<none>              222617 293.12
t1      1     107.0 222510 295.10  0.0125 0.9118
t2      1   19229.9 203387 292.23  2.4583 0.1290
pr      1    5230.8 217386 294.36  0.6256 0.4361
ct      1   15764.7 206852 292.77  1.9815 0.1711
bw      1     448.0 222169 295.06  0.0524 0.8207
cum.n   1   13819.9 208797 293.07  1.7209 0.2010
```

Indeed it appears that none of the remaining predictors, if included in the model, would yield a statistically significant improvement to goodness-offit. Your final model will stay at `nuc.4`, summarized here:

```
R> summary(nuc.4)

Call:
lm(formula = cost ~ date + cap + pt + ne, data = nuclear)

Residuals:
     Min       1Q   Median       3Q      Max
-157.894  -38.424   -2.493   35.363  267.445

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.756e+03  1.286e+03  -3.699 0.000975 ***
date         7.102e+01  1.867e+01   3.804 0.000741 ***
cap          4.198e-01  8.616e-02   4.873 4.29e-05 ***
pt          -1.289e+02  4.950e+01  -2.605 0.014761 *
ne           9.940e+01  3.864e+01   2.573 0.015908 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 90.8 on 27 degrees of freedom
```

```
Multiple R-squared:  0.7519,    Adjusted R-squared:  0.7151
F-statistic: 20.45 on 4 and 27 DF,  p-value: 7.507e-08
```

Forward selection may seem a little cumbersome, and it's sometimes difficult to decide on the "fullest" model to be used as the `scope`, but it's a remarkably good way to remain involved at every stage of the selection process so you can consider each addition carefully. Note, however, that there's an element of subjectivity; it's possible to arrive at different final models by choosing one addition over another, such as if you'd added `pt` instead of `date` (they had similar levels of significance in the very first call to `add1`).

22.2.3 Backward Selection

After learning about forward selection, understanding *backward selection* (or *elimination*) isn't much of a stretch. As the name suggests, where forward selection starts from a reduced model and works its way up to a final model by adding terms, backward selection starts with your most complex model and systematically drops terms. The R functions for this process are `drop1` (to inspect the partial F-tests) and `update`.

The choice of forward versus backward model selection is usually made on a case-by-case basis. If your most complex model is difficult to define and fit, then forward selection is typically preferred. On the other hand, if you do have a natural and easily fitted fullest model, then backward selection can be more convenient to implement. Sometimes, researchers will perform both to see whether the final model they arrive at is different (a perfectly possible occurrence).

Revisit the `nuclear` data example. First, define the fullest model as that which predicts `cost` by main effects of all available covariates (as you did in your use of `scope` when performing forward selection):

```
R> nuc.0 <- lm(cost~date+t1+t2+cap+pr+ne+ct+bw+cum.n+pt,data=nuclear)
R> summary(nuc.0)

Call:
lm(formula = cost ~ date + t1 + t2 + cap + pr + ne + ct + bw +
```

```
    cum.n + pt, data = nuclear)

Residuals:
     Min       1Q   Median       3Q      Max
-128.608  -46.736   -2.668   39.782  180.365

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -8.135e+03  2.788e+03  -2.918 0.008222 **
date         1.155e+02  4.226e+01   2.733 0.012470 *
t1           5.928e+00  1.089e+01   0.545 0.591803
t2           4.571e+00  2.243e+00   2.038 0.054390 .
cap          4.217e-01  8.844e-02   4.768 0.000104 ***
pr          -8.112e+01  4.077e+01  -1.990 0.059794 .
ne           1.375e+02  3.869e+01   3.553 0.001883 **
ct           4.327e+01  3.431e+01   1.261 0.221008
bw          -8.238e+00  5.188e+01  -0.159 0.875354
cum.n       -6.989e+00  3.822e+00  -1.829 0.081698 .
pt          -1.925e+01  6.367e+01  -0.302 0.765401
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 82.83 on 21 degrees of freedom
Multiple R-squared:  0.8394,    Adjusted R-squared:  0.763
F-statistic: 10.98 on 10 and 21 DF,  p-value: 2.844e-06
```

There are clearly several predictors that appear not to contribute significantly to the response, and these same results are evident the first time you use `drop1` to examine the impact on goodness-of-fit that would occur from dropping each variable in isolation:

```
R> drop1(nuc.0,test="F")
Single term deletions

Model:
cost ~ date + t1 + t2 + cap + pr + ne + ct + bw + cum.n + pt
       Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>              144065 291.19
date    1     51230 195295 298.93  7.4677 0.0124702 *
```

```
t1      1      2034 146099 289.64  0.2965 0.5918028
t2      1     28481 172546 294.97  4.1517 0.0543902 .
cap     1    155943 300008 312.67 22.7314 0.0001039 ***
pr      1     27161 171226 294.72  3.9592 0.0597943 .
ne      1     86581 230646 304.25 12.6207 0.0018835 **
ct      1     10915 154980 291.53  1.5911 0.2210075
bw      1       173 144238 289.23  0.0252 0.8753538
cum.n   1     22939 167004 293.92  3.3438 0.0816977 .
pt      1       627 144692 289.33  0.0914 0.7654015
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

One handy feature of `drop1` (and hence backward selection in R) is that its `scope` argument is optional. If you don't include `scope`, it defaults to the intercept-only model as the "simplest" model, which is usually a reasonable choice.

Before diving right into the deletion process, remind yourself of the interpretation of what you're doing. Just as adding any term will always improve the goodness-of-fit in forward selection, deleting any term in backward selection will always worsen the goodness-of-fit. The real question is the perceived significance of these changes. In the same way as earlier, where you wanted to add only those terms that offer a *statistically significant improvement* to goodness-of-fit, when dropping terms, you only want to remove those that *do not* result in a statistically significant *detriment* to goodness-of-fit. As such, backward selection is the complete reverse of forward selection in the way it's carried out, but with the same final goal.

So, from the output of `drop1`, you want to choose the term to remove from the model that has the "least significant" effect of reducing the goodness of the fit. One way to action this is to look for the term with the largest non-significant p-value for its partial F-test, because dropping a term with a small p-value would mean you'd be significantly worsening the predictive capability of the regression model.

In the current example, it seems the predictor `bw` has the single least significant effect on reducing the goodness-of-fit, so let's start the

update by removing that term from `nuc.0`:

```
R> nuc.1 <- update(nuc.0,.~.-bw)
```

Use of `update` in this selection algorithm is the same as before; now, though, you use a `-` to signify the deletion of a term following the standard "what's already there" `.~.` notation.

Repeat the process using the latest model, `nuc.1`:

```
R> drop1(nuc.1,test="F")
Single term deletions

Model:
cost ~ date + t1 + t2 + cap + pr + ne + ct + cum.n + pt
       Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>              144238 289.23
date    1     55942 200180 297.72  8.5326  0.007913 **
t1      1      3124 147362 287.92  0.4765  0.497245
t2      1     30717 174955 293.41  4.6852  0.041546 *
cap     1    159976 304214 311.11 24.4005 6.098e-05 ***
pr      1     27140 171377 292.75  4.1395  0.054122 .
ne      1     86408 230646 302.25 13.1795  0.001479 **
ct      1     11815 156053 289.75  1.8021  0.193153
cum.n   1     24048 168286 292.17  3.6680  0.068557 .
pt      1       930 145168 287.44  0.1419  0.710039

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It would seem that `pt` is the next most sensible main effect to drop. Do so and name the resulting object `nuc.2`:

```
R> nuc.2 <- update(nuc.1,.~.-pt)
```

Keep going, checking `nuc.2` with a call to `drop1` (not shown), and you'll find that the predictor `t1` reveals itself as another viable deletion. Update your model with that predictor deleted; name the model object `nuc.3`:

```
R> nuc.3 <- update(nuc.2,.~.-t1)
```

Recheck the new `nuc.3` with `drop1`. You should now find the effect of `ct` remains non-significant, so delete that and update again, giving you a new model, `nuc.4`:

```
R> nuc.4 <- update(nuc.3,.~.-ct)
```

Perform yet another check with `drop1`, this time on `nuc.4`. At this point, you might hesitate to remove any more predictors, with significance at varying strengths being associated with the effect of their deletion. Note, however, that for at least three of the remaining predictors (`t2`, `pr`, and `cum.n`), you should probably consider the statistical significance borderline at best: All of their *p*-values lie between the conventional cutoff levels of $\alpha = 0.01$ and $\alpha = 0.05$. This again emphasizes the active role a researcher must play in model selection algorithms such as forward or backward elimination; whether you should delete any more variables from here is a difficult question to answer and is left up to your judgment.

Let's remain with `nuc.4` as the final model. Summarizing, you're able to see the estimated regression parameters and the usual post-fit statistics:

```
R> summary(nuc.4)

Call:
lm(formula = cost ~ date + t2 + cap + pr + ne + cum.n, data = nuclear)

Residuals:
     Min       1Q   Median       3Q      Max
-152.851  -53.929   -8.827   53.382  155.581

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -9.702e+03  1.294e+03  -7.495 7.55e-08 ***
date         1.396e+02  1.843e+01   7.574 6.27e-08 ***
t2           4.905e+00  1.827e+00   2.685 0.012685 *
```

```
cap          4.137e-01  8.425e-02   4.911 4.70e-05 ***
pr          -8.851e+01  3.479e+01  -2.544 0.017499 *
ne           1.502e+02  3.400e+01   4.419 0.000168 ***
cum.n       -7.919e+00  2.871e+00  -2.758 0.010703 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 80.8 on 25 degrees of freedom
Multiple R-squared:  0.8181,    Adjusted R-squared:  0.7744
F-statistic: 18.74 on 6 and 25 DF,  p-value: 3.796e-08
```

Immediately, you can see that your final model from forward selection in Section 22.2.2 is different from the final model selected here, despite the fullest model being the same for both. How has that occurred?

The answer, simply put, is that the predictors present in a model affect each other. Remember that the estimated coefficients of present predictors easily change in value as you control for different variables. As the number of predictor terms increases, these relationships become more and more complex, so both the order and direction of the selection algorithm have the potential to lead you on different paths through the selection process and arrive at different final destinations, which is exactly what's happened here.

As a perfect example of this, consider the main effect of `pt` in the `nuclear` data. In forward selection, `pt` was added because it offered the "most significant" improvement to the model `cost~date+cap`. In backward selection, `pt` was removed early, since it offered the least reduction in goodness-of-fit if taken from the model `cost~date+t1+t2+cap+pr+ne+ct+cum.n+pt`. This means that for the latter model, the contribution that `pt` might make in terms of predicting the outcome is already explained by some combination of the other present predictor terms. When performing forward selection, that effect had not yet been explained by a smaller model and `pt` was an attractive addition at that point.

All this serves to highlight the fickle nature of most selection algorithms, in spite of the systematic way they're implemented. It's

important to acknowledge that a final model fit may well vary between approaches; you should view these selection methods more as helpful guidelines for finding the most parsimonious model, as opposed to providing some universal, definitive solution. As always, contextual knowledge of the application at hand is a crucial component of modeling and inference.

22.2.4 Stepwise AIC Selection

The application of a series of partial F-tests is one of the most common *testbased* model selection methods, but it's not the only tool a researcher has at their disposal. You can also define parsimony according to a *criterion-* or *information-based* measure. One of the most famous techniques is known as *Akaike's "An Information Criterion" (AIC)*. You'll have noticed this value in the immediately preceding examples as one of the columns in the output of `add1` and `drop1`.

For a given linear model, AIC is calculated as follows:

$$\text{AIC} = -2 \times \mathscr{L} + 2 \times (p + 2) \quad (22.3)$$

Here, $\mathscr{L}$ is a measure of goodness-of-fit named the *log-likelihood*, and p is the number of regression parameters in the model, excluding the overall intercept. The value of $\mathscr{L}$ is a direct outcome of the estimation procedure used to fit the model, though its exact calculation is beyond the scope of this text. Just know that it takes on larger values for better-fitting models.

Equation (22.3) produces a value that rewards goodness-of-fit with $-2 \times \mathscr{L}$ but simultaneously penalizes complexity with $2 \times (p + 2)$. The negative sign associated with $\mathscr{L}$ coupled with the positive sign of $p + 2$ means the smaller the AIC, the more parsimonious the model.

To find the AIC for a fitted linear model, you use the `AIC` or `extractAIC` functions on the object resulting from `lm`; take a look at the help files of these functions to see the technical differences between the two. The value of $\mathscr{L}$ (and therefore also the AIC) has no direct interpretation and

is useful only when you compare it against the AIC of another model. You can base model selection on the AIC by identifying the fit with the lowest AIC value. This is the reason it's directly reported in the output of `add1` and `drop1`; you could decide on which term to add or drop based on the change that results in a shift to the smallest AIC, instead of focusing exclusively on the *significance* of the change via the *F*-test.

Let's go even further and combine the ideas of forward and backward selection. *Stepwise* model selection allows the option to either delete a present term *or* add a missing term and is typically implemented with respect to AIC. That is, a term is chosen to be added or deleted based on the one "move" out of all possible moves that yields the single biggest reduction in AIC. This affords you more flexibility in exploring candidate models on your way to the final model fit, determined as the model from which no further addition or deletion would result in a reduction to the AIC value.

It's possible to implement stepwise AIC selection yourself using either `add1` or `drop1` at each stage, but fortunately R provides the built-in `step` function to do it for you. Take the `mtcars` data first seen in Section 9.3.1 and used in several other examples in the past couple chapters. Let's try to obtain a parsimonious model for mean mileage that offers the opportunity to include every predictor that's available.

First, take a look at the documentation in `?mtcars` (and perhaps a scatterplot matrix of the data via `pairs`) to remind yourself of the variables and their format in the R data frame object. Then define the starting model (often called the *null* model) as the intercept-only model as follows:

```
R> car.null <- lm(mpg~1,data=mtcars)
```

Your starting model can be anything you like, provided it falls within the domain of the models described by your `scope` argument to be supplied to `step`. In this example, define `scope` as the fullest model to be considered; set this to be the overly complex model with a four-way interaction among `wt`, `hp`, `cyl`, and `disp` (and all relevant lower-order

interactions and main effects, via the cross-factor operator), as well as main effects for `am`, `gear`, `drat`, `vs`, `qsec`, and `carb`. The two multilevel categorical variables, `cyl` and `gear`, are explicitly converted to factors to avoid them being treated as numeric (refer to Section 20.5.4).

The potential for interactions in the final model will serve to highlight an especially important (and convenient) feature of `add1`, `drop1`, and `step`. These functions all respect the hierarchy imposed by interactions and main effects. That is, for `add1` (and `step`), an interactive term will not be provided as an option for addition unless all relevant lower-order effects are already present in the current fitted model; similarly, for `drop1` (and `step`), an interactive term or main effect will not be provided as an option for deletion unless all relevant higher-order effects are already gone from the current fitted model.

The `step` function itself returns a fitted model object and by default provides a comprehensive report of each stage of selection. Let's call it now; for print reasons, some of the output has been snipped out, so I encourage you to bring this up on your own machine:

```
R> car.step <- step(car.null,scope=.~.+wt*hp*factor(cyl)*disp+am
                                    +factor(gear)+drat+vs+qsec+carb)
Start:  AIC=115.94
mpg ~ 1

               Df Sum of Sq     RSS     AIC
+ wt            1    847.73  278.32  73.217
+ disp          1    808.89  317.16  77.397
+ factor(cyl)   2    824.78  301.26  77.752
+ hp            1    678.37  447.67  88.427
+ drat          1    522.48  603.57  97.988
+ vs            1    496.53  629.52  99.335
+ factor(gear)  2    483.24  642.80 102.003
+ am            1    405.15  720.90 103.672
+ carb          1    341.78  784.27 106.369
+ qsec          1    197.39  928.66 111.776
<none>                      1126.05 115.943
```

```
Step:  AIC=73.22
mpg ~ wt

               Df Sum of Sq     RSS     AIC
+ factor(cyl)   2     95.26  183.06  63.810
+ hp            1     83.27  195.05  63.840
+ qsec          1     82.86  195.46  63.908
+ vs            1     54.23  224.09  68.283
+ carb          1     44.60  233.72  69.628
+ disp          1     31.64  246.68  71.356
+ factor(gear)  2     40.37  237.95  72.202
<none>                      278.32  73.217
+ drat          1      9.08  269.24  74.156
+ am            1      0.00  278.32  75.217
- wt            1    847.73 1126.05 115.943

Step:  AIC=63.81
mpg ~ wt + factor(cyl)

                 Df Sum of Sq    RSS    AIC
+ hp              1    22.281 160.78 61.657
+ wt:factor(cyl)  2    27.170 155.89 62.669
<none>                        183.06 63.810
+ qsec            1    10.949 172.11 63.837
+ carb            1     9.244 173.81 64.152
+ vs              1     1.842 181.22 65.487
+ disp            1     0.110 182.95 65.791
+ am              1     0.090 182.97 65.794
+ drat            1     0.073 182.99 65.798
+ factor(gear)    2     6.682 176.38 66.620
- factor(cyl)     2    95.263 278.32 73.217
- wt              1   118.204 301.26 77.752

Step:  AIC=61.66
mpg ~ wt + factor(cyl) + hp

--snip--

Step:  AIC=55.9
```

```
mpg ~ wt + factor(cyl) + hp + wt:hp

--snip--

Step:  AIC=52.8
mpg ~ wt + hp + wt:hp

--snip--

Step:  AIC=52.57
mpg ~ wt + hp + qsec + wt:hp

               Df Sum of Sq    RSS    AIC
<none>                      121.04 52.573
- qsec          1     8.720 129.76 52.799
+ factor(gear)  2     9.482 111.56 53.962
+ am            1     1.939 119.10 54.056
+ carb          1     0.080 120.96 54.551
+ drat          1     0.012 121.03 54.570
+ vs            1     0.010 121.03 54.570
+ disp          1     0.008 121.03 54.571
+ factor(cyl)   2     0.164 120.88 56.529
- wt:hp         1    65.018 186.06 64.331
```

Each block of output displays the current model fit, its AIC value, and a table showing the possible moves (either adding `+`, deleting `-`, or doing nothing `<none>`). The AIC value that would result from each move alone is listed, and these potential single moves are ranked from smallest to largest AIC value.

Looking at the blocks of output from first to last, you see the `<none>` row creeping its way up the table as the algorithm proceeds. For example, in the first table, the value of the AIC for the intercept-only model is 115.94. The biggest reduction in AIC would result from adding a main effect for `wt`; that move is made, the model is fitted, and the effect of subsequent moves on the AIC is reassessed. Also note that the addition of the two-way interaction term between `wt` and `factor(cyl)` is considered only at the third step, after the main effects of those

predictors have been added. That particular two-way interaction never ends up being included, though, because the main effect of `hp` is preferable at that third step, and subsequent interactions involving `hp` then offer a much better reduction in the AIC value in the fourth step. In fact, at the fifth step, actually *deleting* the main effect for `factor(cyl)` is deemed to reduce the AIC most, and so the tables for the sixth and seventh steps no longer include that `wt:factor(cyl)` term as an option. The sixth step suggests that adding the main effect for `qsec` offers a minor reduction in the AIC, so this is done. The seventh table signals the end of the algorithm because "doing nothing" offers the lowest AIC value and doing anything else would increase the AIC (shown through `<none>` taking pole position in that last table).

The final model is stored as the object `car.step`; by summarizing it, you'll note that almost 90 percent of the variation in the response is explained by weight, horsepower, and their interaction, as well as the slightly curious main effect of `qsec` (which itself is not deemed statistically significant):

```
R> summary(car.step)

Call:
lm(formula = mpg ~ wt + hp + qsec + wt:hp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.8243 -1.3980  0.0303  1.1582  4.3650

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 40.310410   7.677887   5.250 1.56e-05 ***
wt          -8.681516   1.292525  -6.717 3.28e-07 ***
hp          -0.106181   0.026263  -4.043 0.000395 ***
qsec         0.503163   0.360768   1.395 0.174476
wt:hp        0.027791   0.007298   3.808 0.000733 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 2.117 on 27 degrees of freedom
Multiple R-squared:  0.8925,    Adjusted R-squared:  0.8766
F-statistic: 56.05 on 4 and 27 DF,  p-value: 1.094e-12
```

From this, it seems it would be worthwhile to investigate this predictor further, to establish validity of the fitted model (see Section 22.3), and perhaps to try transformations of the data (such as modeling GPM instead of MPG; see Section 21.4.3) to see whether this effect persists in subsequent runs of the stepwise AIC algorithm. The presence of `qsec` in the final model illustrates the fact that the selection of the model wasn't based solely on the significance of predictor contribution but on a criterion-based measure aiming for its own definition of parsimony.

The AIC is sometimes criticized for a tendency to err on the side of more complexity and higher p-values. To balance this, you can increase the penalizing effect of extra predictors by increasing the multiplicative contribution of $(p + 2)$ on the right of Equation (22.3), though the standard multiplicative factor of 2 is used in the majority of cases (in a call to `step`, you can use the optional argument `k` to change this). That being said, criterion-based measures are incredibly useful when you have models that aren't nested (hence ruling out the partial F-test) and you want to compare them for quick identification of the one that, potentially, provides the most parsimonious representation of the data.

Exercise 22.1

In Sections 22.2.2 and 22.2.3, you used forward and backward selection approaches to find a model for predicting the cost of the construction of nuclear power plants (based on the `nuclear` data frame in the `boot` package).

a. Using the same "fullest" model seen in these sections (in other words, main effects of all present predictors), use

stepwise AIC selection to find a suitable model for the data.

b. Does the final model found in (a) match either of the models resulting from the earlier use of forward and backward selection? How does it differ?

Exercise 21.2 on page 582 detailed Galileo's ball data. Enter these as a data frame in your current R workspace if you haven't already.

c. Fit five linear models to these data with distance as the response: an intercept-only model and four separate polynomial models of increasing order 1 to 4 in the height variable as the predictor.

d. Construct a table of partial *F*-tests to identify your favored model for distance traveled. Is your selection consistent with Exercise 21.2 (b) and (c)?

You first encountered the `diabetes` data frame in the contributed `faraway` package in Section 21.5.2, where you modeled mean overall cholesterol. Load the package and inspect the documentation in `?diabetes` to refresh your memory of the dataset.

e. There are some missing values in `diabetes` that might interfere with model selection algorithms. Define a new version of the `diabetes` data frame that deletes all rows with a missing value in any of the following variables: `chol`, `age`, `gender`, `height`, `weight`, `frame`, `waist`, `hip`, and `location`.
Hint: Use `na.omit` or your knowledge of record extraction or deletion for a data frame. You can create the required vector of row numbers to be extracted or deleted using `which` and `is.na` (see an example of this toward the end of Section 22.2.1), or you can try using the `complete.cases` function to obtain a logical flag vector; inspect its help file for details.

f. Use your data frame from (e) to fit two linear models with `chol` as the response. The null model object, named `dia.null`,

should be an intercept-only model. The full model object, named `dia.full`, should be the overly complex model with a four-way interaction (and all lower-order terms) among `age`, `gender`, `weight`, and `frame`; a three-way interaction (and all lower-order terms) among `waist`, `height`, and `hip`; and a main effect for `location`.

g. Starting from `dia.null` and using the same terms as in `dia.full` for `scope`, implement stepwise selection by AIC to choose a model for mean total cholesterol and then summarize.

h. Use forward selection based on partial F-tests with a conventional significance level of $\alpha = 0.05$ to choose a model, again starting from `dia.null`. Is the result here the same as the model arrived at in (g)?

i. Stepwise selection doesn't have to start from the simplest model. Repeat (g), but this time set `dia.full` to be the starting model (you don't need to supply anything to `scope` if you're starting from the most complex model). What is the final model selected via AIC if you start from `dia.full`? Is it different than the final model from (g)? Why do you think this is (or isn't) the case?

Revisit the ubiquitous, built-in `mtcars` data frame for the final task.

j. In Section 22.2.4, you used stepwise AIC selection to model mean MPG. The selected model included a main effect for `qsec` and an interaction term between two other predictors. Rerun the same AIC selection process, but this time, do it in terms of GPM=1/MPG (don't forget the special way in which we must designate arithmetic transformations of variables in R formulas; review Section 21.4.1). Does this change the complexity of the final model?

22.2.5 Other Selection Algorithms

Any model selection algorithm will always aim to quantitatively define parsimony and suggest a model that optimizes that definition in light of the available data. There are alternatives to AIC, such as the *corrected AIC (AIC_c)* or the *Bayesian Information Criterion (BIC)*, both of which impose heavier penalties on complexity than the default AIC in Equation (22.3).

Sometimes it's tempting to simply monitor R^2, the coefficient of determination, for a series of models. However, as mentioned in Section 22.2.1, this on its own is inadequate for choosing between models since it doesn't penalize complexity and will generally always increase as you continue to add predictors, regardless of whether they have a statistically significant impact. The *adjusted R^2 statistic*, denoted $\overline{R}^2$ and reported as `Adjusted R-squared` in the output of a call to `summary`, is a simple transformation of the original R^2 that does incorporate a penalty for complexity relative to the sample size n. It's calculated as

$$\overline{R}^2 = 1 - \frac{(1-R^2)(n-1)}{n-p-1}$$

where p is the number of predictor terms (excluding the intercept). The algorithms based on tests and information criteria are always preferable (since interpretation of $\overline{R}^2$ can be difficult), but monitoring $\overline{R}^2$ can be useful as a quick check between nested models: A higher value points to a preferred model.

For further reading, Chapter 10 of Faraway (2014) provides some excellent commentary on the guideline-only nature of both test- and criterionbased model selection procedures. Regardless of which approach you employ, always remember that any final model reached by using these algorithms should still be subject to scrutiny.

22.3 Residual Diagnostics

In previous chapters, you examined the practical aspects of multiple linear regression models, such as fitting and interpretation, dummy coding, transforming, and so on, but you haven't yet looked at methods that are essential for determining the *validity* of your model. The final part of this chapter will introduce you to *model diagnostics*, the primary goal of which is to ensure that your regression model is valid and accurately represents the relationships in your data. For this, I'll return focus to the theoretical assumptions underpinning the components of linear regression models (refer back to Sections 20.2.1 and 21.2.1).

When fitting linear regression models, it is important to be cognizant of the following:

Errors The error term ϵ, which defines the departure of any observation from the fitted mean-outcome model, is assumed to be normally distributed with a mean of zero and a constant variance denoted with σ^2. The error associated with a given observation is also assumed to be independent of the error of any other observation. If a fitted model suggests violation of any of these assumptions, you'll need to investigate further (which usually involves refitting a variation of the model).

Linearity It is critical to be able to assume that the mean response as a function is linear in terms of the regression parameters $\beta_0, \beta_1, \ldots, \beta_p$. Though transformations of individual variables and the presence of interactions can relax the specific nature of the estimated trends somewhat, any diagnostic suggestion that a relationship is nonlinear (and hence not being captured by the fitted model at hand) must be investigated.

Extreme or unusual observations Always inspect extreme data points or data points that strongly influence the fitted model—for example, points that have been recorded incorrectly should be removed from the analysis.

Collinearity Predictors highly correlated with one another can adversely affect an entire model, meaning it can be easy to

misinterpret the effects of any included predictors. Avoid this in any regression.

You can investigate aspects of the first three items on this list after fitting the model using special diagnostic plots and tools. This is important because any violation of these assumptions diminishes the reliability of your model in terms of inference or prediction, sometimes severely. Collinearity or extreme observations can be assessed by basic statistical explorations (for example, viewing scatterplot matrices) of the raw data pre-fit, but any consequential effects are appraised post-fit.

There are some statistical tests you can perform to diagnose a statistical model, but a diagnostic inspection commonly boils down to interpretation of the results of graphical tools designed to target specific assumptions. Interpreting these plots can be quite difficult; it becomes easier with experience. Here, I'll provide an overview of these tools in R and describe some common things to look for. For a more detailed discussion, look to dedicated texts on regression methods such as Chatterjee et al. (2000), Montgomery et al. (2012), or Faraway (2014).

22.3.1 Inspecting and Interpreting Residuals

The plot in Figure 20-2 on page 526 provides a good demonstration of the importance of interpreting the results given by $\hat{y}$ as a mean response value. Under the assumed model, any deviation of the raw observations from the fitted line is deemed to be the result of the (normally distributed) errors defined by the e term in Equation (20.1) on page 523.

Of course, in practice, you don't have the true error values because you don't know the true model of your data, so we estimate them using the fitted model itself. For the ith response observation, y_i, its fitted value is the value predicted by the model when evaluated at the predictor values of the ith individual or entity in the dataset; the fitted value is denoted $\hat{y}_i$. It follows that the estimated "raw" residual for the ith observation, $\hat{e}_i$, is given by the difference between the observed value and the predicted value of the response:

$$\hat{e}_i = y_i - \hat{y}_i$$

Indeed, any call to `summary` on an `lm` model object encourages a post-fit analysis of the estimated residuals by providing you with a five-number summary of the êi above the table of estimated coefficients. This allows you to take a look at their values and do a preliminary numeric assessment of the symmetry of their distribution (as required by the assumption of normality; see Section 22.3.2).

As well as a diagnostic inspection of the collection of raw residuals $\hat{e}_i$ *(*i = 1, . . . , *n)*, you also perform some diagnostic checks by using their *standardized* (or *Studentized* values). The standardized residuals, which I'll denote here by $\hat{e}_i^*$, represent a rescaled version of the raw residuals *ei* that adjusts for a potential non-constant variance, which is helpful if you need to directly compare them to one another. Formally, this is achieved with the calculation

$$\hat{e}_i^* = \hat{e}_i / \left(\widehat{\sigma} \{1 - h_{ii}\}^{0.5} \right)$$

where $\widehat{\sigma}$ is the estimate of the residual standard error and h_{ii} is the *leverage* of the *i*th observation (you'll learn about leverage in Section 22.3.4).

Arguably the most common graphical tool used for a post-fit analysis of the residuals is a simple scatterplot of the "observed-minus-fitted" raw residuals on the vertical axis against their corresponding fitted-model values from the regression—that is, plotting $\hat{e}_i$ against $\hat{y}_i$. If the assumptions concerning ϵ are valid, then the *êi* should appear randomly scattered around 0 (since the errors aren't assumed to be related in any way to the value of the response). Any systematic pattern in the plot suggests the residuals don't agree with the error assumptions; this could be because of nonlinear relationships in your data or the presence of dependent observations (in other words, your data points are correlated and therefore not independent of one another). You can also use such a plot to detect *heteroscedasticity*—a non-constant variance

in the residuals—commonly seen as a "fanning out" of the residuals around 0 as the fitted values increase.

Note once more that it's important these theoretical assumptions are valid because they affect the validity of the estimates of the regression coefficients and the reliability of their standard errors (and thus statistical significance)—in other words, the real-world "correctness" of your interpretation of their impact on the response.

To further connect these concepts, look at the images in Figure 22-1. These provide residuals-versus-fitted-values plots for three hypothetical scenarios.

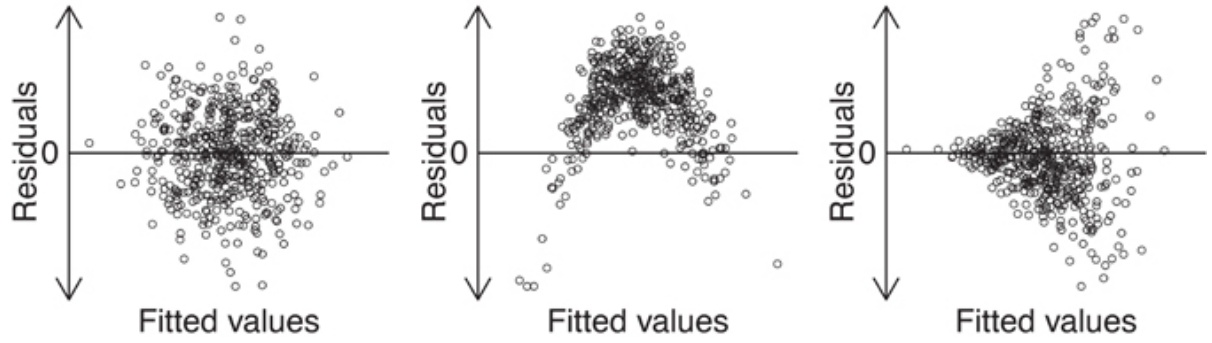


Figure 22-1: Three impressions of a hypothetical residuals-versus-fitted diagnostic plot from a linear regression: random (left), systematic (middle), and heteroscedastic (right)

A plot resembling something like that in the leftmost panel is, more or less, what you're looking for: The residuals appear randomly scattered around 0, and their spread around 0 appears consistent (*homoscedasticity*). In the middle plot, however, you can see systematic behavior in the residuals. Though the variability still seems to remain roughly constant throughout the range of fitted values, the appArent trend suggests the current model isn't explaining some of the relationship between response and predictor (or predictors). On the right, the residuals seem to be scattered randomly about 0 again. However, the variability they exhibit isn't constant. Among other things, this kind of heteroscedasticity will affect the reliability of your confidence and prediction intervals.

It's important to know that even if your graphical diagnostics don't provide a well-behaved plot like the hypothetical example on the left of Figure 22-1, this is not a reason to immediately give up on the analysis. These plots can form an integral part in finding an appropriate model for your data. You can often reduce non-linearity by including additional predictors or interactions, changing the treatment of a categorical variable, or performing nonlinear transformations of certain continuous predictors. Heteroscedasticity, especially the kind in Figure 22-1 where the variability is higher for higher fitted values, is common in some fields of research. A first step to remedy this problem often involves a simple transformation of the response followed by a reinspection of the diagnostics.

It's time for an example. In Section 22.2.4, you used stepwise AIC selection to choose a model for MPG for the `mtcars` data, creating the object `car.step`. Let's now diagnose that same fit to see whether there are any problems with the assumptions of the model (if you need to recreate the `car.step` object, rerun the `step` command as it appears in Section 22.2.4).

When you apply the `plot` function directly to an `lm` object, it can conveniently produce six types of diagnostic plot of the fit. By default, four of these plots are produced in succession. Follow the signal to the user in the console, `Hit <Return> to see next plot`, to progress through them. In the examples that follow, however, you'll select each plot individually using the optional `which` argument (specified by the integers `1` through `6`; see `?plot.lm` for the documentation).

The residuals-versus-fitted plot is given with `which=1`. The line

```
R> plot(car.step,which=1)
```

produces the plot on the left in Figure 22-2.

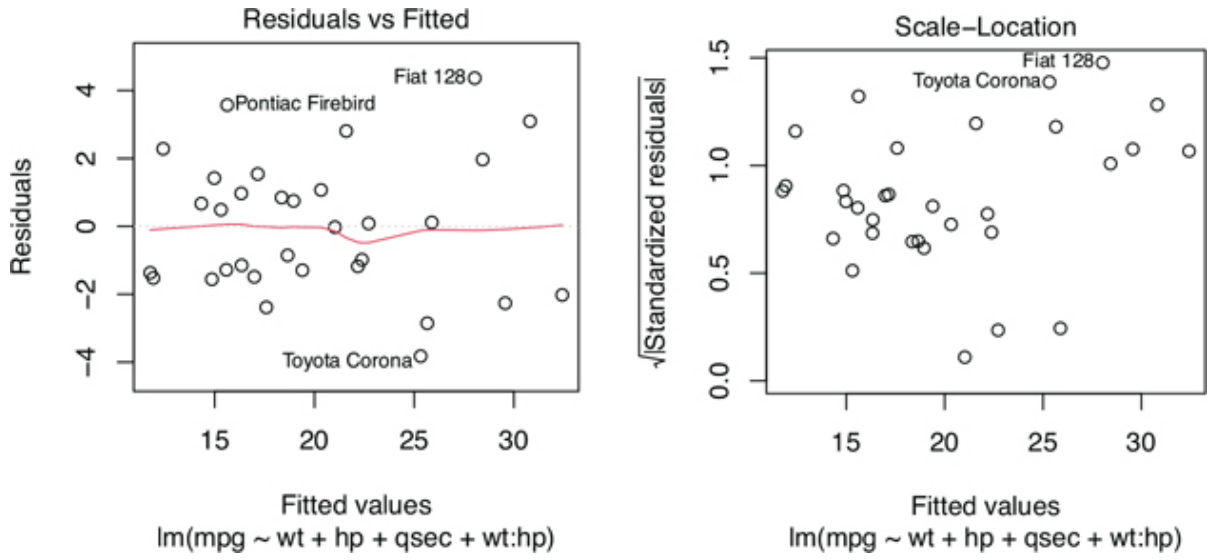


Figure 22-2: The residuals-versus-fitted and scale-location diagnostic plots for the `car.step` *model*

As you can see, R adds a smoothed line to help the user interpret any trend, though this shouldn't be used exclusively in any judgment. By default, the three most extreme points from 0 are annotated (according to the `rownames` attribute of the data frame used in the call that fitted the model). The model formula itself is specified below the horizontal axis label.

From this, you see that the residuals-versus-fitted plot for the `car.step` offers little, if any, cause for concern. There isn't much of a discernible trend, and you can take further comfort in the fact that the errors (e_i) appear homoscedastic in their distribution.

The *scale-location* plot is similar to the residuals-versus-fitted plot, though instead of the raw e_i on the vertical axis, the scale-location plot provides $\left|e_i / \left(\widehat{\sigma}\{1 - h_{ii}\}^{0.5}\right)\right|^{0.5}$, that is, the square root of the absolute value (denoted by $|\cdot|$, which renders all negative values positive) of the standardized residuals. These are plotted against the respective fitted values on the horizontal axis. By restricting attention to the magnitude of each residual in this way, the scale-location plot is used to reveal trends in the size of the departure of each data point from its fitted value, as the fitted values increase. This means such a plot can, for

example, be more useful than the raw residualsversus-fitted plot in detecting things such as heteroscedasticity. Just as with the original residuals-versus-fitted plot, you're looking for a plot with no discernible pattern as an indication that no error assumptions have been violated.

The right plot of Figure 22-2 shows the scale-location plot for `car.step`, selected with `which=3`. The code used to create this plot additionally demonstrates the ability to remove the default smoothed trend line with the `add.smooth` argument and to control how many extreme points are labeled using the `id.n` argument:

```
R> plot(car.step,which=3,add.smooth=FALSE,id.n=2)
```

As with the original residuals-versus-fitted plot, there doesn't seem to be much to be concerned about in the scale-location plot for this `mtcars` model.

Return to Galileo's ball-rolling data first laid out in Exercise 21.2 on page 582. In the following example, the response variable "distance traveled" is given as column `d`, and the explanatory variable "height" is column `h`, in the data frame `gal`. I'll recreate some of the exercise to give you a couple straightforward examples of cause for concern in residual diagnostic plots. Execute the following code to define the data frame of the seven observations and fit two regression models: the first linear in height and the second quadratic (refer to Section 21.4.1 for details on polynomial transformations):

```
R> gal <- data.frame(d=c(573,534,495,451,395,337,253),
                     h=c(1,0.8,0.6,0.45,0.3,0.2,0.1))
R> gal.mod1 <- lm(d~h,data=gal)
R> gal.mod2 <- lm(d~h+I(h^2),data=gal)
```

Now take a look at the three plots in Figure 22-3, created with the following code:

```
R> plot(gal$d~gal$h,xlab="Height",ylab="Distance")
R> abline(gal.mod1)
R> plot(gal.mod1,which=1,id.n=0)
R> plot(gal.mod2,which=1,id.n=0)
```

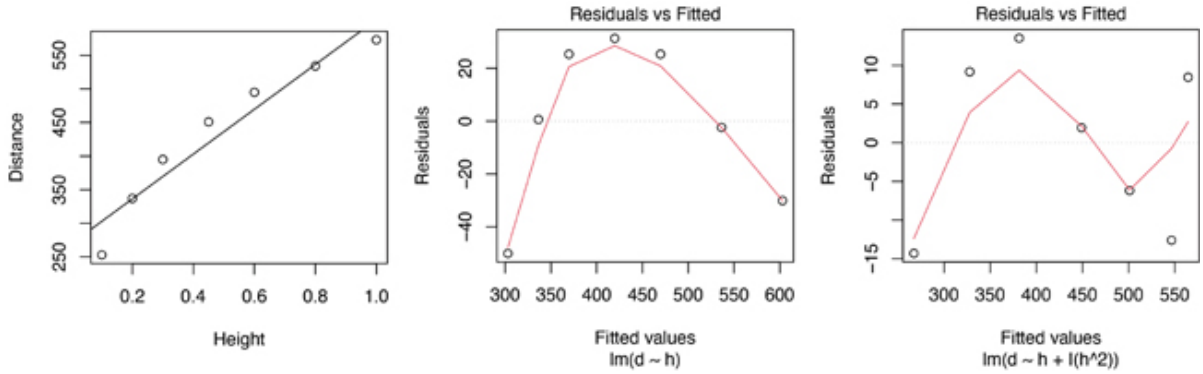


Figure 22-3: Demonstrating residual diagnostics for Galileo's ball-rolling data. Left: The raw data with a simple linear trend corresponding to `gal.mod1` *superimposed. Middle: Residuals-versus-fitted for the linear-trend-only model. Right: Residuals-versus-fitted for the quadratic model* `gal.mod2`*.*

The leftmost plot shows the data and provides the straight line of the simple linear model. Although this clearly captures the increasing trend, this plot suggests some curvature is also present. The diagnostic residuals-versusfitted plot (middle) shows that the linear-trend-only model is inadequate; the systematic pattern throws up a red flag concerning the assumptions surrounding the linear model errors. The rightmost image shows the residualsversus-fitted plot based on the quadratic version of the model in `gal.mod2`. Including a quadratic term in "height" removes this prominent curve in the residuals. However, these latest e_i values still seem to exhibit systematic behavior in a wavelike form, perhaps suggesting you try a cubic model (though we should interpret all of these plots with some caution, given the small sample size).

22.3.2 Assessing Normality

To assess the assumption that the error residuals are normally distributed, you can use a normal quantile-quantile (QQ) plot, as first discussed in Section 16.2.2. Select `which=2` when calling `plot` on an `lm` object to produce a normal QQ plot of the (standardized) residuals. Return to the `car.step` model object you were just investigating in Section 22.3.1 and enter the following line:

```
R> plot(car.step,which=2)
```

This produces Figure 22-4.

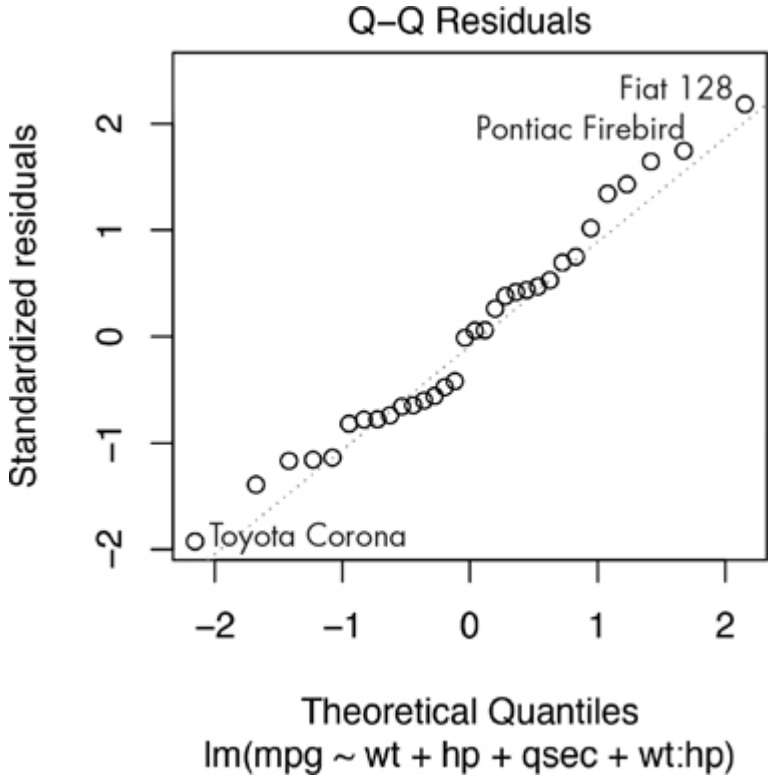


Figure 22-4: A normal QQ plot of the residuals from the `car.step` model

You interpret the QQ plot of the residuals as detailed in Section 16.2.2. The gray diagonal line represents the true normal quantiles, and the plotted points are the corresponding numeric quantiles of the estimated regression errors. Normally distributed data should lie close to the straight line. For the `car.step` model, the points generally seem to follow the path laid out by the theoretical normal quantiles. There is some deviation, which is to be expected, but no appArent major or systematic departure from normality.

There are also other ways to test for normality, such as the famous Shapiro–Wilk hypothesis test. The null hypothesis for the Shapiro–Wilk test is that the data are normally distributed, so a small *p*-value would suggest non-normality of your data (see Royston 1982, for technical details). To execute the procedure, use the `shapiro.test` function

in R. By first extracting the standardized residuals of your fitted model with `rstandard`, you'll see that applying this test to `car.step` results in a large *p*-value:

```
R> shapiro.test(rstandard(car.step))

        Shapiro-Wilk normality test

data:  rstandard(car.step)
W = 0.97105, p-value = 0.5288
```

In other words, there's no evidence (according to this test) that the residuals of `car.step` aren't normal.

Being able to assume normality of the error term supports the methodology used to produce reliable estimates of the regression coefficients. As long as your data are *approximately* normal, though, you shouldn't be too concerned with mild indications of non-normality. Some transformations of your data, and an increase in your sample size, can reduce concerns about more severe deviations from normality.

22.3.3 Illustrating Outliers, Leverage, and Influence

It's important to investigate any individual observations that appear unusual or extreme compared to the bulk of your observations. In general, an exploratory analysis of your data, perhaps involving summary statistics or scatterplot matrices, is a good idea since it can help you identify any such values, which have the potential to adversely affect your model fits. Before going further, let's clarify some frequently used terms:

> **Outlier** This is a general term used to describe an unusual observation in the context of the data, as you saw in Section 13.2.7. In linear regression, an outlier typically has a large residual but is identified as an outlier only if it doesn't conform to the trend of the fitted model. An outlier can, but doesn't always, significantly alter the trends described by the fitted model.

Leverage This term refers to the extremity of the values of the present predictors. A high-leverage point is an observation with predictor values extreme enough to potentially affect the slopes or trends in the fitted model significantly. An outlier can have a high or low leverage.

Influence An observation with high leverage that *does* affect the estimated trends is deemed influential. In other words, influence is judged only when the response value is taken into account alongside the corresponding predictor values.

You can describe a given observation using a combination of these terms. Let's look at some hypothetical examples. Create the following vectors of 10 supposed response (`y`) and explanatory (`x`) values:

```
R> x <- c(1.1,1.3,2.3,1.6,1.2,0.1,1.8,1.9,0.2,0.75)
R> y <- c(6.7,7.9,9.8,9.3,8.2,2.9,6.6,11.1,4.7,3)
```

These will form the bulk of the data. Now consider the following six objects, `p1x` to `p3y`, which will be used to store the predictor and response values for three additional observation points:

```
p1x <- 1.2
p1y <- 14
p2x <- 5
p2y <- 19
p3x <- 5
p3y <- 5
```

That is, point 1 is (1.2, 14); point 2 is (5, 19); and point 3 is (5, 5).

Next, the following uses of `lm` provide four simple linear model fits:

```
R> mod.0 <- lm(y~x)
R> mod.1 <- lm(c(y,p1y)~c(x,p1x))
R> mod.2 <- lm(c(y,p2y)~c(x,p2x))
R> mod.3 <- lm(c(y,p3y)~c(x,p3x))
```

The first is the regression of `y` on `x` only. The next three additionally include points 1, 2, and 3, separately, as an 11th observation.

Now you can use these objects to visually clarify the definitions of *outlier*, *leverage*, and *influence*, as in Figure 22-5.

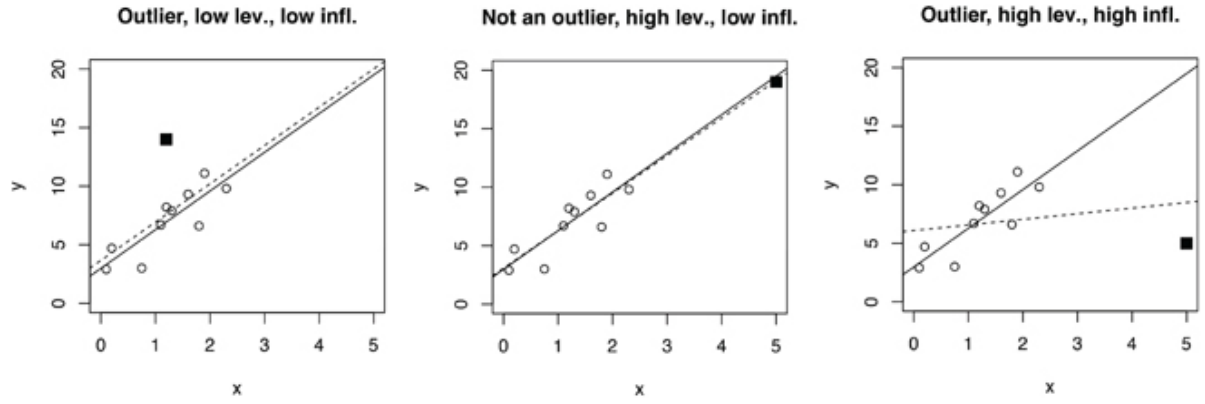


Figure 22-5: Three examples of the definitions of outlier, leverage, and influence in a linear regression context

In each plot, the solid line represents the model fitted to the original observations in `x` and `y`, and the dashed line represents the model fitted including the extra point, plotted with ■.

Enter the following code to initialize the scatterplot with set axis limits of `x` and `y`:

```
R> plot(x,y,xlim=c(0,5),ylim=c(0,20),main="Outlier, low lev., low infl.")
```

Then use `points`, `abline`, and `text` to build the leftmost plot of Figure 22-5, as follows:

```
R> points(p1x,p1y,pch=15,cex=1.5)
R> abline(mod.0)
R> abline(mod.1,lty=2)
```

Create the middle and right plots by replacing `p1x`, `p1y`, and `mod.1` with those corresponding to points 2 and 3 and altering the `main` argument in the initial call to `plot` to change the title accordingly.

In the leftmost plot of Figure 22-5, the additional point is an example of an outlier since it sits away from the bulk of the data *and*

doesn't conform to the trend suggested by the original observations. Despite this, it's considered to have low leverage only because of its predictor value of 1.2 (`p1x`), which isn't deemed unusual compared to that of the other values of `x`. In fact, its proximity to the overall mean of the `x` values indicates that the effect of including it, incorporated in `mod.1`, is only a minor change to the original intercept. You could therefore also classify this as a low influence point: The overall change to the fitted model seems minimal.

The center plot of Figure 22-5 provides an example of an observation that would *not* be considered an outlier. Although the extra point (point 2) does sit apart from the 10 original observations, it conforms quite well to the model fitted to only `x` and `y`, which is important in the context of regression. That being said, the added point is considered to possess a high leverage, since it sits at an extreme predictor value compared to all other values of `x` (in other words, it has the *potential* to dramatically alter the fit should its response value be different). As it stands, it's a low influence point since the model fit itself is barely affected by its inclusion.

Lastly, the rightmost plot shows (via inclusion of point 3) a clear example of an outlier in a high-leverage position that also has a high influence: It sits away from the original 10 observations and isn't a clear part of the original trend; its extreme predictor value means high leverage; and its inclusion substantially alters the entire model by notably dragging down the slope and raising the intercept. These ideas remain the same in higher dimensions (that is, when you have several predictors) for multiple linear regression models.

22.3.4 Calculating Leverage

You calculate leverage by using the design matrix structure $\boldsymbol{X}$ defined in Section 21.2.2. Specifically, if you have n observations, then the leverage of the ith point ($i = 1, \ldots, n$) is denoted h_{ii}. These are the diagonal elements (ith row, ith column) of the $n \times n$ matrix H such that:

$$H = X\left(X^{\mathrm{T}}\,X\right)^{-1}X^{\mathrm{T}} \quad (22.4)$$

In R, constructing the design matrix for the 10 illustrative predictor observations you defined in Section 22.3.3 as the object `x` is achieved in a straightforward fashion using knowledge of `cbind` (refer to Section 3.1.1). H is subsequently calculated using the corresponding functions for matrix multiplication (`%*%`), matrix transposition (`t`), matrix inversion (`solve`), and diagonal element extraction (`diag`). Then you can plot the values h_{ii} against the values of `x` themselves. The following code

```
R> X <- cbind(rep(1,10),x)
R> hii <- diag(X%*%solve(t(X)%*%X)%*%t(X))
R> hii
[1] 0.1033629 0.1012107 0.3487221 0.1302663 0.1001345 0.3723971 0.1711595
[8] 0.1980630 0.3261232 0.1485607
R> plot(hii~x,ylab="Leverage",main="",pch=4)
```

produces Figure 22-6.

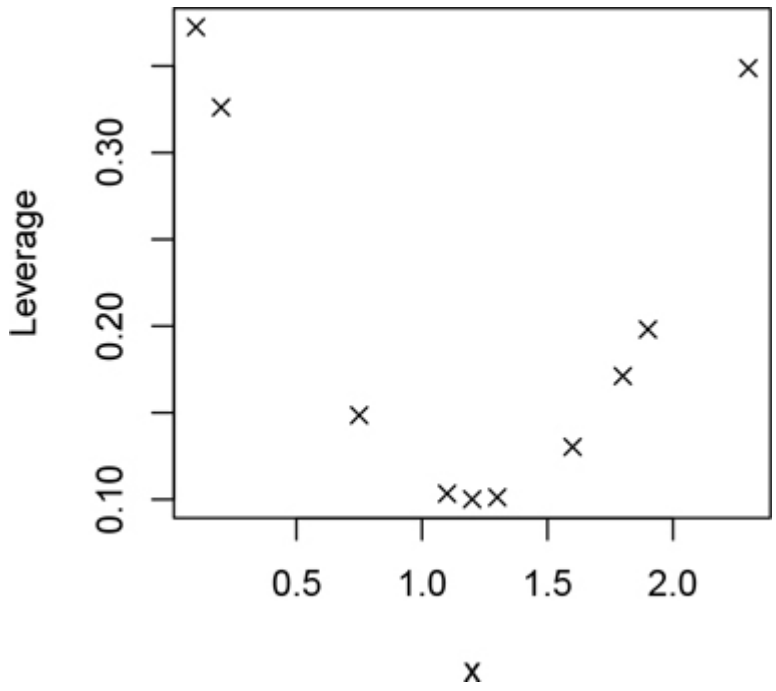


Figure 22-6: Plotting the leverage of the 10 illustrative predictor observations in the object `x`

You would typically use the built-in R function `hatvalues`, named after the nature of the matrix formulation in Equation (22.4), to obtain the leverages (rather than manually constructing the design matrix $\boldsymbol{X}$ and doing the math yourself). Simply provide `hatvalues` with your fitted model object. You can confirm your earlier calculations by using the corresponding `lm` object fitted to the `x` and `y` data (stored previously as the `mod.0` object):

```
R> hatvalues(mod.0)
        1         2         3         4         5         6         7
0.1033629 0.1012107 0.3487221 0.1302663 0.1001345 0.3723971 0.1711595
        8         9        10
0.1980630 0.3261232 0.1485607
```

Looking at Figure 22-6, the appearance of the leverages plotted against the corresponding predictor values themselves makes sense: Leverage gets progressively higher as you move away from the mean of the predictor data in either direction. This is essentially the pattern you'll see for any plot of the raw leverages.

22.3.5 Calculating Cook's Distance

Of course, leverage alone isn't enough to determine the overall influence of each observation on a fitted model. For that, you must also take the response value into account.

Arguably the most well-known measure of influence is *Cook's distance*, which estimates the magnitude of the effect of deleting the *i*th value from the fitted model. Cook's distance for observation *i* (denoted D_i) is given with the following equation:

$$D_i = \sum_{j=1}^{n} \frac{\left(\hat{y}_j - \hat{y}_j^{(-i)}\right)^2}{(p+1)\hat{\sigma}^2}; \quad i,\ j = 1,\ \ldots,\ n \quad (22.5)$$

It turns out that this equation is a specific function of a point's leverage and residual. Here, the value $\hat{y}_j$ is the predicted mean response of observation *j* for the model fitted with all *n* observations, and $\hat{y}_j^{(-i)}$

represents the predicted mean response of observation j for the model fitted *without* the ith observation. As usual, the term p is the number of predictor regression parameters (excluding the intercept), and the value σ is the estimate of the residual standard error.

Put simply, the larger the value of D_i, the larger the influence the ith observation has on the fitted model, meaning outlying observations in highleverage positions will correspond to higher values of D_i. The important question is, how big does D_i have to be in order for point i to be considered influential? In practice, this is difficult to universally answer, and there's no formal hypothesis test for it, but there are several rule-of-thumb cutoff values. One such rule states that if $D_i > 1$, the point should be considered influential; another, more sensitive rule suggests $D_i > 4/n$ (see, for example, Bollen & Jackman 1990, Chatterjee et al. 2000). It's generally advised that you compare multiple Cook's distances for a given fitted model rather than analyzing one single value; you might consider further investigation of a data point corresponding to a comparatively large D_i.

Continue with the objects created in Section 22.3.3, with the 10 illustrative observations in `x` and `y`, as well as the additional point defined in `p1x` and `p1y`. The linear regression model fitted to those data was stored as the object `mod.1`. It's a good exercise to write some code that calculates the Cook's distance values following Equation (22.5). To that end, enter the following code in the R editor:

```
x1 <- c(x,p1x)
y1 <- c(y,p1y)
n <- length(x1)
param <- length(coef(mod.1))
yhat.full <- fitted(mod.1)
sigma <- summary(mod.1)$sigma
cooks <- rep(NA,n)
for(i in 1:n){
    temp.y <- y1[-i]
    temp.x <- x1[-i]
```

```
    temp.model <- lm(temp.y~temp.x)
    temp.fitted <- predict(temp.model,newdata=data.frame(temp.x=x1))
    cooks[i] <- sum((yhat.full-temp.fitted)^2)/(param*sigma^2)
}
```

Here, we first create new objects `x1` and `y1` to hold all 11 observations. The objects `n`, `param`, and `sigma` store the dataset size, the total number of estimated regression parameters (two in this case), and the estimated residual standard error for the model originally fitted to all 11 data points. The latter two items, `param` and `sigma`, represent $(p + 1)$ and $\widehat{\sigma}$ in Equation (22.5), respectively. The object `yhat.full` uses the `fitted` function on the object `mod.1` to provide the fitted mean response values, representing the $\hat{y}_j$ values in Equation (22.5).

To store the Cook's distances, a vector `cooks` of 11 positions is created (initialized to be filled with `NA`s) with `rep`. Now, to calculate each D_i value, set a `for` loop (see Chapter 10) to scroll through each index from `1` to `11`. The first step of the loop is to create two temporary vectors `temp.x` and `temp.y` to be `x1` and `y1` with the observation at index `i` removed. A new temporary linear model is fitted to `temp.y` based on `temp.x`; then `predict` finds the mean responses from `temp.model` for each of the 11 predictor values (in other words, including the one that was deleted). As such, the resulting vector `temp.fitted` represents the $\hat{y}_j^{(-i)}$ values in Equation (22.5). Finally, `sum` and the product of `param` with `sigma^2` compute D_i, and the result is stored in `cooks[i]`.

After highlighting and executing the code, the resulting Cook's distances are as follows:

```
R> cooks
 [1] 2.425993e-03 4.060891e-07 1.027322e-01 1.844150e-03 2.923667e-03
 [6] 7.213229e-02 1.387284e-01 3.021075e-02 7.099904e-03 1.251882e-01
[11] 3.136855e-01
```

Unsurprisingly, the largest value of these is the last one, at around 0.314. This corresponds to the 11th observation in `x1` and `y1`, which is the additional point originally defined in `p1x` and `p1y`. The value 0.314 is

less than 1 and less than 4 / 11 = 0.364, the cutoffs defined by the earlier rules of thumb. This ties in with the assessment of the top-left image in Figure 22-5 that the influence of the point `p1x` and `p1y` is minimal when compared to the influence of a point like `p3x` and `p3y` in the rightmost image.

Just as the `hatvalues` function computes the leverages for you, the built-in `cooks.distance` function does the same for D_i. You can confirm the previous values in `cooks`, which are based on `mod.1`, as follows:

```
R> cooks.distance(mod.1)
           1            2            3            4            5            6
2.425993e-03 4.060891e-07 1.027322e-01 1.844150e-03 2.923667e-03 7.213229e-02
           7            8            9           10           11
1.387284e-01 3.021075e-02 7.099904e-03 1.251882e-01 3.136855e-01
```

R automatically calculates and provides Cook's distances as a diagnostic plot of a fitted linear model object when you select `which=4` in the relevant usage of `plot` applied to a fitted `lm` model object. The following code uses `mod.1`, `mod.2`, and `mod.3` from earlier to produce the three images in Figure 22-7; these correspond to the three datasets in Figure 22-5:

```
R> plot(mod.1,which=4)
R> plot(mod.2,which=4)
R> plot(mod.3,which=4)
R> abline(h=c(1,4/n),lty=2)
```

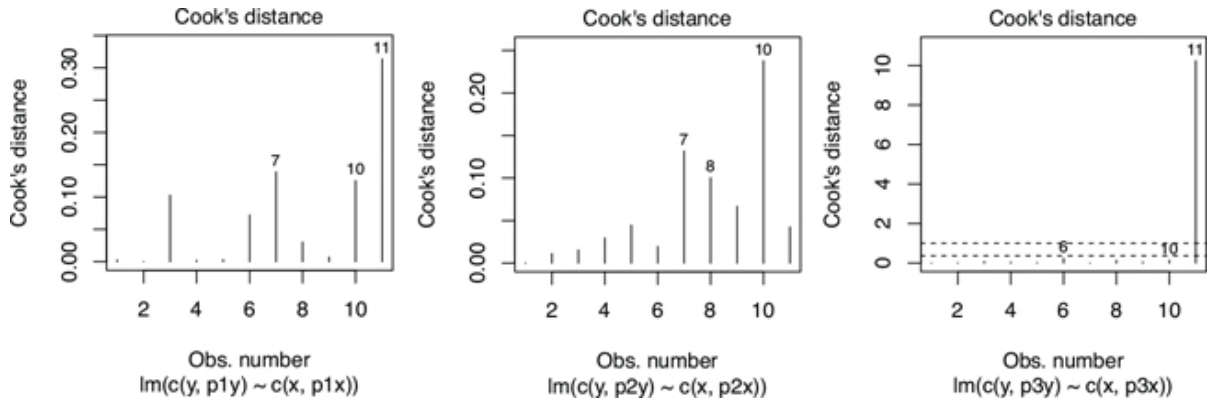


Figure 22-7: Three illustrative examples of the Cook's distance plots produced in R, based on `mod.1` (left), `mod.2` (middle), and `mod.3` (right)

The D_i displayed in the leftmost panel of Figure 22-7 match the values you manually calculated earlier, stored in `cooks`. The influences of all the data points in the middle plot remain relatively small, which reflects what you can see in the top-right plot of Figure 22-5, where the additional point (`p2x`, `p2y`) doesn't greatly affect the overall fit. In the rightmost plot, `abline` superimposes two horizontal lines marking off the values 1 (highest line) and 4 / 11 = 0.364, both of which are clearly breached by the 11th point (`p3x`, `p3y`), just as you'd expect given the corresponding plot of the data in the rightmost panel of Figure 22-5.

Turn your attention back to the `car.step` model, where you modeled MPG using the `mtcars` dataset, with the final fit achieved using stepwise AIC selection in Section 22.2.4 on page 615. You've already studied the residualsversus-fitted and QQ plots in Figures 22-2 and 22-4, respectively. Figure 22-8 provides a plot of Cook's distances for this model with these two lines:

```
R> plot(car.step,which=4)
R> abline(h=4/nrow(mtcars),lty=2)
```

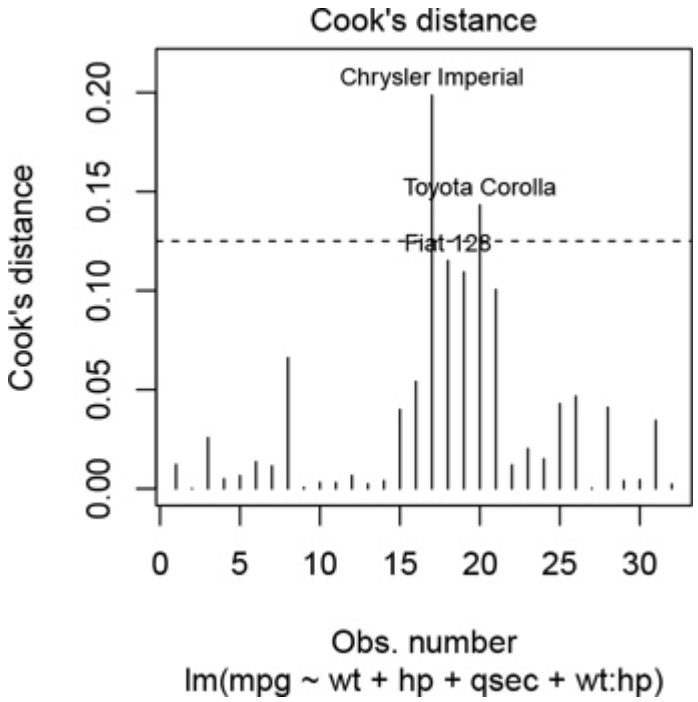


Figure 22-8: Cook's distances for the model in `car.step`; a dashed horizontal line marks off 4/n for the `mtcars` data frame.

The plot labels the three points with the highest D_i by default; two of these breach the $4 / n = 4 / 32 = 0.125$ rule-of-thumb mark. In light of the fitted model based on various effects of car weight (`wt`), horsepower (`hp`), and quarter-mile time (`qsec`), the Chrysler Imperial and Toyota Corolla are deemed to be in high-leverage positions with residuals large enough to be designated as influential. It should also be noted that the Fiat 128, though it doesn't quite breach the 0.125 line, is still rather influential and was in fact also one of the extreme labeled points in the residual plots (Figure 22-2) and the QQ plot (Figure 22-4).

This could reasonably suggest investigating these highly influential observations further. Was everything recorded correctly? Has your model been selected carefully? Are there alternative options for the model, such as additional predictor terms or transformations? You could explore these options and continue to monitor a plot of the Cook's distances (along with the other diagnostics).

Whatever your result, the presence of influential points doesn't necessarily mean there is a serious problem with your model; this is more a tool to help you detect observations that are extreme in terms of their specific combination of the predictor values *and* that have a larger residual, suggesting their response value sits away from the trends predicted by the model itself. This is especially useful in multiple regression, when the high dimensionality of the response-predictor data can make conventional visualization of the raw data in a single plot difficult.

22.3.6 Graphically Combining Residuals, Leverage, and Cook's Distance

The last two diagnostic plots from `plot` combine the standardized residual, the leverage, and the Cook's distance for the *i*th observation. These "combination plots" are especially useful in allowing you to see whether it is the high leverage or large residual, or both, that contributes more to the overall influence of a given observation.

Remaining with the models `mod.1`, `mod.2`, and `mod.3` fitted to the three illustrative datasets, use `plot` with `which=5` to produce the three images in the left column of Figure 22-9:

```
R> plot(mod.1,which=5,add.smooth=FALSE,cook.levels=c(4/11,0.5,1))
R> plot(mod.2,which=5,add.smooth=FALSE,cook.levels=c(4/11,0.5,1))
R> plot(mod.3,which=5,add.smooth=FALSE,cook.levels=c(4/11,0.5,1))
```

These plots show leverage on the horizontal axis and the standardized residuals on the vertical axis for each observation. As a function of residual and leverage, the Cook's distances are represented as *contours* on each of these scatterplots. These contours delineate the spatial areas of the plots that correspond to high influence (in the right extreme corners).

The closer a point falls to the horizontal line at 0, the smaller its residual. A point that lies more left than right has a smaller leverage. If a point lies far enough from the horizontal line given its leverage (x-axis)

position, it will breach the contours marking off certain values of D_i (defaulting to just 0.5 and 1), indicating a high influence. Indeed, you can see by the narrowing of the contours as you move from left to right on these plots that classification as a high-influence point is "easier" if a given observation is in a high-leverage position, which makes perfect sense. In the previous calls to `plot` with `which=5`, the optional `cook.levels` argument is used to include a contour for the rule-of-thumb value of 4/11 for these three examples.

The plot for `mod.1` shows that the added 11th point (`p1x`, `p1y`) has a large residual, but it doesn't breach the 4/11 contour because it's in a lowleverage position. The plot for `mod.2` shows that the added point (`p2x`,`p2y`) is in a high-leverage position but isn't influential because its residual is small. Lastly, the plot for `mod.3` clearly identifies the added 11th point (`p3x`, `p3y`) as highly influential: With a large residual and high leverage, it's in clear breach of the high-level contours. Looking back at all the previous plots of these three illustrative datasets, you should be able to recognize how these diagnostic plots reflect the nature of each of the individually added extra observations.

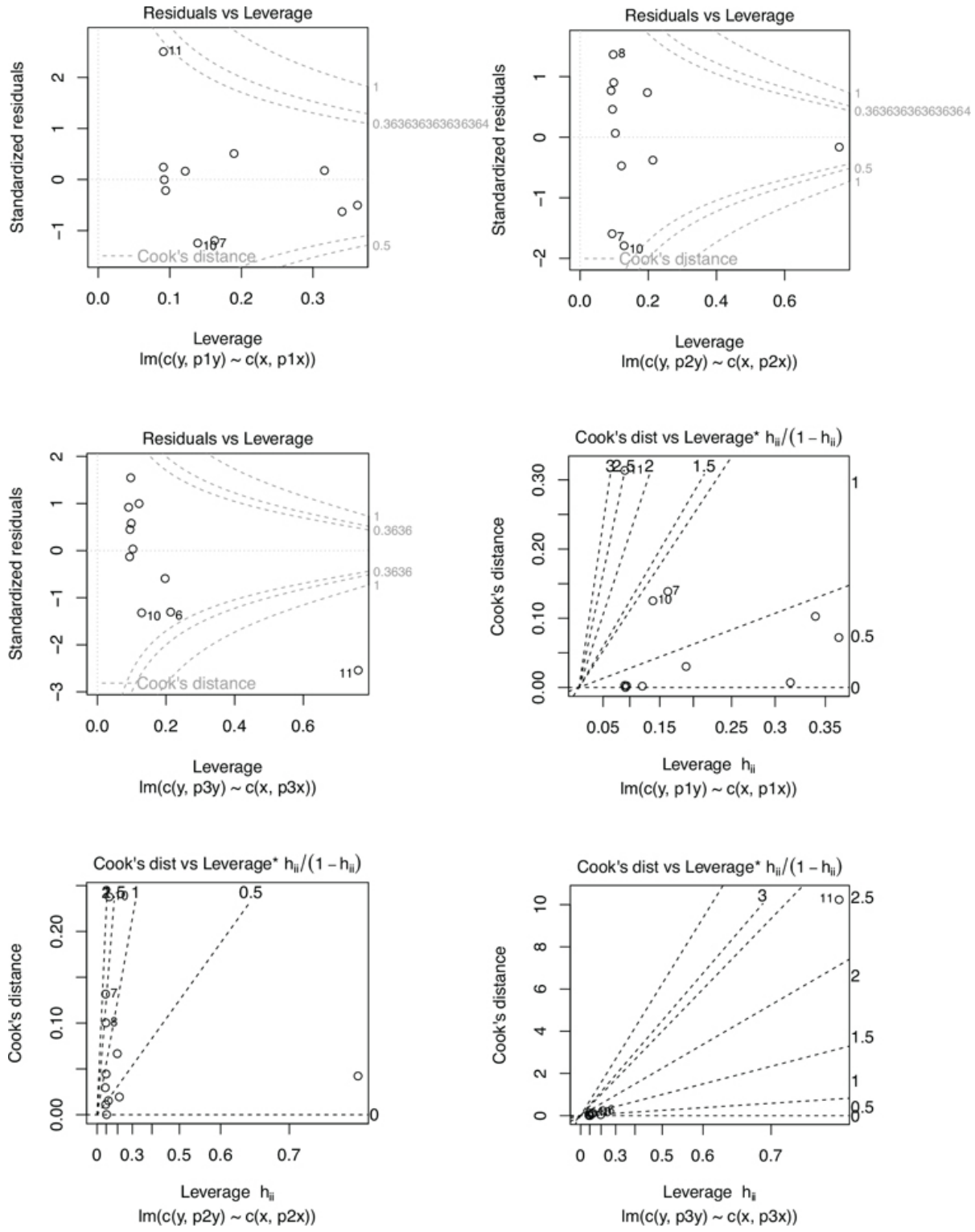


Figure 22-9: Combination diagnostic plots of standardized residuals against leverage (left column) and Cook's distance against leverage (right column) for the three illustrative models `mod.1` *(top row),* `mod.2` *(middle row), and* `mod.3` *(bottom row)*

The final diagnostic plot, achieved using `which=6`, displays the same information as the `which=5` combination diagnostic, but this time the vertical axis displays Cook's distance, and the horizontal axis displays a transformation of the leverage, namely, $h_{ii} / (1 - h_{ii})$. This transformation amplifies larger leverage points in terms of their horizontal position, which is an effect that, in part, indirectly displays itself as a "stretched-out" scale on the x-axis. This is useful if you're particularly interested in the extremity of the observations with respect to the collection of predictor variables. As such, the contours therein define standardized residuals as a function of the scaled leverage and Cook's distance. The following lines produce the three images in the rightmost column of Figure 22-9:

```
R> plot(mod.1,which=6,add.smooth=FALSE)
R> plot(mod.2,which=6,add.smooth=FALSE)
R> plot(mod.3,which=6,add.smooth=FALSE)
```

Points positioned further to the right are high-leverage points; points positioned higher up are high-influence points. Looking down the right column of plots in Figure 22-9, you should be able to see that the three additional points are found where you would expect them to be, according to their characteristics in `mod.1`, `mod.2`, and `mod.3`.

For a real-data example, return to the model stored in `car.step` (see page 615). The following code produces the two combination diagnostic plots in Figure 22-10:

```
R> plot(car.step,which=5,cook.levels=c(4/nrow(mtcars),0.5,1))
R> plot(car.step,which=6,cook.levels=c(4/nrow(mtcars),0.5,1))
```

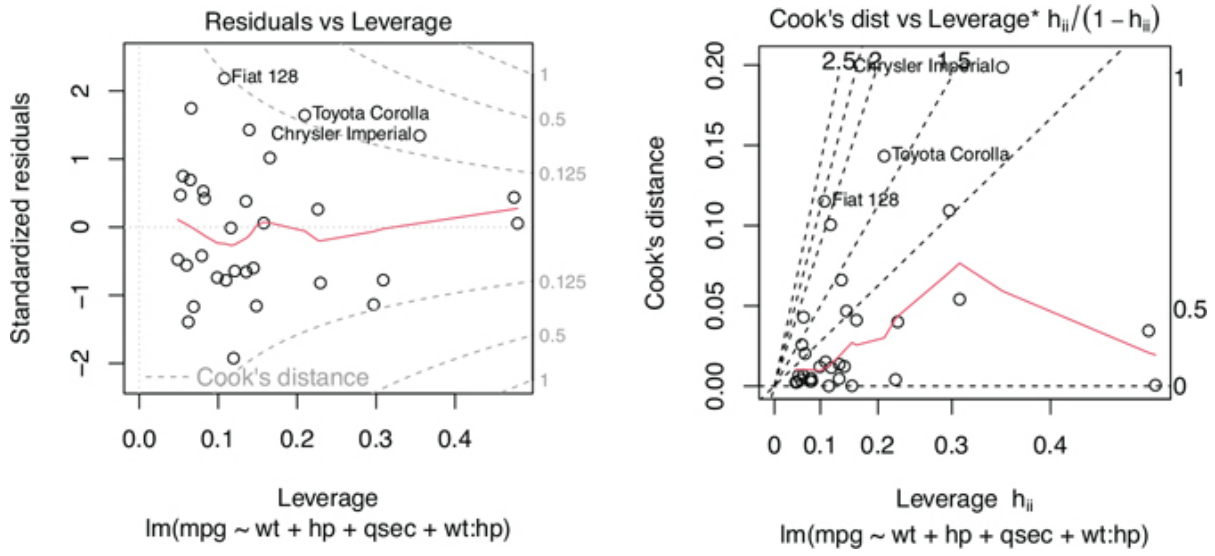


Figure 22-10: A combination diagnostic plot of standardized residuals against leverage (left) and Cook's distance against leverage (right) for the `car.step` model

The two images in Figure 22-10 show the Corolla and the Imperial as influential observations, with D_i values greater than the `4/nrow(mtcars)` ruleof-thumb cutoff. Interestingly, this plot reveals that the Imperial (which was shown to have the largest D_i by far in Figure 22-8) actually has a smaller residual than both the Corolla and the Fiat 128. Its high influence is clearly because of its high-leverage position with respect to the predictor values of the variables present in `car.step`. The Fiat 128, on the other hand, has one of the largest residuals in the entire dataset (which is why it was flagged in some of the earlier diagnostic plots) but misses out on being labeled a highinfluence observation because of its relatively low-leverage position (based purely on the rule-of-thumb cutoff).

Any linear regression model will have observations that influence the model more than others, and these plots aim to help you identify them. But deciding what to actually do with highly influential observations can be difficult and is application-specific. Although it's not ideal to have a single observation exerting heavy influence over the final estimated model, it's also extremely unwise to remove these observations without careful thought; they might be pointing to other

issues, such as deficiencies in your model design or previously undetected trends.

Exercise 22.2

In Section 22.2.2, you used the `nuclear` data frame in the `boot` package to illustrate forward selection, where a model was selected for `cost` as a function of main effects of `date`, `cap`, `pt`, and `ne`.

a. Access the data frame; fit and summarize the model as described.

b. Inspect the raw residuals plotted against the fitted values and a normal QQ plot of the standardized residuals, and comment on your interpretations: Do the assumptions underpinning the error component of the linear model appear satisfied in this case?

c. Determine the value of the rule-of-thumb cutoff, $4/n$, for influential observations based on the Cook's distances. Produce a plot of the Cook's distances and add a horizontal line corresponding to the cutoff. Comment on your findings.

d. Produce a combination diagnostic plot of the standardized residuals against leverage. Set the Cook's distance contours to include the cutoff value from (c) as well as the default contours. Interpret the plot: How are any individually influential points characterized?

e. Based on (c) and (d), you should be able to identify the record in `nuclear` exerting the largest influence on the fitted model. For the sake of argument, let's assume the observation was recorded incorrectly. Refit the model from (a), this time omitting the offending row from the data

frame. Summarize the model: Which coefficients have changed the most? Produce the diagnostic plots from (b) for the new model and compare them to the ones from earlier.

Load the `faraway` package and access the `diabetes` data frame. In Exercise 22.1 (g), you used stepwise AIC selection to choose a model for `chol`.

f. Using the `diabetes` object, fit the multiple linear model identified in the earlier exercise—that is, with main effects and a two-way interaction between `age` and `frame` alongside an isolated main effect for `waist`. By summarizing the fit, determine the number of records that contained missing values in `diabetes` that were deleted from the estimation.

g. Produce the raw residuals-versus-fitted and QQ diagnostic plots for the model in (f). Comment on the validity of the error assumptions.

h. Investigate influential points. Make use of the familiar ruleof-thumb cutoff for Cook's distances (note you'll need to subtract the number of missing values from the total size of the data frame to get the effective sample size n for your model). In the combination plot of the standardized residuals against leverage, use one, three, and five times the cutoff as the Cook's distance contours.

Recall the discussion of reading in web-based files in Section 8.2.3. There, you called in a data frame containing data on the prices of 308 diamonds (in Singapore dollars), as well as weight (in carats: continuous), color (categorical: six levels from `D`, the least yellow and the reference level, to `I`, the most yellow), clarity (categorical: five levels with `IF`, essentially flawless, and the reference level, `VVS1`, `VVS2`, `VS1`, and `VS2`, with the last being the least clear), and certification (categorical: three levels for different diamond certification bodies with levels `GIA` as the reference, `HRD`, and `IGI`).

Seek out the freely available article by Chu (2001) for more information on these data. With an internet connection, run the following lines, which will read in the data as the object `diamonds` and name each variable column appropriately:

```
R> dia.url <- "http://jse.amstat.org/v9n2/4Cdata.txt"
R> diamonds <- read.table(dia.url,stringsAsFactors=TRUE)
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
```

i. Using either base R graphics or `ggplot2` functionality, produce a scatterplot of the price on the y-axis and carat weight on the x-axis to get a feel for the data. Experiment using color to split the points according to the following one at a time:

   - Diamond clarity
   - Diamond color
   - Diamond certification

j. Fit a multiple linear model with `Price` as the response and main effects for the other variables as the predictors. Summarize the model and produce the three diagnostic plots that tell you about the assumptions surrounding the error term. Comment on the plots: Are you satisfied that this is an appropriate model for the diamond prices? Why?

k. Repeat (j) but use a log transformation of `Price`. Again, inspect and comment on the validity of the error assumptions.

l. Repeat (k)—that is, model the log transformation of price—but this time include an additional quadratic term for `Carat` (refer back to Section 21.4.1 for details on polynomial transformations). How do the residual diagnostics look now?

22.4 Collinearity

This final aspect of fitting regression models isn't technically classified as a diagnostic check but still has substantial potential to adversely affect the validity of any conclusions you draw from a fitted model and occurs frequently enough to warrant discussion here. *Collinearity* (also referred to as *multicollinearity*) is when two or more of the explanatory variables are highly correlated with each other.

22.4.1 Potential Warning Signs

High correlation between two predictors implies there will be some level of redundancy in terms of the information they contain when it comes to the response variable. It's a problem since it can destabilize model fitting and therefore be detrimental to any subsequent model-based inference, as noted earlier.

The following items serve as potential warnings of collinearity when you're inspecting a model summary:

- The omnibus F-test (Section 21.3.5) result is statistically significant, but none of the individual t-test results for the regression parameters are significant.
- The sign of a given coefficient estimate contradicts what you would reasonably expect to see—for example, drinking more wine resulting in a lower blood alcohol level.
- Parameter estimates are associated with unusually high standard errors or vary wildly when the model is fitted to randomly selected subsets of the available records.

As the last point notes, collinearity tends to have more of a detrimental effect on the standard errors of the coefficients (and associated outcomes such as confidence intervals, significance tests, and prediction intervals) than it does on point predictions per se. In most cases, you can avoid collinearity simply by being careful. Be aware of the variables present and how the data have been collected. For example,

ensure any given predictors you intend to include in the model don't just represent a rescaled value of another included predictor. It's also advisable to perform an exploratory analysis of your data, producing summary statistics and basic statistical plots. You can tabulate counts between categorical variables or look at estimated correlation coefficients between continuous variables, for example. In the latter case, as a rough guide, some statisticians suggest that a correlation of 0.8 or more between two variables could potentially lead to problems if both are included as predictors in a model.

22.4.2 Correlated Predictors: A Quick Example

Consider the `survey` data of the statistics students again, located in the `MASS` package (first seen in Section 19.3.2). In most models you've looked at for these data, you've attempted to predict student height from certain explanatory variables, often including the handspan of the writing hand (`Wr.Hnd`). The help page `?survey` shows that data have also been collected on the nonwriting handspan (`NW.Hnd`). It's reasonable to expect that these two variables will be highly correlated, which is precisely why I've avoided any use of `NW.Hnd` previously. Indeed, executing

```
R> cor(survey$Wr.Hnd,survey$NW.Hnd,use="complete.obs")
[1] 0.9483103
```

reveals a high correlation coefficient, suggesting a strong positive linear association between the writing and non-writing handspans of the students. In other words, these two variables should represent much the same information as one another when used in any given model.

Now you know from previously fitted models that writing handspan has a significant and positive impact on predicting mean student height. The following code quickly confirms this via a simple linear regression:

```
R> summary(lm(Height~Wr.Hnd,data=survey))

Call:
```

```
lm(formula = Height ~ Wr.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-19.7276  -5.0706  -0.8269   4.9473  25.8704

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9536     5.4416   20.94   <2e-16 ***
Wr.Hnd        3.1166     0.2888   10.79   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.909 on 206 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3612,    Adjusted R-squared:  0.3581
F-statistic: 116.5 on 1 and 206 DF,  p-value: < 2.2e-16
```

The high positive correlation between `Wr.Hnd` and `NW.Hnd` suggests that were we to use `NW.Hnd` instead, that it should have a similar effect in a model for `Height`. Indeed, it does:

```
R> summary(lm(Height~NW.Hnd,data=survey))

Call:
lm(formula = Height ~ NW.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-21.8285  -5.1397  -0.2867   4.5611  25.5750

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 118.0324     5.2912   22.31   <2e-16 ***
NW.Hnd        2.9107     0.2818   10.33   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 8.032 on 206 degrees of freedom
```

```
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3412,    Adjusted R-squared:  0.338
F-statistic: 106.7 on 1 and 206 DF,  p-value: < 2.2e-16
```

Look, however, at what happens if you try to model student height based on both of these predictors at the same time:

```
R> summary(lm(Height~Wr.Hnd+NW.Hnd,data=survey))

Call:
lm(formula = Height ~ Wr.Hnd + NW.Hnd, data = survey)

Residuals:
     Min       1Q   Median       3Q      Max
-20.0144  -5.0533  -0.8558   4.7486  25.8380

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 113.9962     5.4545  20.900   <2e-16 ***
Wr.Hnd        2.7451     1.0728   2.559   0.0112 *
NW.Hnd        0.3707     1.0309   0.360   0.7195
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.926 on 205 degrees of freedom
  (29 observations deleted due to missingness)
Multiple R-squared:  0.3616,    Adjusted R-squared:  0.3554
F-statistic: 58.06 on 2 and 205 DF,  p-value: < 2.2e-16
```

Since the effects of `Wr.Hnd` and `NW.Hnd` on `Height` are intermingled with one another, including both at the same time heavily masks any individual contribution to modeling the response. Statistical significance of the predictors is almost nonexistent; at the least, the effects are both associated with much, much higher *p*-values than in the individual single-predictor fits. That said, the omnibus *F*-test remains highly significant, giving an example of the first warning sign noted in Section 22.4.1.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`anova`	Partial *F*-tests	Section 22.2.1, p. 603
`add1`	Review single-term additions	Section 22.2.2, p. 605
`update`	Make changes to fitted model	Section 22.2.2, p. 606
`drop1`	Review single-term deletions	Section 22.2.3, p. 610
`step`	Stepwise AIC model selection	Section 22.2.4, p. 615
`plot` (used on `lm` object)	Model diagnostics	Section 22.3.1, p. 624
`rstandard`	Extract standardized residuals	Section 22.3.2, p. 627
`shapiro.test`	Shapiro–Wilk test of normality	Section 22.3.2, p. 627
`hatvalues`	Calculate leverages	Section 22.3.4, p. 630
`fitted` (or `fitted.values`)	Fitted response values	Section 22.3.5, p. 631
`cooks.distance`	Calculate Cook's distances	Section 22.3.5, p. 632

PART V
ADVANCED GRAPHICS

23
ADVANCED PLOT CUSTOMIZATION



Many users are first drawn to R because of its impressive graphical capabilities and the ease with which you can control and tailor the resulting visuals. In this chapter, you'll take a closer look at the base R graphics device, and fine-tune the plots you're already familiar with, to get the most use out of your visualizations. In the chapters that follow, you'll expand your repertoire in both `ggplot2` and traditional R graphics.

Much of this chapter assumes you're familiar with the content of Chapters 7 and 14. In specific cases, I'll also assume you're using the standard base R application (for example, *R.app* on a Mac or *Rgui.exe* in Windows; see Appendix A), since the behavior of some of the commands can vary if you're working with R in a different context.

Depending on your operating system, the default software drivers used to render graphical displays on your computer screen are also different. In the standard *R.app* application on a Mac, for example, you'll notice that producing a live plot will open a window with a banner title that looks something like *Quartz 2 [*]*, which is the default graphics device driver for macOS is the Quartz system. On a Windows machine using *Rgui.exe*, you'll see *R Graphics: Device 2 (ACTIVE)*. The

numbering of any graphics device always starts at 2; Device 1 is referred to as the *null device*, meaning there's nothing active currently.

NOTE

For a list of devices your R session has available, enter `?Devices` *at the prompt. The list includes commands such as* `png` *and* `pdf`*, which are the so-called silent graphics devices that enable direct-to-file plotting, as detailed in Chapter 8. You can use a different device for any given plot if you want, though the default is almost always appropriate if you're plotting directly to your screen.*

23.1 Handling the Graphics Device

So far, your plotting has dealt with one image at a time. It's possible to have multiple graphics devices open, but only one will be deemed active at any given time (the banner titles highlight the currently active device with *[*]* or *(ACTIVE)*). This is useful when you're working on several plots at once or want to view or alter one plot without closing any others.

23.1.1 Manually Opening a New Device

The typical base R commands you've met already (such as `plot`, `hist`, `boxplot`, and so on) will automatically open a device for plotting and draw the desired plot, if nothing is currently open. You can also open new device windows using `dev.new`; this newest window will immediately become active, and any subsequent plotting commands will affect that particular device.

As an example, first close any open graphics windows and then enter the following at the R prompt:

```
R> plot(quakes$long,quakes$lat)
```

This generates a plot of the spatial locations of the occurrences of the 1,000 seismic events in the ready-to-use `quakes` data frame (for simplicity, we'll ignore the issue of setting the aspect ratio for geographic data; see Section 14.6.4). If the only device currently available is Device 1, the null device, any plotting command that refreshes a plotting window and produces a new image (such as `plot` here or more specialized commands such as `hist` or `boxplot`) will automatically open a new instance of the default graphics device before actually plotting the data. On my macOS machine running *R.app*, I see *Quartz 2 [*]* open and display the plot of the spatial coordinates. Using the RStudio application on the same machine, the plot by default simply appears in the "Plots" pane.

Now let's say you'd also like to see a histogram of the number of stations that detected each event. Execute the following to open a new plotting window:

```
R> dev.new()
```

In the macOS *R.app*, this new window will be numbered 3 (it usually sits itself on top of the previously open window, so you may want to move it to one side with your mouse). The same thing happens with numbering starting at 4 when using the RStudio application on a Mac. Regardless, you'll see that this becomes the active device: on a Mac the *[*]* is now on the newly opened device window banner; in Windows, the new device's banner will say *(ACTIVE)*, with any previous external device banners now showing *(inactive)*.

At this point, you can enter the usual command to bring up the desired histogram in your just-opened device:

```
R> hist(quakes$stations)
```

If you hadn't used `dev.new`, the histogram would've just overwritten the plot of the spatial locations in your original device. Keep both these plot devices open for the following examples.

23.1.2 Switching Between Devices

To change something in an inactive (but still open) device without affecting the contents of the device that is currently active, we first need to re-activate the inactive one. This is achieved with the `dev.set` command, to which you supply the number of the device you wish to re-activate.

Assume the two devices displaying the plots of the `quakes` data remain open in the state you left them at the end of Section 23.1.1 (if you're using the RStudio editor, remember to pay attention to the different numbering than that which is displayed here). The following code re-activates Device 2 and re-plots the locations of the seismic events with some changes. Now the size of each point is proportional to the number of stations that detected the event, and we have some neater axis labels:

```
R> dev.set(2)
quartz
     2
R> plot(quakes$long,quakes$lat,cex=0.02*quakes$stations,
        xlab="Longitude",ylab="Latitude")
```

Using `dev.set` always confirms the newly active device by printing to the console; the specific text will vary according to your operating system and type of device (the previous output is what you'll see on macOS running *R.app*).

Switching back to the device with the histogram in it (Device 3 under the current numbering), add as a final tweak a vertical line marking the mean number of detecting stations:

```
R> dev.set(3)
quartz
     3
R> abline(v=mean(quakes$stations),lty=2)
```

Figure 23-1 shows the two graphics devices after making these modifications on my macOS system. Keep them open for the next set of examples.

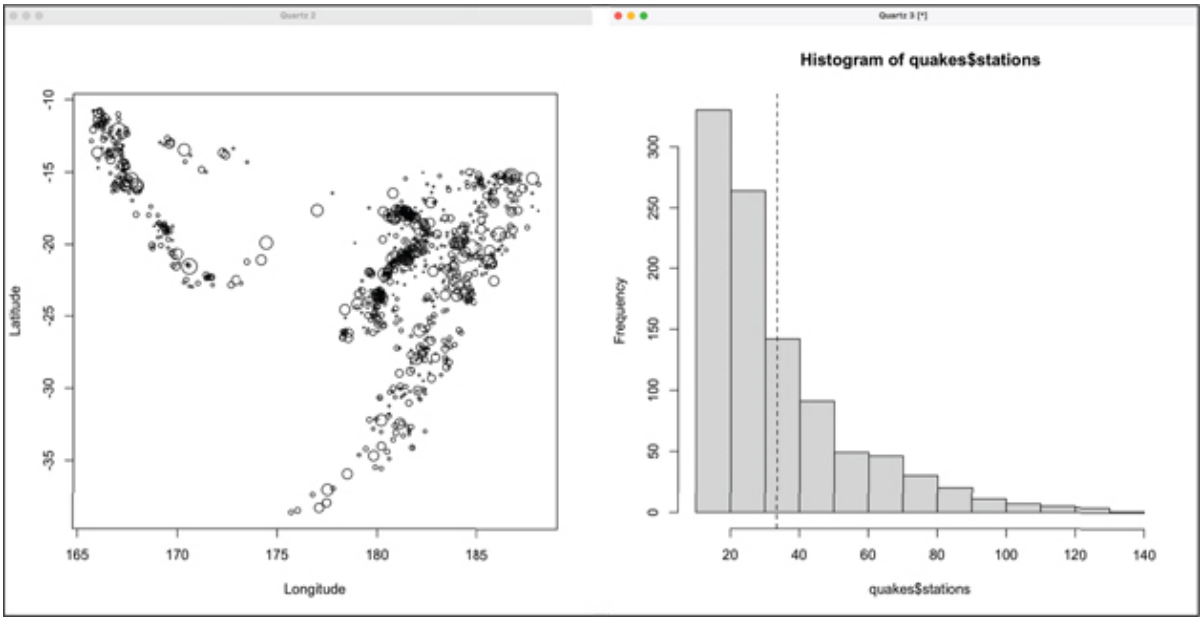


Figure 23-1: My two visible graphics devices, Device 2 (left) and Device 3 (right), showing the final results of producing and manipulating two plots of the `quakes` *data*

23.1.3 Closing a Device

To close a graphics device, either click ✕ with your mouse cursor as you would to close any window, or use the `dev.off` function at the R prompt (you originally saw this command in Chapter 8 when closing a direct-to-file device). Calling `dev.off()` with no arguments simply closes the currently active device. Otherwise, you can specify the device number just as when using `dev.set`.

Continue with the example plots of the `quakes` data; assume they remain in the state you left them in at the end of Section 23.1.2. To close the plot of the spatial locations, leaving the histogram as the active device, call `dev.off` with the appropriate device number (2, in my current instance):

```
R> dev.off(2)
quartz
     3
```

Similar to `dev.set`, the printed output from `dev.off` identifies the active device is after the specified closure.

Now repeat the call without an argument. It will close whatever the active device is. If you've been following along with the `quakes` examples, this will kill the histogram:

```
R> dev.off()
null device
          1
```

When you close the last available actionable device, you're returned to the null device; this is confirmed by the output.

23.1.4 Controlling Multiple Plots in One Device

You can also control the number of individual plots in any one device. There are a few ways to do this; I'll describe the two most direct ways here.

Setting the mfrow (or mfcol) Parameter

Recall there are a number of *graphical parameters* available to tweak features of traditional/base R plots (Section 7.2). Some can be supplied to the relevant plotting command as optional arguments, but more generally, we can use the `par` function to exert control over the behavior of a given graphics device and control precisely how subsequent plots are displayed.

The `mfrow` argument instructs a new (or the currently active) device to "invisibly" or "quietly" divide itself into a grid of the specified dimensions, with each cell holding one plot. You pass the `mfrow` option a numeric integer vector of length 2 in the order of `c(rows,columns)`; its default is `c(1,1)`.

Say you want the two plots of the `quakes` data from the earlier examples—the event locations and the histogram—to be side-by-side in the same device. You can set `mfrow` as a 1×2 grid with the vector `c(1,2)`. This will dictate that any subsequent plots will appear as two side-by-side; `c(1,2)` implies a plotting grid of one row and two columns:

```
R> par(mfrow=c(1,2))
R> plot(quakes$long,quakes$lat,cex=0.02*quakes$stations,
       xlab="Longitude",ylab="Latitude")
R> hist(quakes$stations)
R> abline(v=mean(quakes$stations),lty=2)
```

This results in showing the event locations and the histogram on the left and right, respectively: The creation of new plots will fill the available cells as governed by the value of `mfrow` in a left-to-right (followed by top-to-bottom if there are multiple rows) fashion. Notice that the default square shape of the graphics device means these two plots may look a bit "squashed." Use your mouse to resize the device so that it is roughly twice as wide as it is high. Figure 23-2 shows exactly how the images are displayed once I've done so on my macOS system.

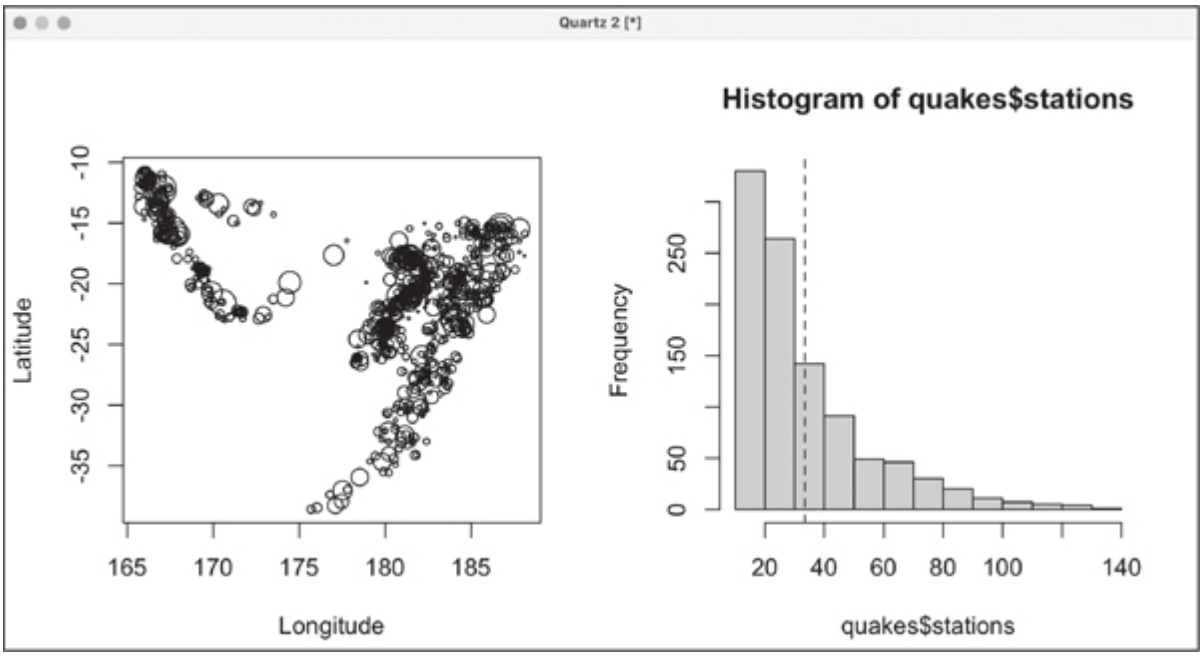


Figure 23-2: Using `mfrow` in `par` to generate a grid of plots in a single graphics device, showing the two plots of the `quakes` data

A close sibling to the `mfrow` parameter is `mfcol`, which works similarly, and its argument takes the same row-column form. However, while `mfrow` fills plot positions in a row-wise manner (left-to-right preceding top-to-bottom), use of `mfcol` dictates plots to be filled column-wise (top-to-

bottom preceding left-to-right). Given that there is only one row of plots, we could've equivalently created the pair of plots shown in Figure 23-2 by first executing the line `par(mfcol=c(1,2))`. As a more instructive comparison, let's run a little example using a `for` loop (Section 10.2.1):

```
R> par(mfrow=c(3,2))
R> for(i in 1:6){ plot(i,i,main=i) }
```

This code results in a set of six placeholder plots (the result is not shown here). Note the positions of the numbers in the titles, 1 through 6, given in the aforementioned row-wise form. Contrast that result with the set of plots obtained by doing everything again, but using `mfcol` instead:

```
R> par(mfcol=c(3,2))
R> for(i in 1:6){ plot(i,i,main=i) }
```

Note that when `par` is used in the way demonstrated here, any tweaks you make will affect *only* the currently active device. Subsequent calls to `dev.new` will open new devices with, for example, `mfrow`/`mfcol` taking their default value equivalent to the "one plot only" arrangement—that is, `c(1,1)`. In other words, if you want to tailor the options of any new graphics device (including the direct-to-file devices seen in Section 8.3.2), you need to set the required values of `par` *after* opening the device but *before* executing any plotting commands.

Defining a Custom Layout

You can refine the arrangements of plots in a single device using the `layout` function. This offers flexible ways to individualize the assignment of individual plots to multiple panels.

Return to the student survey data in `survey` in the `MASS` package, first seen in Section 19.3.2 and used in several examples since. Suppose you want an array of three statistical plots: a scatterplot of height on writing handspan (Section 14.6.1), side-by-side box plots of height split by smoking status (Section 14.4.2), and a bar plot of the frequencies of

exercise of the students (Section 14.1.1). If you want the plots arranged in a square device (as opposed to a single row or column of three plots), using only `mfrow` in `par` may not work best. You could set a square grid with `par(mfrow=c(2,2))`, but you'd end up with a blank space for the fourth cell with no image assigned to it.

When you use `layout`, you provide the dimensions in a matrix `mat` as the first argument; these govern an "invisible" rectangular grid, just like controlling the `mfrow` or `mfcol` options via `par`. The difference is that you can use numeric integer entries in `mat` to tell `layout` which plot number will go where (refer back to Section 3.1.1 for basic details on creating matrices). Now examine the following object:

```
R> lay.mat <- matrix(c(1,3,2,3),2,2)
R> lay.mat
     [,1] [,2]
[1,]    1    2
[2,]    3    3
```

The dimensions of this matrix imply a 2×2 grid of plotting cells, but the values inside `lay.mat` tell R that you want plot 1 to take the upper-left cell, plot 2 to take the upper-right cell, and plot 3 to stretch itself over the two bottom cells.

Calling `layout` as follows will either silently initialize the active device to respect `lay.mat` or prepare a new one (if the null device is the only device currently available):

```
R> layout(mat=lay.mat)
```

If you're ever unsure of the result of your specification, you can use the `layout.show` function to see how plots will be placed. The following line produces the image on the left of Figure 23-3:

```
R> layout.show(n=max(lay.mat))
```

Then, once you've loaded the `MASS` package by calling `library("MASS")` so you can access `survey`, run the following lines to place the desired

plots. These will appear in the order the plotting commands are executed, matched to the integers in `lay.mat`:

```
R> plot(survey$Wr.Hnd,survey$Height,
        xlab="Writing handspan",ylab="Height")
R> boxplot(survey$Height~survey$Smoke,
        xlab="Smoking frequency",ylab="Height")
R> barplot(table(survey$Exer),horiz=TRUE,main="Exercise")
```

Note that if you'd previously closed down the plot arising from `layout.show`, you'll need to reinitialize a new device with the same call to `layout` for these three plots to display as intended. The result should look like that which appears on the right-hand side of Figure 23-3.

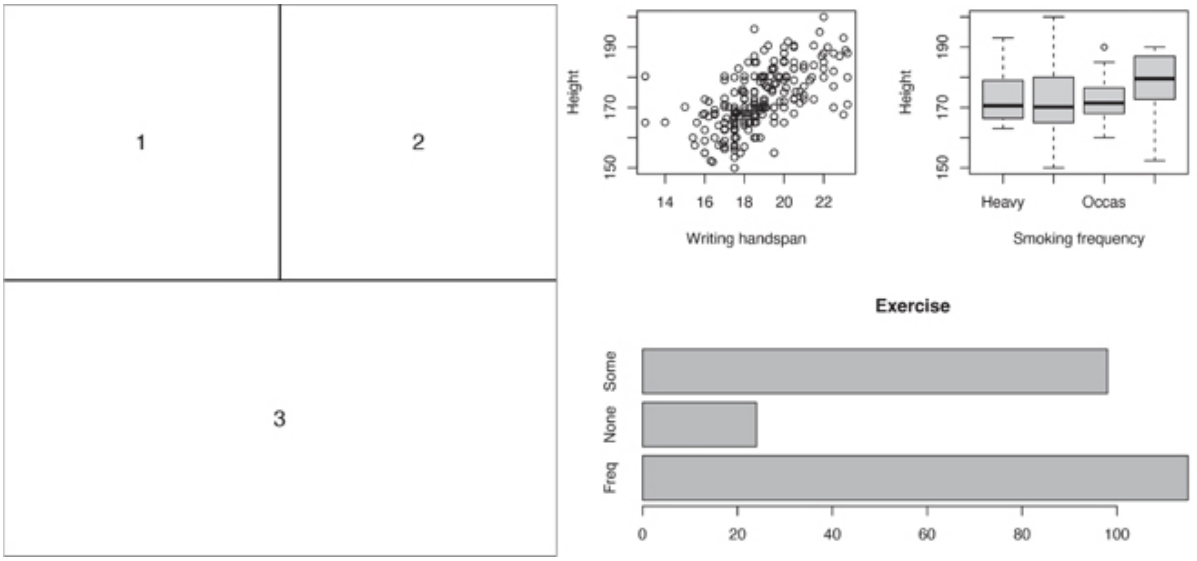


Figure 23-3: Left: Using `layout.show` to visualize the planned plotting layout and order. Right: Demonstrating three plots of the `survey` data arranged according to `lay.mat` through `layout`.

Probably the biggest benefit of `layout` is its ability to relax the rigidity of the standard grid layout achievable with `mfrow`/`mfcol` via `par`. Additional arguments to `layout`—namely, `widths` and `heights`—even allow you to preset the relative widths and heights of the cells as structured in the `mat` argument. See the documentation in `?layout` for details; you'll find some other examples of its flexibility at the bottom of that help file.

NOTE

A consequence of the two methods of creating multi-panel plots discussed here is the inability to edit a previous plot once you've finished it and moved on to the next. For this, there is the built-in `split.screen` *function, which allows you to set up several "screens" in a single device and switch between them. However, this method requires a lot of extra coding and generally doesn't behave well with regard to plotting regions and margins (see the next section) in R. Many users (myself included) prefer to remain with* `layout`*, even if it occasionally means going through a bit of trial and error to perfect things.*

23.2 Plotting Regions and Margins

Although the main concern when plotting is the dataset or model being visualized, it's additionally important to ensure the plot is annotated clearly and accurately to facilitate correct interpretation. To help do this, you need to know how to manipulate and draw in all visible areas of a given device, not just the area where your data lie.

For any single plot created using base R graphics, there are three regions that make up the image:

- The *plot region* is all you've dealt with so far. This is where your actual plot appears and where you'll usually draw your points, lines, text, and so on. The plot region uses the *user coordinate system*, which reflects the value and scale of the horizontal and vertical axes.
- The *figure region* is the area that contains the space for your axes, their labels, and any titles. These spaces are also referred to as the *figure margins*.
- The *outer region*, also referred to as the *outer margins*, is additional space around the figure region that is not included by default but can be specified if necessary.

You can explicitly measure and set margin space in a few different ways. One typical way is in terms of *lines*; this refers to the number of "lines of text" that can fit on top of one another parallel to each edge. You specify these as vectors of length 4 in a particular (clockwise) order; each of the elements corresponds to one of the four sides: `c(bottom, left, top, right)`. The graphical parameters `oma` (outer margin) and `mar` (figure margin) control these amounts; like `mfrow`/`mfcol`, they are initialized through a call to `par` before you begin to draw any new plot.

23.2.1 Default Spacing

You can find your default figure margin settings with a call to `par` in R:

```
R> par()$oma
[1] 0 0 0 0
R> par()$mar
[1] 5.1 4.1 4.1 2.1
```

Note that `oma=c(0, 0, 0, 0)` means there's no outer margin set by default. The default figure margin space is `mar=c(5.1, 4.1, 4.1, 2.1)`—in other words, 5.1 "lines of text" on the bottom, 4.1 on the left and top, and 2.1 on the right.

To illustrate these regions, consider the image on the left of Figure 23-4, created in a fresh graphics device with the following:

```
R> plot(1:10)
R> box(which="figure",lty=2,lwd=3)
```

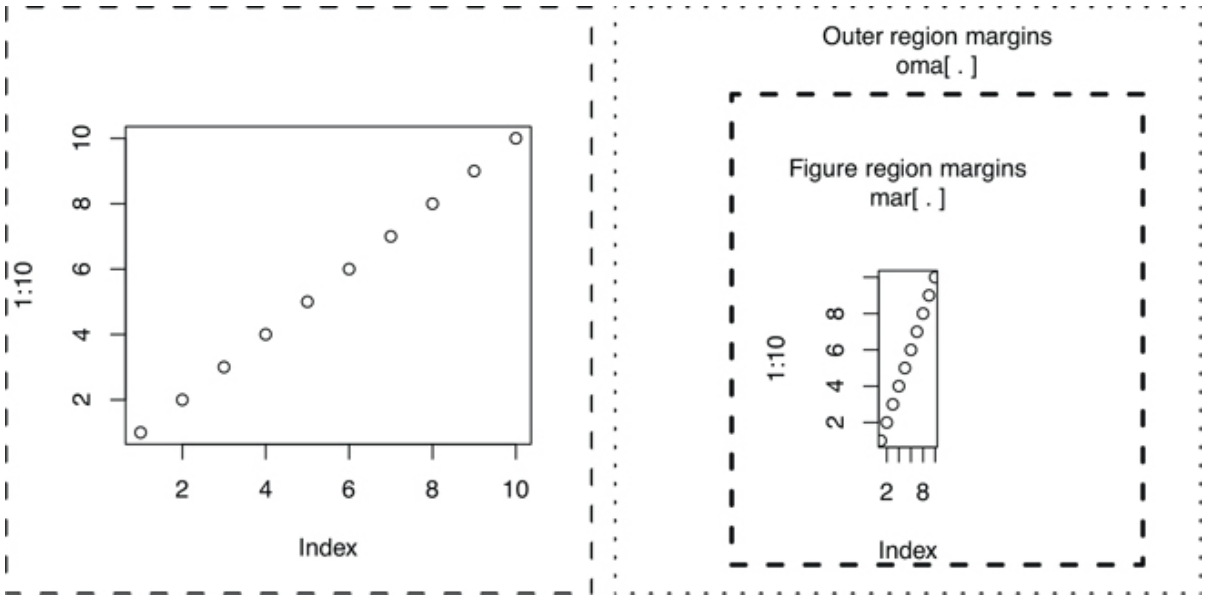


Figure 23-4: Illustrating graphical device regions as treated by traditional (base) R graphics; solid-line boxes show the plot region, dashed-line boxes show the figure region, and a dotted-line box shows the outer region area. Left: Default settings. Right: User specification, through `par`*, of the outer and figure margin areas in "lines of text" via* `oma` *and* `mar`*, respectively.*

If you use the `box` function with the optional argument `which` set to `"figure"`, it shows you the figure region. (The additional specification of `lty=2` draws dashed lines; stipulating `lwd=3` thickens them somewhat.)

If you're looking at this plot in your onscreen graphics device, you should note the dashed lines snug up against the window edges. Examining the default values in `mar` shows that they correctly correspond to the spacing on the four sides of the plot region (given with the default solid box), relatively speaking. The widest figure margin, parallel to the bottom of the plot region, is 5.1 lines; the narrowest figure margin, parallel to the right of the plot region, is 2.1 lines.

23.2.2 Custom Spacing

Let's produce the same plot but with tailored outer margins so that the bottom, left, top, and right areas are one, four, three, and two lines, respectively, and the figure margins are four, five, six, and seven lines. The result of the following code is given on the right of Figure 23-4:

```
R> par(oma=c(1,4,3,2),mar=4:7)
R> plot(1:10)
R> box("figure",lty=2,lwd=3)
R> box("outer",lty=3,lwd=3)
```

Notice that the irregular margins have squashed the plot region in the default-sized square device to accommodate the defined spacing. If you set graphical parameters that squash the plot region into nonexistence (or attempt to plot something in a device so small that the plot itself is unable to render), R will throw an error stating `figure margins too large`.

Since you'd usually manipulate margin space to accommodate particular annotations of the plot, let's look at the `mtext` function, used specifically to produce text in the figure or outer margins. By default, the argument `outer` is `FALSE`, meaning the text will be written in the figure margin. Setting `outer=TRUE` positions the text in the outer region. If you've kept the most recent plot open, the following lines provide the additional margin annotation visible on the right of Figure 23-4:

```
R> mtext("Figure region margins\nmar[ . ]",line=2)
R> mtext("Outer region margins\noma[ . ]",line=0.5,outer=TRUE)
```

Here, you provide the text you want written in a character string as the first argument (notice the use of the escape sequence `\n` to split the text onto two lines, as described in Section 4.2.3), and the argument `line` instructs how many lines of space away from the inside border the text should appear. There's also an optional argument `side` in `mtext`, which dictates where the text appears. It defaults to `3`, setting the text at the top, but you can set `side=1` to place the text on the bottom, use `side=2` to set it on the left, and use `side=4` to set it on the right. Look to `?mtext` for details on even more arguments available for the margin text.

At this point you might also like to investigate the `title` function, which is a specialized implementation of `mtext` often used if figure margin annotation for the four axes of a plot (beyond the basic capabilities of specifying things such as `main`, `xlab`, or `ylab`) is the primary

concern. You'll see this function used in examples in Sections 23.5.2 and 23.6.

23.2.3 Clipping

Controlling *clipping* allows you to draw in or add elements to the margin regions with reference to the user coordinates of the plot itself. For example, you might want to place a legend outside the plotting area or draw an arrow that extends beyond the plot region to embellish a particular observation.

The graphical parameter `xpd` controls clipping in base R graphics. By default, `xpd` is set to `FALSE`, so all drawing is clipped to the available plot region only (with the exception of special margin-addition functions such as `mtext`). Setting `xpd` to `TRUE` allows you to draw things outside the formally defined plot region into the figure margins but not into any outer margins. Setting `xpd` to `NA` will permit drawing in all three areas: plot region, figure margins, and the outer margins.

For an example, let's revisit the built-in `mtcars` data object first used in Section 9.3.1:

```
R> dev.new()
R> par(oma=c(1,1,5,1),mar=c(2,4,5,4))
R> boxplot(mtcars$mpg~mtcars$cyl,xaxt="n",ylab="MPG")
R> box("figure",lty=2,lwd=3)
R> box("outer",lty=3,lwd=3)
R> arrows(x0=c(2,2.5,3),y0=c(44,37,27),x1=c(1.25,2.25,3),y1=c(31,22,20),
          xpd=FALSE)
R> text(x=c(2,2.5,3),y=c(45,38,28),
        c("4 cyl. cars","6 cyl. cars","8 cyl. cars"),
        xpd=FALSE)
```

The result of this code is given as the top-left image in Figure 23-5. I've defined the device region itself with particular figure and outer margins for the purpose of illustration. The plotting of the horizontal axis is suppressed with `xaxt="n"` in the call to `boxplot`, and calls to `box` add the boundaries of the figure and outer margins (dashed and dotted lines,

respectively). Lastly, calls to `arrows` and `text` point to and annotate each boxplot; the label for fourcylinder cars extends into the outer margin, the label for six-cylinder cars extends into the figure region, and the label for eight-cylinder cars remains contained within the plot region.

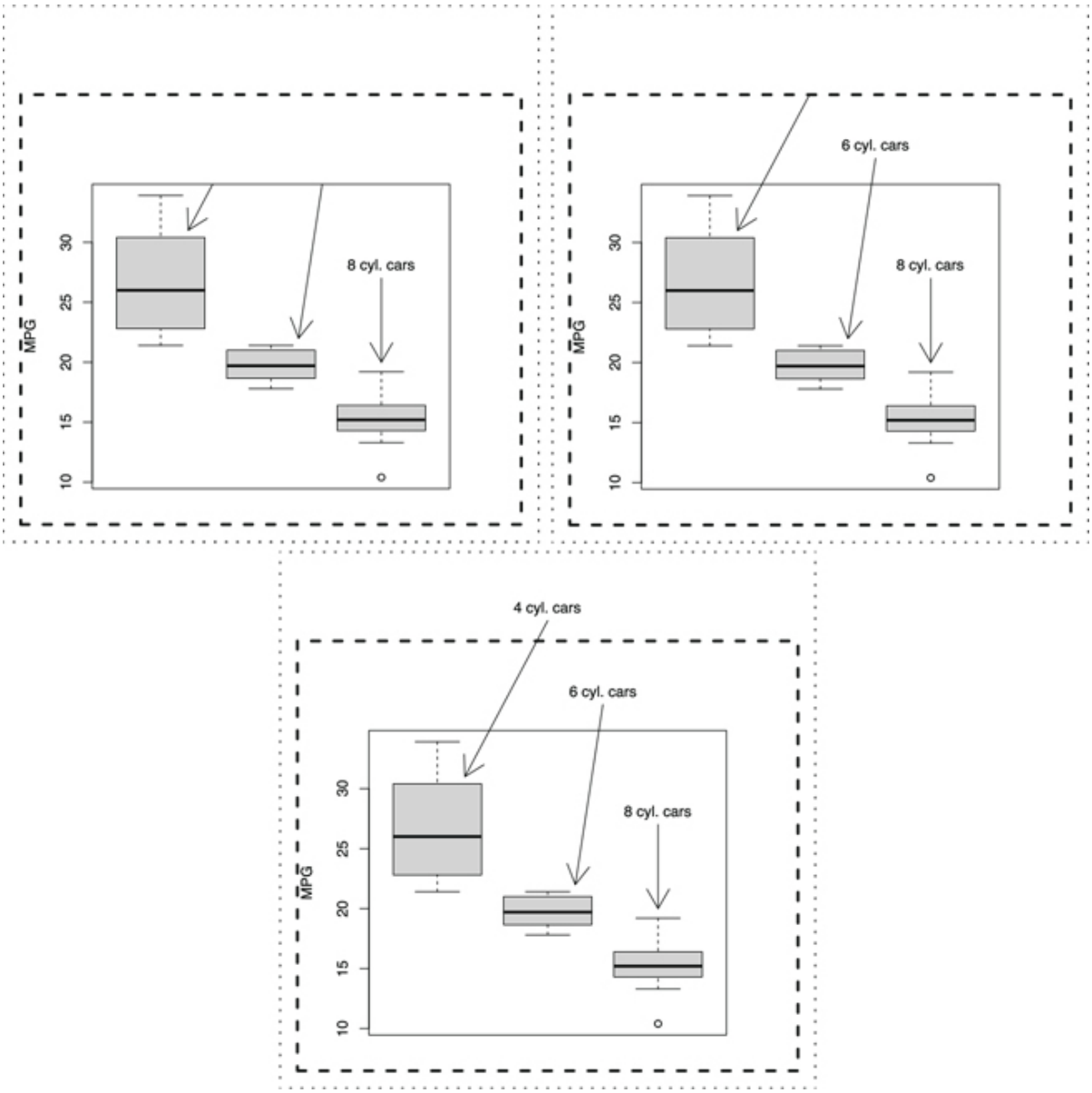


Figure 23-5: Illustrating the behavior of setting `xpd=FALSE` (top left, default), `xpd=TRUE` (top right), and `xpd=NA` (bottom) to enable drawing in figure and outer margins with respect to the user coordinates of the plot region

Note that the graphical parameter `xpd` is specified only in the two "addto-current-plot" functions `arrows` and `text`, explicitly set as the

default `FALSE`. This means all plotting is restricted to the plot region; the text and a portion of the arrows corresponding to four- and six-cylinder cars have been chopped off, or clipped, to this plot region.

Now set `xpd=TRUE` in the calls to `arrows` and `text` and run the entire code chunk again. You'll get a plot that looks like the image in the top right of Figure 23-5. This version of the plot allows the label for the six-cylinder car to be printed in the figure margin instead of being chopped off, but the label for the four-cylinder cars is still clipped given that it falls in the outer margin. Finally, re-running the code chunk with `xpd=NA` in `arrows` and `text` produces the lower plot in Figure 23-5, where drawing beyond the plot and figure margins is permitted.

These effects are usually desirable when you need to annotate the main plot somehow, especially when there isn't enough space in the plot region to squeeze in any additions. Although their code wasn't shown at the time, plots that I've created in earlier chapters, such as those in Figure 16-6 on page 411 (where the legend sits outside the main plot) and Figure 17-3 on page 448 (where I annotated the critical values), were created specifying `xpd=TRUE` in the relevant functions (`legend`, `text`, `segments`, and `arrows`).

As demonstrated, you'd typically set `xpd` in the specific commands that intend to add to the margins (in other words, on a line-by-line basis), so only the results of that particular command will be produced with the given clipping rule. This offers a bit more control over what's visible outside the plot region. You can, however, set `xpd` alongside `oma` and `mar` in the initial call to `par` to make the value of `xpd` "universal" to that device.

23.3 Point-and-Click Coordinate Interaction

Your dealings with the graphics device don't need to be solely commandbased. Under typical circumstances, R can read mouse clicks you make inside the device.

23.3.1 Retrieving Coordinates Silently

The `locator` command allows you to find and return user coordinates. To see how it works, first execute a call to `plot(1,1)` to bring up a simple plot with a single point in the middle. To use `locator`, you simply execute the function (with no arguments for default behavior), which will "hang" the console without returning you to the prompt. Then, on an active graphics device, your mouse cursor will change to a + symbol (depending on your setup, you may need to first click your device once to bring it to the foreground of your computer desktop). With your cursor as the +, you can perform a series of left mouse clicks inside the device, and R will silently record the precise user coordinates. To stop, right-click or press ESC to terminate the command (other options to stop are system dependent; see the help file `?locator`). Once you do so, the coordinates you identified in the device are returned as a list with components `$x` and `$y`. These are printed to the console screen unless you specifically assign the call to `locator` to an R object.

On my machine, I clicked four spots at arbitrary locations around the plotted point at (1, 1), from top left clockwise around to the bottom left. Nothing is shown in the plotting window while you do this; upon completion, the following was the output printed to my console:

```
R> plot(1,1)
R> locator()
$x
[1] 0.8275456 1.1737525 1.1440526 0.8201909

$y
[1] 1.1581795 1.1534442 0.9003221 0.8630254
```

This "silent" use of `locator` is useful if you need to, for example, identify approximate user coordinates in the plot region where you need to place future annotations.

23.3.2 Visualizing Selected Coordinates

You can also use `locator` to plot the points you select either as individual points or as lines. On my machine, I executed the following code,

clicking 12 random locations in the graphics device before terminating the call to `locator`:

```
R> plot(1,1)
R> Rtist <- locator(type="o",pch=4,lty=2,lwd=3,col="red",xpd=TRUE)
R> Rtist
$x
[1] 0.5013189 0.6267149 0.7384407 0.7172250 1.0386740 1.2765699
[7] 1.4711542 1.2352573 1.2220592 0.8583484 1.0483300 1.0091491

$y
[1] 0.6966016 0.9941945 0.9636752 1.2819852 1.2766579 1.4891270
[7] 1.2439071 0.9630832 0.7625887 0.7541716 0.6394519 0.9618461
```

Figure 23-6 shows the result of this activity.

Note that drawing using `locator` requires you to specify the plot `type`, as covered in Chapter 7. Selecting `type="o"` (as opposed to the silent default, `type="n"`) produces the overplotted points and lines in Figure 23-6. For just points, use `type="p"`; for lines, use `type="l"`. The graphical parameters controlling other relevant features, such as point/line type and color, can also be used, as you've seen in conventionally produced plots in Chapter 7. I also used `xpd=TRUE`, shown earlier, to allow the `locator` points or lines to protrude into the figure region margins. The call to `locator` is directly assigned to a new object `Rtist`, illustrating how you can use the clicked coordinates later if needed.

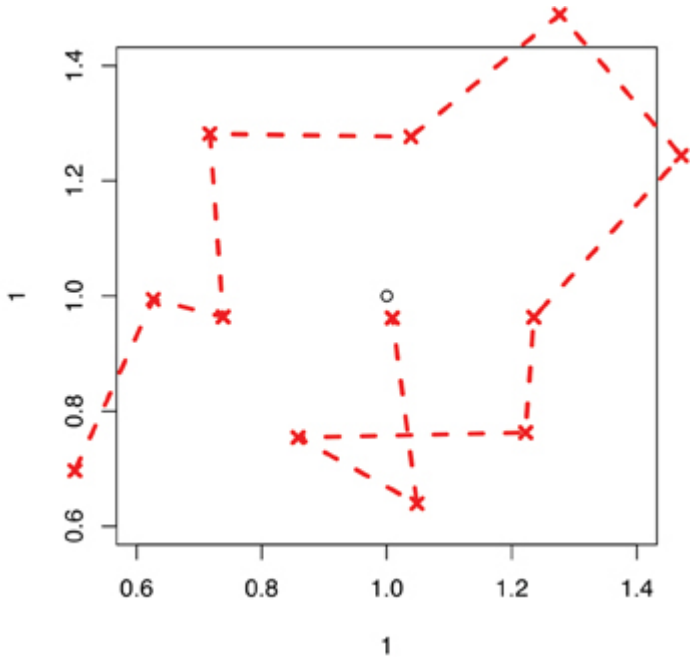


Figure 23-6: Using `locator` to draw an arbitrary sequence of overplotted points and lines

23.3.3 Placing Ad Hoc Annotations

The `locator` function also allows you to place ad hoc annotations, such as legends, on your plot; remember, since `locator` returns valid R user coordinates, these results can directly form the positional argument of most standard annotation functions.

Return to the student survey data in the `MASS` package; first, load the package by calling `library("MASS")`. The following call produces the basic scatterplot of student height by a handspan and sex (a version of which previously appeared in Section 21.3.3):

```
R> plot(survey$Height~survey$Wr.Hnd,pch=16,
        col=c("gray","black")[as.numeric(survey$Sex)],
        xlab="Writing handspan",ylab="Height")
```

When this plot was used in Section 21.3.3 (Figure 21-1 on page 565), I simply used the string `"topleft"` to position a legend identifying the two different types of points. This time, call the following code and use your mouse cursor to left-click somewhere in an empty area of the scatterplot when the + cursor appears on the graphics device:

```
R> legend(locator(n=1),legend=levels(survey$Sex),pch=16,
          col=c("gray","black"))
```

I chose to click the blank space above the cloud of points, producing the result in Figure 23-7.

The optional argument to `locator`, `n`, takes a positive integer for an upper limit on how many points you want to select; it defaults to `512`. If you specify `n=1`, `locator` will automatically terminate after you left-click once in the device, so you don't need to manually exit the function with a right-click.

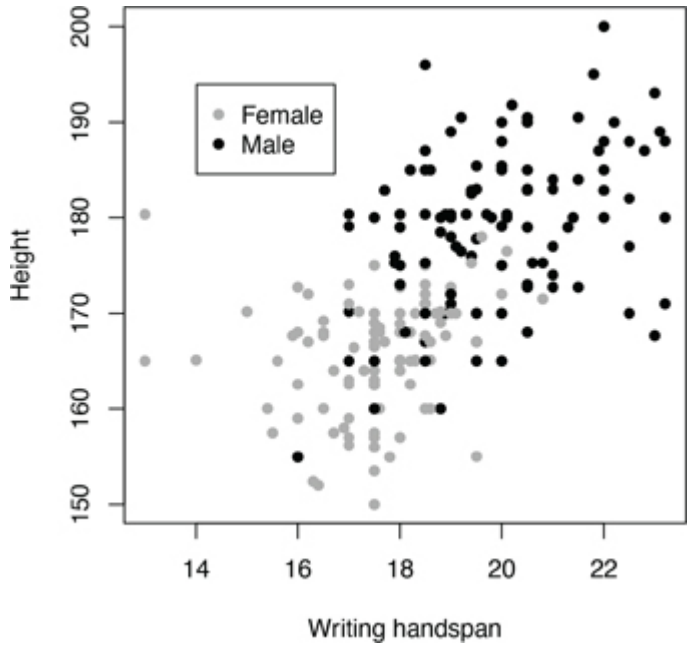


Figure 23-7: An ad hoc placement of a legend on a scatterplot of the `survey` data

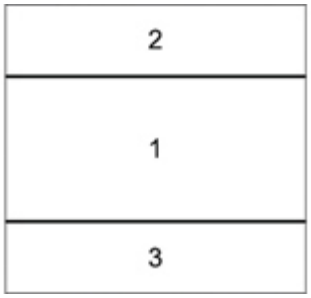
Exercise 23.1

a. In Section 20.5.4 (page 548), I provided code showing a simple linear model fitted with a categorical predictor being treated as continuous (the `mtcars` data with `mpg` as the response and `cyl` as the explanatory variable). Reproduce

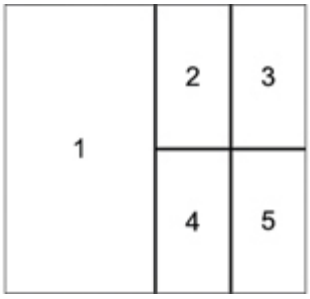
the side-by-side box plots and the scatterplot (with fitted line) from Figure 20-6 on page 549, but this time, use `mfrow` to present the two plots, vertically stacked, in one device.

b. Use `layout` to create the appropriate layout matrices in order to reproduce the following three plots (as they appear in a square device) via `layout.show`:

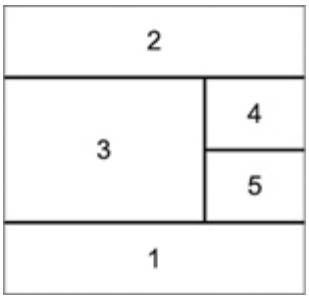
i.



ii.



iii.



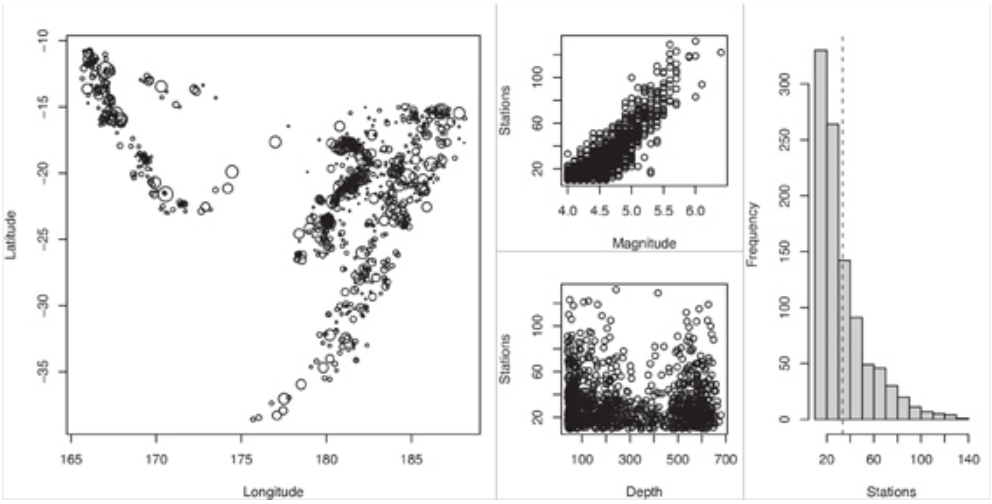
c. Run the following line to open a new graphics device of dimension 9×4.5 inches (if you're using RStudio, you may need to resize your graphics device manually with your mouse so that it's roughly twice as wide as it is high):

```
R> dev.new(width=9,height=4.5)
```

In this new device, use `layout` to set the following layout, and confirm with `layout.show`:



Now use this device to produce the following combined set of plots of the built-in `quakes` data, including the framing around each individual plot (further details to follow):



To achieve this, note the following:

- You may wish to refamiliarize yourself with the `quakes` data object; if so, execute `?quakes` at the prompt or review the examples in Section 23.1.
- After you open the device and set the layout, use `par` to reset plot margins to be four lines, four lines, two lines, and one line of space on the bottom, left, top, and right, respectively.
- You can ignore the aspect ratio in the plot of the spatial coordinates for simplicity.

- After each plot, use `box` to add a gray box corresponding to the figure region to achieve the visible partitions.
- Plots 1 and 4 are the same as the two plots shown in Figure 23-1 (page 650).
- Plots 2 and 3 are scatterplots showing the number of detecting stations on the y-axis, with magnitude and depth on the x-axis, respectively.
- Do not place main titles on any plots (that is, set `main=""`), and ensure the axis titles are neat (compared to their defaults).

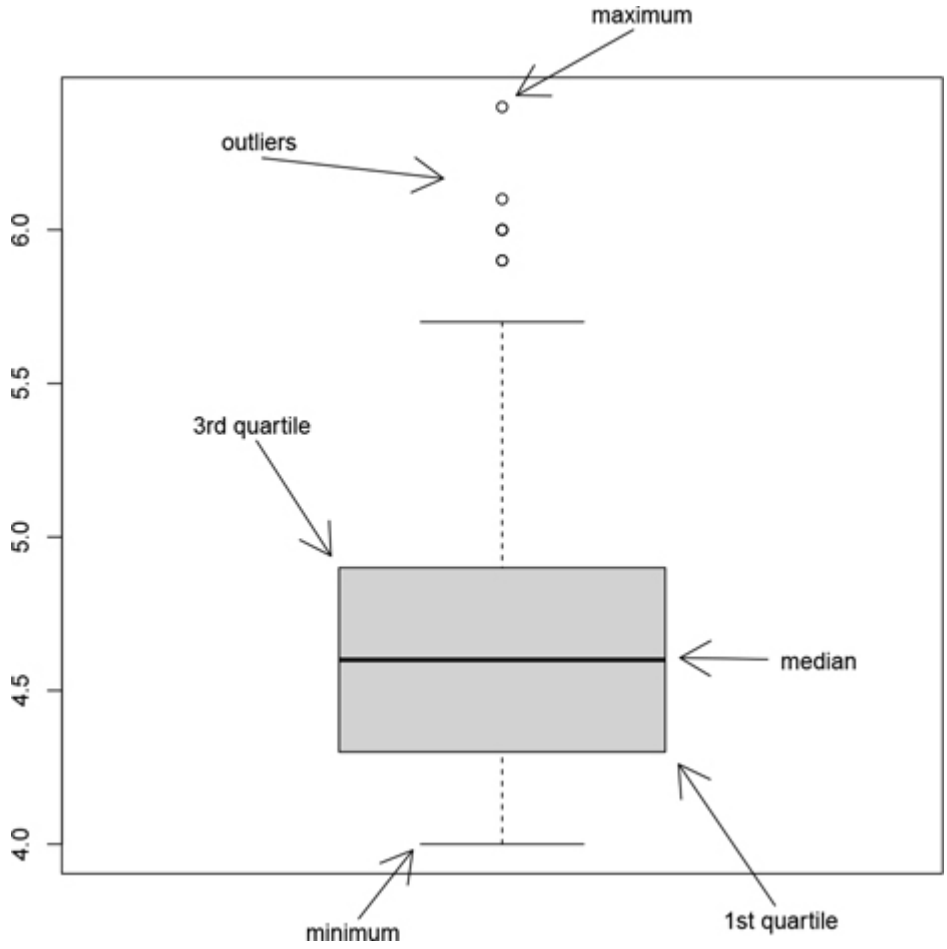
d. Use skills from Chapter 11 and newly obtained knowledge from the preceding content of this chapter to write a little R function named `interactive.arrow`. The purpose of this function is to superimpose an arrow upon any base R plot using two clicks of your mouse. The details are as follows:

   - The crux of your function will be the use of `locator` (Section 23.3.1) to read exactly two mouse clicks. You may assume a suitable active graphics device is already open whenever the function is called. The first click should mean the beginning of the arrow, and the second click should mean the tip of the arrow (where it's pointing to).
   - In the function, the coordinates returned by `locator` should be passed to `arrows` (Section 7.3) to do the actual drawing.
   - The function should take an ellipsis as its first argument, intended to hold additional arguments to be passed directly to `arrows`.
   - The function should take an optional logical argument `label`, which defaults to `NA` but should be

intended to have an optional character string. If `label` is not `NA`, then `locator` should be invoked once more (separately, after drawing the arrow) to select exactly one coordinate. That point will be passed to `text` so that the user can additionally place the character string given to `label` as an annotation (intended to be for the interactively placed arrow). The call to `text` should consistently allow completely relaxed clipping via use of `xpd` as an argument (in other words, any text added in this fashion will still be visible in the figure region and outer margins, if there are any).

Take another look at the rightmost plot of Figure 14-10 on page 350; this is a stand-alone box plot of the magnitude data from the `quakes` data frame. Arrows and labels were superimposed externally pointing out the various statistics summarized by a box plot. Create the same box plot, but use `interactive.arrow` to annotate the same features to your own satisfaction (you'll have to provide an appropriate value for `xpd` and pass it through the ellipsis if you want your arrows/labels to be drawable in the margins).

My result looks like this:



23.4 Customizing Traditional R Plots

Now that you're familiar with the way R places and handles plots in the graphics device, it's time to focus on some of the more technical features of base R graphics. So far, you've largely left the default settings of these in place when plotting.

23.4.1 Graphical Parameters for Style and Suppression

If you want ultra-fine control over an R plot, you'll typically want to begin with a "clean slate." To do this, you need to be aware of the default settings of certain graphical parameters when calling a plotting function and how to alter them if need be, such as suppressing boxes and axes.

For an example image, let's plot MPG against horsepower (from the ready-to-use `mtcars` dataset seen in previous chapters) and set each plotted point to be sized proportionally to the weight of each car. For convenience, create the following objects:

```
R> hp <- mtcars$hp
R> mpg <- mtcars$mpg
R> wtcex <- mtcars$wt/mean(mtcars$wt)
```

The last object is the car weight vector; each individual value is scaled by the sample mean of all weights. This creates a vector where cars less than the average weight have a value < 1 and cars more than the average weight have a value > 1, making it ideal for the `cex` parameter to scale the size of the plotted points accordingly (refer to Chapter 7).

Let's start by focusing on some more graphical parameters usually used in the first instance of a plotting function, such as a call to `plot`, which pave the way for using the `box` and `axis` commands. Executing the following line gives you the default appearance of the plot and its box, axes, and axis labels; this is shown as the leftmost image in Figure 23-8:

```
R> plot(hp,mpg,cex=wtcex)
```

There are two axis "styles," controlled by the graphical parameters `xaxs` and `yaxs`. Their sole purpose is to decide whether to impose the small amount of additional horizontal and vertical buffer space that's present at the ends of each axis to prevent points being truncated at the edges of the plotting region. The default, `xaxs="r"` and `yaxis="r"`, is to include that space. The alternative, setting one or both of these to `"i"`, instructs the plot region to be *strictly* defined by the upper and lower limits of the data (or by those optionally supplied to `xlim` and/or `ylim`), that is, with *no* additional padding space.

For example, the following line produces the middle plot in Figure 23-8:

```
R> plot(hp,mpg,cex=wtcex,xaxs="i",yaxs="i")
```

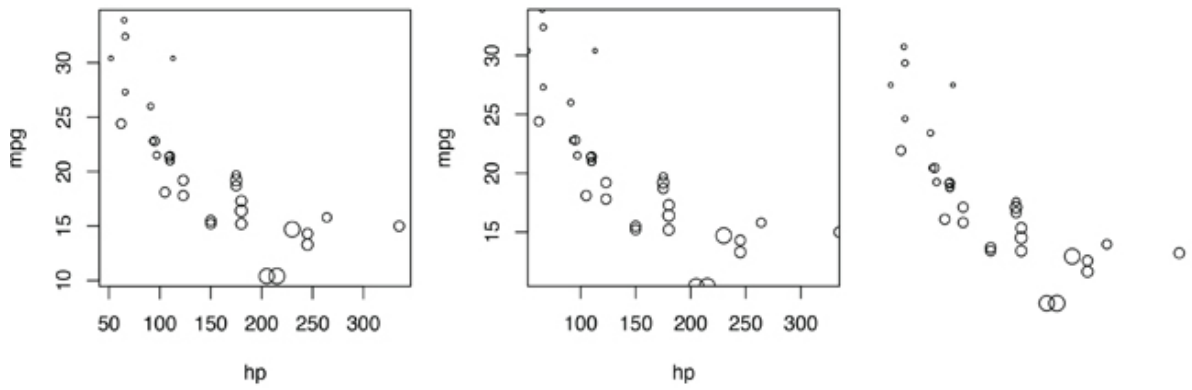


Figure 23-8: Plotting MPG against horsepower for the `mtcars` data; point size proportional to car weight, using a call to `plot` only. Left: Default appearance. Middle: Setting `xaxs="i"` and `yaxs="i"` to prevent the buffer spacing on the limits of the axes. Right: Using `xaxt`, `yaxt`, `xlab`, `ylab`, and `bty` to suppress all box, axis, and label drawing (alternatively achieved by setting `axes=FALSE` and `ann=FALSE`).

This plot is almost the same as the default, but note now that there's no padding space at the end of the axes; the most extreme data points sit right on the axes. Generally, the default axis style `"r"` is fine, but on occasions where you need precise control over the axes' scales and the corresponding plot region, that additional buffer space can be problematic. On those occasions, you'll often see `xlim`/`ylim` being used in conjunction with `xaxs="i"`/`yaxs="i"`.

If you want full control over the specific appearance of any boxes, axes, and their labels, you'll want to start a plot with none of these and add them as per your design. The rightmost plot in Figure 23-8 is the result of suppressing the default drawing of these by a call to either

```
R> plot(hp,mpg,cex=wtcex,xaxt="n",yaxt="n",bty="n",xlab="",ylab="")
```

or

```
R> plot(hp,mpg,cex=wtcex,axes=FALSE,ann=FALSE)
```

This code reveals that you can achieve this either by setting the parameters `xaxt`, `yaxt`, and `bty` to `"n"` and setting the default axis labels `xlab` and `ylab` to the empty string `""`, or by simply setting both `axes` and `ann` to `FALSE` (the former is able to specified only as a plot-specific argument and suppresses all axes and the box, while the latter suppresses any axis

titles). Although the first way is a bit more of a chore to remember, it affords you greater flexibility in suppressing each aspect of a given plot (as opposed to the "total" suppression enforced by the second approach).

23.4.2 Box Customization

When you're starting with a suppressed-box or suppressed-annotation plot, to add a box specific to the current plot region in the active graphics device, use the `box` function, specifying its type with `bty`. For example, if you start with a plot like the one on the right of Figure 23-8 (just run the most recent line of code to get this), then additionally calling the following line provides you with the image given on the left of Figure 23-9:

```
R> box(bty="u")
```

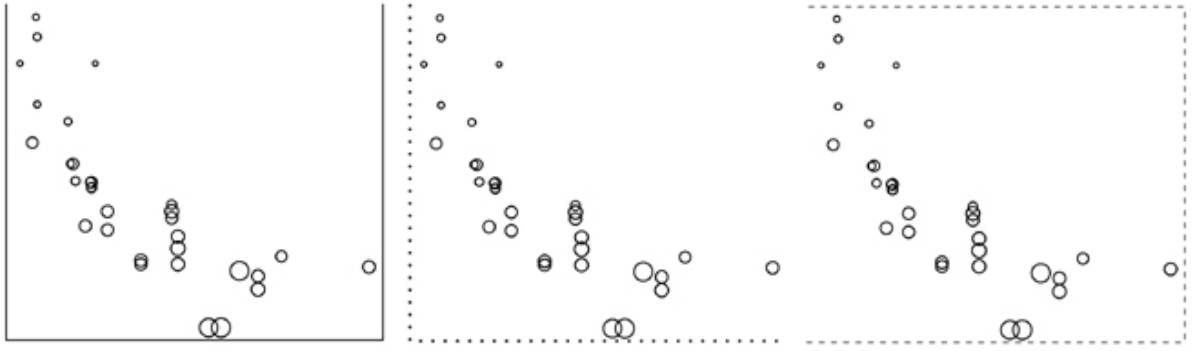


Figure 23-9: Various box configurations added to the `mtcars` scatterplot

The `bty` argument is supplied a single character: `"o"` (default), `"l"`, `"7"`, `"c"`, `"u"`, `"]"`, or `"n"`. The help file entry for `bty` in `?par` (this argument is a general graphical parameter) tells you that based on one of these values, the resulting box boundaries will follow the appearance of the corresponding uppercase letter, with the exception of `"n"` (which, as you saw a moment ago, will suppress the box).

You can use other relevant parameters that you've met already, such as `lty`, `lwd`, and `col`, to further control the appearance of a box. Replot the

data as on the right of Figure 23-8 and then call the following to produce the middle image of Figure 23-9:

```
R> box(bty="l",lty=3,lwd=2)
```

The final example on the right of Figure 23-9 is created with this line:

```
R> box(bty="]",lty=2,col="seagreen4")
```

23.4.3 Axis Customization

Once you have the box the way you want it, you can focus on the axes. The `axis` function allows you to control the addition and appearance of an axis on any of the four sides of the plot region in greater detail. The first argument it takes is `side`, provided with a single integer: `1` (bottom), `2` (left), `3` (top), or `4` (right). These numbers are consistent with the positions of the relevant margin-spacing values when you're setting graphical parameter vectors like `mar`.

The first thing you might want to change on an axis is where the tick marks are drawn. By default, R uses the built-in function `pretty` to find a "neat" sequence of values for the scale of each axis, but you can set your own by passing the `at` argument to `axis`. The following lines make use of the objects created in Section 23.4.1 to produce the plot on the left of Figure 23-10:

```
R> hpseq <- seq(min(hp),max(hp),length=10)
R> plot(hp,mpg,cex=wtcex,xaxt="n",bty="n",ann=FALSE)
R> axis(side=1,at=hpseq)
R> axis(side=3,at=round(hpseq))
```

First, an evenly spaced sequence of 10 values spanning the range of `hp` is stored as `hpseq`. The initial call to `plot` suppresses the x-axis, the box, and any default axis labels; however, the y-axis is permitted to appear as per its default. Then `axis` is instructed to draw the x-axis (`side=1`), with tick marks at `hpseq`. To provide a comparison to that, an axis is also

drawn along the top (`side=3`), but this time the tick marks are drawn at `hpseq` after each of its values is rounded to the nearest integer.

As you can see on the left of the figure, the custom x-axis I've created on the bottom shows 10 tick marks at the sequence of values supplied to `at`. R may suppress some of the labels so they don't overlap one another, which has occurred here. Since these "decimal" values might not be aesthetically pleasing, the axis that's been drawn along the top has tick marks drawn at the nearest integers of `hpseq`, achieved using `round` in the final call to `axis` shown earlier. Although, strictly speaking, this now means that the tick marks are no longer exactly evenly spaced, the rounded values mean shorter default axis labels that can all be displayed in the current device.

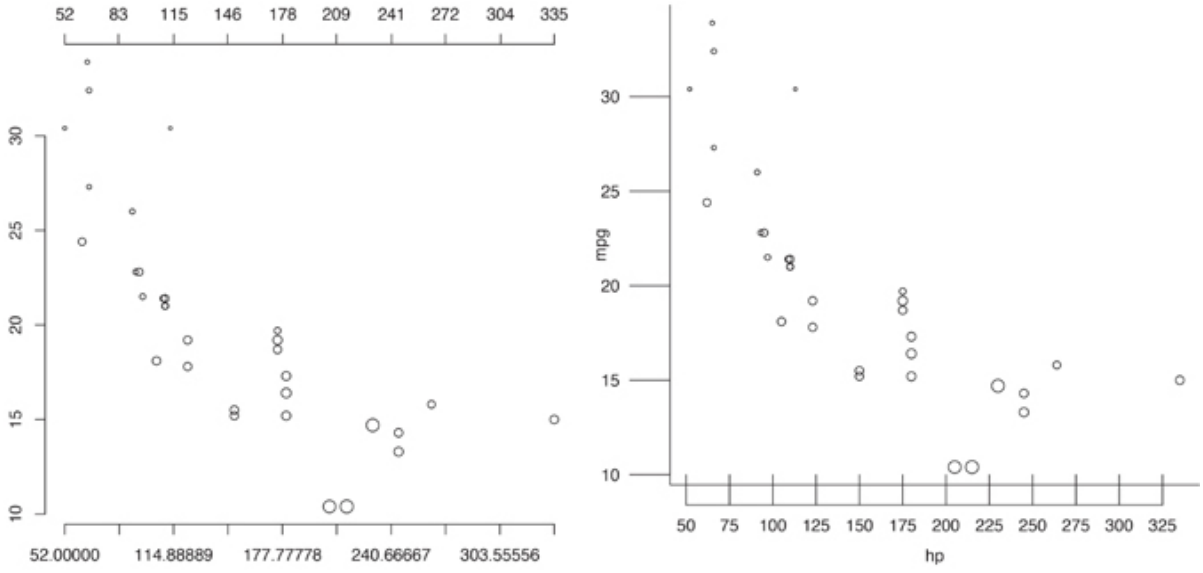


Figure 23-10: Customizing axes of the `mtcars` scatterplot

These difficulties show that tick mark locations are generally best left to R, unless you have specific axis values that you know you want marked out; you'll see an example of this in Section 23.6. For now, let's look at some other tweaks you can make to your axes. In particular, the `tcl` (length of the ticks), `las` (orientation of the labels), and `mgp` (axis spacing) parameters are among those more frequently used. The following code uses the same objects as in the previous example to create the plot on the right of Figure 23-10:

```
R> hpseq2 <- seq(50,325,by=25)
R> plot(hp,mpg,cex=wtcex,axes=FALSE)
R> box(bty="l")
R> axis(side=2,tcl=-2,las=1,mgp=c(3,2.5,0))
R> axis(side=1,at=hpseq2,tcl=1.5,mgp=c(3,1.5,1))
```

After a new sequence, `hpseq2`, is defined as all integers that lie within the recorded range of the data and are evenly spaced by 25 units, the plot is initialized. The box and axes are suppressed, but the default variable names (`"mpg"` and `"hp"`) remain as axis titles.

Next, an L-shaped box and the y-axis (`side=2`) are added. In the latter, the `tcl` parameter governs the length of each tick mark in "parallel lines of text" (recall this is a standard unit of measurement for margin spacing in an R plot); it defaults to `-0.5`. When the value is negative, it draws the tick marks away from the plot region; when it's positive, the tick marks are drawn inward. For this `side=2` axis, `tcl=-2`, which means that the ticks will point outward from the plot but be four times the length they usually are (two whole lines of text as opposed to half a line).

In the same call to `axis`, the `las` parameter (controlling the way the labels for each tick mark are oriented) is set it to `1`, instructing R to produce all tick labels *horizontally*, regardless of axis side. The default, `las=0`, writes all labels *parallel* to the corresponding axis; the alternative `las=2` means labels are always *perpendicular* to the corresponding axes; and using `las=3` orients all labels to be read *vertically*, regardless of axis.

Next up, the `mgp` parameter controls three further aspects of axis spacing and, as such, is supplied a vector of length 3 as per the following definition: `c(`*`axis title,axis labels,axis line`*`)`. Once more, these arguments are expressed in "lines of text." The default value of `mgp` is `c(3,1,0)`, meaning that in every axis you've seen so far, the axis title has sat three lines of text away from the plot region, the tick mark labels one line of text away, and the axis line itself zero lines of text away from the plot region (so it's flush with any drawn plot region box). When used in `axis`, only the second and third elements of `mgp` are relevant. In the vertical axis in the plot on the right of Figure 23-10, the only alteration from

the default was to set the second element (spacing of the axis labels) to `2.5`, pushing the axis labels out to the left, further away from the plot region. The tick marks themselves are considerably lengthened by `tcl`, so this is required to avoid the axis tick mark labels going through those ticks. Try replotting the image and that axis, but without specifying `mgp`, and you'll see that unappealing result.

Moving to the final call to `axis`, we see the addition of the x-axis (`side=1`). You can see tick marks at `hpseq2` being placed via `at`. This time, a positive value has been supplied to `tcl`, instructing the axis to have *inward*-facing tick marks of 1.5 lines of text in length. In `mgp`, note that the third element of the vector is now set to `1`, meaning you want the axis line itself to sit one line of text away from the plot region. Looking at the right of Figure 23-10, you can see that the entire axis has been moved downward, away from the plot region. To account for this in terms of the spacing of the tick mark labeling, the second element of `mgp` has been slightly increased from its default, to `1.5`.

From the preceding example, you can get a sense of just how much finetuning you're able to do when it comes to tailoring your R plots. It's never easy to remember precisely what all of R's graphical parameters are; that said, they are all explained in great detail in the help file `?par`, which I frequently visit! A good way to learn is to simply start experimenting with various tweaks to a simple example plot, such as those created in this section.

23.5 Specialized Text and Label Notation

Now you'll investigate some immediately accessible tools for controlling fonts and displaying special notation, such as Greek symbols and mathematical expressions.

23.5.1 Font

Whenever you're showing text on a base R plot, the displayed font is controlled by two graphical parameters: `family` for the specific font

family and `font`, an integer selector for controlling bold and italic typeface.

Available fonts depend on both your operating system and the graphics device you're using. That said, three generic families—`"sans"` (the default), `"serif"`, and `"mono"`—are always available. These are paired with the four possible values of `font`: `1` (normal text, default), `2` (bold), `3` (italic), and `4` (bold and italic). You can set these two graphical parameters universally for a device using `par`, but like the use of `xpd`, it's just as common (if not more so) to set `family` and `font` in the relevant annotation functions.

Figure 23-11 shows you some variants alongside the corresponding values of `family` and `font`.



Figure 23-11: Displaying font styles through use of the `family` and `font` graphical parameters

To create this plot, start with an empty plot region with preset x- and y-limits, created with the following:

```
R> par(mar=c(3,3,3,3))
R> plot(1,1,type="n",xlim=c(-1,1),ylim=c(0,7),xaxt="n",yaxt="n",ann=FALSE)
```

Then, use `text` (see Section 7.3) to plot six possible variants of text at different coordinate locations in the empty plot region; the plot is completed by executing the following lines:

```
R> text(0,6,label="sans text (default)\nfamily=\"sans\", font=1")
R> text(0,5,label="serif text\nfamily=\"serif\", font=1",
        family="serif",font=1)
R> text(0,4,label="mono text\nfamily=\"mono\", font=1",
        family="mono",font=1)
R> text(0,3,label="mono text (bold, italic)\nfamily=\"mono\", font=4",
        family="mono",font=4)
R> text(0,2,label="sans text (italic)\nfamily=\"sans\", font=3",
        family="sans",font=3)
R> text(0,1,label="serif text (bold)\nfamily=\"serif\", font=2",
        family="serif",font=2)
R> mtext("some",line=1,at=-0.5,cex=2,family="sans")
R> mtext("different",line=1,at=0,cex=2,family="serif")
R> mtext("fonts",line=1,at=0.5,cex=2,family="mono")
```

Note the use of the escape sequence `\n` (see Section 4.2.3) to force a new line in the character string displayed via `text`. The `mtext` function (Section 23.2.2) is subsequently used to add to the top figure margin.

23.5.2 Greek Symbols

For statistically or mathematically technical plots, annotation may occasionally require Greek symbols or mathematical markup. You can display these using the `expression` function, which, among other things, is capable of invoking the `plotmath` mode of R (Murrell & Ihaka 2000, Murrell 2018). Use of `expression` returns a special object that has a class of the same name and can subsequently be passed to any argument in a plotting function that requires the character string to be displayed.

Focusing for the moment on Greek symbols, consider Figure 23-12, which is produced with the following code:

```
R> par(mar=c(3,3,3,3))
R> plot(1,1,type="n",xlim=c(-1,1),ylim=c(0.5,4.5),xaxt="n",yaxt="n",
      ann=FALSE)
R> text(0,4,label=expression(alpha),cex=1.5)
R> text(0,3,label=expression(paste("sigma: ",sigma,"    Sigma: ",Sigma)),
      family="mono",cex=1.5)
R> text(0,2,label=expression(paste(beta," ",gamma," ",Phi)),cex=1.5)
R> text(0,1,label=expression(paste(Gamma,"(",tau,") = 24 when ",tau," = 5")),
      family="serif",cex=1.5)
R> title(main=expression(paste("Gr",epsilon,epsilon,"k")),cex.main=2)
```

If you just want a single special character by itself, then something like `expression(alpha)` is all you need to produce α in the plot, as in the first call to `text` shown in the code chunk. Note that the specification of the special characters is done *without* quotes around the name of the desired symbol. More commonly, however, you'll want a character to appear alongside other components, such as regular text or in an equation. For that, you need to use `paste` inside the call to `expression`, separating the components with commas. This treatment is shown in the remaining three calls to `text`. You can also use `cex` to control size, though use of `family` and `font` affects only quoted regular text, not symbols, as the final call to `text` demonstrates.

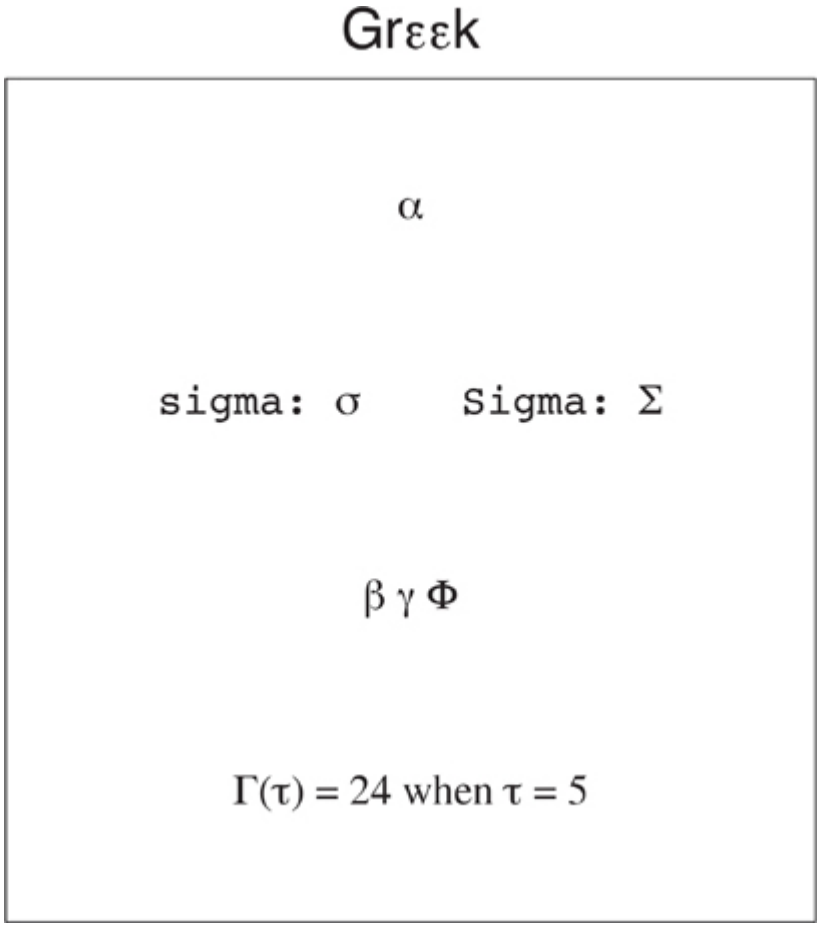


Figure 23-12: Displaying Greek symbols using `expression`

The `title` function, which allows you to add axis and main titles, is then used to add the title "Gr$\in\in$k" by supplying the corresponding `expression` to `main`. I use `cex.main=2` in the same call to double its size (the slightly different tag `cex.main` is required there to distinguish between the size of the main title and any axis titles, controlled via `cex.lab`). The `title` function will be used in a more complete way in the extended example within Section 23.6.

23.5.3 Mathematical Expressions

Formatting entire mathematical expressions to appear in R plots is a bit more complicated and is reminiscent of using markup languages like LATEX. Because of this, I won't give a full exposition of the syntax

required, but I'll provide examples of the kinds of things that are possible, as Figure 23-13 shows.

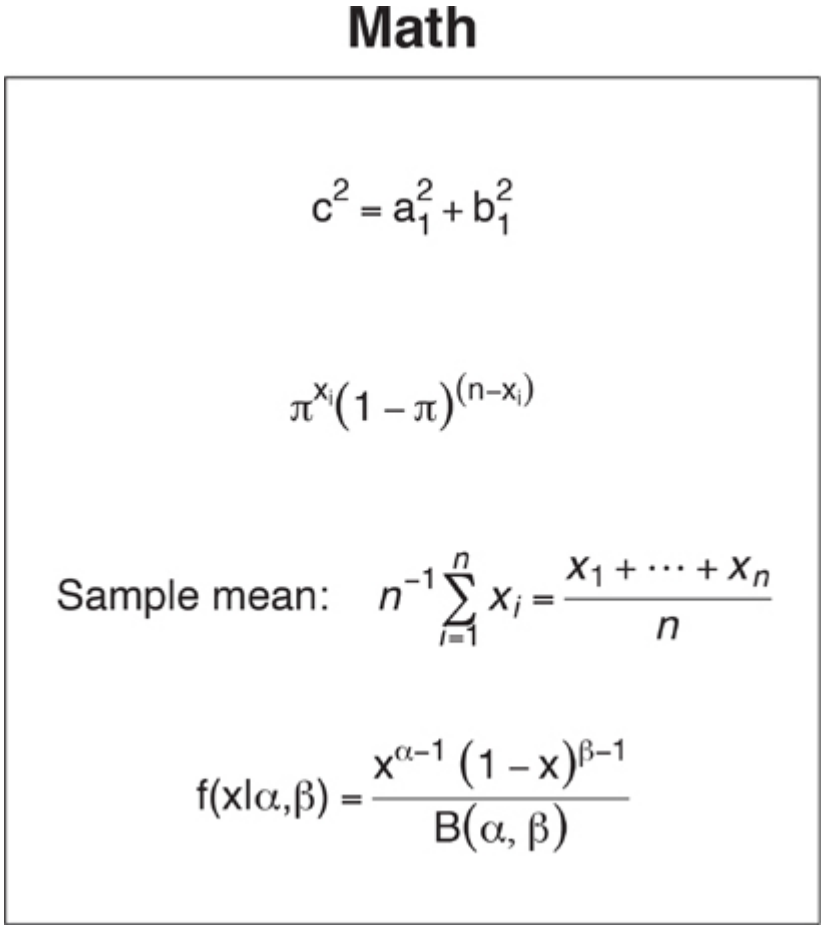


Figure 23-13: Some examples of typesetting mathematical expressions in R plots

To create the image, I first defined four expression objects as follows:

```
R> expr1 <- expression(c^2==a[1]^2+b[1]^2)
R> expr2 <- expression(paste(pi^{x[i]},(1-pi)^(n-x[i])))
R> expr3 <- expression(paste("Sample mean:    ",
                             italic(n)^{-1},
                             sum(italic(x)[italic(i)],
                                 italic(i)==1,
                                 italic(n))
                             ==frac(italic(x)[1]+...+italic(x)[italic(n)],
                                    italic(n))))
```

```
R> expr4 <- expression(paste("f(x","|",alpha,",",beta,
                             ")"==frac(x^{alpha-1}~(1-x)^{beta-1},
                                       B(alpha,beta))))
```

I then used them in the following code:

```
R> par(mar=c(3,3,3,3))
R> plot(1,1,type="n",xlim=c(-1,1),ylim=c(0.5,4.5),xaxt="n",yaxt="n",
        ann=FALSE)
R> text(0,4:1,labels=c(expr1,expr2,expr3,expr4),cex=1.5)
R> title(main="Math",cex.main=2)
```

All Greek and mathematical markup is contained within a call to `expression`. Use `paste` if you require separate components (separated by commas), some of which may or may not be regular text (in other words, provided in quotes) to produce the final result. Here are some key notes:

- Superscripts are given by `^` and subscripts by `[ ]`; for example, `c^2` provides c^2 in `expr1`, and the `a[1]^2` component provides a^2_1.
- You can group components with parentheses `( )`, which are visible (for example, the `(1-pi)^(n-x[i])` component of `expr2`), or with braces `{ }`, which aren't (for example, the `pi^{x[i]}` component).
- Italicized alphabetic variables are drawn with `italic()`; for example, `italic(n)` produces n in `expr3`.
- Constructs for common arithmetic operators already exist, such as `sum( , , )` and `frac( , )`; for example, in `expr3`, calling `sum(italic(x)[italic(i)],italic(i)==1,italic(n))` produces a result that looks like this

  $$\sum_{i=1}^{n} x_i$$

  and `frac(italic(x)[1]+...+italic(x)[italic(n)],italic(n))` produces an expression like this: $x_1 + \ldots + x_n \; n$

$$\frac{x_1 + \ldots + x_n}{n}$$

- There are additional markup tools for proper formatting of expressions, such as combining regular text in quotes directly next to mathematical markup and creating spaces between components without needing to insert quotes. The need for these depends on where the markup contents are exactly (in other words, as a stand-alone component of the call to `paste` or as a component of an operator tool like `frac`). See, for example, the `")"==frac( , )` part of `expr4` and the space separation of the two components `x^{alpha-1}~(1-x)^{beta-1}` with a `~` (these sit on the numerator of the fraction).

There's an extensive amount of functionality built in to R for this type of string formatting in graphical displays that I haven't covered here. If you're interested in seeing more, access the help file by entering `?plotmath` at the prompt as a first step. There's also an extremely useful demonstration you can view in R by executing `demo(plotmath)` at the prompt. This shows much of what's possible alongside the relevant syntax for `expression`.

23.6 Extended Example: A Custom Scatterplot

To provide a final example that covers most of the concepts you've considered so far, let's create a detailed plot of the MPG by horsepower data that was used in Sections 23.4.1 through 23.4.3. The images in Figure 23-14 show the final result as the largest plot on the bottom alongside three smaller interim plots (to illustrate the various stages of production) appearing on the top.

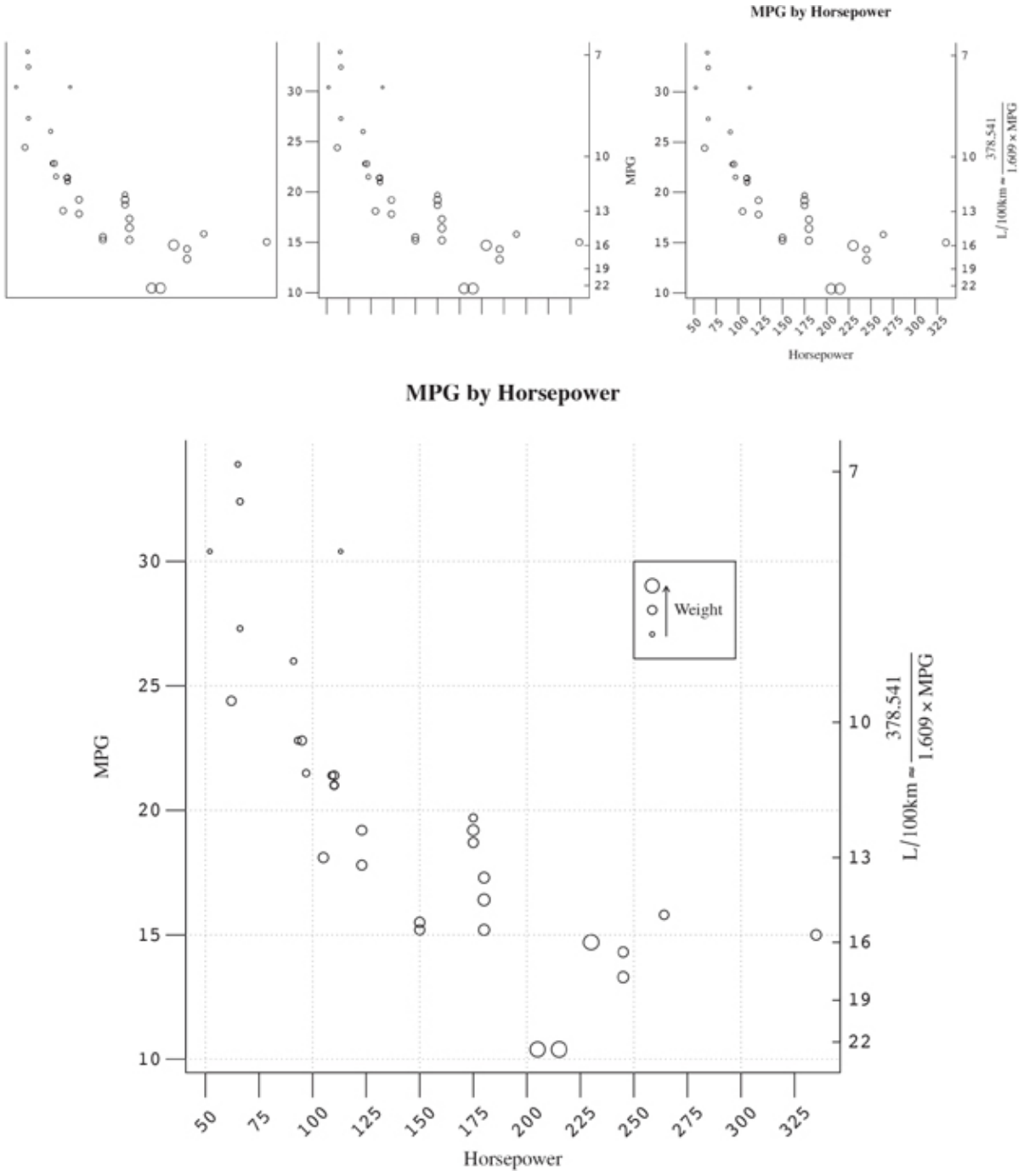


Figure 23-14: A detailed version of the `mtcars` scatterplot of MPG by horsepower, with point size proportional to weight

First, ensure you have the objects `mpg`, `hp`, `wtcex`, and `hpseq2` (defined in Sections 23.4.1 on page 665 and 23.4.3 on page 669) ready in your workspace, since you'll use them to ease the length of the code. Here they are again:

```
R> hp <- mtcars$hp
R> mpg <- mtcars$mpg
R> wtcex <- mtcars$wt/mean(mtcars$wt)
R> hpseq2 <- seq(50,325,by=25)
```

We start the plot, with a slightly wider right margin than default and a U-shaped box, with the following code:

```
R> dev.new()
R> par(mar=c(5,4,4,4))
R> plot(hp,mpg,cex=wtcex,axes=FALSE,ann=FALSE)
R> box(bty="u")
```

This provides the top-left image of Figure 23-14. I've used `dev.new` to explicitly open a new graphics device on my machine, which defaults to 7×7 inches. If you're using the RStudio editor, you may need to call this command twice to ensure a new external graphics device window pops up. You can use `width` and `height` arguments supplied to `dev.new` to alter its initial size if you want, though this will affect the specific coordinates required for placement of the customized legend and certain axis annotations.

Next, add some axes:

```
R> axis(2,las=1,tcl=-0.8,family="mono")
R> axis(1,at=hpseq2,labels=FALSE,tcl=-1)
```

These two lines add the left vertical axis for MPG; the tick marks are lengthened a little using `tcl`, their labels are made horizontal through `las`, and a `"mono"` font is requested. For the horizontal axis (horsepower), longer outward tick marks are drawn at the values in `hpseq2`, but their labels are suppressed by setting `labels=FALSE`. You'll populate those in a moment.

Rather than using MPG as a measure of fuel efficiency, many countries use "liters per one hundred kilometers" (L/100km). So, for their benefit, let's say you want to provide a second vertical axis on the right of the plot that provides L/100km. To do this, you need the

conversion formula. Based on a US gallon, an approximate conversion between the two is given by the following:

$$\mathrm{MPG} = \frac{378.541}{1.609 \times (\mathrm{L}/100\mathrm{km})}$$

It turns out that this function is *involutory*. That is, to convert from MPG back to L/100km, simply swap those two variables in the equation.

A little experimentation with the conversion formula, based on the limits of the observed MPG data, yields a sensible collection of L/100km values at which to mark the right axis:

```
R> L100 <- seq(22,7,by=-3)
R> L100
[1] 22 19 16 13 10  7
```

Note that these are stored in decreasing order for convenience; once you convert these to MPG, the results are increasing:

```
R> MPG.L100 <- (100/L100*3.78541)/1.609
R> MPG.L100
[1] 10.69385 12.38236 14.70405 18.09729 23.52648 33.60925
```

This makes sense; a smaller number for L/100km means a more fuelefficient car. Why do you need the MPG version of these numbers? Well, remember that the plot itself is on the scale of MPG, so to instruct R to place the L/100km tick marks on the right side at the correct locations, you need the L/100km values in MPG "coordinates."

With that done, one final call to `axis` leaves you with the top-middle plot in Figure 23-14:

```
R> axis(4,at=MPG.L100,labels=L100,las=1,tcl=0.3,mgp=c(3,0.3,0),family="mono")
```

Of particular note here is that you've used `at` to specify the tick marks on the MPG scale, at the values in `MPG.L100`, but since they correspond to the L/100km sequence in `L100`, you supply the latter vector to `labels` to actually label said tick marks.

Next, it's time to annotate the axes with some titles and provide the labels for the tick marks on the horizontal axis. Before doing that, construct an `expression` for the MPG-to-L/100km conversion to clarify the right vertical axis:

```
R> express.L100 <- expression(paste(L/100,"km"%~~%frac(378.541,1.609%*%MPG)))
```

In `express.L100`, the `%~~%` provides an "approximately equal to" sign (≈), and `%*%` provides an explicit multiplication symbol (×).

Then, by running the following lines, you get the top-right image in Figure 23-14:

```
R> title(main="MPG by Horsepower",xlab="Horsepower",ylab="MPG",
         family="serif")
R> mtext(express.L100,side=4,line=3,family="serif")
R> text(hpseq2,rep(7.5,length(hpseq2)),labels=hpseq2,srt=45,
        xpd=TRUE,family="mono")
```

The first line uses the `title` function to provide a main title and x- and y-axis titles in a `"serif"` style. Then `mtext` places a `"serif"` version of the arithmetic expression you just created in an appropriate position `line=3` on the right axis (`side=4`). The third line places the `"mono"`-style tick mark labels in `hpseq2` along the x-axis at the appropriate user coordinates in the same vector, with a vertical position of `7.5`, after a little trial and error; an appropriate position is partially dependent on the size of the graphics device. Since you're using `text` to draw in the figure margin, you must set `xpd` to `TRUE`. Special to `text` is the optional `srt` graphical parameter, which allows you to rotate the labels. Here, they've been rotated `45` degrees.

You're now ready to put the final touches on the plot. So far, the scaled sizes of the points according to car weight have been ignored. It'd be helpful to provide at least minimal information about that and other features that can aid in interpretation of the relationship (especially since there are two vertical axes), like an overlaid grid.

Superimposition of a grid on such a plot is straightforward via `grid`:

```
R> grid(col="darkgray")
```

Therein, you can specify the number of cells along the horizontal and vertical axes using the optional arguments `nx` and `ny`, respectively; if not, R will draw the grid lines at what would be the default x- and y-axis tick marks (as I've let it do here). You can alter other aesthetics in the usual way, using arguments such as `col` (color) and `lty` (line type).

The fun part is now trying to work out how to reference the sizes of the plotted points by the weights of the cars. There are a number of ways you might achieve this. For this last example, I'll manhandle the `legend` function in an effort to produce the graphic. The following three lines provide the end result:

```
R> legend(250,30,legend=rep("          ",3),pch=rep(1,3),pt.cex=c(1.5,1,0.5))
R> arrows(265,27,265,29,length=0.05)
R> text(locator(1),labels="Weight",cex=0.8,family="serif")
```

The legend is placed at user coordinates (250, 30), and three points of the default `pch` type `1` are included—one large, one standard, and one small—using `pt.cex` set to `1.5`, `1`, and `0.5`, respectively. Instead of writing text labels for each of these three points by using the `legend` argument, I simply assign them to be empty strings made up of 10 spaces. This widens the box around the legend, creating a space for what will instead accompany the points: a small-headed arrow pointing upward and the word *Weight*. Finding suitable user coordinates for the arrow to fit inside the artificially empty legend box took a little trial and error (and depends on the device size), and I place the `"Weight"` text by invoking the interactive `locator` function (as you saw used in Section 23.3) and left-clicking the desired spot.

Playing with R functionality to produce intricate plots like this is an excellent way to start learning how to handle the traditional graphical abilities of the language. It's not uncommon to use trial and error and little cheats or hacks to reach an end result, though this typically comes at the expense of the robustness of your code. For example, resizing the graphics device even moderately and attempting to reproduce this

scatterplot will likely result in a displeasing misalignment of the arrow in the legend and the x-axis text. To learn more, I recommend the authoritative reference on graphics in R, Murrell (2018), which is a good text to consult once you're familiar with the fundamentals discussed here and want a comprehensive guide on all things visual in R.

Exercise 23.2

For the following tasks, you'll work with the diamond-pricing data as analyzed by Chu (2001). You'll need an internet connection. Read the data in and name the columns as you've done previously with the following code (this is identical to the code used most recently in Exercise 22.2):

```
R> dia.url <- "http://jse.amstat.org/v9n2/4Cdata.txt"
R> diamonds <- read.table(dia.url,stringsAsFactors=TRUE)
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
```

a. Open a new graphics device of 6×6 inches, or ensure your device is at least roughly square in shape. Use `par` with the appropriate graphical parameter to initialize the margin spacing to be zero, four, two, and zero lines on the bottom, left, top, and right of the plot region, respectively. Then, complete the following:

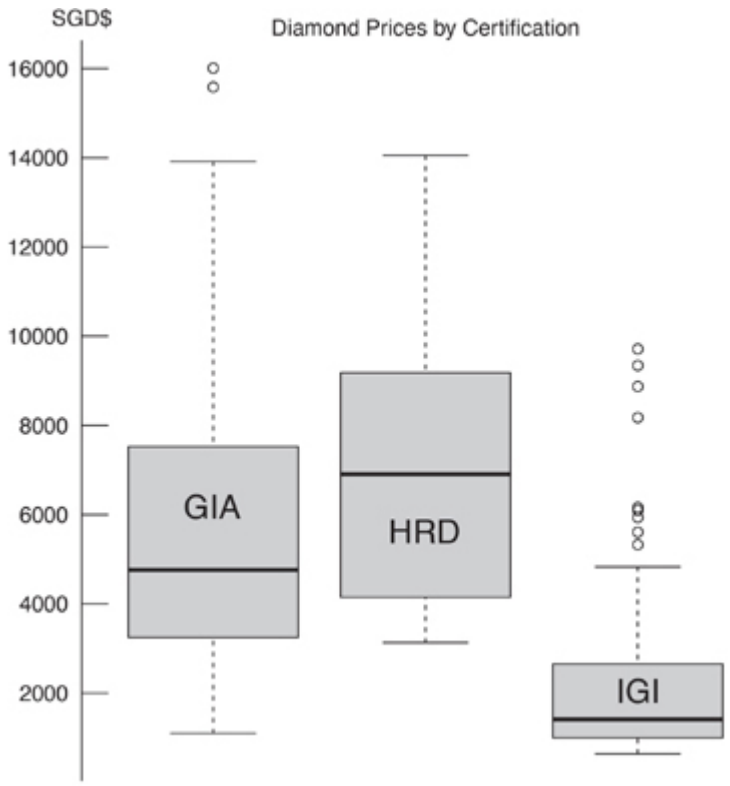
   i. Produce side-by-side box plots of the diamond prices in Singapore dollars (SGD$), split by certification. Suppress all axes, annotations, and the surrounding box, but use a subsequent call to `mtext` to add an informative title.

   ii. Next, insert a vertical y-axis. The axis should have tick marks ranging from SGD$0 to SGD$18,000, progressing in steps of SGD$2,000 (though note

that the axis will be automatically truncated at both ends by the figure region; this is permissible). The axis tick marks should point inward and be one line in length. The axis labels for the tick marks should sit a reasonable distance from the axis and should be horizontally readable.

iii. Finally, use `locator` in conjunction with `text` to add an appropriate title to clarify that the units of the y-axis are in SGD$. Make it perch at the top of the y-axis; relax clipping via the `xpd` parameter. Use the same mouse-click approach to add text to sit inside each of the three box plots; the text should denote the corresponding certification (GIA, HRD, or IGI).

My version of the plot looks like this:



b. Now open a new graphics device of 8×7 inches, or simply ensure it is slightly wider than it is high. Set the figure margins to be two, five, three, and five lines on the bottom, left, top, and right, respectively. Also allow one line of outer margin space on each side other than the bottom, which should itself get two lines of outer margin.

    i. Produce a scatterplot of diamond price on the vertical axis and carat weight on the horizontal axis. Use the colors red, green, and blue to distinguish the points according to certification. Suppress all axes, boxes, labels, and titles in the initial plot. In a subsequent call to `box`, add a U-shaped box.

    ii. Add the horizontal axis. Use `axis` to place tick marks at an evenly spaced sequence of carat values between 0.2 and 1.1, in steps of 0.1. Use a bold, italic, sansstyle font for the labels and adjust the labels to be only half a line from the axis. Then add smaller, outward-facing tick marks between the existing ones. To do this, make a second call to the `axis` function and place the ticks at a sequence of values from 0.15 to 1.05 at steps of 0.1. Set these secondary tick marks to have a length of one-quarter of a line and suppress the axis labels with `labels=FALSE`.

    iii. Add two vertical axes. On the left, ticks should appear at SGD$1,000–17,000 in steps of SGD$2,000. Labels should be horizontally readable and in the same font style as the horizontal axis. On the right, axis ticks should be made in the equivalent of US dollars (USD$) at the sequence USD$1,000–11,000, in steps of USD$1,000, and should be labeled as such. To do

this, just use the rough conversion USD$ = 1.37 × SGD$. Label orientation and font should match the other axes.

iv. Fit a linear model of price on a quadratic (order 2) polynomial of carat weight for the data. Provide a prediction of the model for a sequence of carat values spanning the range of the observed values; include estimation of a 95 percent prediction interval. Use this information to superimpose a gray solid line for the fitted values and gray dashed lines for the prediction interval upon the scatterplot. Refer back to Section 21.4.1 for relevant knowledge.

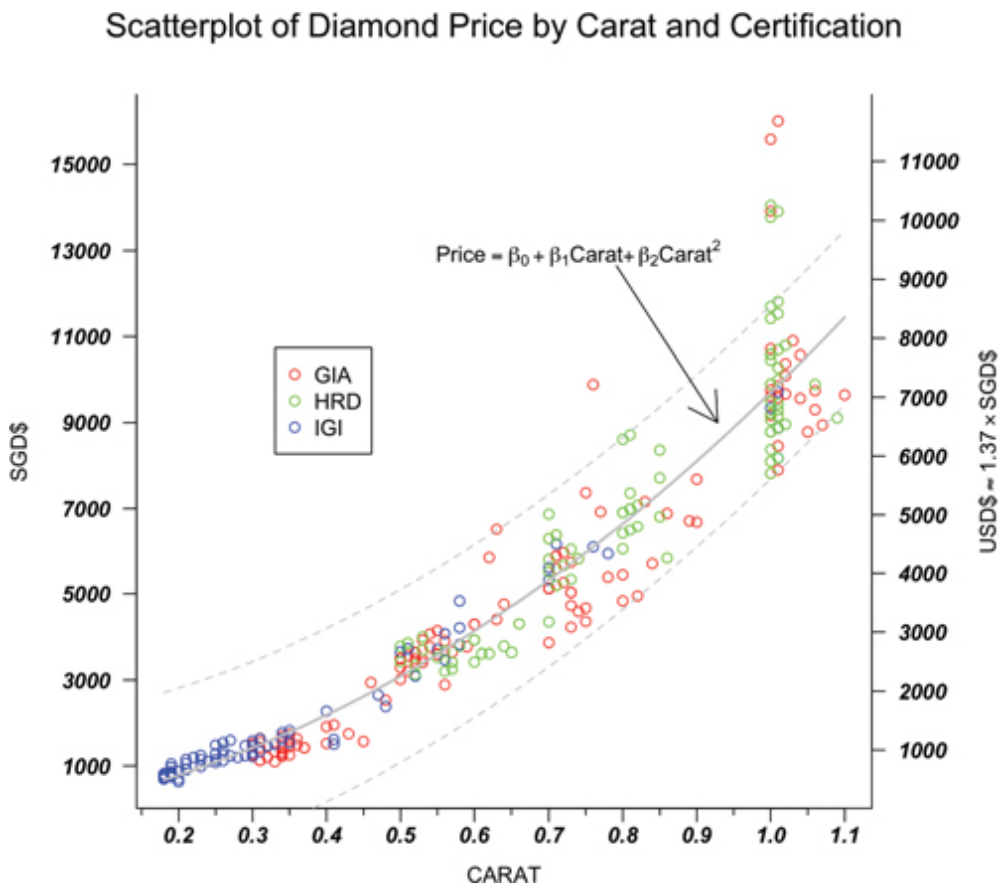
v. Set up expression objects for labeling the approximate US dollar conversion and the regression equation. Name the conversion `expr1`; it should look something like USD$ ≈ 1.37 × SGD$. The regression equation should look similar to Price = $\beta_0 + \beta_1 \text{Carat} + \beta_2 \text{Carat}^2$; name it `expr2`.

vi. Use `mtext` to add an appropriate main title and titles for all three individual axes. You may need to experiment a little with line depth for each one, as well as whether to write in the outer margin or the figure margin, depending on your own spacing preference. The rightmost axis title should make use of `expr1`.

vii. Either via trial and error to find appropriate coordinates, or by using the `interactive.arrow` function you wrote in Exercise 23.1, place an arrow pointing to the fitted polynomial regression line and label it with `expr2`.

viii. Finally, use a call to `locator` to place a legend in any appropriate location, referencing the color of the points according to the appropriate certification.

My version of the plot looks like this:



Important Code in This Chapter

Function/operator	Brief description	First occurrence
`dev.new`	Open new graphics device	Section 23.1.1, p. 648
`dev.set`	Change active device	Section 23.1.2, p. 649

Function/operator	Brief description	First occurrence
`dev.off`	Close device	Section 23.1.3, p. 650
`par`	Set/query graphical parameter(s)	Section 23.1.4, p. 651
`mfrow` (via `par`)	Device grid layout (row-wise)	Section 23.1.4, p. 651
`mfcol` (via `par`)	Device grid layout (column-wise)	Section 23.1.4, p. 652
`layout`	Graphics device custom layout	Section 23.1.4, p. 653
`oma`, `mar` (via `par`)	Device margin spacing	Section 23.2.1, p. 655
`box`	Add box to plot	Section 23.2.1, p. 655
`mtext`	Write text in margins	Section 23.2.2, p. 657
`xpd` (argument or `par`)	Control margin clipping rules	Section 23.2.3, p. 657
`locator`	Interactive coordinates	Section 23.3.1, p. 660
`xaxs` (argument or `par`)	Buffer space, x-axis	Section 23.4.1, p. 666
`yaxs` (argument or `par`)	Buffer space, y-axis	Section 23.4.1, p. 666
`axes` (argument)	Draw/suppress axes	Section 23.4.1, p. 667

Function/operator	Brief description	First occurrence
`ann` (argument or `par`)	Draw/suppress titles	Section 23.4.1, p. 667
`bty` (argument or `par`)	Box style	Section 23.4.2, p. 667
`axis`	Add axis to plot	Section 23.4.3, p. 668
`tcl` (argument or `par`)	Axis tick length	Section 23.4.3, p. 669
`las` (argument or `par`)	Axis label orientation	Section 23.4.3, p. 669
`mgp` (argument or `par`)	Axis spacing	Section 23.4.3, p. 669
`family` (argument or `par`)	Font type (`"serif"`, `"sans"`, `"mono"`)	Section 23.5.1, p. 671
`font` (argument or `par`)	Font style	Section 23.5.1, p. 671
`expression`	Greek/math in plotted text	Section 23.5.2, p. 672
`italic` (in `expression`)	Italicize text	Section 23.5.3, p. 674
`^`, `[ ]` (in `expression`)	Super- and subscripts	Section 23.5.3, p. 674
`sum`, `frac` (in `expression`)	Typeset mathematical operators	Section 23.5.3, p. 674
`title`	Add main/axis titles to plot	Section 23.6, p. 678

Function/operator	Brief description	First occurrence
`srt` (in `text`)	Rotate plotted text	Section 23.6, p. 678
`grid`	Add grid to plot	Section 23.6, p. 678

24
GOING FURTHER WITH THE GRAMMAR OF GRAPHICS



You covered the basics of the `ggplot2` package, which offers an alternative to "base" R graphics, in Section 7.4 and throughout Chapter 14. In this chapter, you'll look at a couple of this package's more popular and useful features, as well as its younger cousin, `ggvis`, which provides an interactive, browser-based experience.

24.1 A Quick Recap

Creating `ggplot2` graphics entails assigning your dataset's variables to specific *aesthetic mappings*, and layering visual features by using the desired *geometric modifiers*, called *geoms*.

As a quick refresher, revisit the student survey data from Section 19.3.2 and used in several examples since. Call `library("MASS")` at the prompt to gain access to the `survey` data frame; we will focus on the height, writing handspan, and sex variables. For convenience, first execute the following line of code:

```
R> surv <- na.omit(survey[,c("Sex","Wr.Hnd","Height")])
```

This creates a subset of the `survey` data frame of just the three variables of interest, utilizing `na.omit` (see Section 6.1.3) to strip away any records with missing values. You'll use this object in additional examples in Section 24.2.

In Chapter 21, you investigated the student survey data with an aim to describe the relationships between height, handspan, and sex using linear regression; see Figure 21-1. Consider now a `ggplot2` plot of the three variables, using skills developed in Section 7.4:

```
R> ggplot(surv,aes(x=Wr.Hnd,y=Height)) + geom_point(aes(col=Sex,shape=Sex))
```

Figure 24-1 shows the result. Note that the handspan and height are mapped to the x- and y-axes, respectively. The point geom, `geom_point`, produces a scatterplot of the global mapping declared in the inital call to `ggplot`; local aesthetics in `geom_point` stipulate the individual points should be classified according to color and shape/type.



Figure 24-1: A `ggplot2` scatterplot of height on handspan for the student survey data

The `ggplot2` package is useful for splitting plot features by one or more categorical variables, as the example here indicates. Note also the automatic production of an appropriate legend; when plotting with base R graphics, this would need to be tailored and added separately.

24.2 Smoothing and Shading

As the preceding example demonstrates, the benefits of `ggplot2` are especially appArent when you're enhancing your plot with features that are more difficult to achieve using base R commands. Two examples of this include adding data-specific smoothed trend estimates, as well as overlaying multiple plots.

24.2.1 Adding LOESS Trends

When you're looking at raw data, it's sometimes difficult to get an overall impression of trends without fitting a parametric model (for example, via linear regression), which means making assumptions about the nature of these trends. You can use certain *nonparametric smoothing* methods to uncover structure in your data without fitting a specific model. These methods are a flexible means by which one might interpret overall trends whatever their form, but the trade-off is that you're not provided with any specific numeric details of the relationships between response and predictors (since you're not estimating any coefficients such as slopes or intercepts) and you lose any reliable ability to extrapolate.

Locally estimated scatterplot smoothing (LOESS) and a closely related special case known as *locally weighted scatterplot smoothing (LOWESS)* represent popular nonparametric smoothing techniques. Application thereof produces a smooth estimate of a trend by using regression methods on localized subsets of the data, step-by-step, over the entire range of the explanatory variable. While I'll leave technical definitions aside for the purposes of this section, the interested reader can consult Chapter 6 of *Applied Nonparametric Regression* (Härdle 1990), as well as

Chapters 2 and 3 of *Introduction to Nonparametric Regression* (Takezawa 2006), for gentle introductions to the methodology.

Turn your attention back to the `surv` object created in Section 24.1 on page 685. Ensure the `ggplot2` package is loaded and execute the following code to produce the image on the left of Figure 24-2:

```
R> ggplot(surv,aes(x=Wr.Hnd,y=Height)) + geom_point(aes(col=Sex,shape=Sex)) +
          geom_smooth(method="loess")
`geom_smooth()` using formula = 'y ~ x'
```

The calls to `ggplot` and `geom_point` are identical to those used to create the basic scatterplot in Figure 24-1. The addition of `geom_smooth` superimposes the LOESS smoother; it takes instruction on what to smooth from the global mappings defined in the initial call to `ggplot` and this is confirmed by the output. By default, a 95 percent confidence interval for the estimated trend is included as a transparent gray-shaded area.

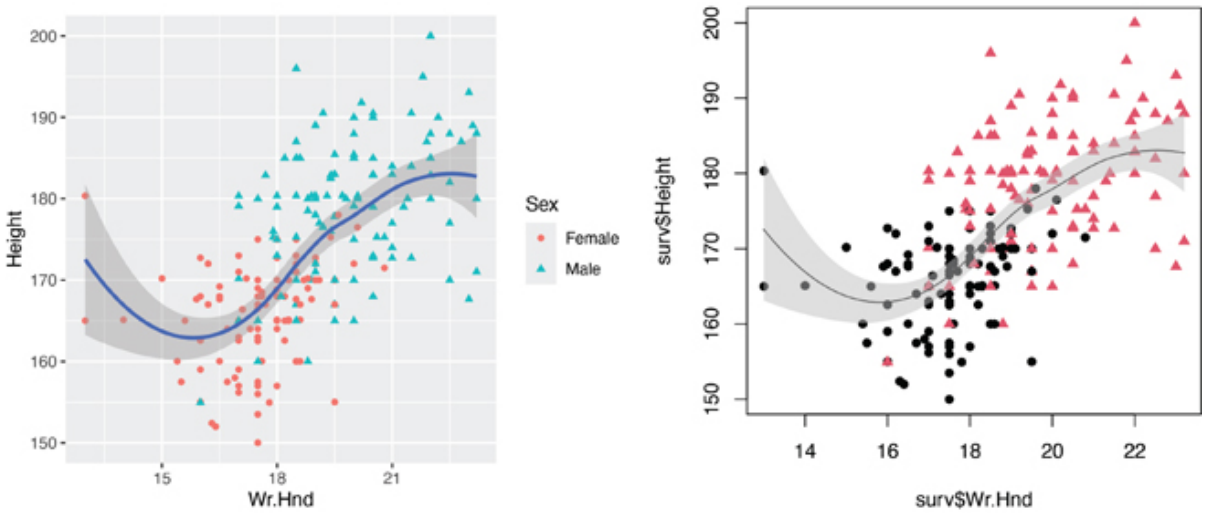


Figure 24-2: Showcasing `ggplot2` (left) and base R graphics (right) for display of nonparametrically estimated trends via LOESS

Now I'll demonstrate how to produce a similar result using base R graphics. Although base R functions, such as `scatter.smooth`, can produce a scatterplot with the smoothed trend relatively quickly, to be able to do things such as shade in the confidence interval region, it's helpful to

instead build the plot up piece by piece. Compare the relative ease of the `ggplot2` approach with the following base R code, which produces the image on the right of Figure 24-2:

```
R> plot(surv$Wr.Hnd,surv$Height,col=surv$Sex,pch=c(16,17)[surv$Sex])
R> smoother <- loess(Height~Wr.Hnd,data=surv)
R> handseq <- seq(min(surv$Wr.Hnd),max(surv$Wr.Hnd),length=100)
R> sm <- predict(smoother,newdata=data.frame(Wr.Hnd=handseq),se=TRUE)
R> lines(handseq,sm$fit)
R> polygon(x=c(handseq,rev(handseq)),
           y=c(sm$fit+2*sm$se,rev(sm$fit-2*sm$se)),
           col=adjustcolor("gray",alpha.f=0.5),border=NA)
```

The first line plots the raw data, and the second line uses the built-in `loess` function to provide the smoothed trend; the syntax is identical to that of `lm`. Just as with linear models fitted by `lm`, for drawing to begin, you need to set up a fine sequence of values of the x-axis variable at which to obtain the point estimates and their standard errors. This is achieved with `seq` in the third line, followed by `predict` in the fourth line with the `se` argument set to `TRUE`. This results in the object `sm`, a list with components `$fit` and `$se` as usual. Refer back to Section 20.4 for details.

The smoothed trend is then drawn by using `sm$fit` in a call to `lines`. Finally, a rough 95 percent confidence interval is calculated for each of the predicted values as the `sm$fit` elements plus and minus twice the corresponding standard errors in `sm$se`. This is done directly in the call to `polygon`, which draws the gray band based on the vertices formed by the confidence interval (therein, the `rev` command reverses the entries in the given `handseq` vector). You need to instruct the gray-filled shape to be transparent with a call to the ready-to-use `adjustcolor` command (the argument `alpha.f` takes a value from 0, which is fully transparent, to 1, which is fully opaque); setting `alpha.f=0.5` sets 50 percent opacity of the specified `"gray"`.

All that, and you've yet to add a legend referencing the two different types of points (representing sex)! This example certainly exposes the extra effort the base R version of the image requires, in terms of the

length of not just the script but also mapping out its construction (for example, putting together the vertices of a polygon for the confidence region appropriately and remembering to adjust the opacity of the filled shape to prevent covering up any pre-plotted content).

This additional effort is compounded the moment you become a little more ambitious with such features. Suppose you wanted to superimpose smoothers for each sex separately; in base R this requires separate estimation of the LOESS functions and a rethink of the plotting strategy. However, such a modification is simple in `ggplot2` terms, requiring only a change in the aesthetic mappings. The following code produces Figure 24-3:

```
R> ggplot(surv,aes(x=Wr.Hnd,y=Height,col=Sex,shape=Sex)) +
       geom_point() + geom_smooth(method="loess")
`geom_smooth()` using formula = 'y ~ x'
```

This moves the aesthetic mapping for color and point type (`col=Sex` and `shape=Sex`, respectively) out of `geom_point`. Instead of being specific to the plotted points only, it's now part of the default mapping declared in the initialization call to `ggplot`. Any layer added afterward (that doesn't reassign the mapping) will follow this global mapping, as is now the case for both `geom_point` *and* `geom_smooth`; the latter is now executed respecting the two different sub-groups of points.

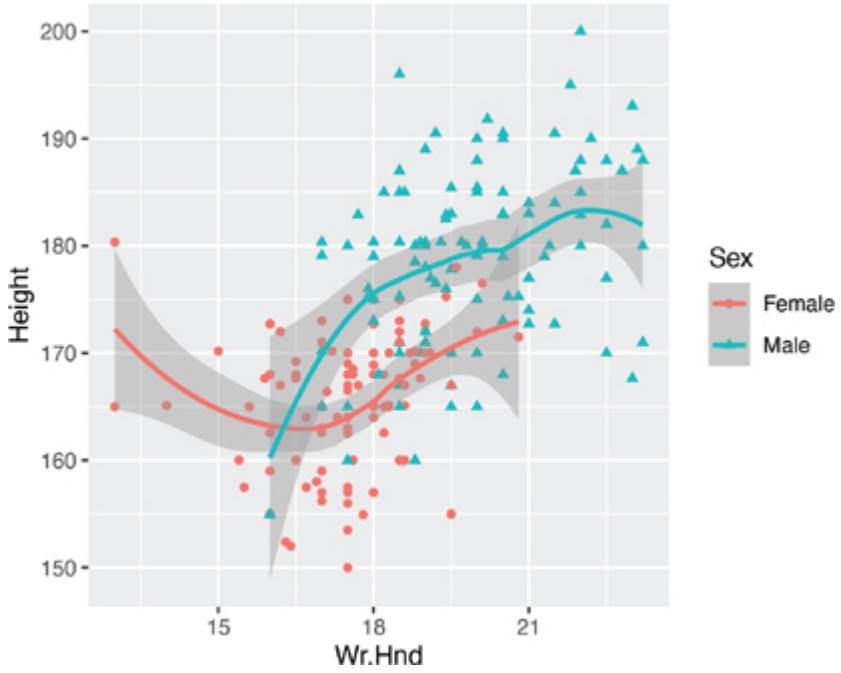


Figure 24-3: Illustrating separate LOESS smoothers for categorical subsets of data, a result of a simple change in aesthetic mapping, with `ggplot2` *functionality*

NOTE

The implementation of LOESS and other trend smoothers depends on specifying the amount of smoothing you want through controlling the proportion of the data to use as each localized weighted subset for each step/location in the estimation procedure. A larger proportion leads to a smoother, less variable trend estimate than a smaller proportion. You can set this value, called the span, *with the optional argument* `span` *in either* `loess` *or* `geom_smooth`. *For quick data exploration, however, the default value of 0.75 is usually adequate. Try changing the value of this argument on the example plots in this section to see the effects on the respective trends.*

24.2.2 Constructing Smooth Density Estimates

Smoothing isn't limited to scatterplot trends. For example, Section 14.3 covered how to use kernel density estimation (KDE) to form a smooth estimate of some unknown probability density function. This is a convenient way to visualize the distribution of a set of continuous

observations if you want to produce a more sophisticated visualization than a histogram.

Since you learned how to use `ggplot2` functionality to produce kernel estimates in Section 14.3.4 (including how to overlay multiple estimates split according to some categorical variable), I'll quickly recap these skills using a new dataset. I'll also demonstrate another of `ggplot2`'s strengths: filled color shading.

Consider the built-in `airquality` data frame. Enter `?airquality` at the prompt to open the documentation, which tells you it contains a number of measurements taken of the air in New York over several months. The following line provides a basic plot of the kernel density estimate for the temperature measurements:

```
R> ggplot(data=airquality,aes(x=Temp)) + geom_density()
```

This is shown on the left of Figure 24-4.

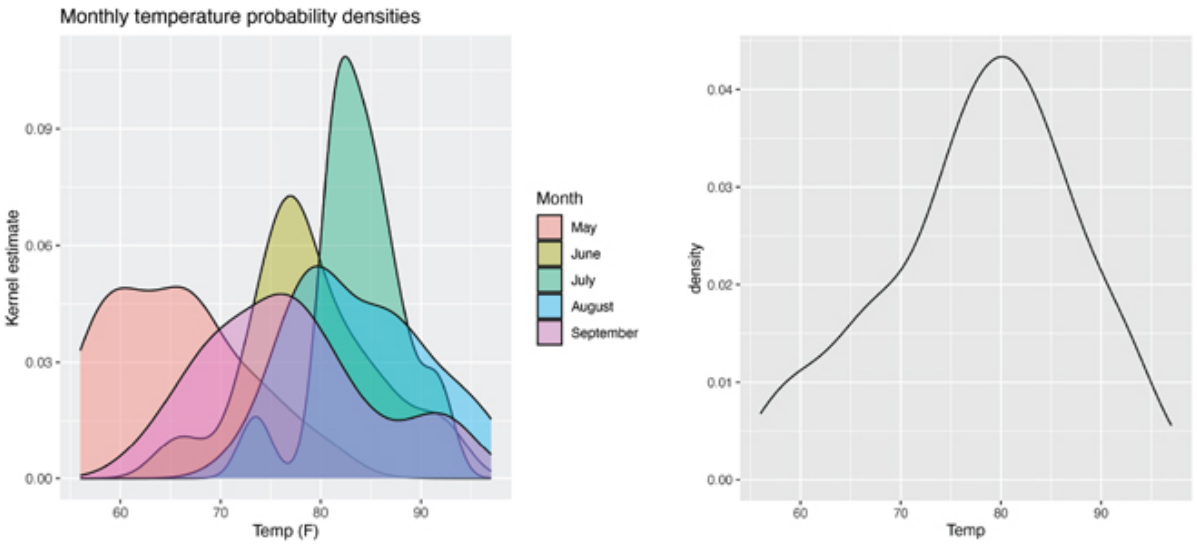


Figure 24-4: Visualizing temperature distributions in the `airquality` data frame via KDE, using `ggplot2` functionality

As illustrated previously in Section 14.3, such a plot is relatively easy to create with base R graphics as well, by using the built-in `density` command to implement KDE for a given data vector. However, `ggplot2` naturally lets you dress up the plot using aesthetic mappings with

relative ease. Suppose you want to visualize the density estimates for temperature separately according to the month of observation. First, execute the following code:

```
R> air <- airquality
R> air$Month <- factor(air$Month,
                       labels=c("May","June","July","August","September"))
```

This creates a copy of the `airquality` data frame in your workspace and re-codes the originally numeric `Month` vector to be a factor (as required for categorical mappings in `ggplot2`), labeling the entries appropriately given the information found in the help file for the `airquality` object. Then, using `air`, the following code produces the right-hand plot in Figure 24-4:

```
R> ggplot(data=air,aes(x=Temp,fill=Month)) + geom_density(alpha=0.4) +
       ggtitle("Monthly temperature probability densities") +
       labs(x="Temp (F)",y="Kernel estimate")
```

The different densities are clearly identified by different color fills, set using `fill=Month` in `aes` in the plot initialization mapped out by `ggplot`. You additionally supply `alpha=0.4` to `geom_density` to set 40 percent opacity so you can see all five curves clearly. The remaining calls to `ggtitle` and `labs` simply tidy up the main and axis titles. Features of the distributions of these measurements are as you might expect: Temperatures for July, the hottest month, are centered over a far higher range of values than, say, those for May.

A key ingredient in the application of any smoothing technique is the desired amount of smoothing to use. Recall from Section 14.3.2 that for KDE, the value that controls this is typically referred to as the *bandwidth*. For the plots in this chapter, the bandwidth used in each instance of a kernel density estimate is automatically chosen using a data-driven technique. This default level of smoothing is generally appropriate when you're just seeking to visually explore your data, but you can experiment with different values by specifying the `bw` argument in `geom_density` (see the help file for further details).

24.3 Multiple Plots and Variable-Mapped Facets

Section 23.1.4 showed different ways to view or lay out several base R plots in a single graphics device. You cannot use these methods (such as setting `mfrow` in a call to `par` or compartmentalizing the device using `layout`) for `ggplot2` graphics, though other functions allow independent `ggplot2` plots to populate a single device. True to form, `ggplot2` also offers a convenient way to construct multiple-plot graphics by using *facets*, where the images are all drawn in one go.

24.3.1 Independent Plots

Say you have several `ggplot2` plots that you've created independently of one another and you'd like to arrange them as a single image. To do so quickly, use the `grid.arrange` function provided in the contributed `gridExtra` package (Auguie 2017). Install the package by running `install.packages("gridExtra")` at the prompt (you'll need an internet connection).

To illustrate the use of `grid.arrange`, continue with the `air` object (the copy of `airquality` you created in Section 24.2) with the factor `Month` column. Now consider the following three `ggplot2` objects, which I'll explain further in a moment:

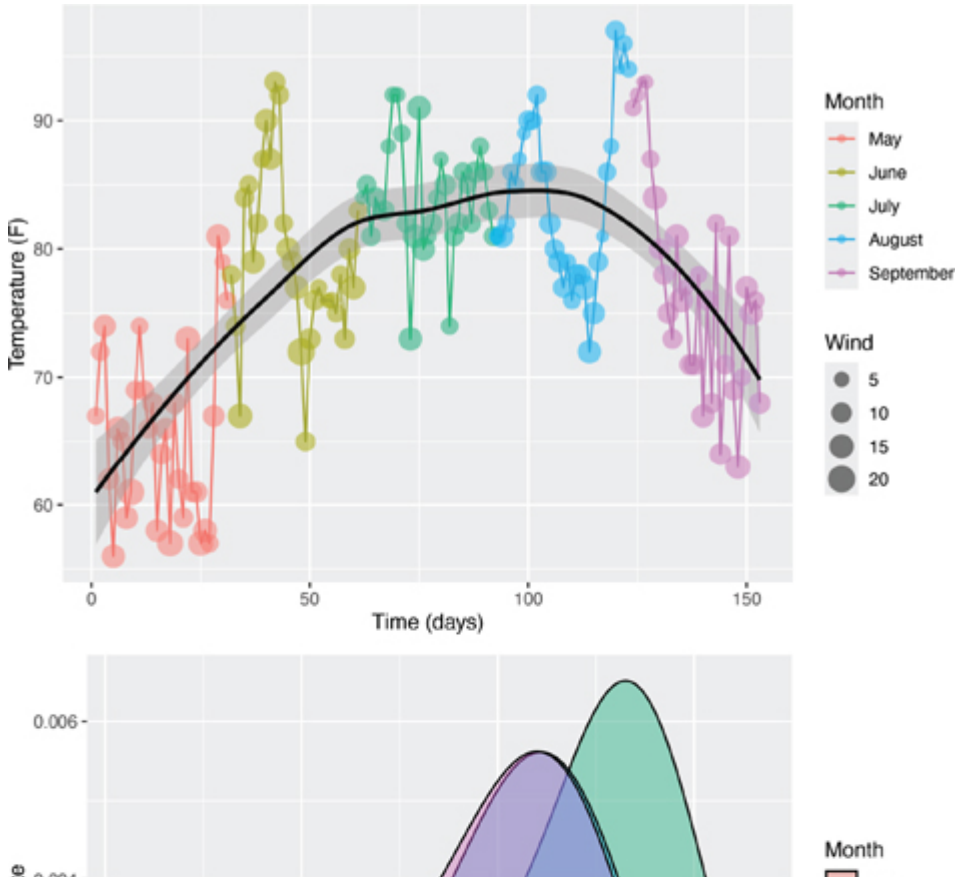
```
R> gg1 <- ggplot(air,aes(x=1:nrow(air),y=Temp)) +
            geom_line(aes(col=Month)) +
            geom_point(aes(col=Month,size=Wind),alpha=0.5) +
            geom_smooth(method="loess",col="black") +
            labs(x="Time (days)",y="Temperature (F)")
R> gg2 <- ggplot(air,aes(x=Solar.R,fill=Month)) +
            geom_density(alpha=0.4) +
            labs(x=expression(paste("Solar radiation (",ring(A),")")),
                 y="Kernel estimate")
R> gg3 <- ggplot(air,aes(x=Wind,y=Temp,color=Month)) +
            geom_point(aes(size=Ozone),alpha=0.5) +
            geom_smooth(method="lm",level=0.9,fullrange=FALSE,alpha=0.2) +
            labs(x="Wind speed (MPH)",y="Temperature (F)")
```

Execute `library("gridExtra")` to load the required package. To view `gg1`, `gg2`, and `gg3` in one window, simply call the following, which produces Figure 24-5:

```
R> grid.arrange(gg1,gg2,gg3)
```

You'll likely see some warning messages telling you there are missing values in the `air` data frame and recommending to resize the window containing the plots.

As you can see, `grid.arrange` is easy to use: You simply create your `ggplot2` images first and store them as objects, then provide them directly to the arrangement function. The `grid.arrange` function decides how to produce the final layout based on the number of objects you give it (in this case, it's a column of the three plots). You can control the order of the plots by changing the order in which the objects are supplied. You can read about further optional arguments, and the features they control, in the `?grid.arrange` documentation file.



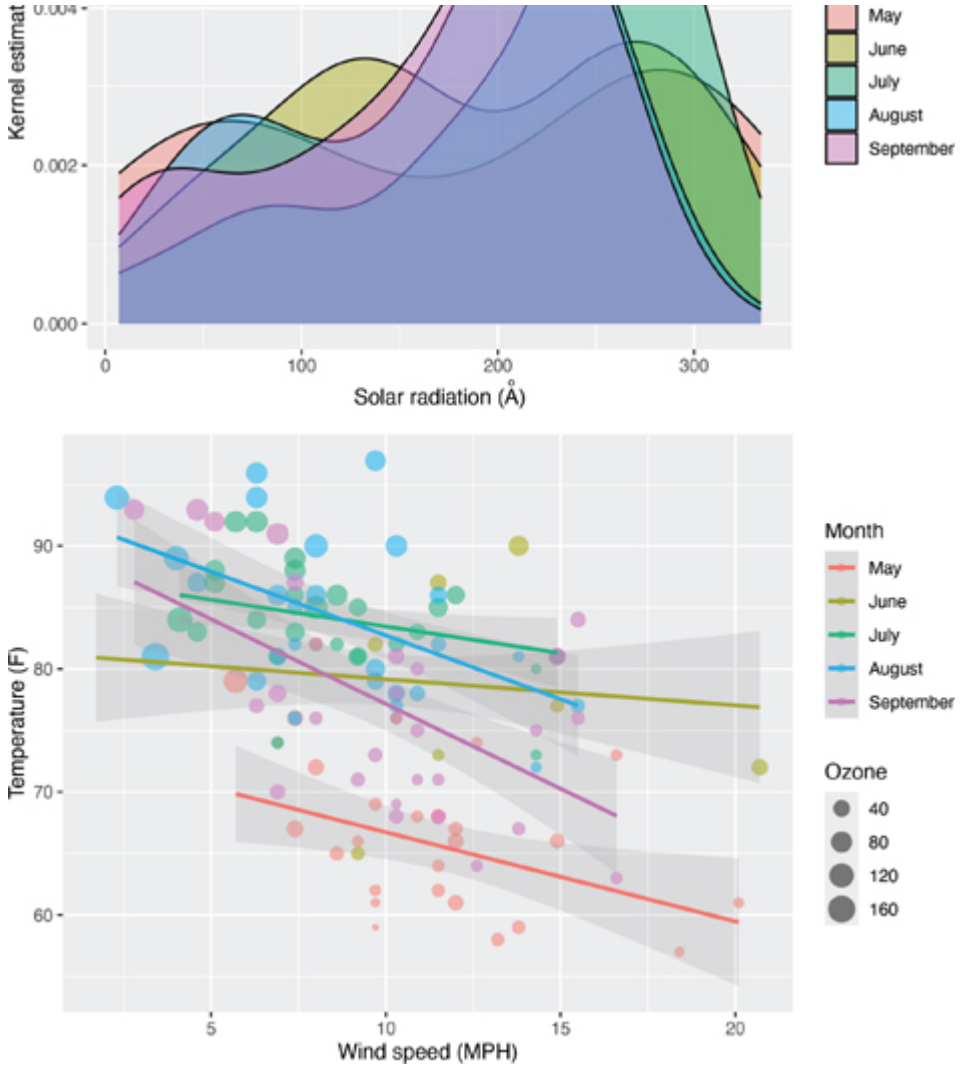


Figure 24-5: A series of three `ggplot2` *graphics of the* `airquality` *data, plotted in the same device window via* `grid.arrange` *in the* `gridExtra` *package. Top: A time series of the temperatures by day, distinguishing between months and wind speed, with an overall LOESS trend with a 95 percent CI. Middle: Kernel density estimates of the distributions of solar radiation by month. Bottom: A scatterplot of temperature by wind speed, using color to delineate month and point size to reference ozone level. Separately fitted simple linear models of temperature on wind speed, split by month, along with 90 percent CIs, are superimposed.*

The plots `gg1`, `gg2`, and `gg3` also provide an opportunity to discuss even more `ggplot2` capabilities. Since there's a lot going on, especially in `gg1` and `gg3`, I'll discuss the code for each object separately:

`gg1` The first plot is of the daily temperature. In setting the default aesthetics in `ggplot`, I create a sequence of integers matching

the number of rows in `air`, to be paired with the relevant `Temp` element. Then `geom_line` and `geom_point` add the interconnecting lines and the raw observations themselves, to be added to the default aesthetic mapping. The interconnecting lines are set to change color according to `Month`. The raw observations also change color according to `Month`, and the point sizes change to be proportional to the wind speed readings; `alpha=0.5` is used to render all points with 50 percent transparency in order to better discern those that overlap. I include an overall LOESS smoother with its default color changed to `"black"`. I remain with the default mapping here; I don't want separate smoothed trends for each month. The final addition of `labs` merely clarifies the axis titles as you've already seen it used.

`gg2` The second plot is a variant of the plot on the right of Figure 24-4. This time, it shows the estimated densities of the solar radiation readings (in angstroms). The opacity is set, as you saw earlier, using `alpha` in `geom_density`. It's also worth noting that I used `expression` in `labs` to approximate the angstrom unit symbol, Å, using `ring(A)`; refer back to Section 23.5 for instruction on how to display special textual symbols and mathematical markup.

`gg3` The last is a scatterplot of temperature by wind speed, where you can see a negative relationship. The color, again to be assigned to each month, is also set as the default aesthetic mapping in the initialization call to `ggplot`. In `geom_point`, the aesthetic enhancement is instructed to plot point size as proportional to the ozone reading (this ensures correct formatting of the corresponding legend, in light of the next addition), and once again, the points are instructed to be somewhat transparent with `alpha=0.5`. Here, you can also see a different kind of use for `geom_smooth`. In setting `method="lm"`, the line(s) I want superimposed correspond to simple linear

model fits according to the `x` and `y` aesthetic mappings as predictor and response, respectively. Additionally, including the factor `Month` in the default mapping ensures separate simple linear models are fitted for the temperature on wind speed data for each month and colored appropriately (it's important to note that the plotted lines do not reflect a multiple linear model that includes all the variables used in the plot). Transparent 90 percent CIs are included with each regression (`level=0.9` and `alpha=0.2`), and setting `fullrange=FALSE` restricts each regression line only to the width of the observed subsets of data for each month.

24.3.2 Facets Mapped to a Categorical Variable

While you can use `grid.arrange` to have independently created `ggplot2` graphics appear in the same window, the `ggplot2` package offers a flexible alternative to quickly view multiple plots. Often, when exploring a dataset, you'll want to create several plots of the same variables based on the levels of one or more important categorical variables. This behavior, referred to as *faceting*, is familiar territory for `ggplot2` using primarily the `facet_wrap` and the `facet_grid` commands.

A Single Categorical Variable

Let's focus on the simplest case, where you want to split up your plots according to a single categorical variable. Remaining with the `air` data frame object (as created from the built-in `airquality` data frame in Section 24.2.2), the following line creates a `ggplot` object of the density plots of the New York temperatures as on the right of Figure 24-4:

```
R> ggp <- ggplot(data=air,aes(x=Temp,fill=Month)) + geom_density(alpha=0.4) +
             ggtitle("Monthly temperature probability densities") +
             labs(x="Temp (F)",y="Kernel estimate")
```

Rather than view the density estimates for the different months as overlapping curves, you can create a plot of each one separately by faceting with respect to the `Month` factor. The individual plots are all

displayed in a single device; the following code contains three different uses of `facet_wrap`:

```
R> ggp + facet_wrap(~Month)
R> ggp + facet_wrap(~Month,scales="free")
R> ggp + facet_wrap(~Month,nrow=1)
```

Figure 24-6 shows the three resulting versions of the plot.

The `facet_wrap` function uses a formula style to specify the faceting variable. By default, it also automatically decides the layout of the individual plots. For all plots in Figure 24-6, the formula is set to `~Month`, to be read as "by Month." The code for the first version, given at the top left, specifies no additional arguments. Without arguments, the x- and y-axes of each individual facet are fixed, so you can compare the plots on the same scale. If you don't want that, you can instruct the axes to be "free," which means each facet is produced on scales specific to its own contents. You can see this in the second version of the plot, at the top right in Figure 24-6, the line of code for which specifies `scales="free"`. If desired, you can also opt to free only the horizontal or vertical axis with `scales="free_x"` or `scales="free_y"`, respectively. Finally, note that facet placement itself can be tailored using the `nrow` and `ncol` arguments. In the third version of the plot, setting `nrow=1` instructs the function to lay the facets out in one row only, giving the horizontal arrangement shown on the bottom of Figure 24-6. For further details on placement, review the documentation by executing `?facet_wrap` at the prompt.

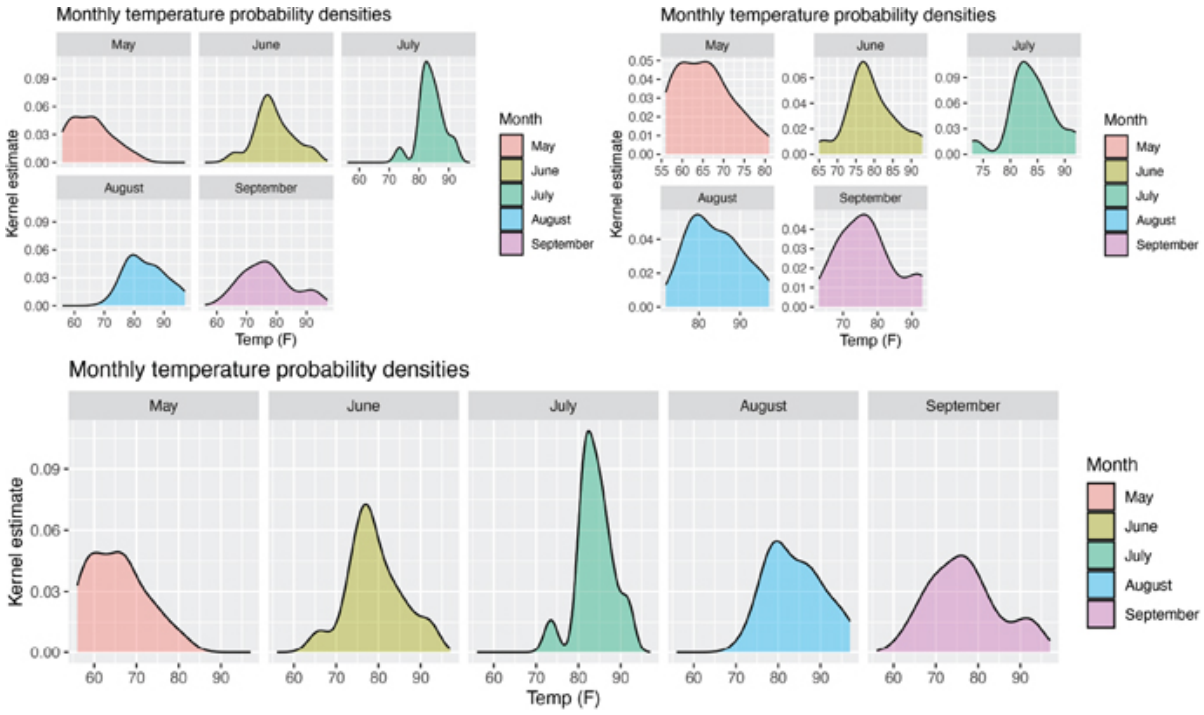


Figure 24-6: Three examples of using `facet_wrap` to display kernel density estimates of the temperature data, split by month

The alternative to `facet_wrap` is `facet_grid`, which does much the same thing but doesn't wrap the plots onto multiple rows if you're faceting by only one categorical variable. Instead, the formula *`var1 ~ var2`* is interpreted as "facet by *`var1`* as rows and by *`var2`* as columns." If you're interested in faceting by only one grouping variable in either columns or rows, then simply replace either *`var1`* or *`var2`* with a dot (.). For example, you can equivalently produce the third version of the faceted plots on the bottom of Figure 24-6 via `facet_grid` as follows:

```
R> ggp + facet_grid(.~Month)
```

You can facet any standard data plot, including those you covered in Chapter 14. For example, recall the `survey` data frame object in the `MASS` package, used most recently in Section 24.1. Ensure the `MASS` package is loaded and then call the following line:

```
R> surv <- na.omit(survey[,c("Sex","Wr.Hnd","Exer")])
```

As you've seen, this creates a version of the `survey` data made up of just three variables—sex, writing handspan, and exercise frequency—and uses `na.omit` to remove rows with any missing values. Now consider the following two objects, which create two independent sets of box plots of the same data in `surv`, but split the individual plots differently:

```
R> ggbox1 <- ggplot(surv,aes(x=Sex,y=Wr.Hnd)) + geom_boxplot() +
                facet_grid(.~Exer)
R> ggbox2 <- ggplot(surv,aes(x=Exer,y=Wr.Hnd)) + geom_boxplot() +
                facet_grid(.~Sex)
```

The first object, `ggbox1`, maps the categorical variable `Sex` to the x-axis and `Wr.Hnd` to the y-axis; box plots are requested with the relevant geom, and a further split according to `Exer` is stipulated using `facet_grid`. By comparison, the second object swaps the roles of the two categorical variables: It instead requests faceting by `Sex`, with `Exer` being the x-axis-mapped variable.

To actually visualize these two `ggplot` objects and hence assess the impact of these choices, you could just print them separately at the prompt. As a more elegant solution, which will display both in the same device, use the `grid.arrange` function covered in Section 24.3.1. Ensure the `gridExtra` package is loaded (for access to the `grid.arrange` function itself), and execute the following line:

```
R> grid.arrange(ggbox1,ggbox2)
```

This creates the plot in Figure 24-7.

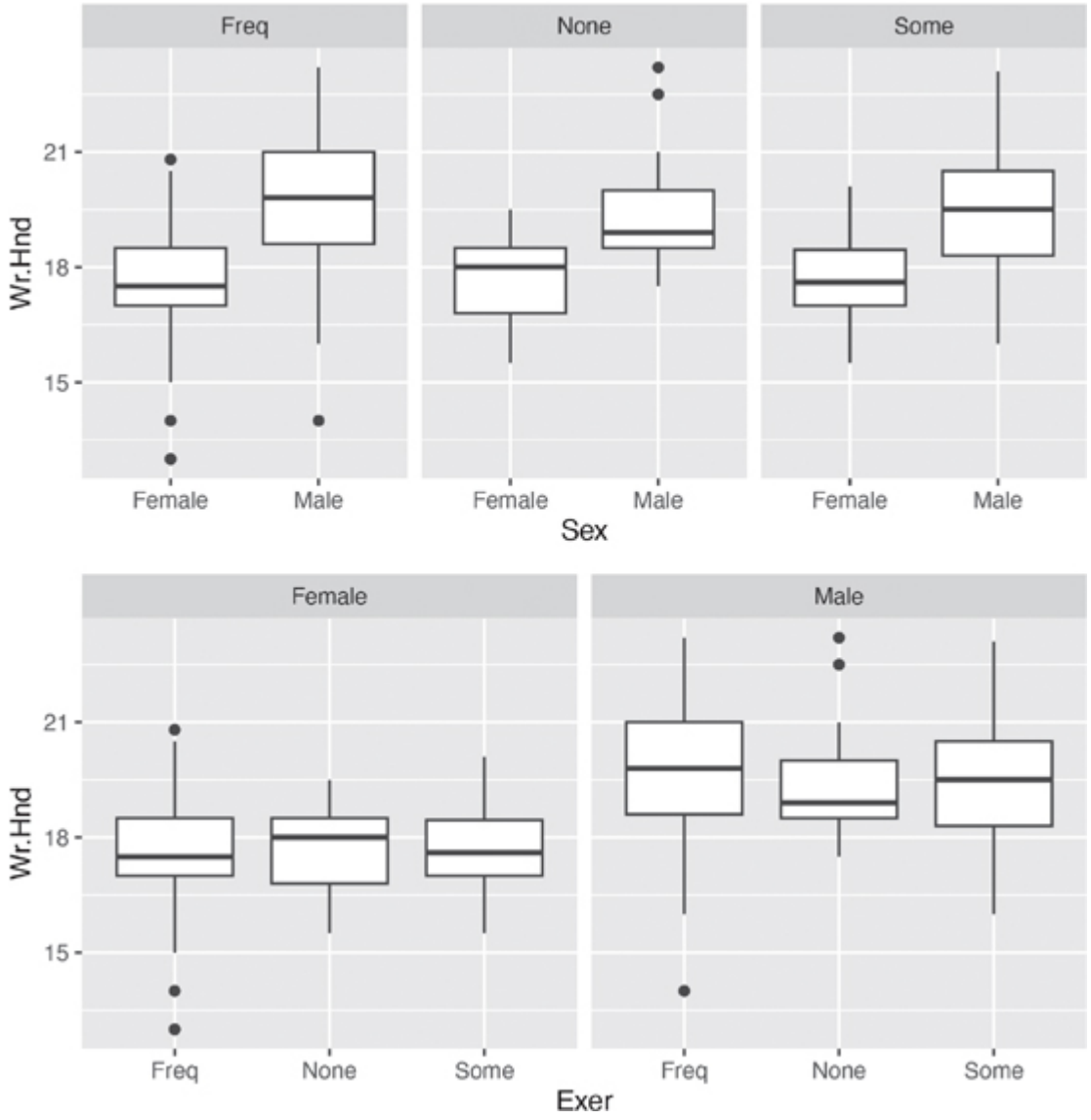


Figure 24-7: Using `grid.arrange` in tandem with faceting behavior to illustrate the effect of different choices of faceting variable when creating box plots of the `survey` data

While both `ggbox1` and `ggbox2` aim to provide side-by-side box plots of the writing handspan observations for the different subgroups, the visual effect of the different configurations of x-axis mapping and faceting variable can be important. There's no right or wrong way to do this in general, but it's worth noting that the physical separation of the individual facets can make a direct visual comparison between the corresponding sub-groups more difficult than if the faceting variable had instead been mapped to the x-axis. For that reason, if you're faced with a similar situation (for example, when you want to create side-by-

side box plots, violin plots, or even bar plots), it's usually sensible to avoid setting the factor whose categories/levels you want to most directly compare (with respect to the continuous variable at hand) as the faceting variable.

Two Categorical Variables

Despite the fact that the most recent example involved two categorical variables, you were still faceting by only one of them in each version of the plot. The next example shows `facet_grid` in action with *two* grouping variables. Turn your attention to the `diabetes` data frame in the `faraway` package, first seen in Section 21.5.2. After loading the package, the following code creates the object with `diab` as the data frame of interest and with missing-value rows deleted. Then, plots of cholesterol level against age, along with simple linear model fits, are faceted according to sex (rows) and body frame type (columns). Points are colored and sized according to height and weight, respectively, and two different point types differentiate the two county locations in Virginia of the study participants:

```
R> diab <- na.omit(diabetes[,c("chol","weight","gender","frame","age",
                               "height","location")])
R> ggplot(diab,aes(x=age,y=chol)) +
       geom_point(aes(shape=location,size=weight,col=height),alpha=0.75) +
       facet_grid(gender~frame) + geom_smooth(method="lm") +
       labs(y="cholesterol")
`geom_smooth()` using formula = 'y ~ x'
```

This subsequently produces Figure 24-8.

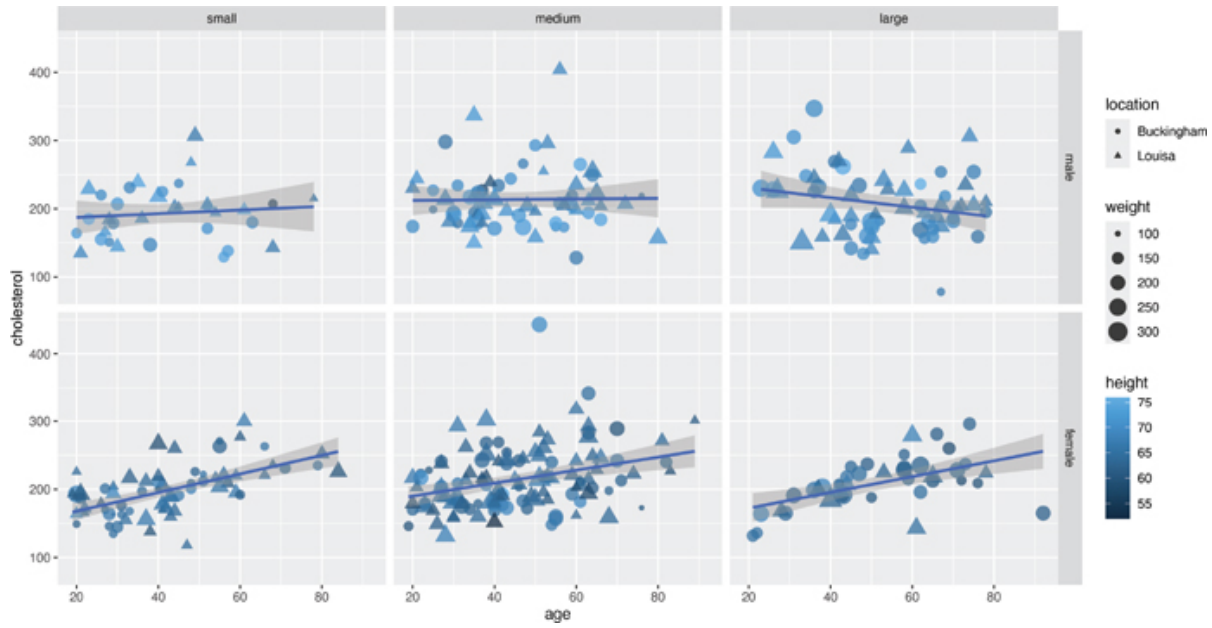


Figure 24-8: Illustrating two-way faceting in `ggplot2`, using the `diabetes` data frame from the `faraway` package

The initial call to `ggplot` tells R to use `diab` and plot total cholesterol against age. Adding `geom_point` sets the shape, size, and color of each plotted point to change according to the county location, the weight, and the height of the individuals, respectively. (As you've already seen for point size based on a continuous variable, point color is also automatically changed to vary on a continuum if the correspondingly mapped aesthetic variable isn't a factor.) Points are universally set to be slightly transparent (`alpha=0.75`) in order to better distinguish those which overlap.

So far, those commands have still defined only a single scatterplot. With the addition of the call to `facet_grid`, the formula `gender~frame` separates the plots into a different scatterplot for males/females (as rows) and for each of the three body frame types, small/medium/large (as columns). You set simple linear model fits to accompany each plot based on the default aesthetic mapping (cholesterol on age) with a call to `geom_smooth`, and a final call to `labs` clarifies the vertical axis title.

The plots themselves reveal, generally, some of the trends you'll have picked up on from an earlier analysis of these data (Section 21.5.2). Increasing age tends to be associated with an increase in mean

cholesterol, though that relationship seems, at least visually, less prominent for males. The overall smaller size of the points in the small frame column on the left makes sense: Those with a smaller frame will typically weigh less than those with larger frames. There is also a tendency for the plots on the bottom row (females) to be of a darker shade than those along the top row, indicating that on average females are typically shorter than males. Any differences between the participants from the two counties, however, is difficult to discern: There doesn't seem to be a systematic departure of the pattern of the points for Buckingham County (●) from that of the points for Louisa County (▲). Faceted plots are thus often useful in informing the next step in a statistical analysis where multivariate relationships abound, such as aiding the design of an appropriate regression model.

As a final note, the plots created in this section serve to further highlight the relative ease with which `ggplot2` functionality can be deployed to visually interrogate your data—typically involving partitioning the observations by one or more factors—in terms of both an individual image and an arrangement of images. Although similar plots are still possible using base R methods, that route demands a somewhat finer or lower-level handling of the details of your data subsets, as well as any varying aesthetic features. This certainly doesn't mean base R graphics should be ignored; said lower-level control offers much greater flexibility in creation of intricate plots (you'll see some nice new tricks in Chapter 25). It's just that you can create certain graphical displays with less coding effort (and often with a prettier end result) by utilizing Wickham's well-received implementation of the grammar of graphics.

Exercise 24.1

Load the `MASS` package and inspect the help file for the `UScereal` data. This data frame provides nutritional and other information

concerning breakfast cereals for sale in the United States in the early 1990s.

a. Create a copy of the data frame and name it `cereal`. Use skills from Section 4.3.5 to collapse the `mfr` (manufacturer) column of `cereal` to be a factor with only three levels, with the corresponding labels `"GeneralMills"` (for those original entries labeled `"G"`), `"Kelloggs"` (for those original entries labeled `"K"`), and `"Other"` (for all other entries—that is, those labeled either `"N"`, `"P"`, `"Q"`, or `"R"`). In addition, convert the `shelf` variable (shelf number from floor) to be a factor.

b. Using `cereal`, construct and store two `ggplot` objects.

    i. A scatterplot of calories on protein. Points should be colored according to shelf position and shaped according to manufacturer. Include simple linear regression lines for calories on protein, split according to shelf position. Ensure tidy axis and legend titles.

    ii. A set of kernel estimates of the distribution of calories, split with filled color differentiating shelf positions. Use 50 percent opacity and again ensure tidy axis and legend titles.

c. Arrange the two plots in (b) on a single device.

d. Produce a faceted graphic of calories on protein, with each panel corresponding to a manufacturer as defined in your `cereal` object. A LOESS smoother with a 90 percent span should be superimposed upon each scatterplot. In addition, the points should be colored according to sugar content, sized according to sodium content, and shaped according to shelf position.

Load the `car` package (downloading and installing it first if you haven't already) and consider the `Salaries` object: a data frame

detailing the salaries (in US dollars) of 397 academics working in the United States during the 2008–2009 school year (Fox & Weisberg 2011). An inspection of the help file `?Salaries` informs you of the present variables, which, in addition to the salary figure, include each academic's rank, sex, and research discipline (as factors), as well as the number of years of service.

e. Produce a `ggplot` object, named `gg1`, of a scatterplot of salary on the vertical axis against years of service on the horizontal axis. Color should be used to distinguish between males and females, along with sex-specific LOESS trends; ensure axis and legend titles are understandable. View your plot.

f. Create the following three additional plot objects—naming them `gg2`, `gg3`, and `gg4`, respectively—again ensuring tidy axis and legend titles:

    i. Side-by-side box plots of salary, split by rank. Each box plot should be further split up according to sex (this can be done simply in the default aesthetic mapping; try assigning the sex variable to either `col` or `fill`).

    ii. Side-by-side box plots of salary, split by discipline, with each discipline split further by sex using color or fill.

    iii. Kernel density estimates of salary, using color fills with 30 percent opacity to distinguish rank.

g. Display your four plot objects (`gg1`, `gg2`, `gg3`, and `gg4`) from (e) and (f) in a single device.

h. Finally, plot the following:

    i. A series of kernel density estimates of salary using color fills of 70 percent opacity to distinguish

between males and females, faceted by academic rank.

ii. Scatterplots of salary on years of service, using color to distinguish between males and females, with two-way faceting by discipline (as rows) and by academic rank (as columns). Each scatterplot should have a sex-specific simple linear regression line with confidence band superimposed and have free horizontal scales.

24.4 Interactive Tools in ggvis

To wrap up this chapter, I'll touch on a more recent addition to the "gg" family, `ggvis`, by Wickham & Chang (2024). The package provides a flexible means by which to create statistical plots you can interact with. The results are provided as web graphics. You'll see the plots you create pop up as a new tab in your default web browser; if you're using the RStudio IDE, the `ggvis` graphic will appear as embedded within the Viewer pane.

For a thorough discussion and set of examples using `ggvis`, visit the `ggvis` website at *https://ggvis.rstudio.com*. The site contains a beginner-friendly tutorial and recipe book of things that are currently possible. Here, I'll just give you an overview of `ggvis`; the package version at the time of writing is 0.4.9.

Install the `ggvis` package along with its dependencies and then load it with a call to `library("ggvis")`. Also make sure you have access to the student survey data, `survey`, by loading the `MASS` package. Create the following object to be used in the upcoming examples:

```
R> surv <- na.omit(survey[,c("Sex","Wr.Hnd","Height","Smoke","Exer")])
```

The usual way to begin a `ggvis` graphic is to declare the data frame of interest with a call to `ggvis`. Similar to static `ggplot` objects, this sets up a

`ggvis` object and defines the variables to be used. Then, you pile on the layers using functions with names prefaced by `layer_` to create the desired plot; this is again similar in nature to the way you might think of `ggplot` graphics and the `geom_` functions. When you use the variables from the declared data frame in `ggvis`, they must be prefaced by a `~`, which explicitly tells R that you're referring to a column of that data frame and not another object of the same name in your workspace. Furthermore, the authors of `ggvis` recommend using pipes (Section 9.3) as the most efficient way to create plots, so I will do the same using the base R pipe operator `|>` (you can also use the older pipe operator `%>%` from the `magrittr` package).

Let's start with a simple, non-interactive static plot as an example. You can obtain the topmost image of Figure 24-9, a histogram of the height measurements, with the following execution:

```
R> surv |> ggvis(x=~Height) |> layer_histograms()
Guessing width = 2 # range / 25
```

The `surv` data frame is declared, and then you pipe this to `ggvis`, which instructs the `Height` variable to be mapped to the x-axis. A pipe to `layer_histograms` produces the graphic, which assigns a default bin width based on the range of the x-mapped data.

So what? You've already created lot of histograms. But wouldn't it be great if you could play with the value of the bin width without needing to create static plot after static plot? The `input_` collection of commands in `ggvis` allows you to instruct the resulting graphic to take interactive input. Consider the following code:

```
R> surv |> ggvis(x=~Height) |>
       layer_histograms(width=input_slider(1,15,label="Bin width:"),
       fill:="gray")
Showing dynamic visualisation. Press Escape/Ctrl + C to stop.
```

The image on the bottom of Figure 24-9 shows a screenshot of my result.

Here, the `width` argument that controls the feature of interest is instructed to take the result of `input_slider`, which sets up an interactive slider button. The range of the slider values for `width` is set as `1` to `15` (inclusive), and the optional argument `label` provides a title for the interactive gadget. Using `fill` in `layer_histogram` sets the color of the bars. Note the particular assignment `fill:="gray"` uses `:=`, not just `=`. The `=` alone is used in `ggvis` for mapping variables—that is, when the feature of interest is to be passed a variable with respect to which some feature will change, essentially like a local aesthetic mapping in `ggplot2` commands. The operator `:=` should be interpreted as setting an overall or global value, that is, when you simply intend to universally impact a certain feature.

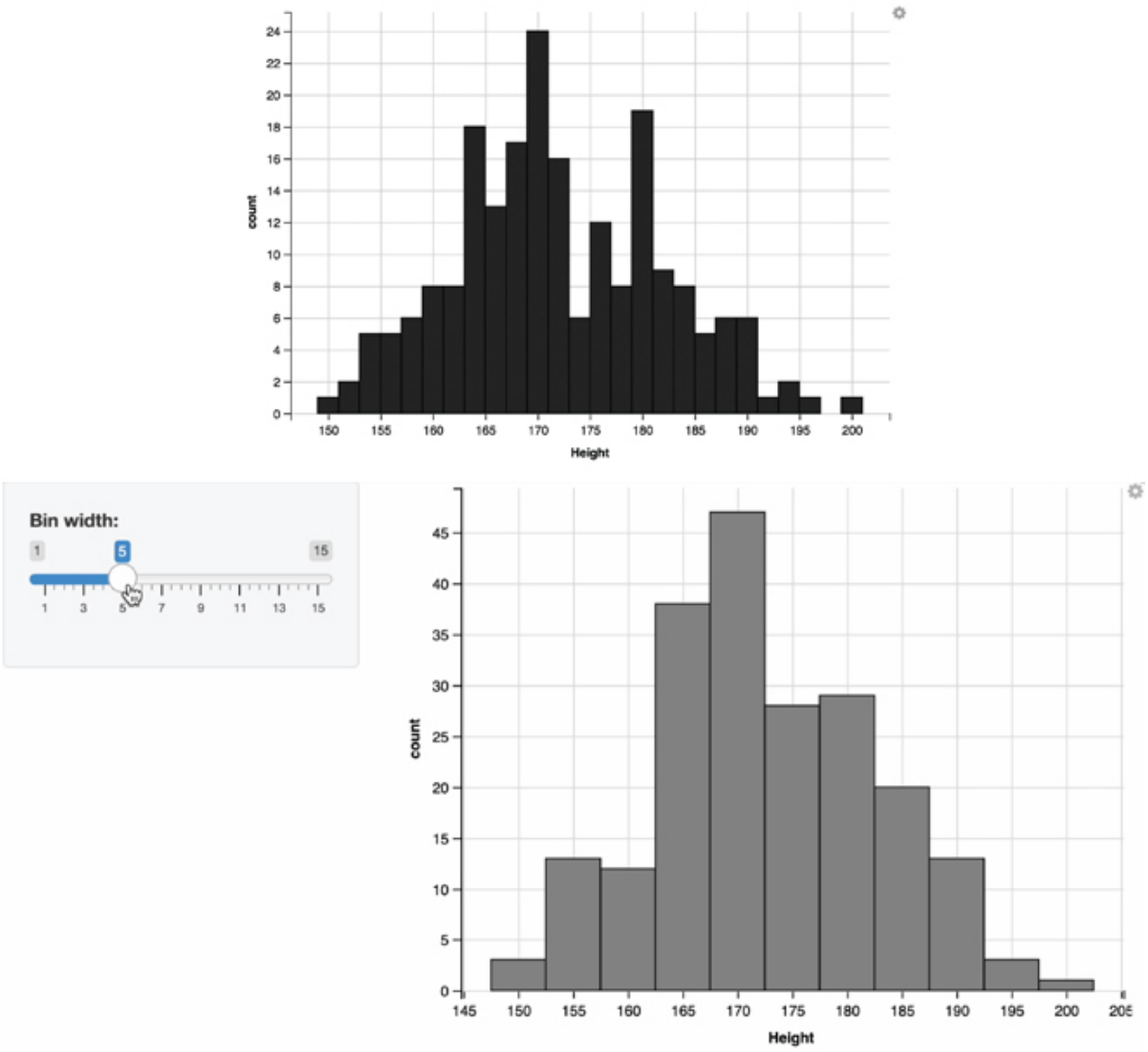


Figure 24-9: Histograms of the height observations from the `survey` data frame, using `ggvis`. Top: Default static plot. Bottom: A screenshot of the result of incorporating a slider button tied to the bin width; the user can interact with, and immediately see, the effect of altering the bins.

Once the code successfully executes, you can experiment with sliding the button for smaller and larger bin widths. It's interesting to gauge just how much your interpretation of the distribution changes along with it. As the text printed out beneath the executed commands in the console tells you, you have to exit the interactive plot to continue using your R session. Pressing ESC will terminate the interactivity and return control to the user at the prompt.

Other kinds of interactive abilities include `input_select` (for a drop-down menu), `input_radiobuttons` (radio button options), and `input_checkbox` (for checkboxes). You can even set up interactive text with `input_text` or numeric input boxes with `input_numeric`. See the relevant help files for these functions or the `ggvis` website for further details on how to use such input to influence an interactive plot.

For another example based on the `surv` object created at the beginning of this section, let's try a scatterplot. For a simple static plot, run the following:

```
R> surv |> ggvis(x=~Wr.Hnd,y=~Height,size:=200,opacity:=0.3) |>
       layer_points()
```

I won't show this result here, but you can see from the call to `ggvis` that you'll be plotting height against writing handspan and that you're universally enlarging the points as well as setting a universal level of 30 percent opacity. The last pipe to `layer_points` produces the image. As with the static plot of the histogram, since there's no interactivity, you don't need to "exit" the plot; you're returned control at the console prompt immediately.

For a more interesting graphic, try this:

```
R> filler <- input_radiobuttons(c("Sex"="Sex","Smoking status"="Smoke",
                                  "Exercise frequency"="Exer"),map=as.name,
                                label="Color points by...")
R> sizer <- input_slider(10,300,label="Point size:")
R> opacityer <- input_slider(0.1,1,label="Opacity:")
R> surv |> ggvis(x=~Wr.Hnd,y=~Height,fill=filler,
                 size:=sizer,opacity:=opacityer) |>
       layer_points() |> add_axis("x",title="Handspan") |>
       add_legend("fill",title="")
Showing dynamic visualisation. Press Escape/Ctrl + C to stop.
```

First, three objects are created for the interactive bits. A set of radio buttons specifies the color of the plotted points according to one of three possible categorical variables (`Sex`, `Smoke`, or `Exer`), and two slider buttons control the point size and opacity. Note that when you intend

to use variables from the data frame as ingredients for interactive behavior, you need to supply their exact names as a vector of character strings and set the optional `map=as.name`; this is done when defining the `filler` object. In the subsequent call to `ggvis`, you pass `filler` to `fill`, using `=`. The two slider buttons in the objects `sizer` and `opacityer` are passed to the relevant arguments with `:=` since they don't depend on variables in the data frame and will be effecting global change. The call to `layer_points` generates the plot, and additional pipes to `add_axis` and `add_legend` simply tidy up the x-axis and legend titles.

The topmost image of Figure 24-10 shows a screenshot of the result, where I've selected the point color to vary according to the exercise frequency variable, reduced the point size, and chosen a moderate-to-high level of opacity.



Figure 24-10: Two examples of `ggvis` scatterplots of height on handspan of the student survey data. Top: Color (fill) changed using radio buttons based on sex, smoking status, or exercise frequency and slider buttons for point size and opacity. Bottom: Splitting points by sex using color and superimposing sex-specific LOESS smoothers and corresponding CIs; their smoothing span is controllable via a slider button.

Finally, let's produce the same scatterplot but settle on using sex to color the points. You can add separate LOESS smoothers to males and

females (look back at Figure 24-3) and dynamically control the degree of smoothing using a slider button. This last example, a screenshot of which appears on the bottom of Figure 24-10, is the result of executing the following code:

```
R> surv |> ggvis(x=~Wr.Hnd,y=~Height,fill=~Sex) |> group_by(Sex) |>
       layer_smooths(span=input_slider(0.3,1,value=0.75,
                                       label="Smoothing span:"),
                     se=TRUE) |> layer_points() |>
       add_axis("x",title="Handspan")
Showing dynamic visualisation. Press Escape/Ctrl + C to stop.
```

LOESS smoothers are added with the `layer_smooths` command, whose `span` argument, the target parameter of interest, is assigned an `input_slider`. As you've seen previously, its range of possible values is set with the first two arguments. The optional `value` argument (also applicable to other `input_` functions) specifies the starting value when the plot is first initialized. Additionally, the `se=TRUE` argument to `layer_smooths` ensures 95 percent CIs accompany the smoothed trends. Note that the pipe to `layer_smooths` is preempted by a pipe through `group_by(Sex)`. Without it, the smoothers would simply be applied to the `x` and `y` data as a whole (also note, at the time of writing, you don't preface the variable name with `~` in `group_by`).

In summary, `ggvis` offers the user a more dynamic experience for visual data exploration. These tools are especially useful in media such as presentations or websites, where you can offer your audience an interactive appreciation of your data in a grammar-of-graphics style. If you're interested in using these tools, see the `ggvis` website noted at the beginning of this section as your first port of call.

Exercise 24.2

Ensure the `car` and `ggvis` packages are loaded. Revisit the `Salaries` data frame you looked at in Exercise 24.1; inspect the help file `?Salaries` to remind yourself of the present variables.

a. Produce an interactive scatterplot of salary on the vertical axis and the years of service on the horizontal axis. Employ radio buttons to color points according to either academic rank, research discipline, or sex. Use pipes to `add_legend` and `add_axis` to omit a legend title (simply set the title to be an empty string) and to tidy up the axis titles, respectively.

b. Note that a pipe to `layer_densities` (which you've not yet met) is used to produce kernel density estimates, similar to those appearing in Figures 24-4, 24-5, and 24-6.

    i. Use `ggvis` to create a static (non-interactive) plot of kernel density estimates of salary distributions, split up according to academic rank. To do this, assign the salary variable to `x` and the rank variable to `fill`, followed by a pipe to `group_by` to explicitly instruct grouping by the rank variable (this function was used in the most recent example in this section). Lastly, piping to `layer_densities` (no additional arguments needed) will generate the graphic. Your result should resemble the `gg4` object from Exercise 24.1 (f).

    ii. Just like the `width` argument to `layer_histograms` is used to control the appearance of a histogram, the `adjust` argument in `layer_densities` can be used to control the degree of smoothness of the kernel estimates. Reproduce the rank-specific kernel estimates from the previous plot, but this time the graphic should be interactive: Implement a slider button with a range of 0.2 to 2 and a label of `"Smoothness"` to control the smoothing adjustment.

At your discretion, either suppress or clarify the axis and legend titles of the result.

Ensure you have the `MASS` package loaded, once more gaining access to the `UScereal` data frame. If you haven't already done so, inspect the help file `?UScereal` and re-create the `cereal` object exactly as specified in Exercise 24.1 (a). Then do the following:

c. Set up an object for radio buttons to choose among the (simplified) manufacturer, shelf, and vitamin variables. Make sure the labels for each radio button are clear, and set up an appropriate title label for what will form the collection of options to color the points. Name the object `filler`.

d. Borrowing the `sizer` and `opacityer` objects from Section 24.4 (page 704) and using the object you just created in (c) to control `fill`, create an interactive scatterplot of calories (y-axis) on protein (x-axis). Tidy up the axis titles and suppress the legend title (set it to be an empty string) for the point color fill. The result should essentially be the same, in terms of functionality, as the graphic appearing as the topmost screenshot in Figure 24-10.

e. Create a new object for the same set of radio buttons as specified in (c) that will control the shape of the points (in other words, the characters used to plot points). Modify the title label accordingly. Name this object `shaper`.

f. Finally, re-create the interactive scatterplot of calories on protein exactly as in (d), but this time additionally assigning `shaper` from (e) to the `shape` modifier in your call to `ggvis`. Use additional pipes to `add_axis` and `add_legend` to clarify or suppress axis and legend titles. When adding the legend for point shape, use the call

```
add_legend("shape",title="",
           properties=legend_props(legend=list(y=100)))
```

which simply moves the legend for the `shape` modifier vertically downward to ensure it won't overlap the legend for point fill. Lastly, pipe to the following call, which eliminates the slight animation delay that occurs by default when switching between options in the interactive graphic:

```
set_options(duration=0)
```

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`geom_smooth`	Trend line geom (`ggplot2`)	Section 24.2.1, p. 687
`loess`	Calculate LOESS	Section 24.2.1, p. 688
`Rev`	Reverse vector elements	Section 24.2.1, p. 688
`adjustcolor`	Alter color opacity	Section 24.2.1, p. 688
`grid.arrange`	Multiple `ggplot2` plots (`gridExtra`)	Section 24.3.1, p. 692
`facet_wrap`	One-factor faceting (`ggplot2`)	Section 24.3.2, p. 695
`facet_grid`	Two-factor faceting (`ggplot2`)	Section 24.3.2, p. 696
`ggvis`	Initialize `ggvis` plot	Section 24.4, p. 702
`layer_histograms`	`ggvis` histogram layer	Section 24.4, p. 702

Function/operator	Brief description	First occurrence
`input_slider`	Interactive slider (`ggvis`)	Section 24.4, p. 702
`:=`	Universal `ggvis` assignment	Section 24.4, p. 702
`layer_points`	`ggvis` points layer	Section 24.4, p. 704
`input_radiobuttons`	Interactive buttons (`ggvis`)	Section 24.4, p. 704
`add_legend`	Add/alter `ggvis` legend	Section 24.4, p. 704
`layer_smooths`	`ggvis` trend line layer	Section 24.4, p. 705
`add_axis`	Add/alter `ggvis` axis	Section 24.4, p. 705

25
DEFINING COLORS AND PLOTTING IN HIGHER DIMENSIONS



Now that you've mastered some fundamental data visualization skills, you can start to go beyond the standard x- and y-axes by, for example, coloring points according to a third continuous variable, or adding a y-axis to build a three-dimensional (3D) plot. Higher-dimensional plots allow you to visually explore your data or models using more variables than would otherwise be possible.

In this chapter, you'll first cover some technical details about handling colors and creating custom color palettes in R. You'll then proceed to learn how to create four new types of plot: 3D scatterplots, contour plots, pixel image plots, and perspective plots.

25.1 Representing and Using Color

Color plays a key role in many plots. While color can be used for purely aesthetic reasons, it can more importantly aid in interpreting your data/models when used to represent an additional variable. It's beneficial to understand the basics of how R formally represents and handles

colors before learning about more complicated data and model visualization tools. In this section, you'll examine common ways to create and represent specific colors and learn how to define and use a cohesive collection of colors, known as a *palette*.

25.1.1 Red-Green-Blue Hexadecimal Color Codes

When specifying colors in plots, your instruction to R has so far been given either in the form of an integer value from `1` to `8` or as a character string (see relevant comments in Section 7.2.3). For programming purposes, you need a more objective representation of these colors.

One of the most common methods of color specification is to specify different *saturations* or *intensities* of three primaries—red, green, and blue (RGB)—which are then mixed to form the resulting target color. Conventionally, each primary component of the standard RGB system is typically assigned an integer from 0 to 255 (inclusive). Such mixtures are therefore able to form a total of 256^3 = 16,777,216 possible colors.

You always express these values in (R, G, B) order; the result is commonly referred to as a *triplet*. For example, (0, 0, 0) represents full black, (255, 255, 255) represents full white, and (0, 255, 0) is full green.

When creating many kinds of R plots, the `col` argument lets you select one of eight colors by supplying an integer from 1 to 8. You can find the actual identifiers of these colors with the following call:

```
R> palette()
[1] "black"   "#DF536B" "#61D04F" "#2297E6" "#28E2E5" "#CD0BBC"
[7] "#F5C710" "gray62"
```

We obtain a set of character strings; the first and last entries are humanreadable. These two—`"black"` and `"gray62"`—are specific entries of the socalled "named" list of 650+ color presets you can choose from when creating plots. Try typing `colors()` at the prompt, and you'll get the complete list (R is somewhat famous for use of delightfully imaginative color names):

```
R> colors()
  [1] "white"                "aliceblue"
  [3] "antiquewhite"         "antiquewhite1"
  [5] "antiquewhite2"        "antiquewhite3"
  [7] "antiquewhite4"        "aquamarine"
  [9] "aquamarine1"          "aquamarine2"
 [11] "aquamarine3"          "aquamarine4"
 [13] "azure"                "azure1"
 --snip--
```

The other six entries in the output of `palette()`, beginning with a `#`, are coded in *hexadecimal* format, which I'll explain in a moment.

You can interrogate any valid character string that R recognizes as a color, be it one of the named presets or in a hexadecimal form, to determine the contributing RGB levels. To do this, supply the desired color names as a vector of character strings to the built-in `col2rgb` function. Here's an example:

```
R> myrgbs <- col2rgb(c("black","gray62","#DF536B","turquoise"))
R> myrgbs
      [,1] [,2] [,3] [,4]
red      0  158  223   64
green    0  158   83  224
blue     0  158  107  208
```

The result is a matrix of RGB values, with each column representing one of your specified colors. Notice that `"black"` is all zeros and that `"gray62"` is the result of a perfectly equal RGB concentrations (this makes sense, as full white would have all three at maximum concentrations of 255). The third column, corresponding to `"#DF536B"` (the second entry in the standard set of eight returned by the `palette` command), indicates this color must be some kind of softened red because its highest individual concentration is the contribution by `red`; this is tempered by weaker inputs of `blue` and `green`. Lastly, the fourth column corresponds to another color from the longer named list of presets, `"turquoise"`; it makes sense that this greenish-blue color shares the strongest contributions from those two RGB components. This should give you a good

impression of what R actually understands, in an RGB sense, when you request colors using the corresponding character strings in creating any given plot.

A hexadecimal, or *hex code*, is a numeric coding system often used to represent RGB triplets in computing. As you've seen, these are provided in R as a character string with a leading `#` followed by six alphanumeric characters: Valid characters are the letters *A* through *F* and the digits 0 through 9. The first pair of characters represents the red component, and the second and third pairs represent green and blue, respectively.

If you have (or create) one or more RGB triplets, you can turn them into hex codes for R to use in any subsequent plotting through the `rgb` function. This command takes a three-column matrix of RGB values; as such, it expects each (R, G, B) specification to be a row of that matrix (not a column like the output of a call to `col2rgb` would provide). You'll also need to tell `rgb` that your maximum color value, as per the standard RGB format, is 255 (since it scales this by default and uses 1).

Suppose now we want to turn the RGB values obtained for the four colors in the previous example, stored as `myrgbs`, into hex codes. We can use the matrix transpose function, `t` (refer to Section 3.1.3), to turn the `myrgbs` object into the required three-column format first:

```
R> t(myrgbs)
     red green blue
[1,]   0     0    0
[2,] 158   158  158
[3,] 223    83  107
[4,]  64   224  208
```

The following call uses this form to execute `rgb`, specifying `maxColorValue` accordingly:

```
R> rgb(t(myrgbs),maxColorValue=255)
[1] "#000000" "#9E9E9E" "#DF536B" "#40E0D0"
```

The output tells you the hexadecimal codes for the RGB values R refers to with strings `"black"`, `"gray62"`, `"#DF536B"`, and `"turquoise"`, respectively.

Unsurprisingly, the third entry has simply returned back to you the originally supplied hex code.

I won't go into the specifics of converting a standard RGB triplet to a hexadecimal here because it's beyond the scope of this book, but know that R represents any colors you create using RGB triplets as hex codes. You should be able to at least recognize a hexadecimal when you're working with colors and color palettes in plotting.

For an even more colorful exploration, let's write a modest little function to plot points in individual colors and label them appropriately with RGB triplets and corresponding hex codes. Consider the following in the editor:

```
pcol <- function(cols){
        n <- length(cols)
        par(mar=rep(1,4))
        plot(1:5,1:5,type="n",xaxt="n",yaxt="n",ann=FALSE)
        for(i in 1:n){
                pt <- locator(1)
                rgbval <- col2rgb(cols[i])
                points(pt,cex=4,pch=19,col=cols[i])
                text(pt$x+1,pt$y,family="mono",
                     label=paste("\"",cols[i],"\"","\nR: ",rgbval[1],
                                 "  G: ",rgbval[2],"  B: ",rgbval[3],
                                 "\nhex: ",rgb(t(rgbval),maxColorValue=255),
                                 sep=""))
        }
}
```

The function `pcol` takes one argument, `cols`, intended to be a character vector of color names recognized by R. When you execute `pcol`, it opens a new graphics device and equalizes the figure margin settings to be one line on each side. A plot is begun, fully suppressed except for a box. This is so you can use `locator` (see Section 23.3) to place points in the plot region, implemented in a `for` loop, one after the other. Each point represents one of the `cols`, and after its coordinates are returned from `locator`, `points` puts down a large dot of the color at hand,

with `text` providing annotation to the right of each point achieved using `paste` (refer to Section 4.2). This annotation includes the R color name, the RGB triplet, and the hex code on top of one another; the latter two are found using `col2rgb` and `rgb` exactly as demonstrated earlier.

The following code sets up the device, first storing 14 valid R color names (chosen randomly) in the character vector `mycols`. After exhausting these with mouse clicks in different areas of the plot region, the execution is complete:

```
R> mycols <- c("black","blue","royalblue2","pink","magenta","purple",
               "violet","coral","lightgray","seagreen4","red","red2",
               "yellow","lemonchiffon3")
R> pcol(mycols)
```

When I execute `pcol` as shown here, starting with a click in the upper-left corner, I click through the points, producing two rough columns of seven each on my graphics device. Figure 25-1 shows the result in a roughly square plotting window.



Figure 25-1: Various named R colors alongside their corresponding RGB triplets and hex codes.

In essence, you can obtain any color you want (in other words, you have far greater flexibility beyond the list of named preset colors that are built into R) by specifying an RGB triplet and obtaining its hex code. These hexadecimals can be supplied as-is to any of the traditional R graphics functions where you specify color (commonly to a `col` argument). You'll see this as the chapter progresses. Since they're just character strings, you can also assign a hex code or a vector of hex codes (for example, if you're creating your own custom colors) to a new object in your R workspace so you can use it in subsequent plotting.

25.1.2 Built-in Palettes

Being able to implement your own colors is most useful when you need many colors, the collection of which is referred to as a *palette*. You'll typically need a palette when using color to describe something on a continuum, like the various shades of blue representing the height measurements in Figure 24-8 on page 698.

There are a number of color palettes built into the base R installation. These include the functions `hcl.colors`, `rainbow`, `heat.colors`, `terrain.colors`, `topo.colors`, `cm.colors`, `gray.colors`, and `gray`. With the exception of `gray`, you directly specify the number of colors you want, and they'll be returned as a character vector of hex codes representing an equally spaced sequence over the entire color range of that particular palette.

The `hcl.colors` function actually encompasses a richer collection of well over 100 palettes born out of a well-received contributed package for handling colors in R, `RColorBrewer` (Neuwirth 2022). This function has an optional argument, `palette`, used to specify the particular choice (to find the list, execute `hcl.pals()` at the R prompt).

It's easiest to see these various options in action with a visualization. The following code generates exactly 600 colors from each palette:

```
R> N <- 600
R> hclN <- hcl.colors(N)
R> rbowN <- rainbow(N)
R> heatN <- heat.colors(N)
R> terrN <- terrain.colors(N)
R> topoN <- topo.colors(N)
R> cmN <- cm.colors(N)
R> gry1N <- gray.colors(N)
R> gry2N <- gray(level=seq(0,1,length=N))
```

First, note that I haven't specified the optional `palette` argument in use of `hcl.colors`; note the resulting `N` colors therefore reference the default palette, `palette="viridis"`. Second, note that instead of a single integer, `gray` expects a numeric vector of values between 0 (full black) and 1 (full white) to provide a grayscale. Its counterpart function, `gray.colors`, works

the same as the other built-in palettes but defaults to a slightly narrower visual range between the extremes of black and white. These can be reset using the optional arguments `start` and `end`, which you'll see shortly.

The next code chunk uses skills from Chapter 23 to initialize a new plot with custom margin spacing and then uses vector repetition to place 600 points for each palette in a single call to `points`. These 600 points are colored correspondingly using the vectors of hex codes arising from the different palette functions:

```
R> par(mar=c(1,9,1,1))
R> plot(1,1,xlim=c(1,N),ylim=c(0.5,8.5),type="n",xaxt="n",yaxt="n",ann=FALSE)
R> points(rep(1:N,8),rep(8:1,each=N),pch=19,cex=3,
          col=c(hclN,rbowN,heatN,terrN,topoN,cmN,gry1N,gry2N))
R> axis(2,at=8:1,labels=c("hcl.colors","rainbow","heat.colors",
                          "terrain.colors","topo.colors","cm.colors",
                          "gray.colors","gray"),
        family="mono",las=1)
```

Figure 25-2 shows the result.

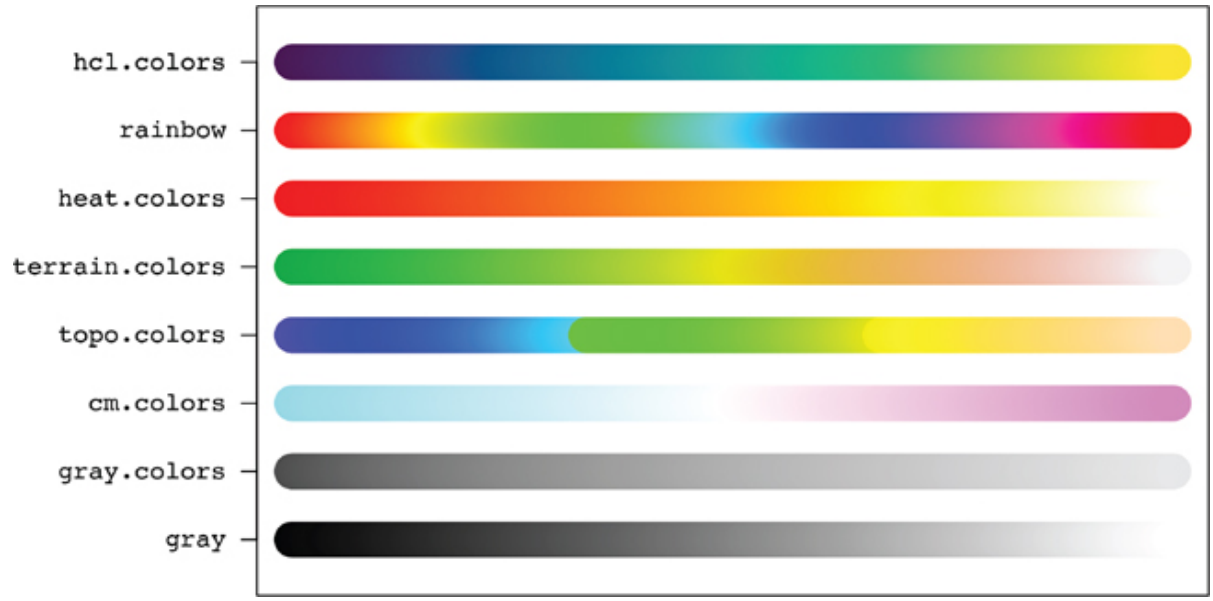


Figure 25-2: Showcasing the color ranges of the built-in palettes, with default limits used in `gray.colors`.

Access the help files for further details on these palettes and usage of their corresponding functions. The grayscale palettes are described under the help files `?gray.colors` and `?gray`, while the others are all listed under the same file accessible by typing any one of the relevant function names, such as `?rainbow`.

25.1.3 Custom Palettes

You're not restricted to the ready-to-use color designs. The `colorRampPalette` function allows you to create your own palettes; you supply it two or more desired key colors, and it creates a palette that transitions between them. The result of a call to `colorRampPalette` is itself a function, one that behaves exactly like the built-in palette functions noted in Section 25.1.2.

Say you'd like to be able to generate colors on a scale between purple and yellow. You specify the key colors to be interpolated, in the desired order, as a character vector of names from the collection that R recognizes. The following line creates this palette function:

```
R> puryel.colors <- colorRampPalette(colors=c("purple","yellow"))
```

Let's create another one, this time picking a palette that will show up a little clearer in case a color plot that ends up using it is printed in grayscale (in which case, sticking to monochromatic palettes is a good idea):

```
R> blues <- colorRampPalette(colors=c("navyblue","lightblue"))
```

Here are a couple that use more than two colors:

```
R> fours <- colorRampPalette(colors=c("black","hotpink","seagreen4","tomato"))
R> patriot.colors <- colorRampPalette(colors=c("red","white","blue"))
```

Having created a handful of custom palette functions, you can now generate any number of colors from each range just like before. After doing so, you can adapt the earlier plotting code to get a plot like that in Figure 25-2. The following code does just this:

```
R> N <- 600
R> py <- puryel.colors(N)
R> bls <- blues(N)
R> frs <- fours(N)
R> pat <- patriot.colors(N)
R> par(mar=c(1,9,1,1))
R> plot(1,1,xlim=c(1,N),ylim=c(0.5,4.5),type="n",xaxt="n",yaxt="n",ann=FALSE)
R> points(rep(1:N,4),rep(4:1,each=N),pch=19,cex=3,col=c(py,bls,frs,pat))
R> axis(2,at=4:1,labels=c("puryel.colors","blues","fours","patriot.colors"),
       family="mono",las=1)
```

The resulting plot is given in Figure 25-3.

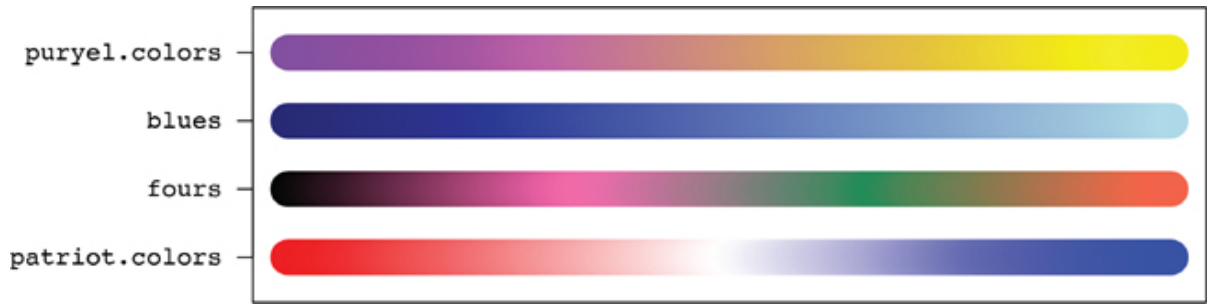


Figure 25-3: Some examples of custom color palettes created using `colorRampPalette`.

25.1.4 Approaches to Color Indexing

You've now seen how to identify groups based on a categorical variable by using color: Simply give the data corresponding to a certain level of a factor a distinct color from the others. However, assigning colors appropriately to values on a continuum requires a little more thought. There are two methods for this: through categorization or through normalization of your continuous values. Let's look first at the former approach.

Via Categorization

One way to color values according to a continuous variable is by turning it into the familiar problem of coloring points of a categorical variable. You can do this by binning your continuous values into a fixed number

of *k* categories, generating *k* colors from your palette, and matching each observation to the appropriate color based on the bin it falls into.

In Section 20.1, you plotted height against writing handspan for the `survey` data from the `MASS` package. This time, let's use color to additionally inform the non-writing handspan variable. Load the package and execute the following line:

```
R> surv <- na.omit(survey[,c("Wr.Hnd","NW.Hnd","Height")])
```

This creates the data frame object `surv`, which is made up of only the three required columns. For the sake of simplicity, any rows with missing values are removed via a call to `na.omit` (refer to Section 6.1.3).

The first thing to do is decide on your color palette. You can use one of the built-in palettes from Section 25.1.2, or create your own as per Section 25.1.3:

```
R> NW.pal <- colorRampPalette(colors=c("red4","yellow2"))
```

Here, you create a palette that goes from a dark, dull red at the lower end of the scale to a slightly faded yellow at the higher end (similar to the builtin `heat.colors` palette; see Figure 25-2). Next, you need to decide how many bins, *k*, you're going to construct for the continuous values. This determines how many distinct colors to generate from `NW.pal`. For this example, set $k = 5$:

```
R> k <- 5
R> ryc <- NW.pal(k)
R> ryc
[1] "#8B0000" "#A33B00" "#BC7700" "#D5B200" "#EEEE00"
```

The `ryc` object now stores your five `NW.pal` colors.

Now you need to actually bin the continuous values by using the `cut` function. First, set $k + 1$ break points for the bins (refer to Section 4.3.7 for a refresher). To do so, use `seq` to construct an evenly spaced sequence:

```
R> NW.breaks <- seq(min(surv$NW.Hnd),max(surv$NW.Hnd),length=k+1)
R> NW.breaks
[1] 12.5 14.7 16.9 19.1 21.3 23.5
```

These values span the range of the students' non-writing handspans, delineating your five intended bins. Then `cut` factorizes the observations with respect to those bins. You can use `as.numeric` to specifically return the indexes for extracting the appropriate color for each observation from your five ordered hex codes in `ryc` (full output is suppressed here for brevity):

```
R> NW.fac <- cut(surv$NW.Hnd,breaks=NW.breaks,include.lowest=TRUE)
R> as.numeric(NW.fac)
 [1] 3 4 3 4 3 3 3 4 3 3 2 4 3 3 4 4 4 5 4 3 4 4 5 4 3 3 4 4 2 3 5
[32] 3 2 3 4 1 3 5 5 3 3 5 4 3 4 5 3 2 3 4 5 3 4 3 3 4 3 3 3 4 2 3
[63] 2 3 3 3 3 4 3 5 3 3 3   --snip--
R> NW.cols <- ryc[as.numeric(NW.fac)]
R> NW.cols
  [1] "#BC7700" "#D5B200" "#BC7700" "#D5B200" "#BC7700" "#BC7700"
  [7] "#BC7700" "#D5B200" "#BC7700" "#BC7700"   --snip--
```

You're now ready to plot:

```
R> plot(surv$Wr.Hnd,surv$Height,col=NW.cols,pch=19,
        xlab="Writing handspan (cm)",ylab="Height (cm)")
```

The result is given on the left of Figure 25-4.

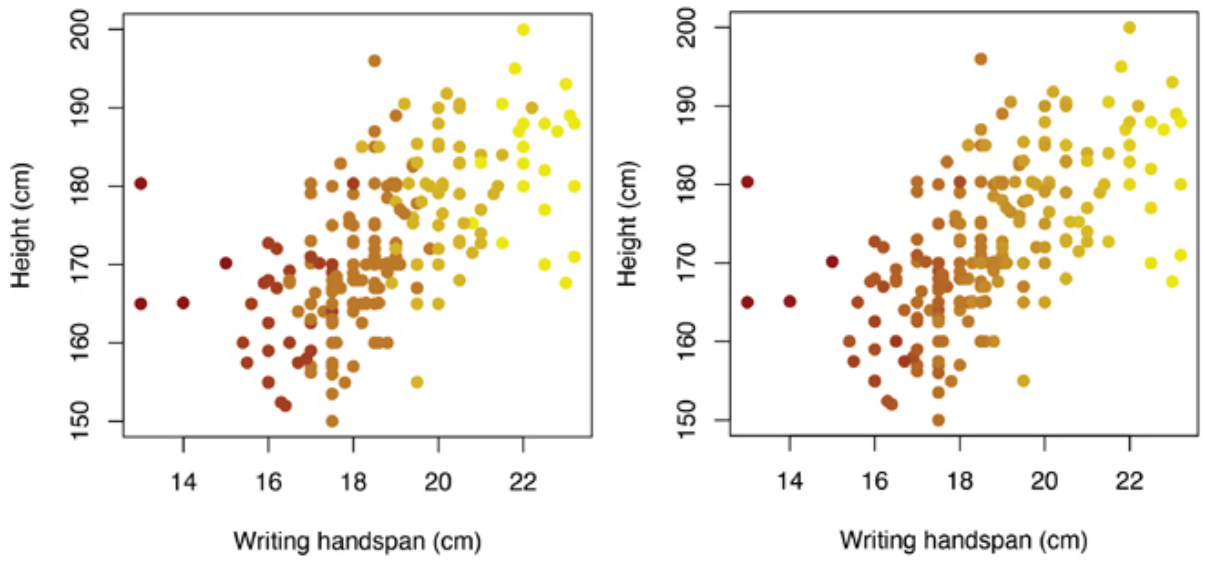


Figure 25-4: Illustrating two ways to assign color to points based on a continuous value: via categorization (left) and via normalization (right)

Via Normalization

Using categorization to index a continuum with color is perhaps a little unsophisticated. There are plenty of ways you can bin your observations, for example, so your plot might look very different from the same plot designed by someone else. In a statistical sense, it's more precise to leave your continuous data as-is.

Recall the built-in `gray` palette mentioned in Section 25.1.2. This function behaved a little differently from the others. Instead of simply asking for a number of colors from the specified palette, you're required to provide a numeric vector of values to tell R, on a continuous scale from 0 through 1, how "far along" the palette to go. This type of behavior suits the current task perfectly, since your raw data are also on a continuous scale. To implement it, you need two things: a way to create a palette that will behave like `gray` and a *normalized* version of your continuous values that falls within the acceptable standardized range of 0 to 1.

We'll use the `colorRamp` function, which is used in the same way as `colorRampPalette`, but the result is a color palette function that expects a numeric value between 0 and 1. You'll see that in a moment. First, to

transform a collection of n original values $\{x_1, \ldots, x_n\}$ to, say, $\{z_1, \ldots, z_n\}$, where $0 \leq z_i \leq 1$; $i = 1, \ldots, n$, you can employ the following equation:

$$z_i = \frac{x_i - \min x_i}{\max x_i - \min x_i} \quad (25.1)$$

Let's make that an R function by writing the following in the R editor:

```
normalize <- function(datavec){
    lo <- min(datavec,na.rm=TRUE)
    up <- max(datavec,na.rm=TRUE)
    datanorm <- (datavec-lo)/(up-lo)
    return(datanorm)
}
```

Based on a vector `datavec` as its only argument, `normalize` implements Equation (25.1), using the optional `na.rm` argument to ensure any missing values in `datavec` don't contaminate the calculation of the minimum and maximum values (see Section 13.2.1).

Copy `normalize` into your workspace and execute the following code, which shows the original non-writing handspan values (from the object `surv` you created earlier) and their corresponding normalized values (output snipped for brevity):

```
R> surv$NW.Hnd
 [1] 18.0 20.5 18.9 20.0 17.7 17.7 17.3 19.5 18.5 17.2 16.0 20.2
[13] 17.0 18.0 19.2 20.5 20.9 22.0 20.7     --snip--
R> normalize(surv$NW.Hnd)
[1] 0.50000000 0.72727273 0.58181818 0.68181818 0.47272727
[6] 0.47272727 0.43636364 0.63636364      --snip--
```

Now you need to create a new version of the color palette with `colorRamp`. This new palette spans the same colors as your first palette:

```
R> NW.pal2 <- colorRamp(colors=c("red4","yellow2"))
```

Next, use `NW.pal2` to generate the corresponding colors for each observation based on the normalized values:

```
R> ryc2 <- NW.pal2(normalize(surv$NW.Hnd))
```

Looking at the returned object in `ryc2`, you'll note it's a matrix of RGB triplets corresponding to each normalized value you supplied to your `colorRamp` function `NW.pal2` (non-integer values end up being coerced to integers). These need to be converted to hex codes before you can use them in plotting. Using `rgb` just as you saw in Section 25.1.1, you get the vector you need (snipped for brevity):

```
R> NW.cols2 <- rgb(ryc2,maxColorValue=255)
R> NW.cols2
[1] "#BC7700" "#D3AD00" "#C48A00" "#CEA200" "#B97000" "#B97000"
[7] "#B66700" "#CA9700" "#C18100" "#B56500"     --snip--
```

Note the difference between the hex codes you obtain here in `NW.cols2` and those of your first palette (`NW.cols`). Here, you get a hex code for each unique value, but for the categorized `NW.cols`, you have only one hex code for each bin (so just $k = 5$ unique colors).

You're ready to plot. This line produces the image on the right of Figure 25-4:

```
R> plot(surv$Wr.Hnd,surv$Height,col=NW.cols2,pch=19,
        xlab="Writing handspan (cm)",ylab="Height (cm)")
```

In terms of this relatively simple example, the visual difference between the two approaches is minimal, though looking closely you can notice the "smoother" color transition in the normalized version. As you increase k when using the categorization approach, the visual result will become closer to that of the normalization technique. That said, the normalization approach should generally be preferred, since it more closely reflects the continuous nature of the values you're trying to visualize, and it tends to work more effectively for values with a skewed distribution or when you're working with a complex color palette.

25.1.5 Color Legends

Now that you can use color to index a continuous variable in your plots, you'll need a legend to reference the color scale. It's possible to create a legend using base R tools alone, but it can be simpler to use contributed functionality in R instead. Here, we'll look at at the `colorlegend` function from the `shape` package (Soetaert 2024). While there are a few different options (I'll note some more at the end of this section), the functionality provided by the `shape` package offers the most direct control over referencing continuous values on arbitrary scales.

First, download and install `shape` from CRAN (for a refresher on this, see either Section 1.3.5 or Section A.2.3). In your R session, ensure it is loaded with a call to `library("shape")`. The following code reproduces the most recent plot (based on the `surv` object created at the beginning of Section 25.1.4 and shown on the right of Figure 25-4) and draws the color strip legend:

```
R> plot(surv$Wr.Hnd,surv$Height,col=NW.cols2,pch=19,
        xlab="Writing handspan (cm)",ylab="Height (cm)")
R> colorlegend(NW.pal(200),zlim=range(surv$NW.Hnd),zval=seq(13,23,by=2),
               posx=c(0.3,0.33),posy=c(0.5,0.9),main="Non-writing handspan")
```

This result is given on the left of Figure 25-5.

The `colorlegend` function belonging to `shape` assumes that you already have a plot present in an active graphics device. The first thing you supply to it is the color span of the values you want to reference. This is easiest with a color palette function like those listed in the help file `?rainbow` or created using `colorRampPalette`—in other words, a function that takes an integer value telling it how many colors to generate. Doing so with a large number of colors gives you a smooth color strip, so I use `NW.pal(200)`. Next, you provide `colorlegend` with the range of the values that will be referenced by the legend using `zlim`—in this case, the range of the non-writing handspans `range(surv$NW.Hnd)`. The `zval` argument takes in the values at which you want to place tick marks. In this example, the values of a sequence between 13 and 23, in steps of 2, are chosen.

The `posx` and `posy` arguments position and size the color legend. Rather than taking user coordinates, each of these must be a vector of

length 2 describing, in *relative device coordinates*, the horizontal (`posx`) and vertical (`posy`) lengths of the strip. In this example, `posx=c(0.3,0.33)` tells the function to draw the width of the legend from 30 percent of the left of the device to 33 percent; the width is therefore 3 percent of the overall device width and positioned to the left of the center. Setting `posy=c(0.5,0.9)` says you want the length of the strip to span 40 percent of the device, from 50 percent of the way up to 90 percent of the overall device height. Lastly, you can add an optional title to the legend by supplying a character string to `main`.

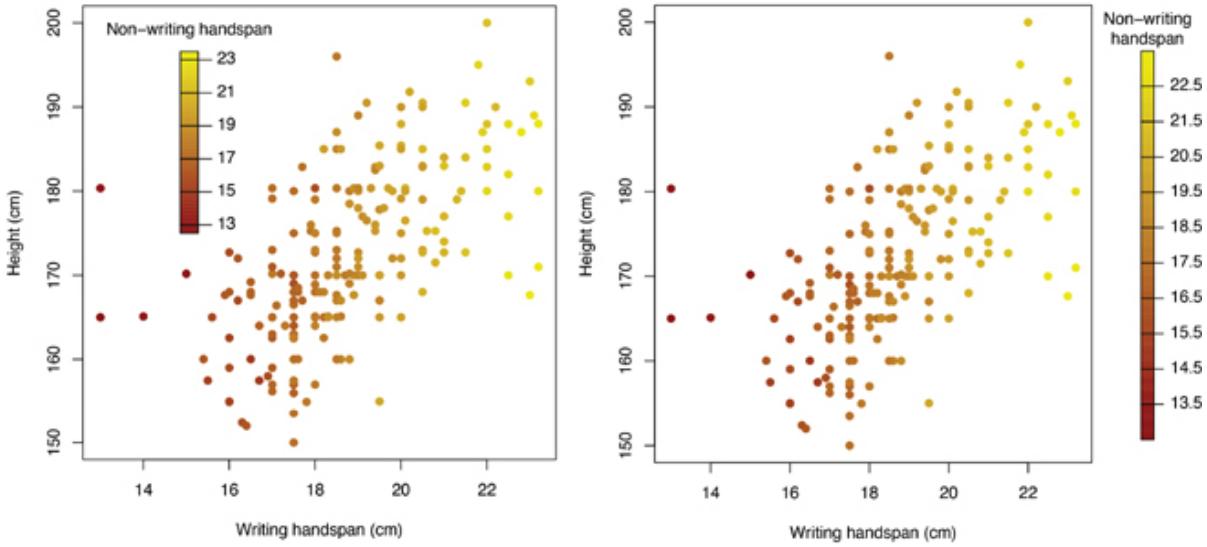


Figure 25-5: Two examples of implementing a color strip legend using the `colorlegend` *function from the contributed* `shape` *package*

You'll probably need to experiment a bit in a trial-and-error fashion to get the positioning and sizing (and appropriate tick marks using `zval`) you want. The device-specific nature of `posx` and `posy` means that if you resize your device interactively after the fact, you might well need to re-evaluate the values of these arguments.

If you want to create extra space to construct a larger color legend, you can use the `xlim` argument in the initial call to `plot` to widen the horizontal size of the plot (or use `ylim` to widening the vertical area). Alternatively, you can change the figure or outer margin spacing (refer to Section 23.2) to give you enough room to put the legend outside the

plot region altogether. This next chunk of code does just that by initializing a wider-than-default right margin (the fourth entry in the `mar` argument to `par`), replotting the scatterplot, and inserting a color legend into that extra space:

```
R> par(mar=c(5,4,4,6))
R> plot(surv$Wr.Hnd,surv$Height,col=NW.cols2,pch=19,
        xlab="Writing handspan (cm)",ylab="Height (cm)")
R> colorlegend(NW.pal(200),zlim=range(surv$NW.Hnd),zval=13.5:22.5,digit=1,
               posx=c(0.89,0.91),main="Non-writing\nhandspan")
```

The result is given on the right of Figure 25-5. The legend is slightly narrower than before, taking up only 2 percent of the device width on the right with `posx=c(0.89,0.91)`. With no specification of `posy`, `colorlegend` has used the default of `c(0.05,0.9)`, giving a color strip that spans almost the entire height of the device. The tick marks and labeling of the new legend are now placed in increments of 1 from 13.5 to 22.5; note that to display decimal places (in other words, *significant digits*), you need to increase the `digit` argument from its default, `0`, to reveal them. Here, `digit=1` prints the tick mark labels to one decimal place.

There are a number of other contributed packages that offer functions for drawing color legends, some of which are geared toward other, more specific statistical and computational methodologies. At the time of writing, the packages `bamlss` (Umlauf et al. 2021) and `corrplot` (Wei & Simko 2024) each have their own functions named `colorlegend`. There's also the `plotrix` package (Lemon 2006), with the ever-so-slightly differently named `color.legend` function. You may wish to investigate these options if the functionality offered by the `shape` package described here isn't quite what you need.

NOTE

These equivalently named functions across multiple independent packages serve as a good reminder of the masking *behavior of R in the event you*

happen to have loaded more than one of these packages in the same R session. Refer back to Section 12.3.1, where it was noted that you can use the "double-colon" notation to ensure the intended function is used in any such scenario. For example, if you have both the `corrplot` *and* `shape` *packages loaded in the same session, then one of the* `colorlegend` *functions will be masked (depending on the order in which the packages were loaded). In that case, it'd therefore be prudent to preface any given call to* `colorlegend` *with the relevant package name of the desired function, for example,* `shape::colorlegend`*, to avoid unintended problems.*

25.1.6 Opacity

Another useful feature of working with colors in R is the ability to specify their opacity/transparency. All of the palette functions described earlier, which provide the user with hex codes, have an optional argument `alpha`, the valid range of which depends on the function (a quick check of the corresponding documentation will tell you). For example, the `rgb` function uses `maxColorValue` to set the upper bound on opacity, and palette functions like `rainbow` all use the normalized range from 0 through 1 (just like in the `ggplot2` plots created throughout Chapter 24).

By default, R assumes full opacity when you're creating colors. However, hex codes change slightly when opacity is explicitly set using `alpha`. Rather than six characters after the `#`, eight will appear, with the last two containing the additionally required information. Consider the following lines of code, which generate four different versions of full red: default, zero opacity (fully transparent), 40 percent opacity (0.4 × 255 = 102), and full opacity, respectively:

```
R> rgb(cbind(255,0,0),maxColorValue=255)
[1] "#FF0000"
R> rgb(cbind(255,0,0),maxColorValue=255,alpha=0)
[1] "#FF000000"
R> rgb(cbind(255,0,0),maxColorValue=255,alpha=102)
[1] "#FF000066"
```

```
R> rgb(cbind(255,0,0),maxColorValue=255,alpha=255)
[1] "#FF0000FF"
```

Note here that the first and last colors are identical; it's just that the last hex code explicitly specifies full opacity.

You can always adjust the opacity of any color you've already got with the `alpha.f` argument (which takes values in the range 0 through 1) of the ready-to-use `adjustcolor` function. The following line takes the default red hex code created by the first line in the previous example and turns it into a 40 percent opaque version, verifying the third hex code obtained previously:

```
R> adjustcolor(rgb(cbind(255,0,0),maxColorValue=255),alpha.f=0.4)
[1] "#FF000066"
```

You briefly came across the `adjustcolor` command in Section 24.2.1, when creating a transparent gray confidence interval for a LOESS-smoothed trend using base R graphics. The function is applicable to any vector of hex codes produced via a built-in or custom palette function.

You'll put opacity to the test using the built-in `quakes` data frame already used in a few examples in this book, which consists of data on 1,000 seismic events near Fiji. Let's re-create the plot in Figure 13-6 on page 328, which shows the "number of detecting stations" against "event magnitude," and dress it up using color to identify the continuous "depth" data. Since there are many overlapping observations, reducing opacity of the individual points would be a sensible idea for data visualization. The code

```
R> keycols <- c("blue","red","yellow")
R> depth.pal <- colorRampPalette(keycols)
R> depth.pal2 <- colorRamp(keycols)
```

sets up a custom three-color palette both ways (as a function expecting an integer, `depth.pal`, and as a function expecting a value between 0 and 1, `depth.pal2`; refer to Sections 25.1.3 and 25.1.4). Then, the following line uses the normalization approach (with the `normalize` function defined

in Section 25.1.4 on page 719) to obtain the appropriate colors for the points to be plotted according to the "depth" variable of the dataset:

```
R> depth.cols <- rgb(depth.pal2(normalize(quakes$depth)),maxColorValue=255,
                     alpha=0.6*255)
```

The request for 60 percent opacity is made through `alpha` in the call to `rgb`; recall that the value of the `alpha` argument in this function must respect `maxColorValue` (hence the expression `alpha=0.6*255`). You can create the plot with the following call, which assigns each point its corresponding color stored in `depth.cols`:

```
R> plot(quakes$mag,quakes$stations,pch=19,cex=2,col=depth.cols,
        xlab="Magnitude",ylab="No. of stations")
```

This plot affords another opportunity to showcase the `colorlegend` function from the `shape` package, discussed in Section 25.1.5. Assuming you have already called `library("shape")` in the current R session, the next line adds a corresponding color legend inside the plot region of the scatterplot (on a default-size square device):

```
R> colorlegend(adjustcolor(depth.pal(20),alpha.f=0.6),
               zlim=range(quakes$depth),zval=seq(100,600,100),
               posx=c(0.3,0.32),posy=c(0.5,0.9),left=TRUE,main="Depth")
```

Here, you can see another demonstration of the use of `adjustcolor`, where the color sequence generated with the call to `depth.pal(20)` is then reduced to 60 percent opacity to match the plotted points. Again, `posx` and `posy` are used to position the legend, and the optional logical argument `left` is set to `TRUE` to make the tick marks and color legend labels appear on the left side of the strip. Figure 25-6 shows the final result.

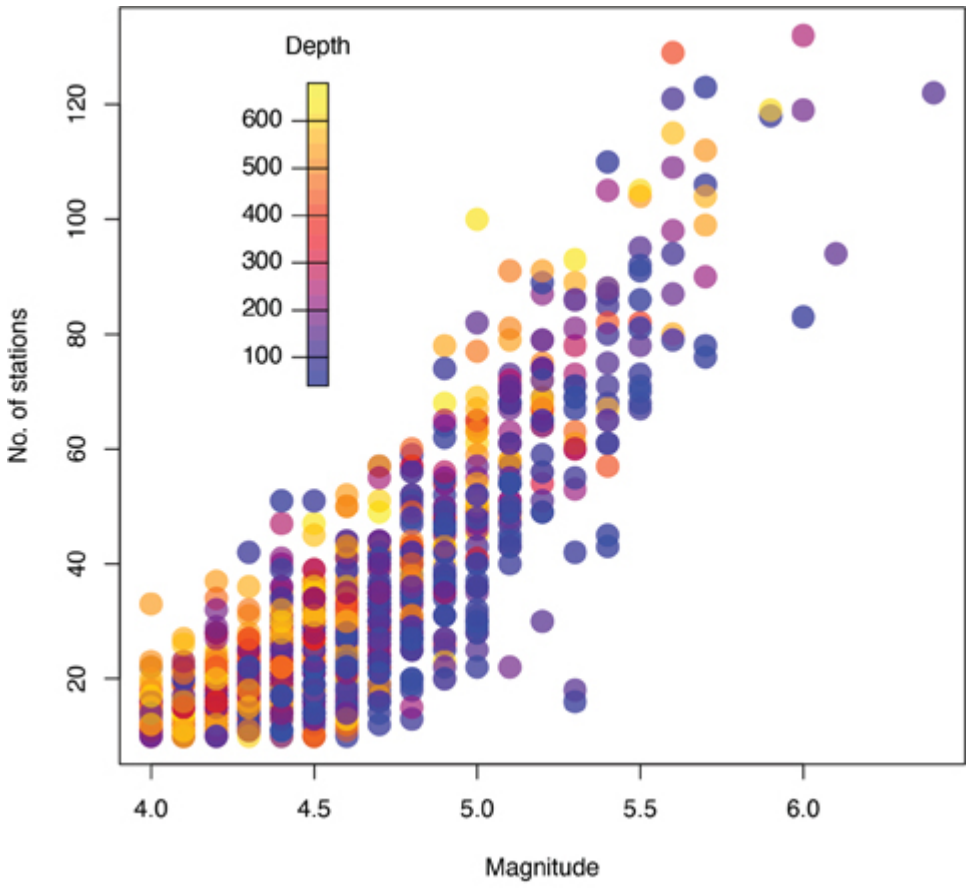


Figure 25-6: Altering the color opacity in a custom palette, used to index the continuous "event depth" observations in a plot of "number of stations" against "magnitude" for the `quakes` *dataset, and a corresponding color legend using* `colorlegend` *from the* `shape` *package*

25.1.7 RGB Alternatives and Further Functionality

RGB triplets aren't the only way to specify color in R. Other specifications include hue-saturation-value (HSV) and hue-chroma-luminance (HCL), available through the built-in `hsv` and `hcl` functions. These work in much the same way as `rgb`, where you specify the strength of influence of the three components and out pop corresponding character string hex codes that form valid R colors for any relevant plotting command. In fact, the HSV parameterization is what's used internally by the built-in palettes detailed in Section 25.1.2 such as `rainbow` and `heat.colors`, and the `hcl.colors` palette is representative of results that are achieved under a HCL system.

Contributed functionality offers even more flexibility. The `colorspace` package (Zeileis et al. 2020), which translates between different color formats, is worth noting, as is the previously mentioned `RColorBrewer` package (Neuwirth 2022), which is based directly on the well-received color schemes designed by Cynthia Brewer and collaborators (see *https://colorbrewer2.org*). That said, from an introductory perspective, you should find the use of RGB coordinate space and the base R functionality as discussed here sufficient for most visual explorations of your data and models.

Exercise 25.1

Ensure the `car` package is loaded. Revisit the `Salaries` data frame from Exercises 24.1 (page 699) and 24.2 (page 706) and take a look at the help file `?Salaries` to remind yourself of the variables present. Your task is to use color, point size, opacity, and point character type to reflect "years since Ph.D.," "sex," and "rank" in a scatterplot of "salary" against "years of service," by completing the following steps:

a. Set up a custom color palette that goes from `"black"` to `"red"` to `"yellow2"`. As per Section 25.1.3, create two versions of this palette: one that expects a number of colors and one that expects a vector of normalized values between 0 and 1.

b. Create two vectors that will control point character and character expansion adhering to the following guidelines in (i) and (ii). Each of these can be achieved in a single line by vector subsetting/repetition based on a numeric coercion (by way of `as.numeric`) of the corresponding factor vector in the data frame.

    i. Use the point characters `19`, `17`, and `15` to reference the three increasing academic ranks in that order.

ii. Use a character expansion of `1` for females and a character expansion of `1.5` for males.

c. Use the `normalize` function defined in Section 25.1.4 on page 719 to create a [0, 1] normalized version of the range of values of the "years since Ph.D." variable. Then use the appropriate palette from (a) along with `rgb` to convert these to the required hex codes.

d. Use `adjustcolor` to modify the entries of the vector of colors you just created in (c), adjusting opacity. Colors in the vector that correspond to females should be reduced to 90 percent opacity; colors that correspond to males should be reduced to 30 percent opacity.

e. Now start the plot; alter the default figure margins to be 5, 4, 4, and 6 lines wide on the bottom, left, top, and right, respectively. Plot salary on the y-axis against years of service on the x-axis. Set the corresponding point colors according to your vector from (d) and the point characters and character expansion according to your vectors from (b). Ensure tidy and informative x-axis and y-axis titles.

f. Incorporate two separate legends following the guidelines in (i) and (ii). Both legends should be horizontal, and you should use the `xpd` argument to relax clipping, thereby allowing their placement in figure margins (refer to Section 23.2.3).

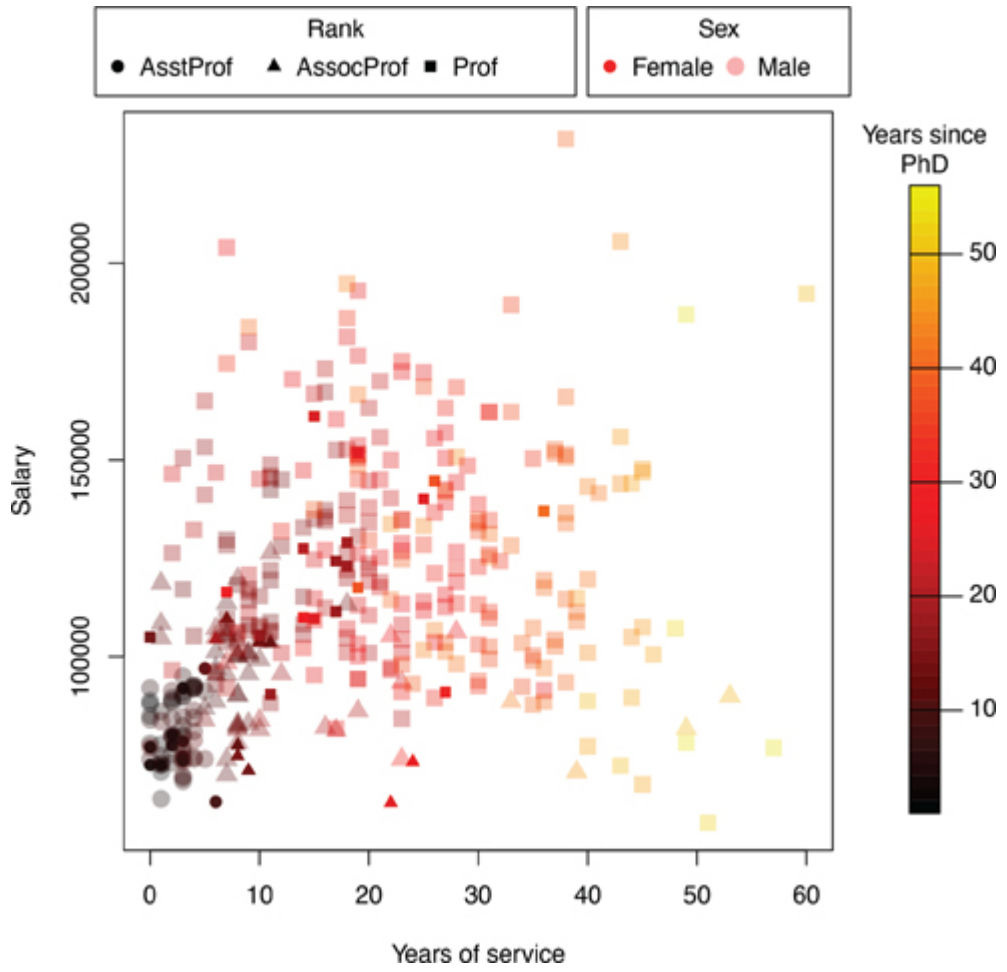
   i. Place the first legend at the user coordinate given by `x=-5` and `y=265000`. It should use the levels of the "rank" factor as the referencing text and pair these with the corresponding `pch` symbols as assigned. Include an appropriate title for the legend.

   ii. The second legend should be placed next to the first, using an x-coordinate of `40` and the same y-coordinate value. This legend should show two

points, both red and of type `19`, but reference the two levels of sex by altering the character expansion and opacity of these to reference points as assigned.

g. Lastly, ensure the `shape` package is loaded and use the function `colorlegend` along with 50 colors generated from the appropriate palette from (a) to reference "years since Ph.D." You can leave the horizontal and vertical placements of the legend at their default values. The `zlim` range should simply be set to match the range of the observed data, and the tick mark values set via `zval` should be a sequence between 10 and 50, increasing in steps of 10. Include an appropriate title for the color legend.

Your plot should look something like this:



Your next task is a little different. The goal is to plot a standard normal probability density function (Section 16.2.2) but use color to shade in some specifically defined polygons underneath the curve to denote "distance from mean." To achieve this, complete the following:

h. Generate a vector of exactly 25 colors from the built-in palette `terrain.colors` (Section 25.1.2) and name it `tcols`. Create a reversed version of `tcols` by executing `tcols[25:1]`, and append both of these together to form a new vector of length 50 (containing the first 25 colors shading one way and then the same 25 shading the opposite way).

i. Next, create and store an evenly spaced sequence of exactly 51 values between –3 and 3 inclusive; name it `vals`. Use `dnorm` (Section 16.2.2) to calculate and store the corresponding 51

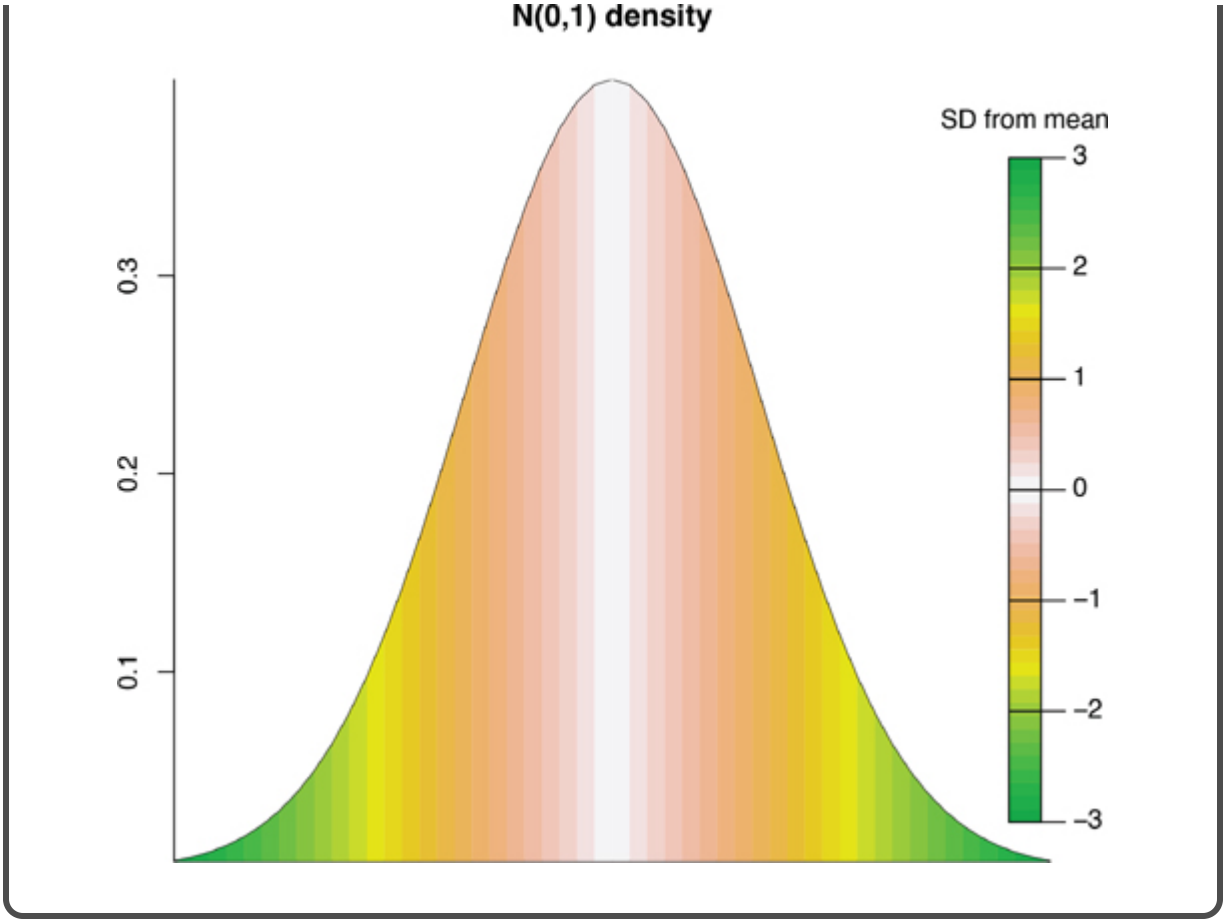
values of the standard normal density curve; name it `normvals`.

j. Draw the normal density curve by plotting the values in (i) as a line (recall setting `type="l"`; see Section 7.2.1). In the same call to `plot`, use knowledge found in Chapter 23 to set both the x-axis and y-axis styles to be of type `"i"`; suppress both axis titles with empty strings; change the surrounding box to be an *L* shape; and suppress the drawing of the x-axis. Give the plot a suitable main title.

k. To shade the different colors underneath the curve, use a `for` loop, iterating through the integers 1 to 50. At each iteration, the loop should call `polygon` (refer to Section 15.2.3, page 383). Assuming your indexer is `i`, the vertices of each polygon should be formed by the vectors `vals[rep(c(i,i+1),each=2)]` and `c(0,normvals[c(i,i+1)],0)`. Each polygon should suppress its border and be colored according to the relevant `i`th entry in your color vector of length 50 created in (h).

l. Lastly, ensure the `shape` package has been loaded and use your length 50 color vector to produce a color legend with default placement to reference "distance from mean." You can easily set the `zlim` and `zval` arguments in the call to `colorlegend` using your `vals` vector. Include an appropriate title for the legend.

For reference, my result looks like this:



25.2 3D Scatterplots

This section will look at creating 3D scatterplots, which allow you to plot raw observations based on three continuous variables at once, as opposed to only two in a conventional 2D scatterplot. You'll then learn how to enhance your 3D scatterplot to represent more variables and make it easier to interpret. As per usual, there are several ways to create three-variable scatterplots in R, but a good place to start is the `scatterplot3d` function in the contributed package of the same name (Ligges & Mächler 2003).

25.2.1 Basic Syntax

The syntax of the `scatterplot3d` function is similar to the default `plot` function. In the latter, you supply a vector of x- and y-axis coordinates; in the former, you merely supply an additional third vector of values

providing the z-axis coordinates. With that additional dimension, you can think of these three axes in terms of the x-axis increasing from left to right, the y-axis increasing from foreground to background, and the z-axis increasing from bottom to top.

Install and load the `scatterplot3d` package, and let's go straight into an example. Recall the famous iris flower data, which you first encountered in Section 14.6. This dataset contains measurements on four continuous variables (petal length/width and sepal length/width) and one categorical variable (flower species); the `iris` data frame is immediately accessible from the R prompt, so there's no need to load anything. Enter the following so you have quick access to the measurement values that make up the data:

```
R> pwid <- iris$Petal.Width
R> plen <- iris$Petal.Length
R> swid <- iris$Sepal.Width
R> slen <- iris$Sepal.Length
```

The most basic 3D scatterplot of, say, petal length, petal width, and sepal width, is achieved with the following:

```
R> library("scatterplot3d")
R> scatterplot3d(x=pwid,y=plen,z=swid)
```

It's as simple as that; the result of this code is given on the left of Figure 25-7. Here, you can observe a general positive relationship among all three plotted variables. There's also a clearly isolated cluster of observations in the foreground that have relatively large sepal widths but small petal measurements.

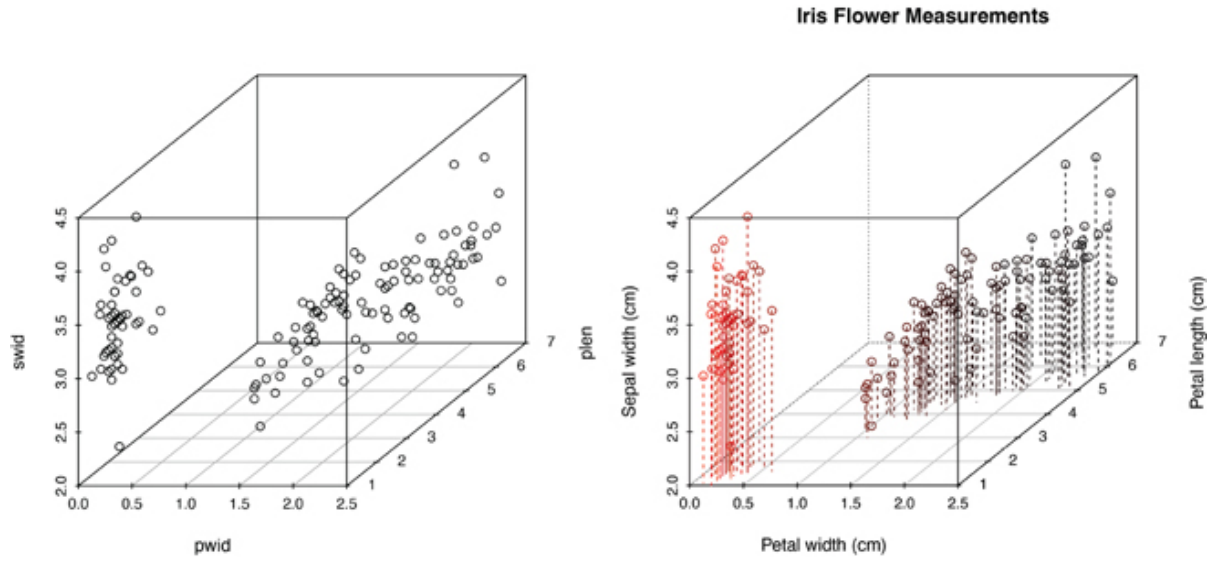


Figure 25-7: Two 3D scatterplots of the famous `iris` data with petal width, petal length, and sepal width on the x-, y-, and z-axis, respectively. Left: Basic default appearance. Right: Tidying up titles and adding visual enhancements to emphasize 3D depth and legibility via color and vertical line marks.

25.2.2 Visual Enhancements

It can be difficult to clearly perceive "depth" in the plotted cloud of points, even with the box and x-y plane grid lines that are drawn by default. For this reason, there are a couple of optional enhancements you can make to a `scatterplot3d` plot: coloring the points to help make the transition from foreground to background clearer and setting the `type="h"` argument to draw lines perpendicular to the x-y plane.

The right-hand plot in Figure 25-7 shows the plot with these enhancements and is the result of the following line:

```
R> scatterplot3d(x=pwid,y=plen,z=swid,highlight.3d=TRUE,type="h",
                 lty.hplot=2,lty.hide=3,xlab="Petal width (cm)",
                 ylab="Petal length (cm)",zlab="Sepal width (cm)",
                 main="Iris Flower Measurements")
```

Notice that the vertical lines make reading the values of the points with respect to the z-axis much easier. By default those lines, achieved via `type="h"`, are solid, but you can alter this with the `lty.hplot` argument (which behaves in the same way as the standard graphical control

argument `lty`); setting `lty.hplot=2` requests dashed lines. Similarly, you can alter the line type of the "non-visible" sides of the box; setting `lty.hide=3` instructs the plot to draw those lines as dotted. Setting `highlight.3d=TRUE` emphasizes 3D depth by applying color transitioning from red to black based on the y-axis position of a point. Finally, `xlab`, `ylab`, `zlab`, and `main` are used to tidy up the corresponding axis and main titles.

The depth-emphasizing color transition is useful, but there's an important consequence: It means you can no longer use color to represent a fourth variable with such a plot. Along that line of thought, remember that the `iris` data has a fourth continuous variable, sepal length (stored as `slen` in Section 25.2.1), that you're not displaying in either plot in Figure 25-7. You're also not displaying the categorical variable of flower species, so let's fix that. First, set up relevant color palette objects and map them to the observations in `slen` using knowledge from Section 25.1.4 and our custom `normalize` function defined therein:

```
R> keycols <- c("purple","yellow2","blue")
R> slen.pal <- colorRampPalette(keycols)
R> slen.pal2 <- colorRamp(keycols)
R> slen.cols <- rgb(slen.pal2(normalize(slen)),maxColorValue=255)
```

The following code produces the 3D scatterplot, which supplies the created `color` argument with the sepal length–indexed colors and uses the `pch` argument to distinguish between the three different species:

```
R> scatterplot3d(x=pwid,y=plen,z=swid,color=slen.cols,
                 pch=c(19,17,15)[as.numeric(iris$Species)],type="h",
                 lty.hplot=2,lty.hide=3,xlab="Petal width",
                 ylab="Petal length",zlab="Sepal width",
                 main="Iris Flower Measurements")
```

Here, I've used the vector `c(19,17,15)`, with the numeric coercion of the `iris$Species` factor object passed to the square brackets, to pair `pch` character numbers as follows: `19` with *Iris setosa* (the first level of the

factor), 17 with *Iris versicolor* (the second level), and 15 with *Iris virginica* (the third level). Refer to Section 4.3.4 for a reminder on the numeric representation of factors and to Figure 7-5 on page 151 for the different types of base R point characters.

You can then insert a legend referencing species type with a familiar call to `legend`:

```
R> legend("bottomright",legend=levels(iris$Species),pch=c(19,17,15))
```

And with a little experimentation, you can include a color strip legend too (making sure you've loaded the `shape` package so you have access to the `colorlegend` function, as per Section 25.1.4):

```
R> colorlegend(slen.pal(200),zlim=range(slen),zval=5:7,digit=1,
               posx=c(0.1,0.13),posy=c(0.7,0.9),left=TRUE,
               main="Sepal length")
```

The final result of all this is the plot shown in Figure 25-8.

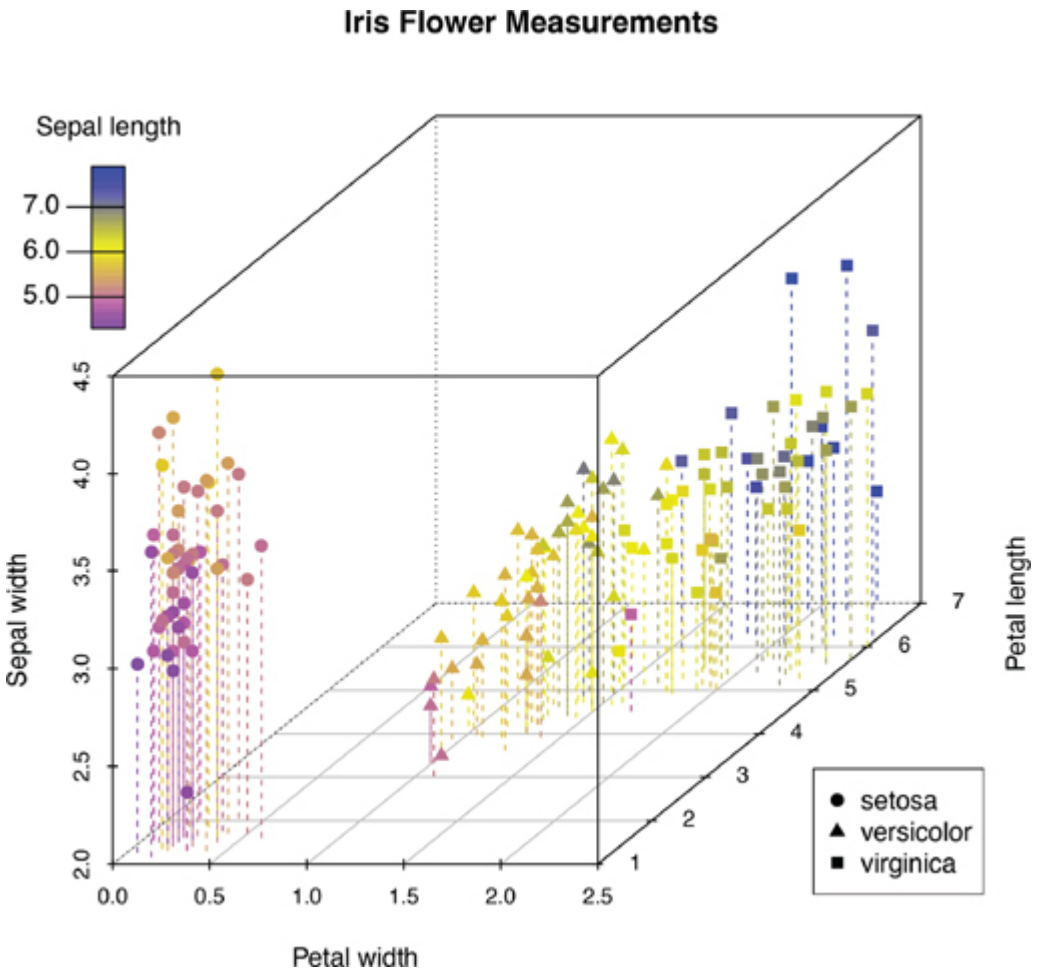


Figure 25-8: A 3D scatterplot of the famous `iris` data, displaying all five present variables with the additional use of color (for sepal length) and point character (for species).

With the creative use of color and point type, you've now been able to display the five-dimensional data in a single 3D scatterplot. This reveals important information about the data. For instance, you can now identify *Iris setosa* as the clearly separate group of points in the foreground and see that while *Iris setosa* tend to have smaller petal widths and lengths, and larger sepal widths, than the other two species (especially *Iris versicolor*), the purple coloring at the lower end of the scale suggests they tend to have smaller sepal *lengths*.

Exercise 25.2

Ensure the `scatterplot3d` library has been loaded in your current R session.

a. Turn your attention back to the `diabetes` data frame found in the `faraway` package (you first looked at these data in Section 21.5.2). Your goal is to produce a `scatterplot3d` plot of weight, hip, and waist measurements as per the following guidelines:

   - Hip, waist, and weight variables should correspond to the x-axis, y-axis, and z-axis, respectively; provide tidy and informative axis titles.
   - Use the appropriate argument to ensure the 3D depth is highlighted by color.
   - Choose two different point characters to reflect `gender`.
   - Place a simple legend referencing these two point characters by gender in the blank space in the topleft area.

b. Create a 3D scatterplot of the built-in `airquality` data, which you first met in Section 24.2.2, according to the following guidelines:

   - Create a copy of the data frame using `na.omit` to remove all rows that contain missing values and work with this copy.
   - Plot wind speed and solar radiation against the x- and y-axes, respectively, using the z-axis to plot temperature.
   - Use the appropriate argument to apply vertical dotted lines reaching up from the x-y plane to each observation.

- The data in `airquality` consist of measurements taken over five months, from May to September. Each plotted point should take on the corresponding `pch` value from `1` to `5` respective to the order of these months.
- With a vector of 50 colors generated from the built-in `topo.colors` palette, use the categorization approach to ensure each plotted point is colored according to its ozone value.
- Set a legend to reference the five point types according to month.
- Set a color legend (using functionality from the `shape` package) to reference the ozone value accordingly.
- Ensure the plot has neat axis, main, and legend titles.

25.3 Preparing a Surface for Plotting

In the rest of this chapter, you'll look at three types of 3D plots geared to visualize a *bivariate surface*. Such plots are required when you have two variables, based on which a function, estimate, or model has been defined, and you want to use the third available axis (in other words, the z-axis) to map out the resulting surface. You've seen examples of bivariate functions already, through the response surfaces for the `mtcars` data in Section 21.5.4 (where you looked at mean MPG as a function of car weight and horsepower) and through the study of diagnostic tools for linear regression models in Section 22.3.6 (where you saw how Cook's distance can be expressed as a function of residual and leverage).

Before you look at producing these plots, it's important to understand how they're created in R. For some function/estimate/model of interest, f, think of its result as comprising individual values z that form a plane or surface that vary according to

continuous, two-dimensional x-y coordinates. Mathematically, you can imagine this in the following sense:

$$z = f(x, y).$$

Plotting a completely continuous surface is technically impossible since that would require you to evaluate *f* at an infinite number of coordinates. Therefore, evaluation of the surface is typically performed on a finite *grid* of evenly spaced coordinates along both the x- and y-axes. The result of the function at each unique pair of coordinates is stored in a corresponding position in an appropriately sized matrix (the size of which depends directly upon the resolution of the evaluation grid in the x- and y-axes). I will generically refer to this discretized result as the *z-matrix*.

Since all the traditional R graphics commands that plot these bivariate functions operate in much the same way (using a z-matrix), it's critical to understand how this matrix is constructed, arranged, and interpreted by those commands to ensure you're correctly drawing the outcome. In this section, you'll prepare for the specific plot types in the remainder of this chapter by getting familiar with this construct in a simplified, hypothetical setting.

25.3.1 Constructing an Evaluation Grid

Say you have a bivariate function that results in a continuous surface *z*, defined between 1 and 6 on the x-axis and 1 and 4 on the y-axis. You can define evenly spaced sequences over each of these coordinate ranges using `seq`; for simplicity, let's do so in straight-out integers:

```
R> xcoords <- 1:6
R> xcoords
[1] 1 2 3 4 5 6
R> ycoords <- 1:4
R> ycoords
[1] 1 2 3 4
```

This implies that you're planning to draw your surface based on evaluation of the bivariate function of interest upon the grid of *x*-*y* values defined by 6 × 4 = 24 unique positions.

When passed two vectors, the built-in `expand.grid` function explicitly generates all unique coordinate pairs of two supplied vectors, by simply repeating each value in the second vector against the entire length of the first vector:

```
R> xycoords <- expand.grid(x=xcoords,y=ycoords)
```

This result takes the form of a two-column data frame object with 24 rows. If you look at `xycoords` object in the R console, you'll see *x* values from 1 to 6 all paired with a repeated *y* value of 1, then *x* from 1 to 6 paired with *y* as 2, and so on.

In practice, you'd now use the evaluation grid coordinates in `xycoords` to calculate the result of your bivariate function *f*. For this hypothetical example, let's just say that your bivariate function has resulted in the 24 letters *a* through *x*, corresponding to the order of the unique evaluation coordinates in `xycoords`. To make this even clearer, take a look at the following columnbind (see Section 3.1.1) of the hypothetical function result with the corresponding evaluation coordinate (note that the ready-to-use `letters` object in R allows you to generate letters of the alphabet quickly):

```
R> z <- letters[1:24]
R> xyz <- cbind(xycoords,z)
R> xyz
   x y z
1  1 1 a
2  2 1 b
3  3 1 c
4  4 1 d

--snip--

21 3 4 u
```

```
22 4 4 v
23 5 4 w
24 6 4 x
```

Quite simply, the `xyz` object emphasizes that each unique x-y evaluation coordinate, expressible via `expand.grid`, will have a *z* value associated with it. This set of *z* values will propagate the resulting surface.

25.3.2 Constructing the z-Matrix

The 3D plots used to visualize a bivariate function require the *z* values corresponding to the *x*-*y* evaluation grid in the form of an appropriately constructed matrix. The size of the z-matrix is determined directly by the resolution of the evaluation grid; the number of rows corresponds to the number of unique *x* grid values, and the number of columns corresponds to the number of unique *y* grid values.

You therefore need to take some care turning your *z* values into a matrix. When your vector of z-axis values corresponds to the evaluation grid that is arranged in the standard `expand.grid` fashion as illustrated in Section 25.3.1 (that is, where coordinates are stacked by increasing *x* values and repeated *y* values), be sure that your resulting z-matrix is filled in the default columnwise fashion (see Section 3.1.1), with the number of rows and columns being exactly representative of the number of values in each of the *x*- and *y*-value sequences, respectively. In the current example, you know that the resulting z-matrix needs to be of size 6×4 because there are six x locations (given in the `xcoords` object in Section 25.3.1) and four y locations (the `ycoords` object).

The following is the correct matrix representation of the hypothetical "function result" vector `z` from the previous section:

```
R> nx <- length(xcoords)
R> ny <- length(ycoords)
R> zmat <- matrix(z,nrow=nx,ncol=ny)
R> zmat
     [,1] [,2] [,3] [,4]
```

```
[1,] "a"  "g"  "m"  "s"
[2,] "b"  "h"  "n"  "t"
[3,] "c"  "i"  "o"  "u"
[4,] "d"  "j"  "p"  "v"
[5,] "e"  "k"  "q"  "w"
[6,] "f"  "l"  "r"  "x"
```

25.3.3 Conceptualizing the z-Matrix

The key concept to grasp at this stage is how the current arrangement of the z-matrix translates into x-y coordinate–based plotting. Using the objects created for the current running example, compare `zmat` from Section 25.3.2 to the `xyz` object in Section 25.3.1. You can see that moving down a column of `zmat` translates to an increase in the x-coordinate value for a given y-coordinate value. In other words, when this hypothetical surface of letters is "plotted," moving down a column of the matrix will correspond to moving horizontally from left to right on the corresponding plot, given a particular vertical y position.

Let's visualize this interpretation. Figure 25-9 provides a conceptual diagram of the hypothetical surface, indexed by `zmat`, as per the 24 unique coordinates defined via `xcoords` and `ycoords`. Therein, the left-to-right arrows denote "moving down" each column of `zmat`; the bottom-to-top arrows denote "moving across" each row of `zmat`. (The code to produce this is not shown here but is included in the R script files for this book at *https://www.nostarch.com/book-of-r-2nd-edition*.)

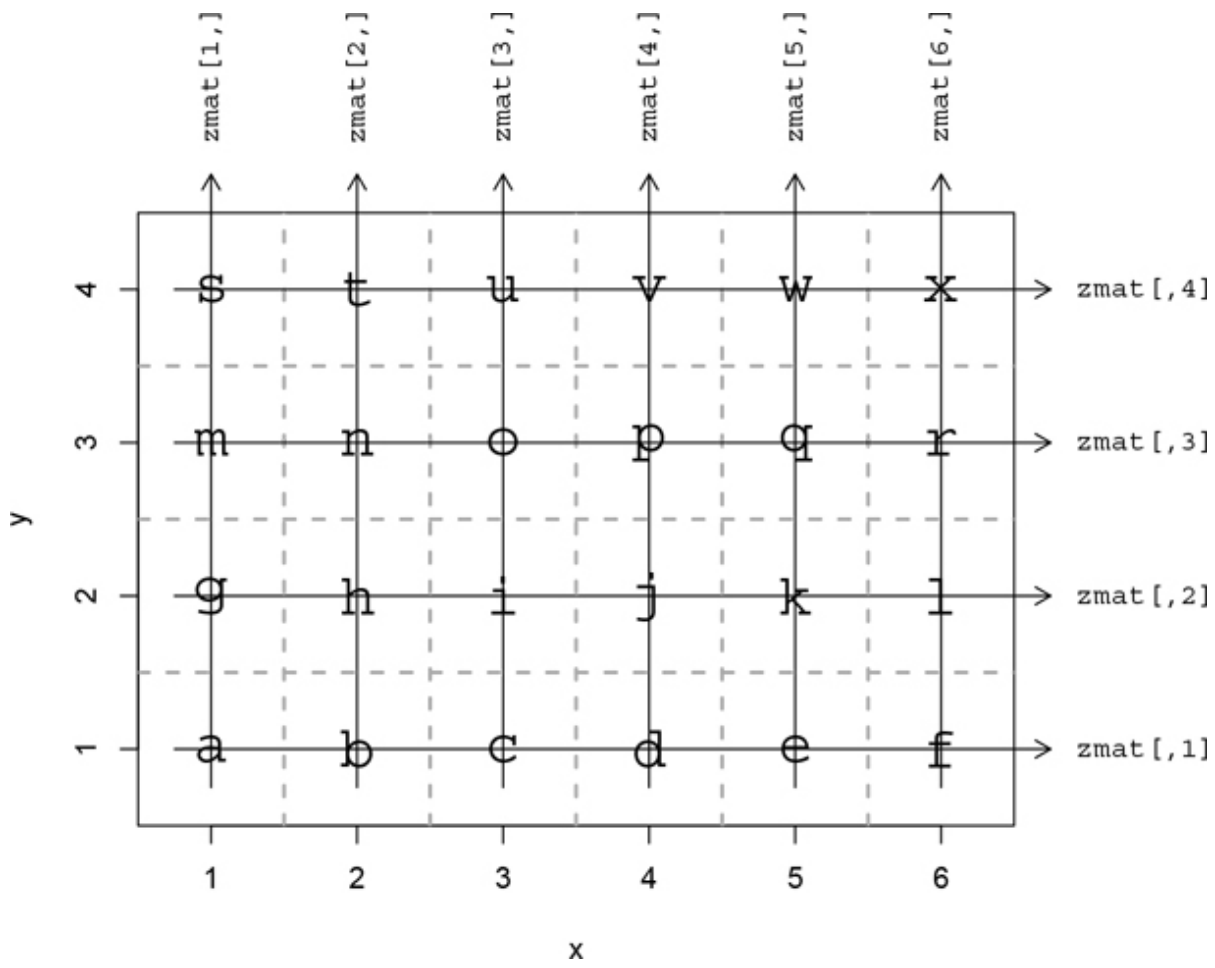


Figure 25-9: A conceptual diagram of a 24-element z-matrix corresponding to a 6×4 grid of x-y coordinates, as interpreted by R

As you now progress to plot some actual surfaces of interest, you should keep the concept of the z-matrix as illustrated in Figure 25-9 in mind. The 6×4 grid used in this hypothetical example is coarse. In practice, you'll usually use far finer grids, in terms of the resolution of the x- and y-sequences, to improve the visual appearance of the surface.

25.4 Contour Plots

One of the most common plots used to display a surface based on evaluation of a function over a grid of bivariate coordinates is the *contour plot*. Contour plots are most easily explained as a series of lines—the

contours—drawn over the 2D evaluation grid, with each contour marking off a specific level of the surface of interest.

25.4.1 Contour Lines

Based on a given numeric z-matrix, you use the R function `contour` to produce the contours connecting x-y coordinates that share the same z value.

Example 1: Topographical Map

For an example, you'll use another built-in dataset: the `volcano` object. This dataset is simply a matrix containing approximate measurements of the elevation above sea level (in meters) of *Maungawhau*, a dormant volcano in the Auckland region of New Zealand. The data are given over a rectangular area; see the documentation in `?volcano` for details. To view the topography, you need the `volcano` object (which is your z-matrix) and the relevant x- and y-coordinate sequences. For our purposes, it suffices to use integers corresponding to the size of the `volcano` matrix (you can obtain row and column numbers with a simple call to `dim`; see Section 3.1.1):

```
R> dim(volcano)
[1] 87 61
R> contour(x=1:nrow(volcano),y=1:ncol(volcano),z=volcano,asp=1)
```

The x- and y-sequences are provided to `x` and `y`, respectively, and the z-matrix corresponding to the grid of x-y coordinates is passed to `z`. Notice also the optional argument `asp=1` to force a 1-to-1 unit treatment of the coordinate axes (refer back to Sections 7.2.6 and 14.6.4).

Figure 25-10 shows the result of the call to `contour`. By default, R automatically chooses the levels of `z` at which to draw the individual contours for an aesthetically pleasing result. Contours are also selectively labeled with their corresponding z value. Looking at the topography in this plot, you can see the highest peak is a rim on the left,

marked by an oblong contour at 190 m, with a depression (to around 160 m) falling immediately to the right of this peak.

Contours are able to show you not only the peaks and troughs in a surface like this but also the "steepness" of any such features. The closer together the contour lines lie, the more rapid the change in the value of z as the x-y coordinates change in a localized area.

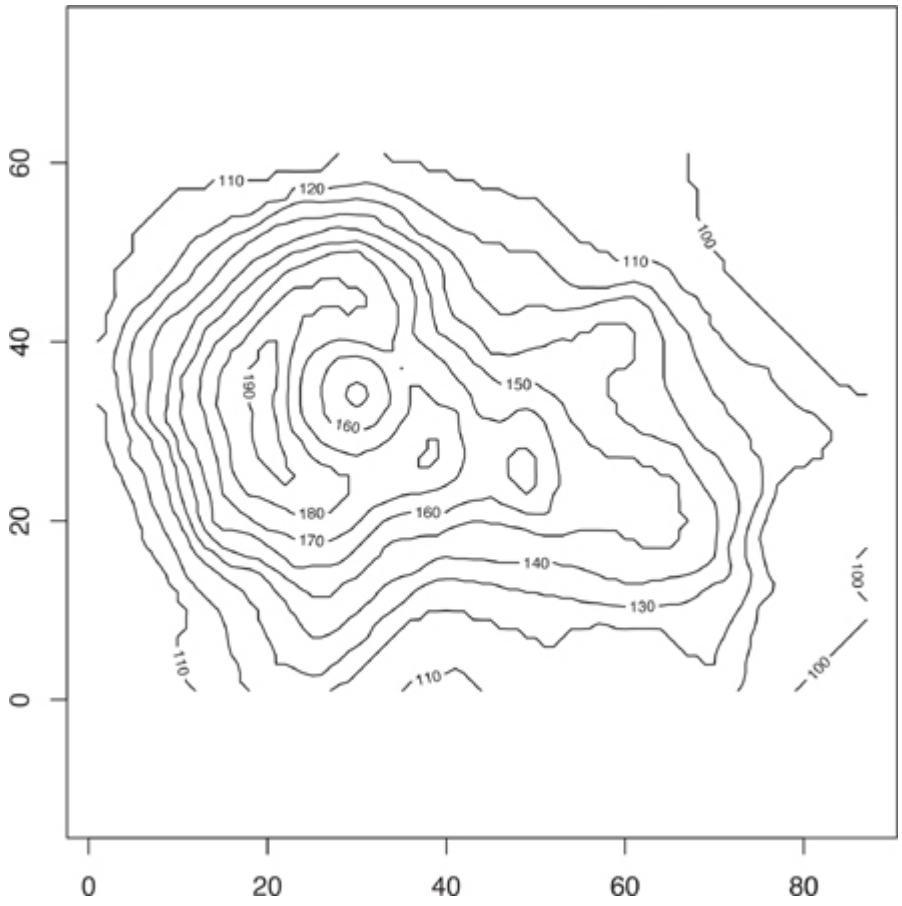


Figure 25-10: Using `contour` to produce a topographic map of the `volcano` data

Example 2: Parametric Response Surface

As a different kind of example, consider the multiple linear model fitted to the `mtcars` data mentioned earlier—that is, of MPG modeled by horsepower, weight, and an interaction between the two predictors. As in Section 21.5.4, you can get the fitted model object with the following:

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> car.fit

Call:
lm(formula = mpg ~ hp * wt, data = mtcars)

Coefficients:
(Intercept)           hp           wt        hp:wt
   49.80842     -0.12010     -8.21662      0.02785
```

The goal is to plot the response, mean mileage, as the corresponding function of horsepower and weight. To do this, you'll need to predict the mean MPG for a grid of horsepower and weight values. Consider the following code:

```
R> len <- 20
R> hp.seq <- seq(min(mtcars$hp),max(mtcars$hp),length=len)
R> wt.seq <- seq(min(mtcars$wt),max(mtcars$wt),length=len)
R> hp.wt <- expand.grid(hp=hp.seq,wt=wt.seq)

R> nrow(hp.wt)
[1] 400
R> hp.wt[1:5,]
         hp    wt
1  52.00000 1.513
2  66.89474 1.513
3  81.78947 1.513
4  96.68421 1.513
5 111.57895 1.513
```

Here, you create evenly spaced sequences (each of length 20, spanning the range of the observed data) in both `hp` and `wt`; these are your x- and y-sequences. This implies there will be 20 × 20 = 400 unique coordinates at which you'll be evaluating the fitted model; these coordinates are obtained using `expand.grid` as in Section 25.3.1.

Next, you can use `predict` to get the 400 corresponding mean MPG (z) values; since it's already a data frame in the required format, `hp.wt` can be passed directly to the `newdata` argument (refer back to Section 21.3.6

for the initial discussion on predicting from a fitted multiple linear regression model):

```
R> car.pred <- predict(car.fit,newdata=hp.wt)
```

Then, you simply need to arrange the resulting vector as the appropriate 20×20 z-matrix:

```
R> car.pred.mat <- matrix(car.pred,nrow=len,ncol=len)
```

Finally, you plot the result as contours:

```
R> contour(x=hp.seq,y=wt.seq,z=car.pred.mat,levels=32:8,lty=2,lwd=1.5,
           xaxs="i",yaxs="i",xlab="Horsepower",ylab="Weight",
           main="Mean MPG")
```

Figure 25-11 shows the result.

In this call to `contour`, you can see the use of the optional `levels` argument. Rather than let R automatically decide at which values of z to show contours, you can supply a numeric vector to this argument containing the specific levels at which to draw the lines. Here, I stipulated contours at all integer levels from 32 through 8. I also employ the familiar arguments `lty` and `lwd` to control the appearance of the contour lines themselves, which are set here as dashed and slightly thicker than the default.

Furthermore, for contour plots in particular, you'll often want to deviate from the default axis limit style because the small amount of additional "padding" space that's included in the default plot region (refer to Section 23.4.1) can be rather prominent. Take another look at the volcano contour plot in Figure 25-10. As shown previously, setting `xaxs` and `yaxs` to `"i"` restricts all plotting to the exact limits imposed by `x` and `y`.

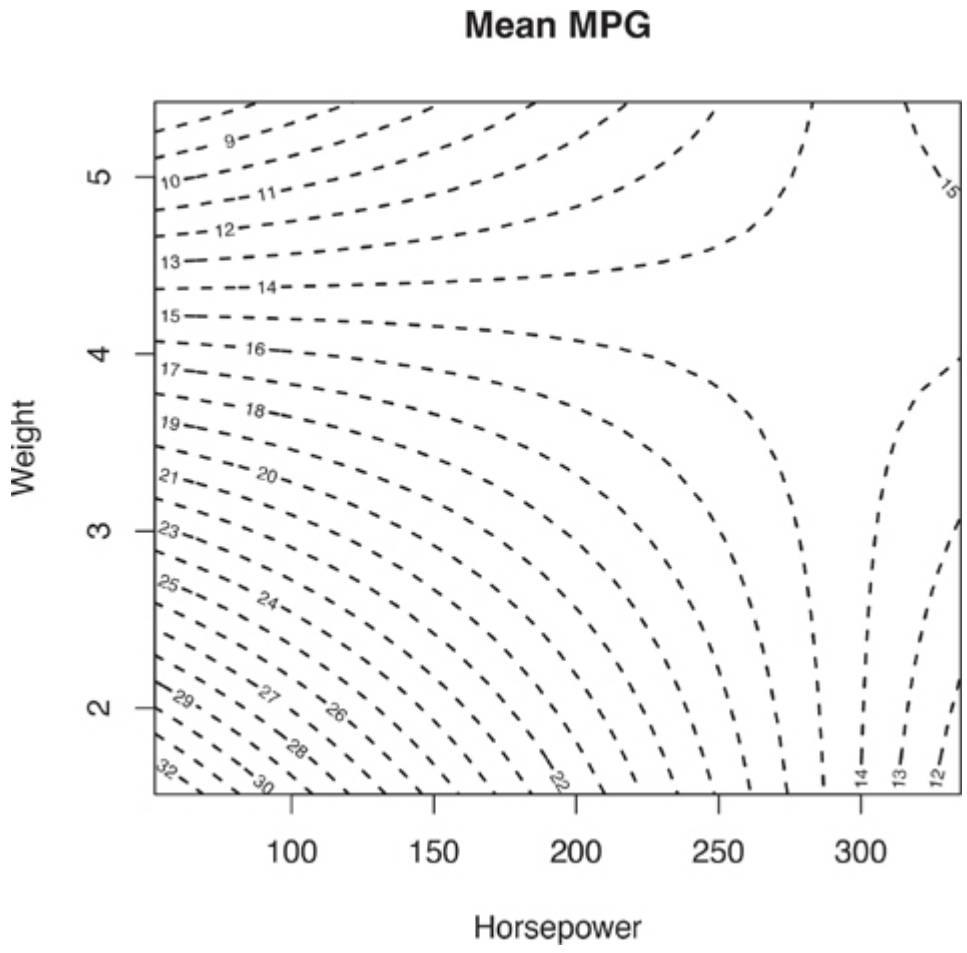


Figure 25-11: Contour lines depicting the response surface based on a multiple linear model of MPG by horsepower and weight, from the `mtcars` data

Example 3: Nonparametric Bivariate Density Estimate (Earthquake Data)

Contour plots and the other plots in this chapter also fulfill the useful role of visualizing bivariate density functions.

Section 14.3 introduced the idea of kernel density estimation (KDE) as a method by which to construct smooth estimates of the probability density functions of your data—essentially, sophisticated histograms. KDE extends to higher dimensions quite naturally so that you can also estimate the density of bivariate observations in the x-y plane. This again involves visualizing a z-matrix over a fixed grid of coordinates. For theoretical details on multivariate KDE, see Wand & Jones (1995).

Turn your attention back to the built-in `quakes` data frame and recall the plots of the spatial coordinates of the 1,000 seismic events (for example, Figure 13-1 on page 307 and Figure 23-1 on page 650). To estimate the joint probability density function of these points, you can use the `kde2d` function in the `MASS` package. Load `MASS` and execute the following line to produce the kernel estimate of the observed two-dimensional data:

```
R> quak.dens <- kde2d(x=quakes$long,y=quakes$lat,n=100)
```

You supply the bivariate data as the `x` and `y` arguments for the horizontal and vertical axes. You use the optional argument `n` to specify the number of evaluation coordinates (along each of the two axes) at which to actually return the estimated density surface. This defines the size of the matrix returned by a call to `kde2d`. Here, you've asked for KDE to be performed on a 100×100 evenly spaced grid over the range of the observed data. You can adjust intricacies of the estimate itself, such as the amount of smoothing to use, with optional arguments to `kde2d`. The argument `h` controls the smoothing bandwidth (see the documentation in `?kde2d`); in this case, I've left this at its default value, which is usually sensible for exploratory plots.

The resulting object, `quak.dens`, is simply a list with three members. The components accessed through `$x` and `$y` contain the evenly spaced evaluation grid coordinates in the corresponding axis directions, and `$z` provides you with the corresponding z-matrix. By entering either `quak.dens$x` or `quak.dens$y` at the prompt, you can confirm that they are indeed increasing sequences spanning the ranges of the observed data. Entering the following confirms the size of the matrix of interest:

```
R> dim(quak.dens$z)
[1] 100 100
```

With that, you have all the ingredients you need to display contours of the KDE surface. The next line produces the default contour plot:

```
R> contour(quak.dens$x,quak.dens$y,quak.dens$z)
```

This is given on the top left of Figure 25-12.

There are many more optional arguments available to `contour` for displaying your continuous surface. It can also be helpful to simultaneously view other data or raw observations (if they've been used in some way to create the surface, as is the case with bivariate KDE). The next line of code replots the `quakes` kernel estimate with unpadded axes, different contour levels to the defaults, and the raw observations. Recall also that it's prudent to consider an appropriate aspect ratio when plotting geographical data (Sections 7.2.6 and 14.6.4); we previously established that a value of approximately 1.069 is suitable for the `quakes` data. You can see the result of the following code on the top right of Figure 25-12:

```
R> contour(quak.dens$x,quak.dens$y,quak.dens$z,nlevels=50,drawlabels=FALSE,
          asp=1.069,xaxs="i",yaxs="i",xlab="Longitude",ylab="Latitude")
R> points(quakes$long,quakes$lat,cex=0.7)
```

Rather than using `levels` to determine the exact levels at which to draw the contours (as you did with Example 2), you can use the `nlevels` argument to specify the *number* of levels to display, and the function will choose the specific values. This most recent call to `contour` requested 50 levels to be drawn. You can suppress the automatic labeling of the displayed contours by setting `drawlabels=FALSE`, also done here, followed by a call to `points` to add the original observations to the image. Naturally, the smooth contours delineating the nonparametric density estimate reflect the heterogeneous spatial patterning of the data.

Changing the appearance of your plotted contours needn't be done universally; you can also alter the appearance of each individual contour level. This can be handy if, for example, you want to display contours at a handful of specific levels without the default labeling (to focus on the shape of the surface itself) but still want to be able to discern the values of those contours. You might also want to superimpose contours on an existing plot that already depicts other data or model-based results of interest. The third plot of the earthquake KDE surface, given on the

bottom of Figure 25-12, shows how you can achieve both of these things.

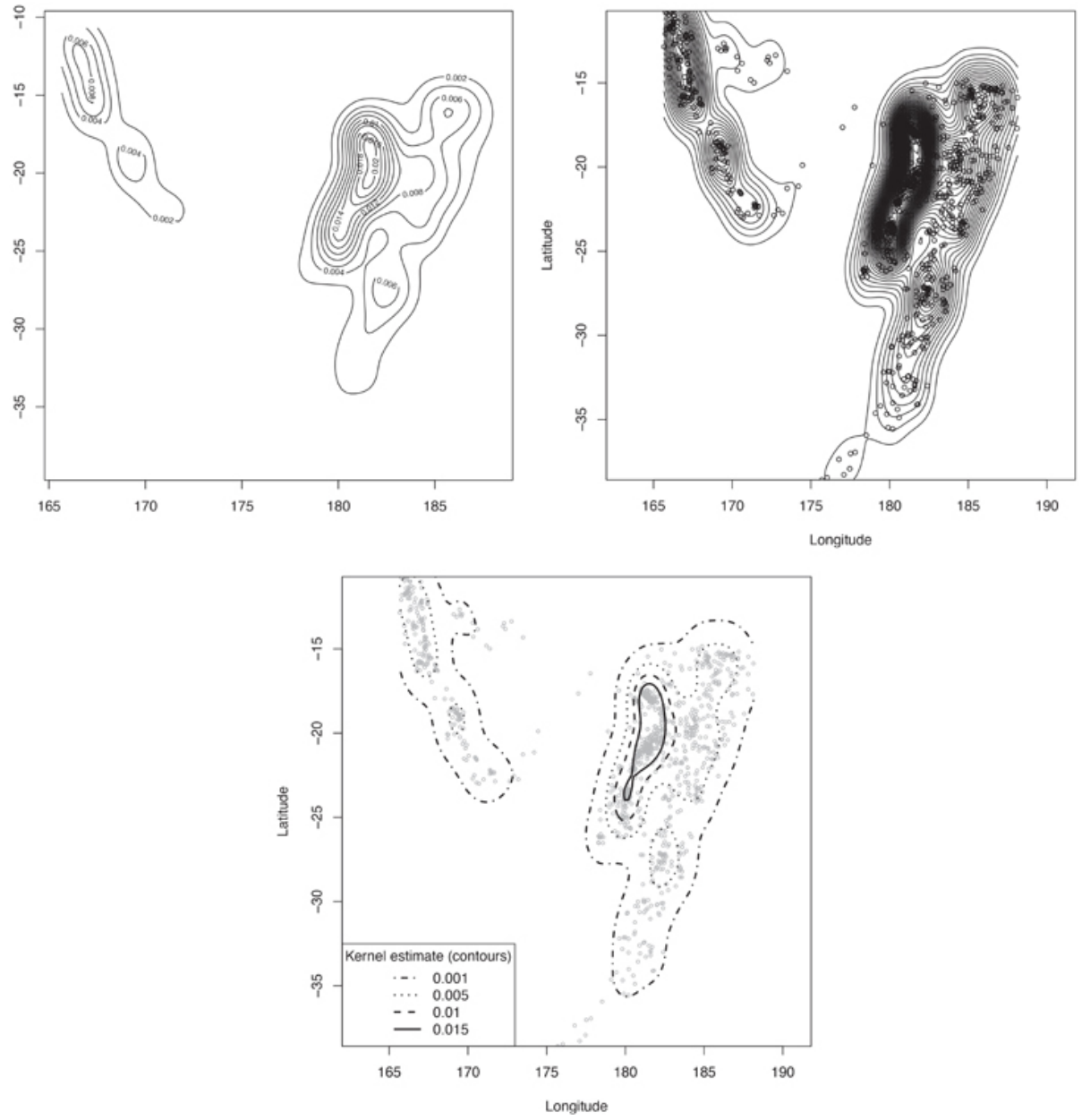


Figure 25-12: Three examples of a contour plot of the bivariate kernel estimate of the probability density function of the spatial earthquake locations given in the `quakes` *dataset*

To start the plot, the spatial locations of the earthquake data are drawn as half-size gray dots using `plot`, the style of the axes are set using `xaxs` and `yaxs` to remove the artificial padding around the edges of the plot region, the aspect ratio is locked as before, and axis titles are added:

```
R> plot(quakes$long,quakes$lat,cex=0.5,col="gray",xaxs="i",yaxs="i",
        asp=1.069,xlab="Longitude",ylab="Latitude")
```

Then, before calling `contour`, store the desired levels at which to draw contours in a vector named `quak.levs` (again, choosing appropriate contour levels depends entirely on what kind of surface you're plotting; you need to be at least roughly aware of the values stored in the relevant z-matrix):

```
R> quak.levs <- c(0.001,0.005,0.01,0.015)
```

Now remember that, by default, `contour` refreshes the graphics device and starts a new plot, but you want to avoid that when adding contour lines to an existing plot. To do so, you need to explicitly specify `add=TRUE`. You then provide the four specified levels in `quak.levs` to `levels` and suppress labeling with `drawlabels=FALSE`. To control the appearance of contour lines at individual levels, you supply the sequence of integers `4:1` to `lty`, the first entry of which, `4`, defines the line type of the contour at $z = 0.001$. The second entry, `3`, specifies the line type of the $z = 0.005$ contour, and so on. Lastly, set all drawn contours to double thickness with the single supplied value `lwd=2`:

```
R> contour(quak.dens$x,quak.dens$y,quak.dens$z,add=TRUE,levels=quak.levs,
           drawlabels=FALSE,lty=4:1,lwd=2)
```

(You could supply a vector with four elements here too, if you want differing line thicknesses for the different contours. The same element-wise contour specification extends to other relevant aesthetics, such as color via `col`.)

As a final touch, since the automatic labeling was suppressed in `contour`, add a legend in the bottom-left corner of the plot region, referencing the values of the contours through the four different line types:

```
R> legend("bottomleft",legend=quak.levs,lty=4:1,lwd=2,
          title="Kernel estimate (contours)")
```

NOTE

Many built-in and contributed base R plotting functions that by default initialize, refresh, or open a new plot include an `add` *argument as shown here. This allows you to use the graphics produced by these functions as additions to an already existing graphic. Look in the relevant help file to see whether this is the case for a given command.*

25.4.2 Color-Filled Contours

For a straightforward variation on the contour plot, you can use color to fill the gaps between the different levels that are drawn. Combined with a color legend, this removes the need to label the contour lines and in certain cases can make it easier to visually interpret fluctuations in the plotted z-matrix surface.

The `filled.contour` function does this for you. You need to supply the increasing sequences of grid coordinates in both the x-axis and y-axis directions, as well as the corresponding z-matrix, to the arguments `x`, `y`, and `z` in exactly the same way as in `contour`. The easiest way to specify the colors is to supply a color palette to the `color.palette` argument (which defaults to the built-in `cm.colors` palette; refer to Figure 25-2), and R does the rest.

Let's use the `mtcars` response surface from Example 2: Parametric Response Surface, found in Section 25.4.1, for a quick demonstration. If need be, rerun the code from that section to obtain the relevant fitted multiple linear regression model, the evaluation grid coordinates, and the prediction thereof. With the objects `hp.seq`, `wt.seq`, and `car.pred.mat`, the following call produces Figure 25-13:

```
R> filled.contour(x=hp.seq,y=wt.seq,z=car.pred.mat,
                  color.palette=colorRampPalette(c("white","red4")),
                  xlab="Horsepower",ylab="Weight",
                  key.title=title(main="Mean MPG",cex.main=0.8))
```

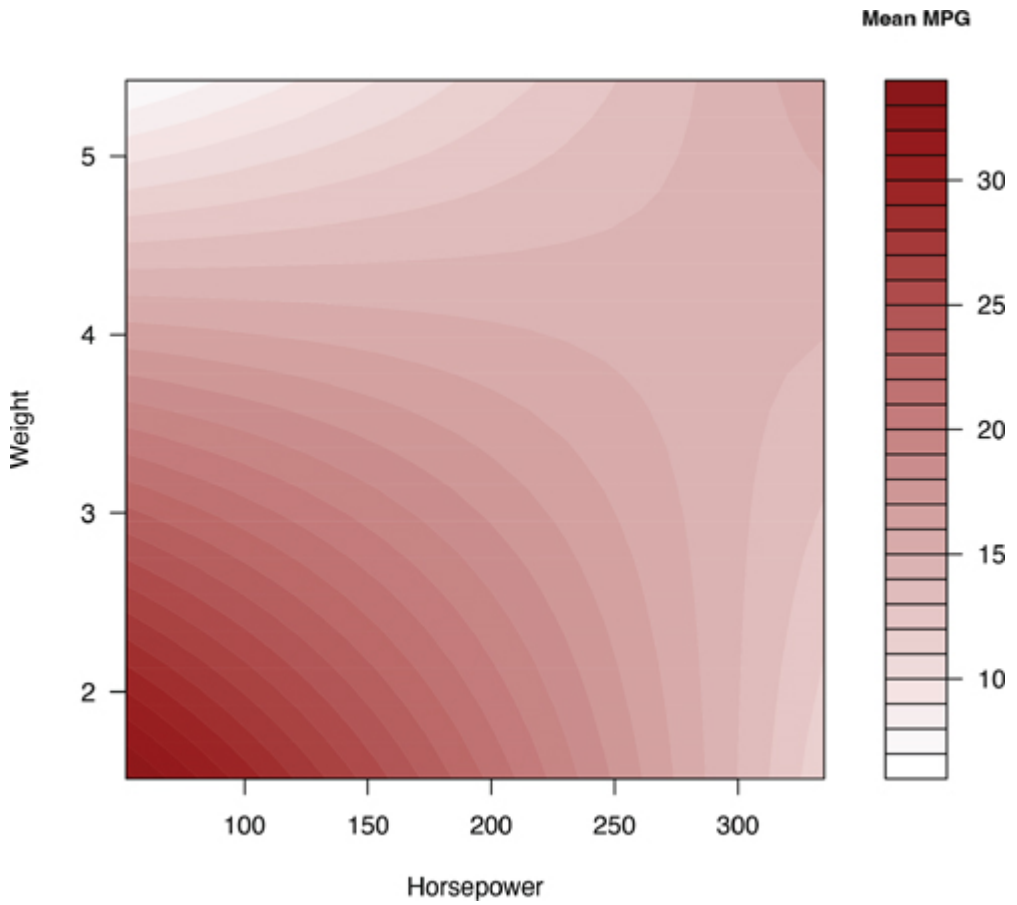


Figure 25-13: A filled contour plot of the response surface for the fitted multiple linear model of the `mtcars` data

The default color palette hasn't been used in this plot. Instead, you've supplied a custom palette (produced as a direct result of an appropriate call to `colorRampPalette`; refer to Section 25.1.3) to the relevant argument, moving from white at the lower end to dark red at the upper end. Note also that although the x-axis and y-axis titles are provided as usual to `xlab` and `ylab`, you have to supply the title for the color legend inside a call to `title` to the `key.title` argument. This is because `filled.contour` actually produces two plots, one for the image itself and one for the color legend, and uses the `layout` command (Section 23.1.4) to place them next to one another.

This internal use of `layout` isn't directly a problem, but, as briefly noted in Section 23.1.4, it complicates matters somewhat if you want to annotate the filled contour plot after the fact (for example, by adding

points to an existing graphic). This is because the original user coordinate system for the main image, which is rendered before the color ribbon, is lost.

Turn your attention back to the two-dimensional kernel estimate of the spatial `quakes` data (use the code from Section 25.4.1, page 740, to re-create the `quak.dens` object if you haven't already got it in your workspace). The following code creates a filled contour plot of the density surface using the built-in `topo.colors` palette, modifies the number of drawn levels from the default of 20 to 30, and locks the aspect ratio as previously deemed appropriate. In the same call, you can superimpose the points of the raw observations onto the image through special use of the optional `plot.axes` argument:

```
R> filled.contour(x=quak.dens$x,y=quak.dens$y,z=quak.dens$z,asp=1.069,
                  color.palette=topo.colors,nlevels=30,xlab="Longitude",
                  ylab="Latitude",key.title=title(main="KDE",cex.main=0.8),
                  plot.axes={axis(1);axis(2);
                             points(quakes$long,quakes$lat,cex=0.5,
                                    col=adjustcolor("black",alpha=0.3))})
```

Figure 25-14 shows the result.

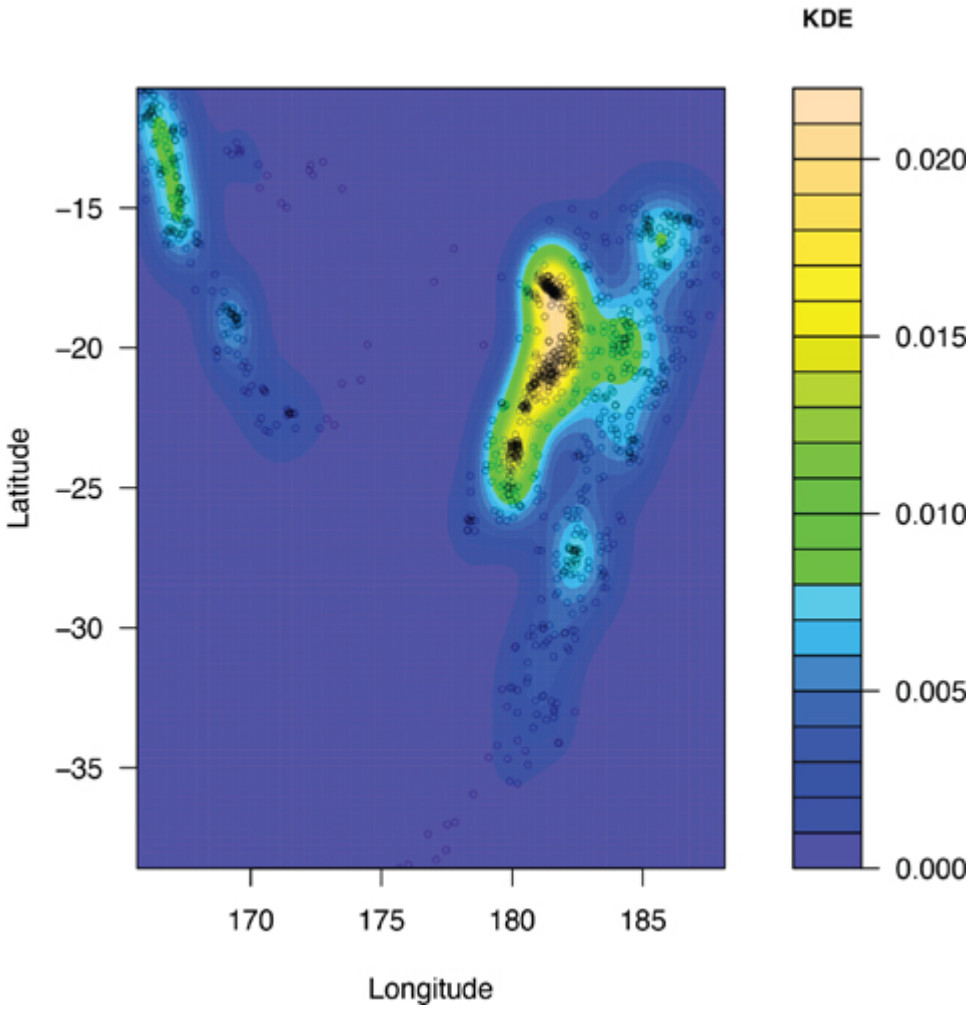


Figure 25-14: A filled contour plot of the kernel estimate of the probability density function of the spatial `quakes` *data, with raw observations superimposed.*

Look at how `plot.axes` is used; it effectively requires its own chunk of code. When `plot.axes` is invoked, you must explicitly tell it to mark the x- and y-axes if you want the labeled tick marks to remain. This is done with two calls to `axis` (refer to Section 23.4.3; `axis(1)` specifies the x-axis, and `axis(2)` is used for the y-axis). You add the data points with a call to `points`; in this example, these are instructed to plot at half size, with 30 percent opacity imparted with `adjustcolor`. Since you're supplying multiple separate commands at once to the `plot.axes` argument, each command needs to be separated by a semicolon (;) inside braces (`{ }`).

To summarize, while basic production of filled contour plots is straightforward, post-hoc annotation requires a little more forethought

since you're required to manually add the axes and perform any additional plotting actions within the same call to `filled.contour`. It won't work to, for example, produce a filled contour plot like the `quakes` KDE surface and then superimpose the raw observations with `points` as a second, separate line of code. If you try that, the observed data points will be unable to align correctly with their original user coordinates as indicated on the axes. See the documentation accessible via `?filled.contour` for more on these technicalities.

Exercise 25.3

Remember that you inspected multiple linear regression models of the cost of nuclear power plant construction in Chapters 21 and 22. The goal now will be to visually assess the impact of including or excluding an interactive term between two continuous predictors using contours. Revisit the `nuclear` dataset, available when you load the `boot` package, and bring up the help file to refresh your memory of the variables present.

a. Fit and summarize two linear models with construction cost as the response variable according to the following guidelines:
    i. The first should account for main effects of the two predictors concerning the date of issue of the construction permit and plant capacity.
    ii. The second, in addition to the two main effects, should include an interaction between permit issue date and capacity. (Refer back to Section 21.5 for details on fitting linear models with interactions.)

b. Set up appropriate z-matrices for plotting each of these response surfaces, as per Sections 25.3.1 and 25.3.2. Each one should be based on a 50×50 evaluation grid constructed using evenly spaced sequences in the capacity and date variables.

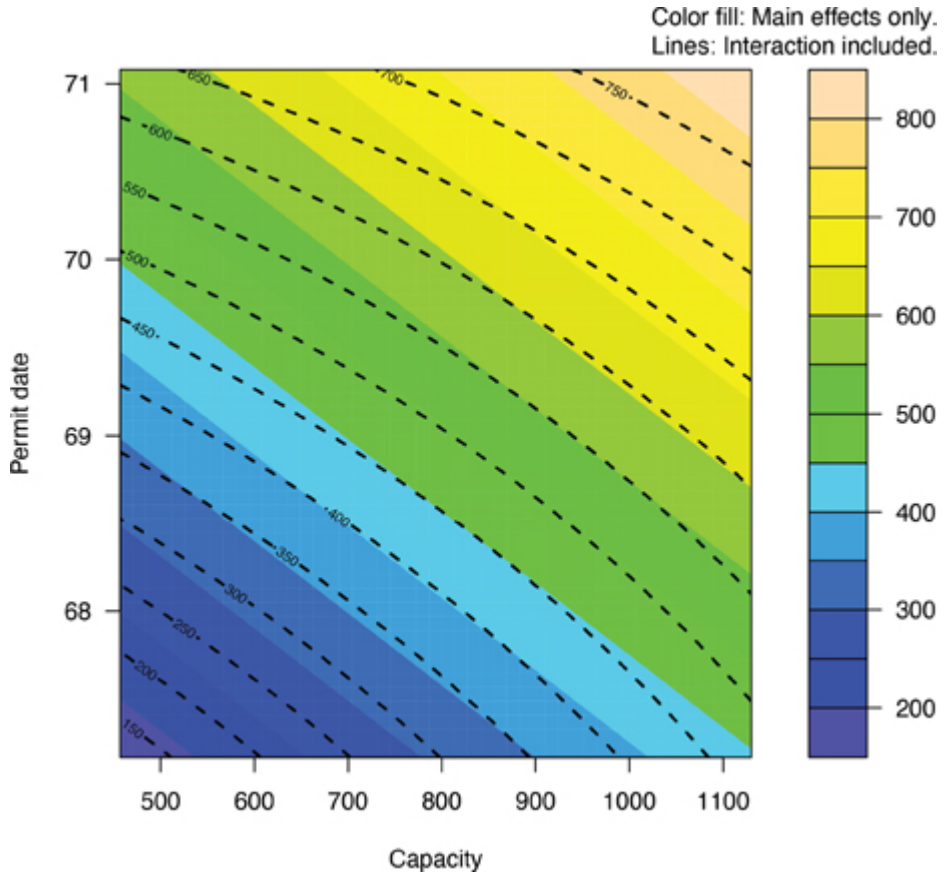
c. Specify `mfrow` in `par` so that you can display default-appearance contour plots for the two response surfaces as specified by (a)(i) and (a)(ii) next to one another. Do they appear similar? Does this tie in with the statistical significance (or lack thereof) of the interaction term in (a)(ii)?

d. To directly compare the two surfaces, use your choice of built-in color palette to produce a filled contour plot of the main-effects-only model and superimpose the contour lines of the interactive model on it. Take note of the following:

   - This plot is achieved in a single call to `filled.contour`. Recall the special way you use the `plot.axes` argument to draw additional features on an existing color-filled contour plot.
   - In use of `plot.axes`, the contour lines of the interactive model can be added with an appropriate call to `contour`. Recall the use of the optional argument `add`.
   - The superimposed contours should be dashed lines of double thickness.
   - The x- and y-axes should be included and given tidy titles.
   - Add some brief text describing the filled contours versus the contour lines (which will be different), with reference to the two versions of the construction cost model and with an additional call to `text` that makes use of a single mouse-clicked location from

`locator` (see Section 23.3). Note that this call will need to use the `xpd` argument (Section 23.2.3) to fully relax clipping for the text to be visible in any of the margins.

For reference, my result is shown on the next page.



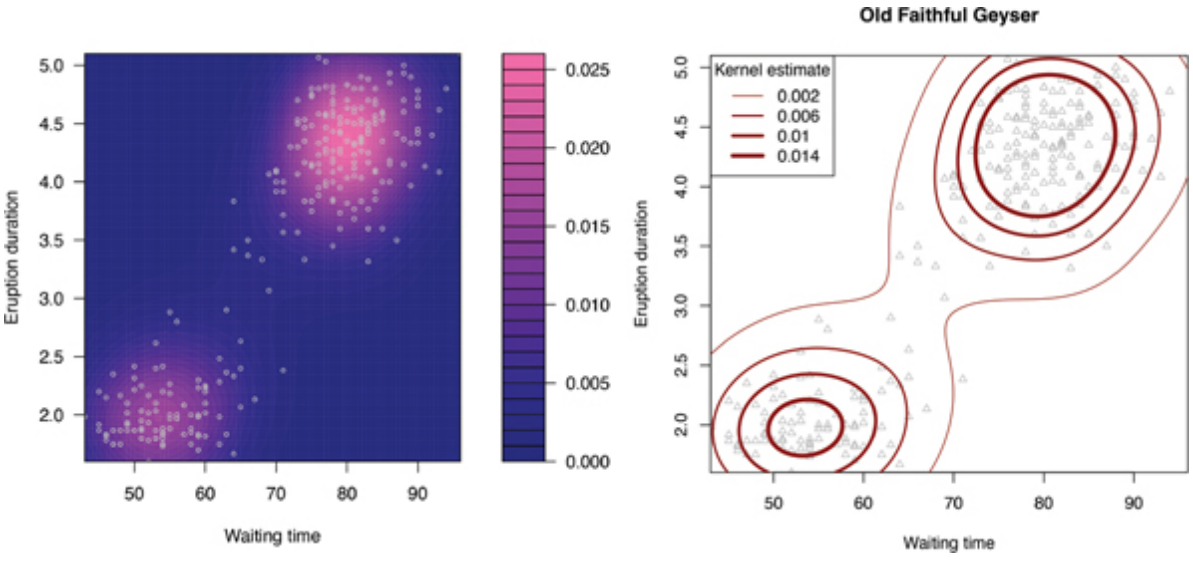
e. Recall the built-in `faithful` data, first seen in Section 14.5.1. This dataset contains observations of waiting times and durations of eruptions of the Old Faithful geyser in Yellowstone National Park, Wyoming. See the documentation in `?faithful` for details. Plot the data as a simple point-based plot with duration on the y-axis and waiting time on the x-axis.

f. Following details in Example 3 of Section 25.4.1 (starting on page 740), estimate the bivariate density of these data

via KDE using a 100×100 evaluation grid on the duration and waiting time variables. Produce a default-appearance contour plot thereof.

g. Create a filled contour plot of the kernel estimate using a custom palette (refer back to Section 25.1.3) that ranges from `"darkblue"` to `"hotpink"`. Include the raw data as halfsize gray points (remember the special use of the `plot.axes` argument in the `filled.contour` command). Label the axes and titles appropriately.

h. Re-plot the raw points as gray, 3/4-sized, type `2` point characters; set the style of the axes to restrict to exactly the ranges of the observed data; and ensure tidy axis titles and a main title. To this plot, add contour lines of the density estimate at the specific levels obtained in a sequence from 0.002 to 0.014 in steps of 0.004. Suppress the labeling of the contours. The contour lines should be dark red and increase in line width thickness for higher levels of the density. Add a legend referencing the density level at each of these lines.
My plots for (g) and (h) are shown here.



25.5 Pixel Images

A *pixel image* is arguably the most literal visual representation of a continuous surface approximated by a discretized evaluation grid. Its appearance is similar to a filled contour plot, but an image plot gives you more direct control over the display of each entry of the relevant z-matrix.

25.5.1 One Grid Point Equals One Pixel

Consider each entry of your z-matrix as a little rectangle whose color depicts its relative value. These rectangles, or *pixels*, are exactly what's depicted as the cells formed by the dashed gray lines making up the conceptual diagram of the z-matrix in Figure 25-9 on page 736. This emphasizes the important fact that the fineness of your evaluation grid sequences (in both the x- and y-coordinate directions) directly defines the size of each pixel and therefore the smoothness of the resulting image within a graphics device of a given size. A smaller pixel means an increase to the image's *resolution*.

The built-in `image` function plots pixel images. Much as with `contour` and `filled.contour`, you supply your x- and y-axis evaluation grid coordinates as increasing sequences to the `x` and `y` arguments, with the corresponding z-matrix supplied to `z`. Refer back to the `volcano` dataset first looked at in Example 1 of Section 25.4.1. The following line produces a pixel image version of Figure 25-10:

```
R> image(x=1:nrow(volcano),y=1:ncol(volcano),z=volcano,asp=1)
```

The result is provided in Figure 25-15.

Note again that you use the optional argument `asp=1` to enforce a oneto-one aspect ratio of the horizontal and vertical axes. This plot consists of exactly 87 × 61 = 5,307 pixels; each one represents a particular entry in the `volcano` matrix. Visually, the reflection of this image in the contour plot of the same data in Figure 25-10 is clear.

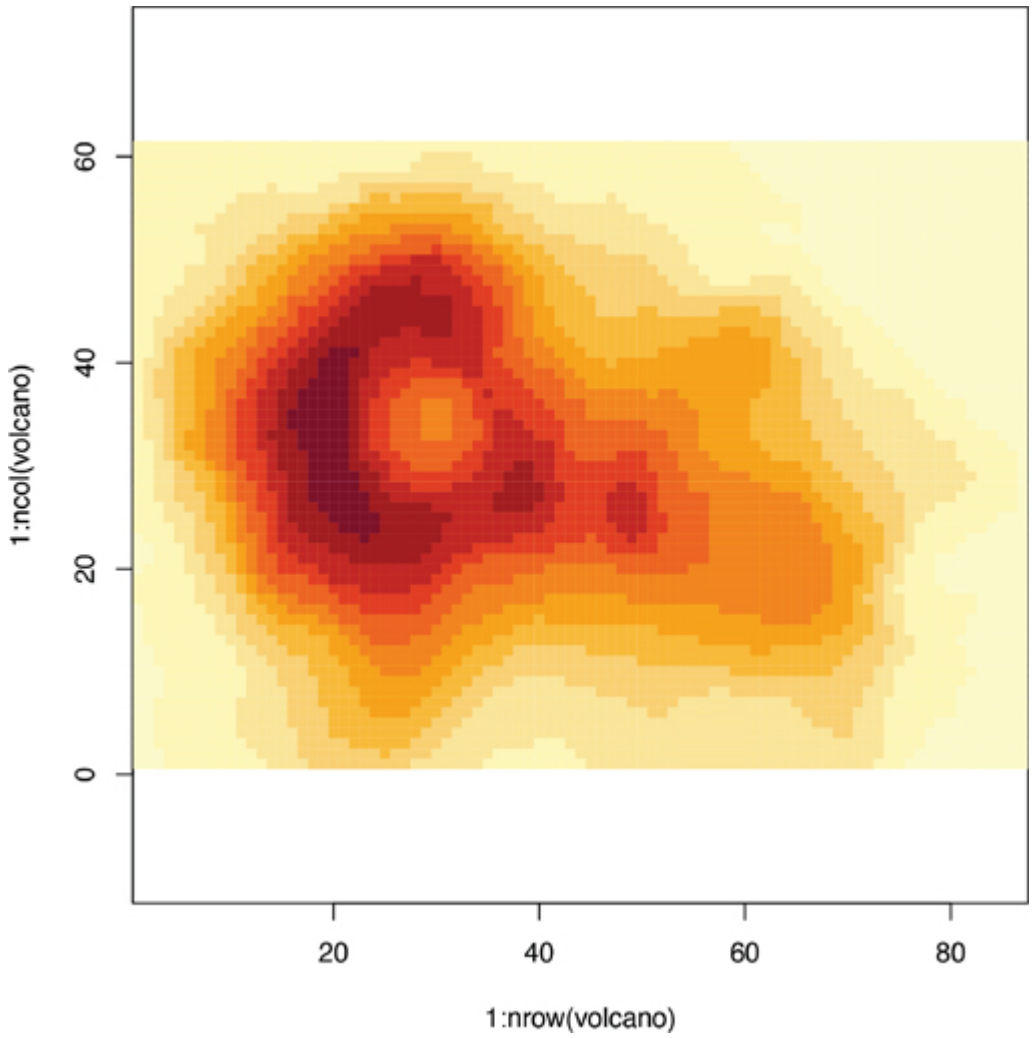


Figure 25-15: Using `image` *to produce a pixel image of the* `volcano` *topographical data*

The `image` command expects a vector of colors, usually supplied as hex codes from a palette, to be passed to its `col` argument. If this isn't specified, it defaults to 12 colors spanning one of the palettes available via the `hcl.colors` function (Section 25.1.2). One immediate concern, however, is the lack of a default color legend. Contributed tools such as the `colorlegend` function from the `shape` package (refer to Section 25.1.5) can therefore prove useful for these plots.

Return to the `mtcars` response surface from Example 2 of Section 25.4.1, which concerns the multiple linear regression model of MPG on horsepower and weight (and an interactive effect between the two predictors). The code for the necessary objects is reproduced here in a

shortened form for convenience (refer back to Section 25.4.1 for a fuller explanation of the operations):

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> len <- 20
R> hp.seq <- seq(min(mtcars$hp),max(mtcars$hp),length=len)
R> wt.seq <- seq(min(mtcars$wt),max(mtcars$wt),length=len)
R> hp.wt <- expand.grid(hp=hp.seq,wt=wt.seq)
R> car.pred.mat <- matrix(predict(car.fit,newdata=hp.wt),nrow=len,ncol=len)
```

Just as earlier, you've set up a matrix of 400 elements in `car.pred.mat`, which is based on sequences of length 20 in both continuous predictors.

Now make sure the `shape` package is loaded so you have access to its `colorlegend` function. The code that follows first sets up a custom palette of blue colors, sets new margin spacing that widens the area on the rightmost axis, and then plots the 20×20 response surface reflecting the predicted MPG before finally producing a corresponding color legend:

```
R> blues <- colorRampPalette(c("cyan","navyblue"))
R> par(mar=c(5,4,4,5))
R> image(hp.seq,wt.seq,car.pred.mat,col=blues(10),
         xlab="Horsepower",ylab="Weight")
R> colorlegend(col=blues(10),zlim=range(car.pred.mat),zval=seq(10,30,5),
               main="Mean\nMPG")
```

The result is shown on the left of Figure 25-16.

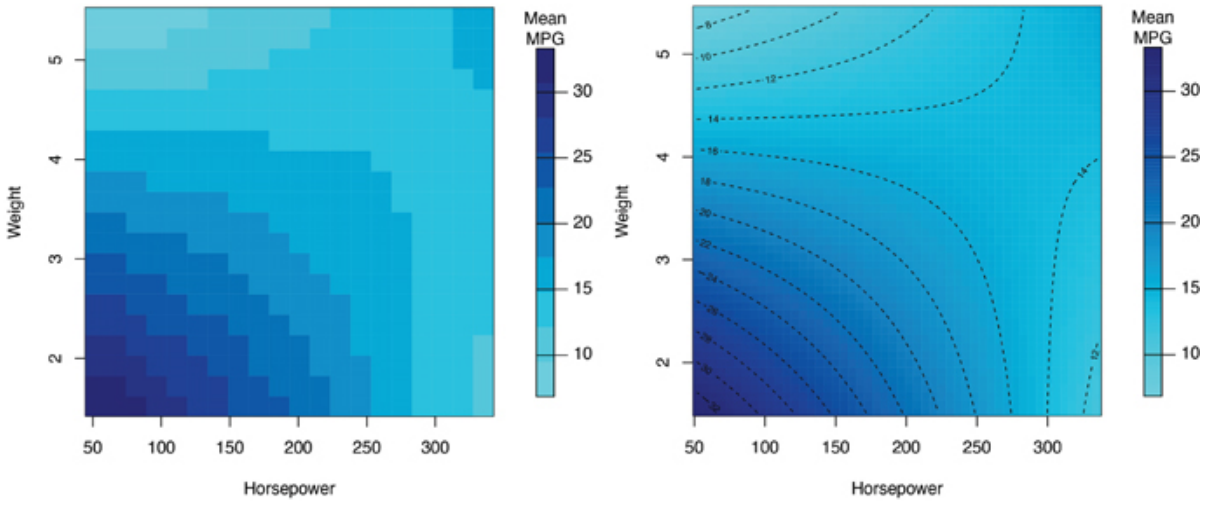


Figure 25-16: Two pixel images of the `mtcars` mean MPG response surface, with accompanying color legends. In terms of the evaluation grid in the horsepower and weight variables, the surface on the left has a resolution of 20^2; the image on the right is based on a finer 50^2 grid. Contours are superimposed upon the rightmost plot.

With a relatively coarse evaluation grid, the individual pixels making up the surface are prominent. You can easily increase the resolution of the parametric response surface by using finer sequences for the `hp.seq` and `wt.seq`. The code that follows does just that, by increasing `len` to `50`, and overwrites the objects used previously to construct the finer grid and the corresponding model-based predictions:

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> len <- 50
R> hp.seq <- seq(min(mtcars$hp),max(mtcars$hp),length=len)
R> wt.seq <- seq(min(mtcars$wt),max(mtcars$wt),length=len)
R> hp.wt <- expand.grid(hp=hp.seq,wt=wt.seq)
R> car.pred.mat <- matrix(predict(car.fit,newdata=hp.wt),nrow=len,ncol=len)
```

Then, the right-hand image of Figure 25-16 is produced with the following code:

```
R> par(mar=c(5,4,4,5))
R> image(hp.seq,wt.seq,car.pred.mat,col=blues(100),
         xlab="Horsepower",ylab="Weight")
R> contour(hp.seq,wt.seq,car.pred.mat,add=TRUE,lty=2)
R> colorlegend(col=blues(100),zlim=range(car.pred.mat),zval=seq(10,30,5),
               main="Mean\nMPG")
```

The newly plotted surface consists of $50^2 = 2{,}500$ pixels, as opposed to the previous image of merely $20^2 = 400$ pixels. The improvement to the picture as a consequence of this increase in resolution is obvious. In plotting the new image, the number of colors used (from the custom `blues` palette) is increased to 100 to provide smoother color transitions. Note also the use of `add` in a call to `contour`, which superimposes contour lines upon the image to provide further visual emphasis of the fluctuating surface over the evaluation grid. A legend is again added with an appropriate call to `colorlegend` as a final touch.

25.5.2 Surface Truncation and Empty Pixels

Because of its one-to-one literal representation of the z-matrix, a pixel image is especially useful when you want to plot a surface defined irregularly over the standard rectangular evaluation grid spanning the x- and y-axes. To carefully demonstrate this kind of manipulation, let's turn to a new dataset from the contributed suite of `spatstat` packages (Baddeley & Turner 2005, Baddeley et al. 2015), which comprise a vast array of statistical methods for spatial data analysis. Install the `spatstat` suite with a call to `install.packages("spatstat")`. Note that `spatstat` has a number of dependencies; see Appendix A.2.3 if you have any trouble downloading and installing the required content.

Example 4: Nonparametric Bivariate Density Estimate (Chorley–Ribble Data)

Once the `spatstat` functionality is installed and loaded into your current R session with a call to `library("spatstat")`, inspect the help file brought up by entering `?chorley` at the prompt. This details the Chorley–Ribble cancer data: spatial locations of 1,036 cases of cancer of the larynx and lung collected in the late 1970s and early 1980s in a particular region of England (data first analyzed by Diggle 1990). The `chorley` object is of a special class specific to `spatstat` (a `"ppp"` object: *planar point pattern*), but its components can be extracted just as if you're referencing members of a named list. Note that while `spatstat` offers many custom functions for producing and visualizing the kind of estimate we'll study here, it's instructive to use the existing base R functionality as much as possible in the following example.

You can retrieve the coordinates of the observations as the `$x` and `$y` components. To view the spatial dispersion of the observations using a basic R plot, the following line gives you the top-left image of Figure 25-17; therein, use `asp` to lock a 1-to-1 aspect ratio (the observations are expressed in kilometers):

```
R> plot(chorley$x,chorley$y,xlab="Eastings (km)",ylab="Northings (km)",asp=1)
```

Your goal is to display a kernel estimate of the two-dimensional probability density function of the cancer distribution over space, similar to what you did with the earthquake data in Example 3. You'll use the `kde2d` function for this; execute `library("MASS")` to gain access to it. Then, exactly as you used it for the spatial locations of `quakes`, the default KDE surface for the observed Chorley–Ribble data is given with the following:

```
R> chor.dens <- kde2d(x=chorley$x,y=chorley$y,n=256)
```

Note the specification of a fine 256×256 easting-northing evaluation grid.

To display the density estimate, use the built-in `rainbow` palette and engage the optional `start` and `end` arguments to restrict the total range of the palette to begin at red at the lower end and end at magenta/pink at the upper end (these arguments were mentioned briefly in Section 25.1.2; refer to the help file `?rainbow` for more details on the use of `start` and `end`). Prestore 200 colors from this palette with the following line:

```
R> rbow <- rainbow(200,start=0,end=5/6)
```

The image is produced by calling the following line:

```
R> image(x=chor.dens$x,y=chor.dens$y,z=chor.dens$z,col=rbow,asp=1)
```

Keep the plot open, so you can add to it.

Another component of the `chorley` data object, named `$window`, contains the vertices of an irregular polygon. This polygon defines the geographical study region in which the observations themselves were made. The `$window` component also happens to be another special object class of `spatstat`, namely, `"owin"` for *observation window*. Although it's possible to extract the specific vertices of the polygon and plot it manually with built-in functionality, the authors of `spatstat` have provided a standard `plot` method to use for this purpose.

With the plot resulting from running the previous call to `image` left open, calling the following line superimposes the border of the study

region upon the pixel image:

```
R> plot(chorley$window,add=TRUE)
```

The final result is given on the top right of Figure 25-17.

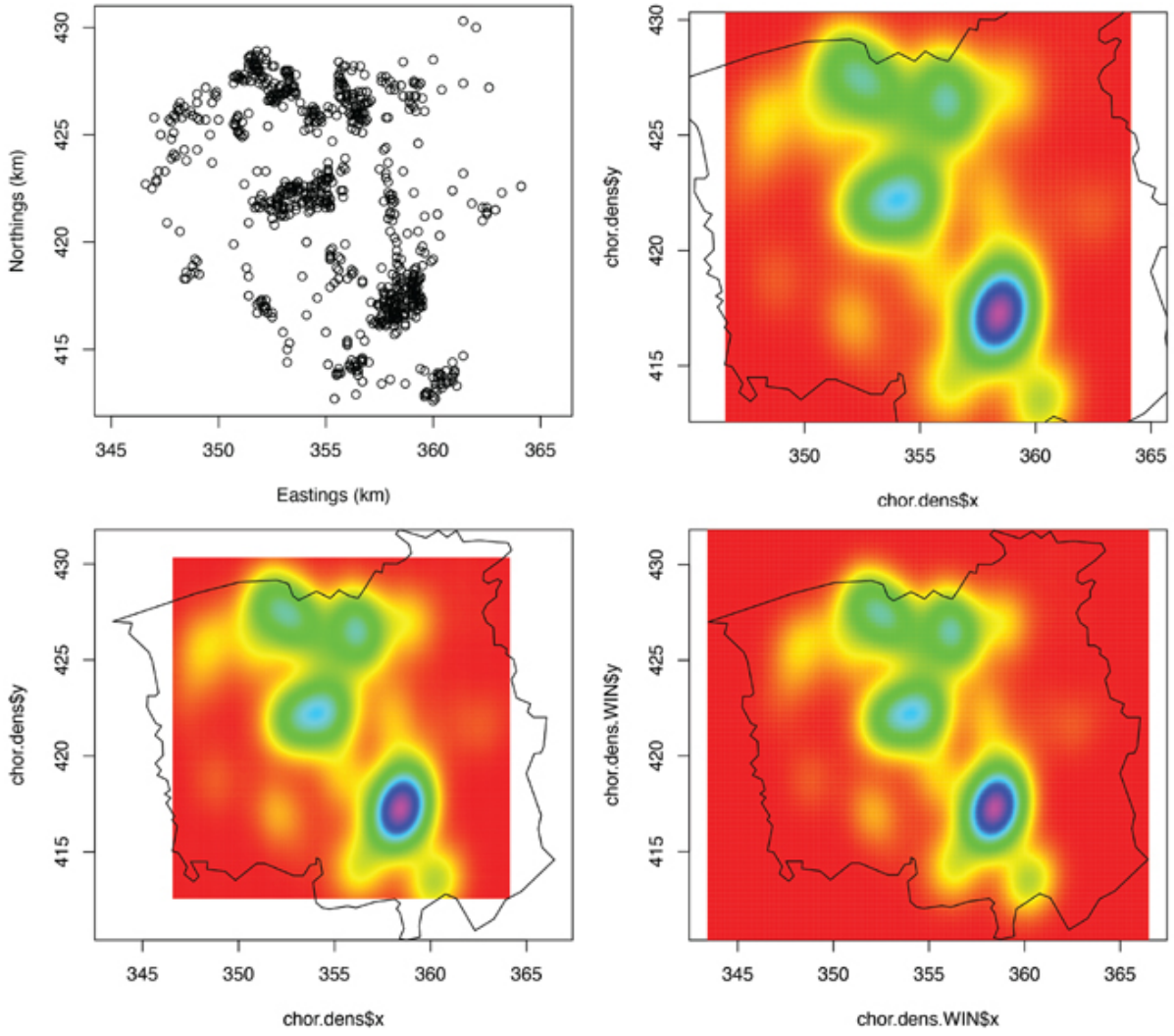


Figure 25-17: A visual experimentation showing progressive attempts to plot a two-dimensional kernel estimate of the probability density function of the Chorley–Ribble cancer data as a base R pixel image. Top left: The raw data. Top right: The default `kde2d` *result based on the data ranges with the study region superimposed. Bottom left: Expanding the* `xlim` *and* `ylim` *of the call to* `image` *when plotting the original density estimate. Bottom right: A revised density estimate, using the full x- and y-ranges of the study region to define the evaluation grid.*

You'll notice that the geographical region in which the data were collected is a little wider than the x- and y-ranges of the observations

themselves, so the current plot hasn't been able to show the region in its entirety. The following code confirms this numerically:

```
R> chor.WIN <- chorley$window
R> range(chorley$x)
[1] 346.6 364.1
R> WIN.xr <- chor.WIN$xrange
R> WIN.xr
[1] 343.45 366.45
R> range(chorley$y)
[1] 412.6 430.3
R> WIN.yr <- chor.WIN$yrange
R> WIN.yr
[1] 410.41 431.79
```

The x- and y-ranges of the study region can be obtained as the `$xrange` and `$yrange` components of the `$window` component (which is stored in the first line as the object `chor.WIN`). You can see that the overall study region is slightly larger when you compare its limits to the results of calling `range` on the raw data.

That's not the only problem, either. From the plot, you can also see that the KDE surface has been estimated and drawn in some areas that are actually *outside* the study region, so that will need to be fixed as well. (You'll look at that in a moment.)

So, first off, what can you do to ensure the entire geographical region is displayed? Well, you could use the ranges of the region as stored previously in the vectors `WIN.xr` and `WIN.yr` and supply them to the familiar optional `xlim` and `ylim` arguments when calling `image`:

```
R> image(chor.dens$x,chor.dens$y,chor.dens$z,col=rbow,
         xlim=WIN.xr,ylim=WIN.yr,asp=1)
R> plot(chor.WIN,add=TRUE)
```

The result of these two lines is given on the bottom left of Figure 25-17. Unfortunately, the original density estimate is still defined in terms of the original x- and y-ranges of the raw data, which gives you an "empty" border at which the estimate has not been evaluated. There

also remain certain areas at which the estimate *has* been evaluated but that fall outside the defined geographical observation window.

All this emphasizes the important fact that a z-matrix is specific to a predefined evaluation grid. The only way to get your density estimate to span the geographical study region for the Chorley–Ribble data is to revise your kernel estimate so that it's produced on an evaluation grid that spans the limits of the region. Fortunately, the `kde2d` function allows you to set optional x-y limits of the evaluation grid with the `lims` argument. This expects a numeric vector of length 4, with the x-axis lower and upper values followed by the y-axis lower and upper values, in that order. The following code reestimates the density using the study region limits and plots it:

```
R> chor.dens.WIN <- kde2d(chorley$x,chorley$y,n=256,lims=c(WIN.xr,WIN.yr))
R> image(chor.dens.WIN$x,chor.dens.WIN$y,chor.dens.WIN$z,col=rbow,asp=1)
R> plot(chor.WIN,add=TRUE)
```

The result is given on the bottom right of Figure 25-17.

With that, you've solved the problem of ensuring your surface spans the desired area. However, this definitely highlights the second problem: The data that were actually observed fall strictly within the defined polygon, but you can see plotted pixels outside the geographical region, which doesn't make sense. You can control precisely which pixels are plotted in any given pixel image by setting the relevant entries in your z-matrix to be `NA` if you don't want them drawn.

You'll need a mechanism that can decide whether a given cell entry in your z-matrix—namely, `chor.dens.WIN$z`—corresponds to a location inside or outside the polygon (the object `chor.WIN`). If it falls outside, you'll want to force that entry to be `NA`. In general, this type of decision-making requires you to test each element of the matrix with respect to its coordinate value on the evaluation grid, possibly using your own R function. Fortunately, in this case, the `inside.owin` function of `spatstat` does exactly that, but the principle remains the same whenever you need control over precisely which pixels are plotted and which aren't.

Given one or more two-dimensional (x,y) coordinates and an object of class `"owin"`, the `inside.owin` function returns a corresponding logical vector with a `TRUE` for those coordinates inside the defined region and a `FALSE` for any other coordinate. As a quick demonstration, observe the following result:

```
R> inside.owin(x=c(355,345),y=c(420,415),w=chor.WIN)
[1]  TRUE FALSE
```

This confirms what you can see from Figure 25-17: that the coordinate (355, 420) lies well within the polygon and that the coordinate (345, 415) doesn't.

Now you need to use the `inside.owin` function on every coordinate in the evaluation grid that z-matrix `chor.dens.WIN$z` sits on. First, create the full set of grid coordinates using `expand.grid`, in the same way as illustrated in Section 25.3.1:

```
R> chor.xy <- expand.grid(chor.dens.WIN$x,chor.dens.WIN$y)
R> nrow(chor.xy)
[1] 65536
```

Calling `nrow` on the resulting data frame of coordinates confirms you have exactly $256^2 = 65{,}536$ grid points as defined in the `chor.dens.WIN` KDE object. The following call then takes the two columns of `chor.xy` and makes use of logical negation (using `!`) to produce a logical vector that flags grid coordinates that are located *outside* the defined geographical region:

```
R> chor.outside <- !inside.owin(x=chor.xy[,1],y=chor.xy[,2],w=chor.WIN)
```

The final step is now at hand. For clarity, recast the long `chor.outside` vector as a 256×256 matrix to emphasize that it corresponds exactly to the z-matrix of interest. Then use this logical flag matrix to directly overwrite the "outside" entries in the z-matrix to be `NA`:

```
R> chor.out.mat <- matrix(chor.outside,nrow=256,ncol=256)
R> chor.dens.WIN$z[chor.out.mat] <- NA
```

All that's left is to plot the image with the newly manipulated z-matrix. Make sure you have the `shape` package loaded for the finishing touch of a color legend. Still using the `rbow` object created at the beginning of this example to specify colors, the following code creates the KDE pixel image plot with pixel points restricted to the geographical region defined by `$window` only:

```
R> par(mar=c(5,4,4,7))
R> image(chor.dens.WIN$x,chor.dens.WIN$y,chor.dens.WIN$z,col=rbow,
        xlab="Eastings (km)",ylab="Northings (km)",bty="l",asp=1)
R> plot(chor.WIN,lwd=2,add=TRUE)
R> colorlegend(col=rbow,zlim=range(chor.dens.WIN$z,na.rm=TRUE),
               zval=seq(0,0.02,0.0025),main="KDE",digit=4,posx=c(0.85,0.87))
```

First, you specify a wide right margin that will incorporate the color legend; this has been done in previous examples. Next, you invoke `image` to plot, specifically using an $\mathscr{L}$-shaped box (recall the interpretation of the `bty` argument as detailed in Section 23.4.2), and remain with a strict 1-to-1 x-y aspect ratio, and then you add the region polygon with slightly thicker lines than default. Finally, you execute `colorlegend` to obtain an appropriately positioned legend referencing the color values (the specific positioning and tick marks of which were found after a little trial and error). You can see the final result in Figure 25-18.

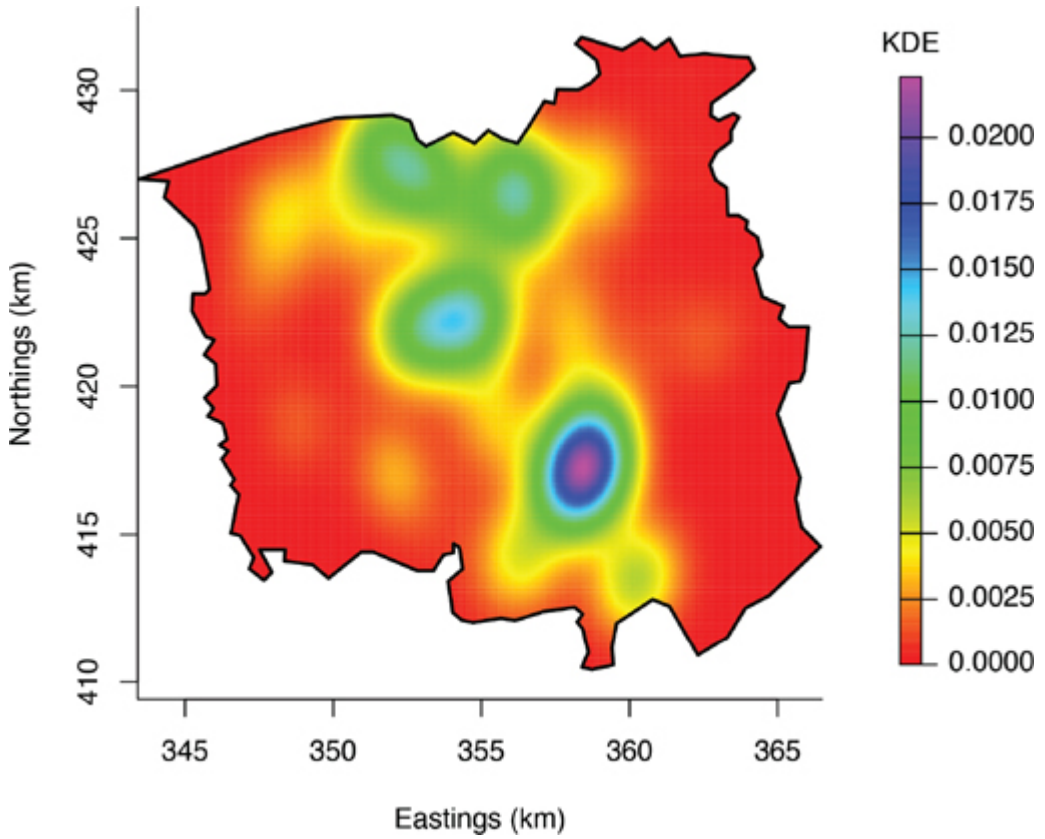


Figure 25-18: A final pixel image plot of the Chorley–Ribble KDE surface, restricted to the geographical study region of the originally collected data.

NOTE

In truncating the kernel estimate of the bivariate density estimate that was originally defined over the full rectangular evaluation grid, technically you no longer have a valid probability density function as a result (since the integral over the irregular region will no longer evaluate to a total probability of 1). A more mathematically sound approach requires a deeper knowledge of multivariate KDE and is beyond the scope of this text. Nevertheless, being able to truncate pixel plots like this is useful in any situation where you want to define your surface on a (possibly irregular) subset of an overall rectangular evaluation grid.

Exercise 25.4

Revisit the built-in `airquality` dataset and take a look at the help file to refresh your memory of the variables present. Create a copy of the data frame; select the columns pertaining to daily temperature, wind speed, and ozone level and use `na.omit` to remove any records with missing values.

a. From your explorations of these data in Chapter 24, there appears to be associations among daily temperature, wind speed, and ozone level. Fit a multiple linear regression model that aims to predict mean temperature based on the wind speed and ozone level, including an interactive effect. Summarize the resulting object. (Refer back to Section 21.5 for details on fitting linear models with interaction terms.)

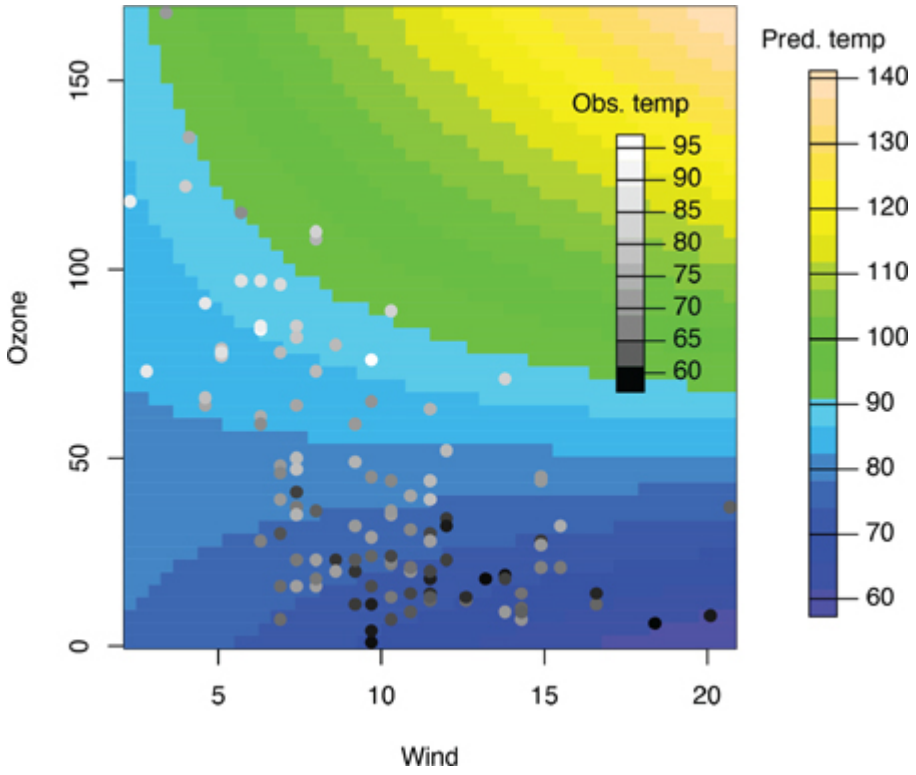
b. Using the model from (a), construct a z-matrix of predicted mean daily temperature based on a 50×50 evaluation grid in both wind speed and ozone. (Follow details in Example 2 of Section 25.4.1, starting on page 738, for guidance.) Create a pixel image of the response surface, superimposing the raw observations as per the following instructions:

   - A graphics device should be initialized based on bottom, left, top, and right margin lines of 5, 4, 4, and 6, respectively.
   - Twenty colors from the built-in `topo.colors` palette should be used to produce the image; include tidy axis titles.
   - Revisit the `normalize` function from Section 25.1.4 (page 719) and use the built-in function `gray` to generate a vector of gray colors (Section 25.1.2) based on the normalized raw temperature observations. Superimpose the raw observations based on wind speed and ozone onto the pixel image,

using the gray color vector to indicate the corresponding temperature observations.

- Make two separate calls to `colorlegend` of the `shape` package. The first should appear in the space on the right margin, referencing the surface itself. The second should use the built-in `gray.colors` function, setting the optional arguments `start=0` and `end=1`, to generate 10 shades of gray for use in the legend that references the raw temperature observations of the superimposed points. This legend should reside on top of the pixel image itself (in the upper-right quadrant where there are no raw observations).
- Both legends should have appropriate titles, and you may need to experiment a little with the `posx` and `posy` arguments to find satisfactory placement.

My result of this plotting exercise follows.



In Section 25.5.2, you used the `chorley` dataset (from the `spatstat` suite of packages) in creating a pixel image truncated to a subset of the overall rectangular evaluation grid. Ensure the `spatstat` library is loaded in your current R session and execute the following two lines:

```
R> fire <- split(clmfires)$intentional
R> firewin <- clmfires$window
```

This extracts the 1,786 locations of fires recorded as intentionally lit in a particular region of Spain. The spatial coordinates can be extracted as the `$x` and `$y` members of `fire`, and the geographical region itself is stored as a polygon in `firewin` (of the same class as the `chorley$window` object you looked at earlier). See the documentation obtained with `?clmfires` for further details on these data.

Now use details of Example 4 (starting on page 752) to complete the next few tasks.

d. Using the total x- and y-range of the study region, use `kde2d` from the `MASS` package to calculate a bivariate kernel estimate of the probability density function of the spatial dispersion of intentionally lit fires, stored in the `fire` object created previously. The KDE surface should be calculated based on a 256×256 evaluation grid.

e. Identify all points on the rectangular evaluation grid that fall outside the geographical region (stored as the `firewin` object) using `expand.grid` in conjunction with `inside.owin`. Set all corresponding pixels of the density surface to `NA`.

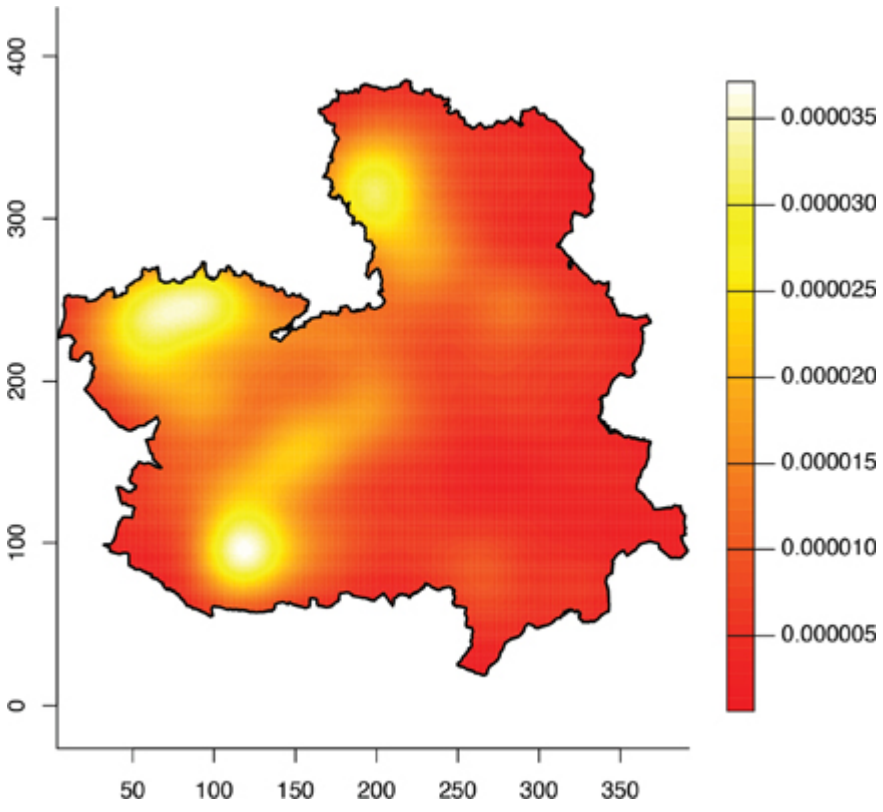
f. Construct a pixel image of the truncated density, as per the following:

   - The graphics device should have three lines of space on the bottom, left, and top of the plot region and

should have seven lines on the right.

- In producing the image itself, you should use 50 colors generated from the built-in `heat.colors` palette. A 1-to-1 aspect ratio should be maintained, the axis titles should be suppressed, and the box type set to be an $\mathscr{L}$ shape.
- The geographical study region should be superimposed onto the image using a double-width line.
- Using `colorlegend` from the `shape` package, a color legend referencing the density with an appropriate title should be placed to the right of the image. You'll need to experiment with the `posx` argument for placement. Label the legend at a sequence from `5e-6` to `35e-6` in steps of `5e-6` (refer to Section 2.1.3 for an explanation of e-notation); also, use the appropriate argument of `colorlegend` to ensure these labels are able to display up to six decimal places of precision.

For reference, my result is given here:



25.6 Perspective Plots

The last kind of plot you'll look at in this chapter is the *perspective plot*, sometimes referred to as a *wireframe plot*. Unlike contour plots and pixel images, where fluctuations in the surface are emphasized with line patterns or colors, a perspective plot uses a third dimension against which the z-value is plotted.

25.6.1 Interpreting Basic Plots and Adjusting Angles

Perspective plots are especially useful when you want to emphasize the fluctuating nature of the values populating your z-matrix. For example, in some applications you might want to get a good impression of the relative extremity of any present peaks or troughs in the plotted surface, which is harder to do in, for example, a pixel image or contour plot.

Recall the `mtcars` response surface, the result of predicting from a multiple linear model fit, plotted as contours and as pixel images (refer

to Sections 25.4.1 and 25.5.1). You created a 20×20 evaluation grid in the horsepower and weight variables, as well as a corresponding z-matrix of 400 giving the predicted mean MPG result:

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> len <- 20
R> hp.seq <- seq(min(mtcars$hp),max(mtcars$hp),length=len)
R> wt.seq <- seq(min(mtcars$wt),max(mtcars$wt),length=len)
R> hp.wt <- expand.grid(hp=hp.seq,wt=wt.seq)
R> car.pred.mat <- matrix(predict(car.fit,newdata=hp.wt),nrow=len,ncol=len)
```

Use the built-in R function `persp` to create perspective plots. Its basic usage is the same as `contour`, `filled.contour`, and `image`. Your increasing sequences in the x- and y-axis directions, which define the evaluation grid, are passed to `x` and `y`, with your corresponding y-matrix passed to `z`. Bring up the default appearance for the 20×20 `mtcars` response surface with the following line:

```
R> persp(x=hp.seq,y=wt.seq,z=car.pred.mat)
```

The result appears in the top left of Figure 25-19.

Interpretation of a perspective plot is straightforward. The default viewing angle shows the x-axis in the foreground, increasing from left to right, and the y-axis on the left side, increasing from the foreground to deeper in the background. In this way, you can think of the evaluation grid as lying flat along the bottom in the resulting 3D graphic, with the z-axis against which your surface is rendered increasing from the bottom upward to the top.

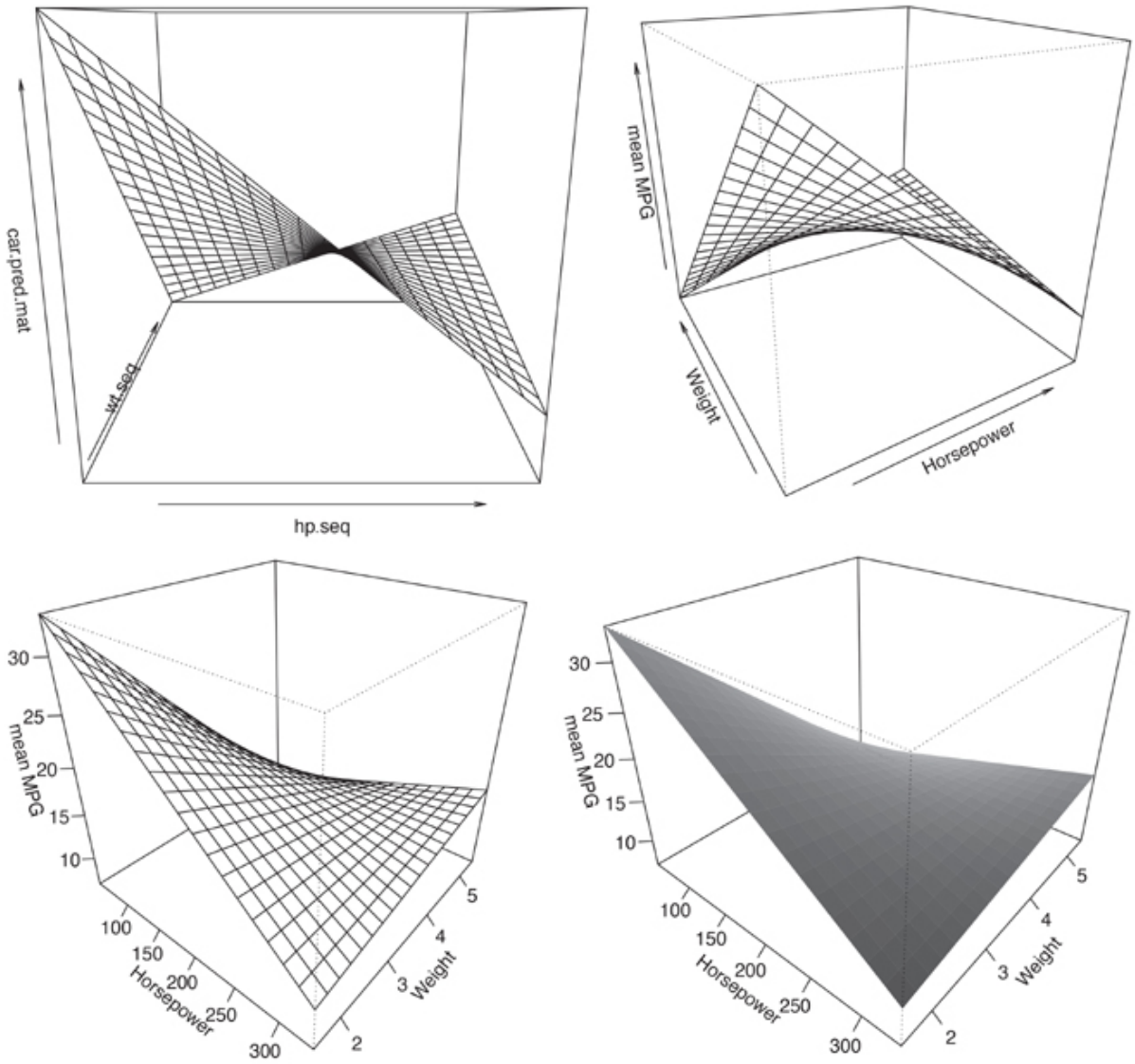


Figure 25-19: Perspective plots of the 20×20 `mtcars` response surface created using `persp`. Top left: Default appearance. Top right: Using `theta` and `phi` to adjust the viewing angle. Bottom left: Setting `ticktype="detailed"` to provide detailed axis labeling. Bottom right: Adding depth shading using `shade` and removing facet border lines with `border=NA`.

The viewing angle is one of the most important aspects of such a plot. In `persp`, you can control it with the two optional arguments `theta`, which spins the plot around horizontally, and `phi`, which adjusts the vertical viewing position. Both are specified in degrees; `theta` defaults to `0`, so you're looking directly at the x-axis spanning left to right in front of you, and `phi` defaults to `15` to give a slightly elevated viewing position so you can see the y-axis extending foreground to background. In

general, you can think of the possible value of `theta` as anywhere from `0` to `360`, representing a complete horizontal spin of the plot. Think of the possible values of `phi` as being between `90` to `-90`, the range of which moves you from a bird's-eye view (directly from the top looking down) to a submarine view (directly from the bottom looking up), respectively.

This second example demonstrates a manipulation of the default viewing angle:

```
R> persp(x=hp.seq,y=wt.seq,z=car.pred.mat,theta=-30,phi=23,
        xlab="Horsepower",ylab="Weight",zlab="mean MPG")
```

In fact, it's this line of code that originally produced the rightmost image in Figure 21-9 on page 593 (when you were introduced to the concept of an interactive term between two continuous predictors in a multiple linear regression model). The graphic is reproduced here in the top right of Figure 25-19. The axis titles are tidied up using `xlab` and `ylab`, with `zlab` used to control the title for the third, vertical axis. The use of `theta` and `phi` in this instance has elevated the viewing point slightly more than the default and rotated the plot so that the origin (in other words, the lower vertex denoting the lower limit of the x-y plane) is prominent in the foreground. It's worth noting that increasing `theta` from `0` rotates the plot in a clockwise-horizontal fashion, but you could also supply a negative value to that argument to rotate the plot in the other direction. Setting `theta=-30`, as shown here, has the same effect as setting `theta=330`.

By default, there are no tick marks or labels included, only directional arrows. You can remedy this by setting the optional `ticktype` argument to `"detailed"`. You can find the result of the following line in the bottom left of Figure 25-19, which also offers another viewing angle:

```
R> persp(x=hp.seq,y=wt.seq,z=car.pred.mat,theta=40,phi=30,ticktype="detailed",
        xlab="Horsepower",ylab="Weight",zlab="mean MPG")
```

The help file `?persp` details a host of other arguments specific to controlling the presentation of any given perspective plot. As a few

examples, you could shade the surface in grayscale to emphasize the 3D depth of the image, you could change the color or suppress the plotting of the grid lines making up the surface itself, or you could change the relative length of the z-axis. The final plot of the `mtcars` response surface illustrates such actions. The result of the following call is visible in the bottom right of Figure 25-19:

```
R> persp(x=hp.seq,y=wt.seq,z=car.pred.mat,theta=40,phi=30,ticktype="detailed",
         shade=0.6,border=NA,expand=0.8,
         xlab="Horsepower",ylab="Weight",zlab="mean MPG")
```

With the same viewing angle as the previous plot, this plot uses the `shade` argument to shade the surface facets. This produces a lighting-style effect, enhancing the perceptive depth slightly. The calculations for the shading rely on a non-negative numeric value; setting `shade=0.6` provides a moderatestrength effect. You might like to experiment with larger or smaller values. If you're shading the surface in this way, it's usually best to suppress the grid lines that by default make up the surface; you can set `border=NA` to achieve this (the `border` argument can also be used to simply change the surface grid color by supplying any valid R color to it). Finally, the `expand` argument is used to adjust the size of the z-axis. Specifying `expand=0.8` requests a vertical axis that is 80 percent the size of the axes in the evaluation grid, producing a slightly "squashed down" prism in which the surface is drawn. You could also use values greater than 1 for `expand`, in which case the effect would be to "stretch out" the plot along the vertical.

25.6.2 Coloring Facets

Like most traditional R plotting commands, you can use the optional `col` argument to color the facets of a perspective surface. To color a perspective surface with a constant color throughout, provide `col` with a single value.

If you're interested in `col`, however, it's often the case that you want to color the surface according to the fluctuating z-values to highlight the changing value of the bivariate function. To successfully do this for

the facets making up the surface, it's important to understand that these facets aren't the same as the pixels that would make up a pixel image of the same z-matrix. Where `image` pixels are directly represented by the entries of, say, your $m \times n$-sized z-matrix, `persp` facets should be interpreted as the space *between* border lines drawn at those matrix entries, leaving you with $(m - 1) \times (n - 1)$ facets. In other words, in a perspective plot, each z-matrix entry lies at an intersection of the drawn lines; the z-matrix entries are *not* situated in the middle of each facet.

To illustrate this, take another look at Figure 25-9 on page 736. When you use `image`, R automatically calculates the pixel sizes based on your x- and y-axis evaluation grid sequences and plots the surface based on the rectangles formed by the dashed gray lines, with the z-matrix entries `a`, `b`, `c`, and so on, represented directly. When you use `persp`, however, the visible border lines are represented by the solid-line grid, intersecting at each entry; the facets of the resulting surface are formed by the space between these lines, each one defined by four adjacent entries. Figure 25-20 shows a section of the hypothetical grid in Figure 25-9, where I've used thickened-line boxes to highlight one pixel as interpreted by `image` and one facet as interpreted by `persp`. With that, you can see why, in Figure 25-9, there would be exactly $6 \times 4 = 24$ pixels in an image plot but $5 \times 3 = 15$ facets in a perspective plot.

The `col` argument needs to specify the $(m-1) \times (n-1)$ facet colors (assuming an $m \times n$ z-matrix passed to `z`). The typical way to find this in R if you're intending to color the facets according to the z-value is to first calculate each facet's z-value, which will be the average of the four adjacent z-matrix entries. Only thereafter can you deploy one of the color assignment approaches from Section 25.1.4.

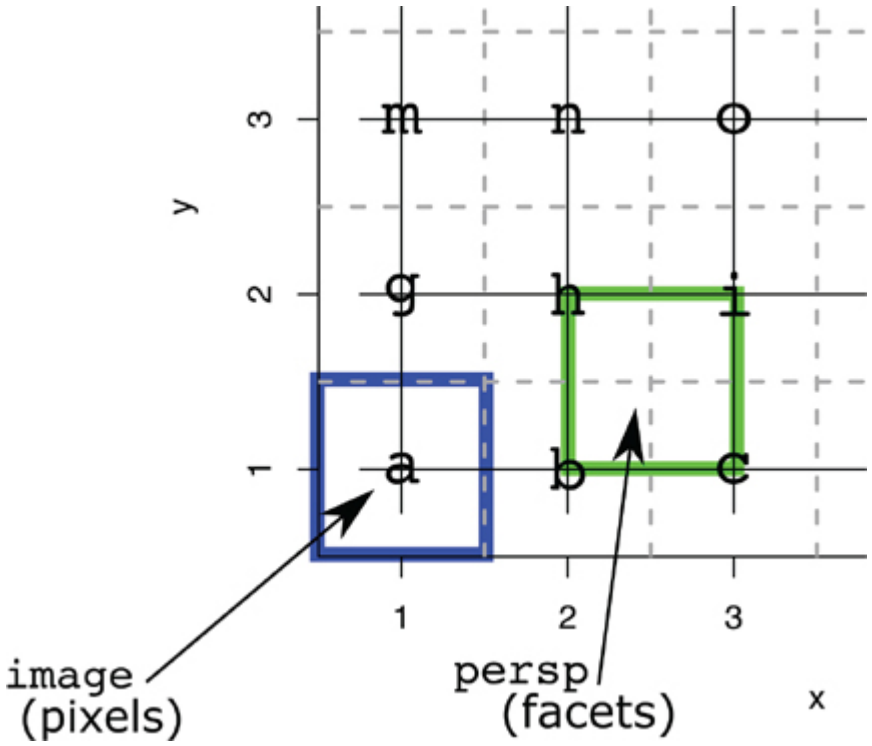


Figure 25-20: Illustrating the difference in treatment of the z-matrix in a pixel image and in a perspective plot. The highlighted box in the bottom-left corner represents an `image` *pixel of the value* `a` *in the z-matrix; the highlighted box to the right represents a* `persp` *facet formed by the values* `b`, `h`, `i`, *and* `c`. *For coloring, the* `z` *value of the highlighted facet will be calculated as the mean of those four entries—in other words,* `(b + h + i + c)/4`.

Let's recast the pixel image of the Chorley–Ribble kernel density estimate (Example 4; Figure 25-18), complete with z-axis-specific coloring, as a perspective plot. First, make sure you have the packages `spatstat` and `MASS` already loaded. Then repeat the code from earlier to obtain the kernel estimate on the appropriate evaluation grid, truncated to the geographical study region. It is reproduced here:

```
R> chor.WIN <- chorley$window
R> chor.dens.WIN <- kde2d(chorley$x,chorley$y,n=256,
                          lims=c(chor.WIN$xrange,chor.WIN$yrange))
R> chor.xy <- expand.grid(chor.dens.WIN$x,chor.dens.WIN$y)
R> chor.out.mat <- matrix(!inside.owin(x=chor.xy[,1],y=chor.xy[,2],w=chor.WIN),
                          256,256)
R> chor.dens.WIN$z[chor.out.mat] <- NA
```

Next, you need to calculate all the facet y-values; the following code does this en masse:

```
R> zm <- chor.dens.WIN$z
R> nr <- nrow(zm)
R> nc <- ncol(zm)
R> zf <- (zm[-1,-1]+zm[-1,-nc]+zm[-nr,-1]+zm[-nr,-nc])/4
R> dim(zf)
[1] 255 255
```

The first three lines simply store the z-matrix as the object `zm` and its total rows and columns (both 256, in this case) as `nr` and `nc`, respectively, for compactness of the code.

The fourth line is where the relevant calculations happen, giving a matrix of the facet z-values. It does this systematically, by element-wise summation of four versions of the original z-matrix: `zm[-1,-1]` (first row and first column omitted), `zm[-1,-nc]` (first row, last column omitted), `zm[-nr,-1]` (last row, first column omitted), and `zm[-nr,-nc]` (last row, last column omitted). When the four alternates are summed in this way and divided by 4 at the end, the result is a matrix `zf`, each internal element of which is the four-point average of the corresponding "rectangle" of four adjacent entries in the original z-matrix, exactly as noted in the discussion and caption of Figure 25-20. The final call to `dim` on `zf` confirms the size of the result. Since there are a total of 256×256 evaluation grid lines in the defined z-matrix, these encapsulate a total of 255×255 perspective facets.

The hard work is done, and all you need to do now is assign the colors from your palette to the calculated facet z-values in `zf`. You can do this using either the categorization or the normalization approach, as noted in Section 25.1.4; for simplicity, let's stick to categorization. Consider the following code:

```
R> rbow <- rainbow(200,start=0,end=5/6)
R> zf.breaks <- seq(min(zf,na.rm=TRUE),max(zf,na.rm=TRUE),length=201)
R> zf.colors <- cut(zf,breaks=zf.breaks,include.lowest=TRUE)
```

The first line is repeated from Section 25.5.2 to generate the same 200 colors from the built-in `rainbow` palette as were used in the pixel images. The second line sets up an evenly spaced sequence spanning the range of the calculated facet z-values to form the category break points that are required by the categorization approach. Note the use of `na.rm=TRUE` in the required calls to `min` and `max` to avoid all the `NA` entries present in `zf` (remember, the surface has been truncated to the irregular polygon representing the geographical study region). The sequence is one more in length than the number of generated colors; again, refer to Section 25.1.4 for this necessary feature of the categorization approach. Lastly, `cut` (Section 4.3.7) assigns each of the `zf` facet value entries an appropriate rank with respect to the 200 ordered bins. As you've learned, the `zf.colors` ranks will subsequently be used to index the vector of 200 colors stored in `rbow` when plotting.

With that, you can enjoy the fruits of your labor! The following code plots the bivariate kernel density estimate of the Chorley–Ribble observations as a perspective plot using facet coloring to reflect the relative height of the surface along the z-axis. Border lines are suppressed to show off the color clearly, the z-axis is scaled down slightly, and a color legend is inserted on the right side (ensure the `shape` package has been loaded for that; refer back to Section 25.1.5) after manipulating the default figure margins via `mar` in a call to `par` to create extra space for it:

```
R> par(mar=c(0,1,0,7))
R> persp(chor.dens.WIN$x,chor.dens.WIN$y,chor.dens.WIN$z,border=NA,
         col=rbow[zf.colors],theta=-30,phi=30,scale=FALSE,expand=750,
         xlab="Eastings (km)",ylab="Northings (km)",zlab="Density")
R> colorlegend(col=rbow,zlim=range(chor.dens.WIN$z,na.rm=TRUE),
               zval=seq(0,0.02,0.0025),main="KDE",digit=4,
               posx=c(0.85,0.87),posy=c(0.2,0.8))
```

You can find the result in Figure 25-21.

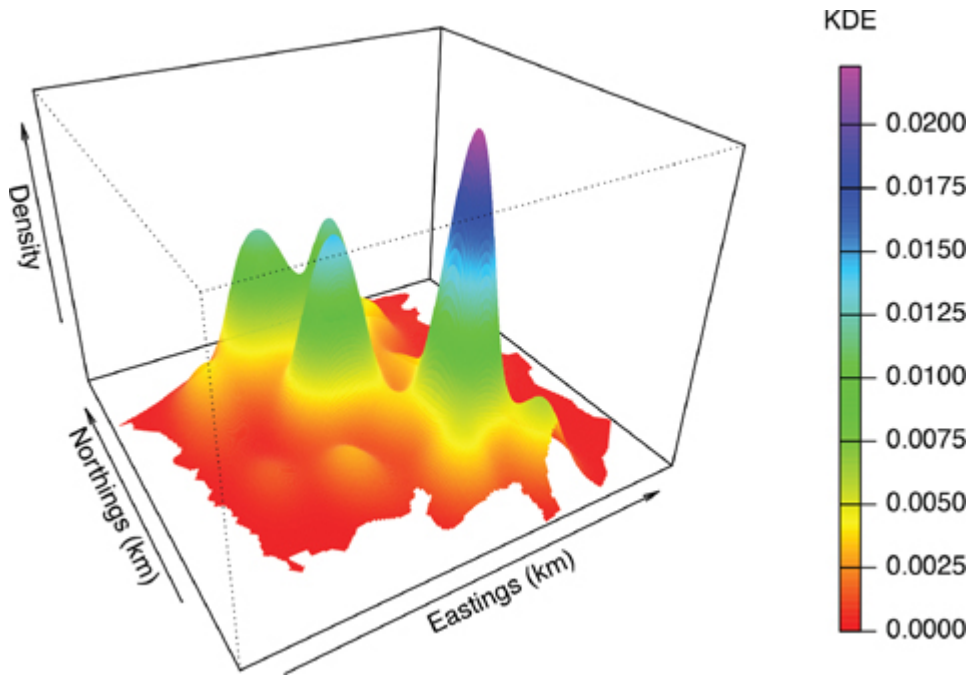


Figure 25-21: A perspective plot of the Chorley–Ribble density estimate demonstrating facet coloring that changes according to the z-value of the surface.

Note I've included the optional argument `scale=FALSE` in the execution of `persp`. This retains a 1-to-1 aspect ratio in the x- and y-coordinate directions; this is useful since you're looking at geographical data. Unfortunately, this also forces the density estimate values on the z-axis to be scaled in the same way, which makes no sense in the context of the current plot. To avoid the small scale resulting in a supremely flat appearance of the surface itself, you need to use `expand` to artificially amplify the surface along the third axis. In this instance, multiplying it by a factor of around 750 provides a visually pleasing result. Note that this would not be necessary if you left the `scale` argument at its default `TRUE` value (since, in that case, R internally scales all three axes for a 1-to-1-to-1 aspect ratio).

25.6.3 Rotating with Loops

There's one last bit of fun you can have with perspective plots if you want to get a good overall view of the plotted surface. Using a simple `for` loop (Section 10.2) to increment either `theta` or `phi`, you can perform a series of repeated calls to `persp`, each one at a slightly new angle. Doing

this in sequence results in an animation—essentially a cartoon—of a rotating surface, allowing you to see it from all different sides.

Consider the following basic function in the R editor:

```
persprot <- function(skip=1,sleep=0.05,...){
    for(i in seq(90,20,by=-skip)){
        persp(phi=i,theta=0,...)
        Sys.sleep(sleep)
    }
    for(i in seq(0,360,by=skip)){
        persp(phi=20,theta=i,...)
        Sys.sleep(sleep)
    }
}
```

Using an ellipsis (see Section 11.2.4), `persprot` simply takes all the arguments you'd usually supply to a call to `persp`, barring `theta` and `phi`. Then comes a `for` loop, which immediately calls `persp` with `theta=0` and the content of the ellipsis. The `for` loop alters the vertical viewing angle, starting with `phi=90` (birds-eye view) and moving down to a mildly elevated `phi=20`. A second `for` loop then completes a full 360-degree horizontal rotation by altering `theta`.

The only two formally tagged arguments are `skip`, which determines the amount `phi` and `theta` increment by at each iteration, and `sleep`, which controls how long to "pause" between frames (refer to Section 12.2.1 for the effect of the `Sys.sleep` function). The defaults, `skip=1` and `sleep=0.05`, move through the integer-valued angles with a minimal delay between each individual perspective plot. Increasing `skip` will reduce the time it takes to complete the rotation for a given value of `sleep`, though it makes for a more jagged animation.

Depending on your hardware and the type of graphics device you're using, you may want to experiment with `skip` or `sleep`. Note that not all graphics device types will be well suited to the animation effect sought by running this rather crude function (as you'll see, if you're using the RStudio IDE, it's best to use `dev.new` first). That said, when running the

base R GUI applications on macOS or Windows, I find `persprot` works well under default graphics settings.

Import the function to try it; let's do so here for a perspective plot of a kernel estimate of the probability density function from the spatial `quakes` locations you first examined in Example 3, Section 25.4.1. With the `MASS` package already loaded, produce the density estimate on a 50×50 evaluation grid with the following line:

```
R> quak.dens <- kde2d(x=quakes$long,y=quakes$lat,n=50)
```

Then use `persprot` just as you'd use `persp`, without needing to specify either `theta` or `phi`. For simplicity, we'll ignore locking the y/x aspect ratio (as we've done for previous plots of these spatially referenced data). If you're using RStudio, first run the following line to ensure a base R graphics device is opened, bypassing the in-built RStudio plot viewer:

```
R> dev.new(noRStudioGD=TRUE)
```

Play the cartoon by executing the following line:

```
R> persprot(x=quak.dens$x,y=quak.dens$y,z=quak.dens$z,border="red3",shade=0.4,
           ticktype="detailed",xlab="Longitude",ylab="Latitude",
           zlab="Density")
```

Figure 25-22 shows a series of screenshots of the rotating plot.

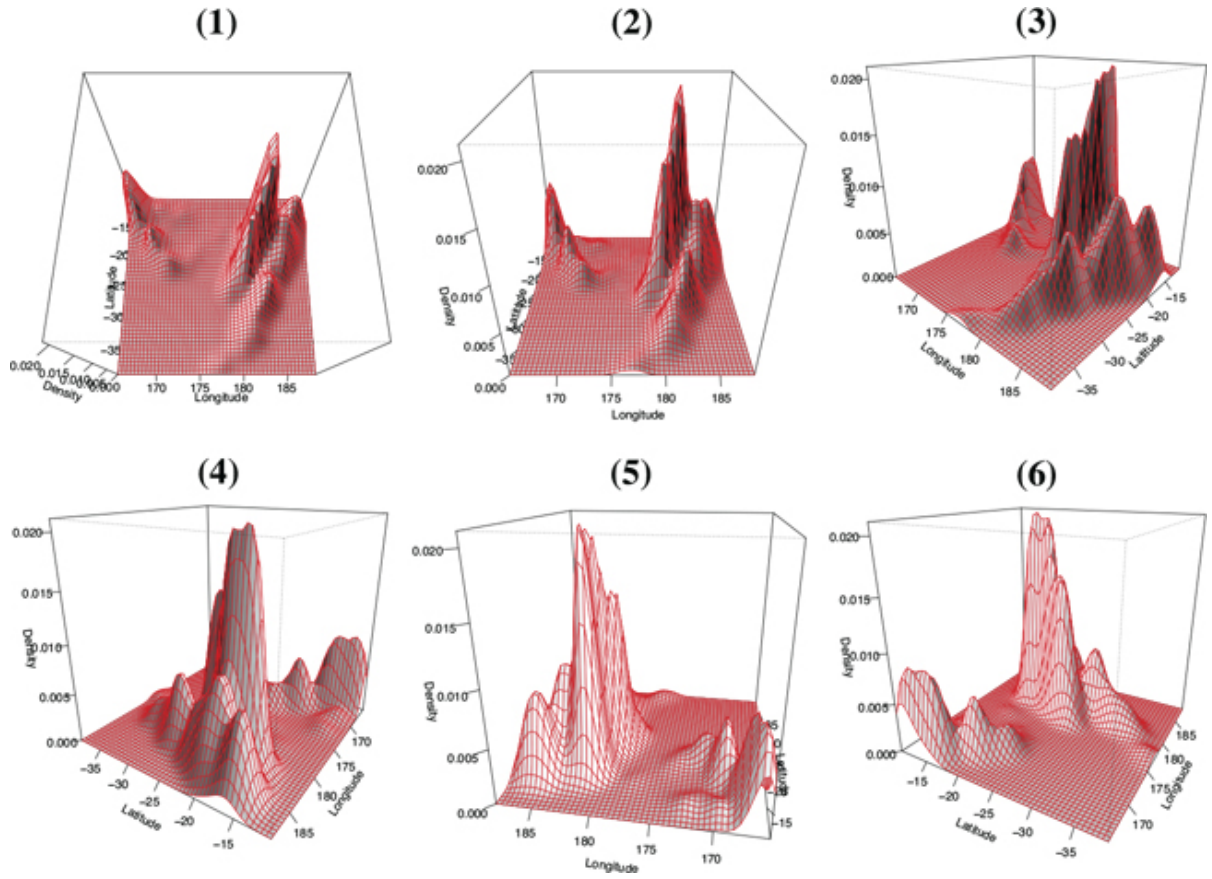


Figure 25-22: Screenshots of a rotating perspective plot of a KDE surface for the spatial earthquake locations after a call to the custom `persprot` *function (numbering follows the progression of the rotation)*

Exercise 25.5

In Exercise 25.3 (a), you revisited the `nuclear` dataset from the `boot` package and fitted two multiple linear regression models aiming to model mean construction cost by permit date issue and plant capacity: one with main effects only and the other including an interaction term between the two continuous predictors.

a. Refit these two versions of the model and produce perspective plots of the response surfaces based again on a 50×50 evaluation grid (relevant detail can be found in the

discussion of Example 2 starting on page 738), taking the following into account:

- Use `mfrow` in a call to `par` to display the two perspective plots next to each other. In the same call to `par`, override the default figure margins to have only one line of space on each side (recall that `par` is explored in these roles in Sections 23.1.4 and 23.2.2).
- Use `zlim` to ensure both plots are displayed on the same scale of vertical axis, spin each one horizontally 25 degrees, and ensure detailed axis markings and tidy titles.
- Is there any visual indication that the presence of the interaction term has had any meaningful impact on modeling the response?

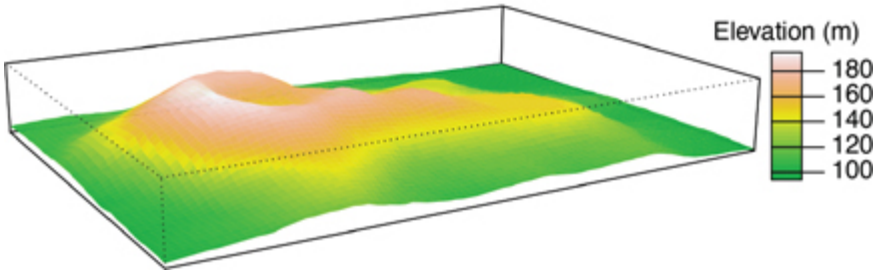
b. Start a fresh plot. To get a better idea of the difference between the two surfaces, produce a perspective plot of the z-matrix obtained as the elementwise difference between the two individual z-matrices for the two fitted models in (a). What, in general, is the effect of including the interaction term?

Return your attention to the *Maungawhau* topographical information, stored as the built-in R object `volcano`: an 87×61 matrix of elevation values (in meters). You first looked at this in Section 25.4.1 as a contour plot.

a. Produce the most basic, default-appearance perspective plot of the volcano, using simple integer sequences for the x- and y-coordinates.

b. The plot in (c) is decidedly unappealing for a number of reasons. Produce a more realistic depiction of the volcano as per the following:

- Use a new graphics device with the margin widths reset to one, one, one, and seven lines on the bottom, left, top, and right, respectively.
- The help file `?volcano` reveals the x- and y-coordinates to which the volcano z-matrix corresponds are in 10-meter units. Using `scale` and altering `expand`, replot the surface with the correct aspect ratio in all three axes.
- Suppress all axis tick marks and notation using `axes`.
- The facets should be colored according to 50 colors generated from the built-in `terrain.colors` palette, and the facet border lines should be suppressed (follow the details of the example in Section 25.6.2).
- Find your choice of visually appealing viewing angle.
- Use `colorlegend` from the `shape` package to place a color legend referencing elevation in meters in the space to the right of the plot. Given it's dependent on the specific size of your graphics device, experiment with the arguments to find appropriate placement and tick mark labels.
  Here's my version of the improved plot:



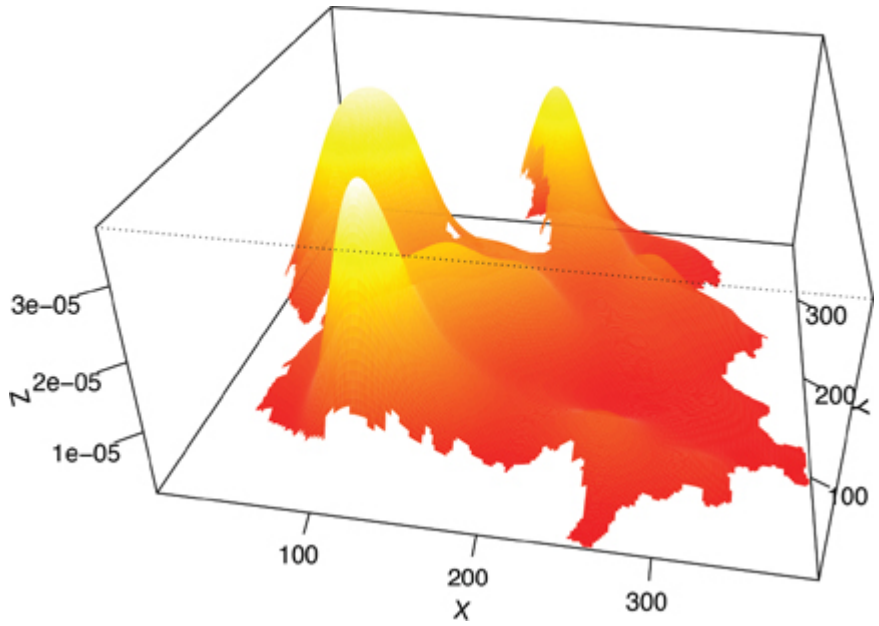
In Exercise 25.4, you looked at the spatial distribution of intentionally lit fires in a region of Spain. Ensure the `spatstat` package is loaded, and then re-run the following lines to obtain the relevant data objects:

```
R> fire <- split(clmfires)$intentional
R> firewin <- clmfires$window
```

e. Borrow the code from Exercise 25.4 (d) and (e) to reproduce the kernel density estimate of this dispersion of observations, based on a 256×256 evaluation grid, truncated to the study region. Then, display it as a perspective plot according to the following:

- Just as with the pixel image, use 50 colors from the built-in `heat.colors` palette to color the facets by z-value. Note the truncated z-matrix for this function contains `NA` values; refer to the code given for Example 4 in Section 25.5.2 (starting on page 752) for guidance.
- Border lines on the surface should be suppressed, and you should find your preferred choice of viewing angle.
- Use `scale` to ensure the correct spatial aspect ratio. In doing so, you'll also need to adjust the z-axis expansion by a factor of around 5,000,000 for the density surface to be visible along the vertical, given the natural scaling of the density estimate on the specified evaluation grid.
- Employ detailed axis labeling and simply entitle the axes `"X"`, `"Y"`, and `"Z"` as appropriate.

My result is given on the next page.



f. Use the `persprot` function defined in Section 25.6.3 to view the surface from (e), setting `skip=20` and `sleep=0.1`. (You may wish to experiment with a satisfactory value for the `sleep` argument.)

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`palette`	List integer colors	Section 25.1.1, p. 710
`colors`	Built-in R color names	Section 25.1.1, p. 710
`col2rgb`	Named color to RGB	Section 25.1.1, p. 711
`Rgb`	RGB to hex code	Section 25.1.1, p. 712

Function/operator	Brief description	First occurrence
`hcl.colors`, `rainbow`, `heat.colors`, `terrain.colors`, `topo.colors`, `cm.colors`, `gray.colors`, `gray`	Built-in palettes	Section 25.1.2, p. 714
`colorRampPalette`	Custom palette (integer)	Section 25.1.3, p. 715
`colorRamp`	Custom palette ([0,1] interval)	Section 25.1.4, p. 719
`colorlegend`	Color legend (`shape`)	Section 25.1.5, p. 720
`scatterplot3d`	3D scatterplot (`scatterplot3d`)	Section 25.2.1, p. 729
`expand.grid`	All unique evaluation coordinates	Section 25.3.1, p. 734
`letters`	Alphabet characters	Section 25.3.1, p. 734
`contour`	Contour plot	Section 25.4.1, p. 737
`kde2D`	Bivariate KDE (`MASS`)	Section 25.4.1, p. 740
`filled.contour`	Color-filled contour plot	Section 25.4.2, p. 744
`image`	Pixel images	Section 25.5.1, p. 749
`inside.owin`	Test inside region (`spatstat`)	Section 25.5.2, p. 756

Function/operator	Brief description	First occurrence
`persp`	Perspective plot	Section 25.6.1, p. 761

26
INTERACTIVE 3D PLOTS



It's important to be able to view 3D plots from different angles to interpret the function or surface of interest. The `rgl` package by Murdoch & Adler (2024) offers some fantastic, simple-to-use R functions that allow you to rotate and zoom in on three-dimensional plots with your mouse. In this chapter, you'll look at a few examples that show off the possibilities of `rgl`, from relatively simple 3D scatterplots to interactive perspective plots and trivariate surfaces. Under the hood, `rgl` utilizes OpenGL—a standard application programming interface—to render the graphics on your computer screen. Install `rgl` (for example, by calling `install.packages("rgl")` at the prompt) and call `library("rgl")` to load it.

26.1 Point Clouds

Let's begin with the most basic of 3D plots: point clouds. In statistics, these typically take the form of scatterplots of three continuous variables, as you saw when you created static 3D scatterplots in the previous chapter.

For the remainder of this section, we'll work with the built-in `iris` data, first seen in Section 14.6; recall this dataset comprises four measurements taken on three species of flower. For accessibility, store the following objects (as done previously in Section 25.2.1):

```
R> pwid <- iris$Petal.Width
R> plen <- iris$Petal.Length
R> swid <- iris$Sepal.Width
R> slen <- iris$Sepal.Length
```

We'll use these in the ensuing examples.

26.1.1 Basic 3D Cloud: Points and Spheres

You use the `plot3d` function of `rgl` to display an interactive 3D cloud of points. It's called by supplying the x-, y-, and z-coordinates to the `x`, `y`, and `z` arguments, respectively. The following line opens an RGL device and produces a default-appearance scatterplot of petal width, length, and sepal width from the `iris` data:

```
R> plot3d(x=pwid,y=plen,z=swid)
```

To get a better view of the data, increase the size of the device window with your mouse. Then, by left-clicking the plot and holding the button down, you can move the mouse to rotate the plot in any direction you like. If you hover your cursor over the plot and scroll up or down, you can zoom in or out (this effect can also be achieved by holding a right-click and moving your mouse up or down). The axis tick marks and their labels automatically appear on different sides based on your viewing angle.

To tidy up the axis and main titles, use the `main`, `xlab`, `ylab`, and `zlab` arguments. For example, the following adds more useful axis titles than the default plot provides:

```
R> plot3d(x=pwid,y=plen,z=swid,xlab="Petal width",ylab="Petal length",
          zlab="Sepal width")
```

Figure 26-1 shows a screenshot of the result.

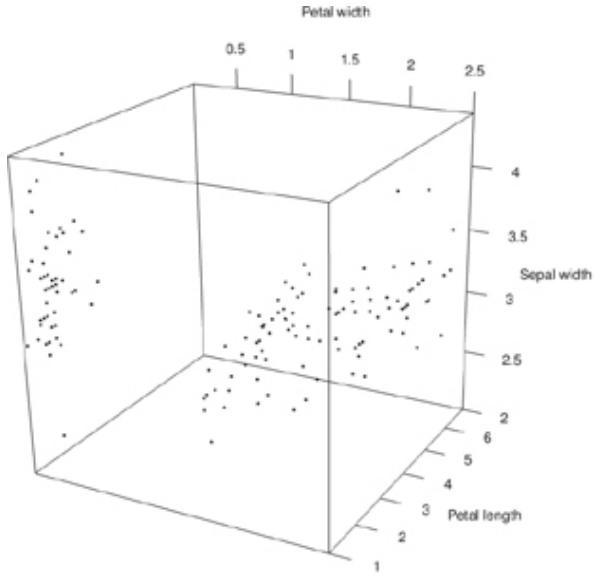


Figure 26-1: An interactive 3D scatterplot of the `iris` data using the `plot3d` function of `rgl`

The `plot3d` function also allows you to render points as spheres, which possess a light-reflection effect and are a little prettier than simple dots. To do this, set the optional argument `type="s"` (the default is `type="p"`, for "points"). Figure 26-2 shows a screenshot resulting from the following code:

```
R> plot3d(x=pwid,y=plen,z=swid,type="s",radius=0.1,
          xlab="Petal width",ylab="Petal length",zlab="Sepal width")
```

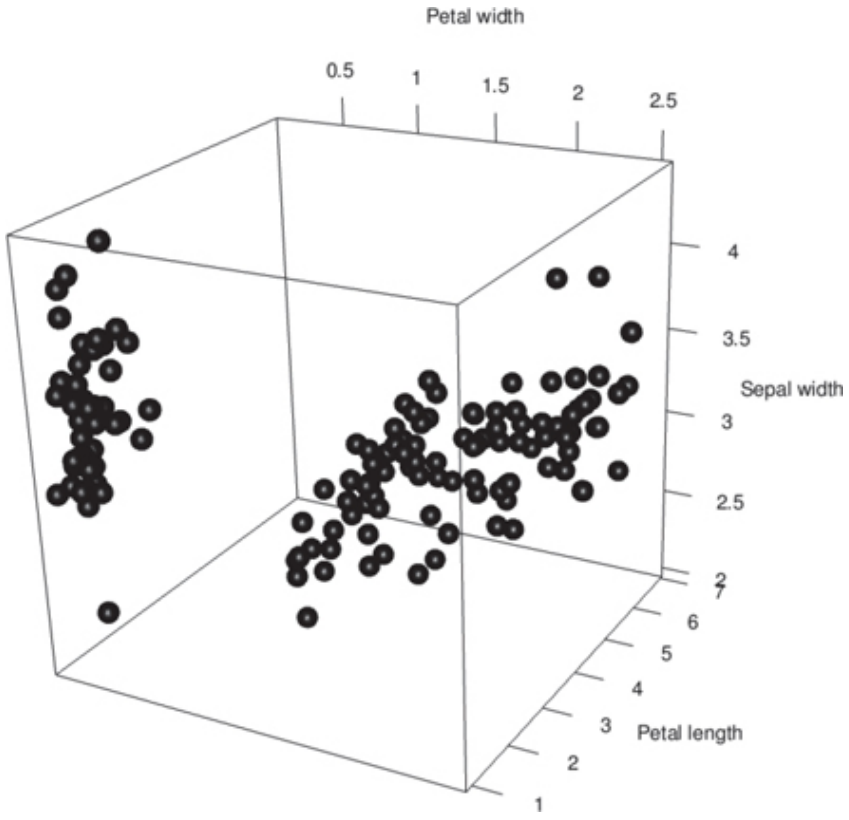


Figure 26-2: Another 3D scatterplot of the `iris` data using `type="s"` to plot spheres instead of the default points

The `radius=0.1` argument scales the size of the spheres down slightly compared to their default size (which is a bit on the large side). The `radius` argument is interpreted as the radius of the spheres roughly on the scale of the observed data. You can also change the size of the spheres (and the default-appearance points) by using the optional `size` argument, the value of which is provided on a scale relative to the size of the plotting region (see the documentation `?plot3d` for further detail). Either way, it sometimes takes a little experimentation to get a pleasing value.

To add more points or spheres to an existing RGL graphic, use the `points3d` and `spheres3d` functions, respectively. Think of these as the `rgl` analogs of the base R `points` function (Section 7.3). You'll experiment with `points3d` in Exercise 26.1 and see it in action in Section 26.2.

26.1.2 Color and Legends

You approach simple aesthetic enhancements like color similarly to nonRGL graphics; for example, use the `col` argument to specify color. To

illustrate, let's replot the `iris` data and use color to distinguish between the three different species. First, close any currently open RGL graphics devices. Then, execute the following line; resize the graphics device to your liking, and keep the plot open:

```
R> plot3d(x=pwid,y=plen,z=swid,type="s",radius=0.08,
          col=as.numeric(iris$Species),xlab="Petal width",
          ylab="Petal length",zlab="Sepal width",
          main="Iris Flower Measurements")
```

This will start a new RGL device with a plot that colors spheres according to numeric representation of the flower species factor (1, black, for *setosa*; 2, red, for *versicolor*; and 3, green, for *virginica*). Just like the base R `plot` function for 2D scatterplots, passing the `col` argument a vector of the same length as the plotted coordinates will assign the specified color to the corresponding point in an element-wise fashion. Here, I also reduced the `radius` value to 0.08 units (resulting in spheres slightly smaller than those in Figure 26-2) and included a main plot title via `main`.

The plot produced by the most recent code is devoid of a useful legend referencing the imparted color. You can remedy this with the `legend3d` function, which changes the background image upon which the interactive plot sits. Assuming the scatterplot from the previous line remains open, execute the following line:

```
R> legend3d("topright",col=1:3,legend=levels(iris$Species),pch=16,cex=2)
```

This inserts a static, unmovable legend referencing the plotted species by color. The `legend3d` function actually calls the base R `legend` function, so they are conveniently used in the same way. With the static legend in place, the scatterplot remains fully interactive, and you can continue to rotate and zoom. Figure 26-3 shows a screenshot of the result.

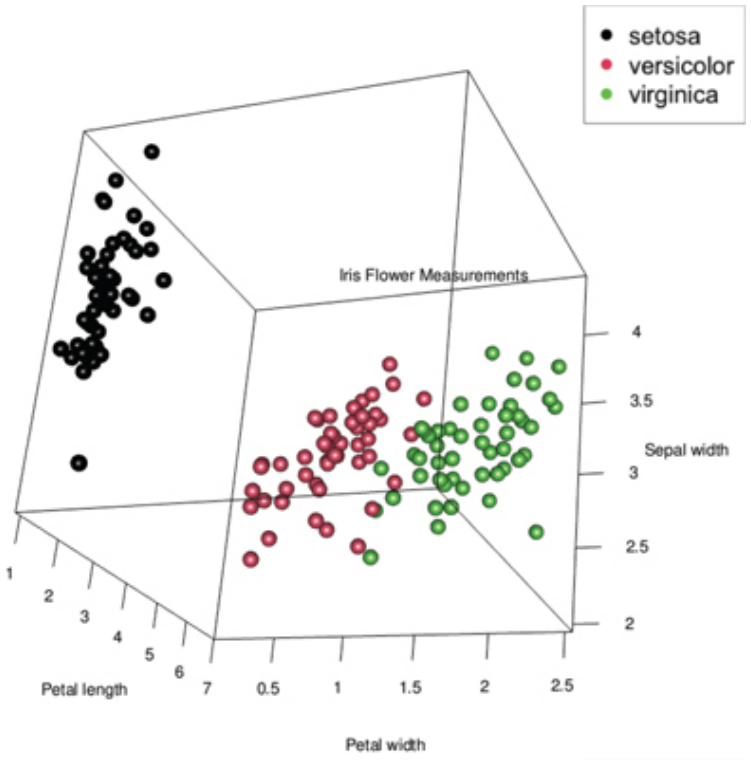


Figure 26-3: Replotting the `iris` petal width, length, and sepal width data with `plot3d`. Observations are plotted as circles and colored according to species type, and a legend is added via `legend3d`.

Given the `legend3d` function changes the background canvas, it is important to open a new device, produce your main plot, and get your device into a preferred size manually before you add the legend. If you now produce a new `rgl` plot in the same device without closing it first or resetting the background, the flower species legend will still be there; if you resize your device using your mouse after adding a legend, it may appear stretched or squashed.

If you're making multiple `rgl` plots consecutively in the same device, you can reset the background to its default white canvas at any time by calling the following:

```
R> bg3d(color="white")
```

Executing this line of code with the most recent plot still active will cause the flower species legend to disappear while the scatterplot remains. Alternatively, you can just close your RGL device when you're done so that a new device will be used for any subsequent plot.

26.1.3 Alternate 3D Point Characters

When using base R to produce simple 2D coordinate–based plots, recall that you can choose different point characters via the `pch` argument (refer back to the center plot of Figure 7-5 on page 151). The `rgl` package offers an approximation of the 25 available 2D point characters for use in 3D RGL graphics, allowing you to mimic base R aesthetics in this regard.

This capability is provided by the `pch03d` function. This is a stand-alone function that can either start a new RGL plot or be called after some initial execution—for example, of `plot3d`—to add to a current RGL graphic. You use its `pch` argument in the same way as you would the argument of the same name in default, base R 2D plots.

Start another version of the iris scatterplot with the following line:

```
R> plot3d(x=pwid,y=plen,z=swid,type="n",xlab="Petal width",ylab="Petal length",
         zlab="Sepal width",main="Iris Flower Measurements")
```

You'll see an "empty" plotting region pop up, devoid of points. This is due to the specification `type="n"`, which suppresses all initial plotting but still draws the axes and annotations as specified. Use `pch03d` to add the observations; note we're now using different point characters to distinguish between the species:

```
R> pch03d(x=pwid,y=plen,z=swid,pch=c(2,5,8)[as.numeric(iris$Species)],
        radius=0.1)
```

Here, the numeric representation of the `Species` factor is used to index the vector `c(2,5,8)`, which in turn means point character 2 will be associated with *setosa*, 5 with *versicolor*, and 8 with *virginica*. Notice that the `radius` argument specifies the size of the characters in the same way it scaled the spheres in Section 26.1.1.

Because the `rgl` point characters in `pch03d` are designed to mirror the 25 options in base R, it's easy to add a legend referencing these three character types (similar to how we referenced color in Section 26.1.2):

```
R> legend3d("topright",pch=c(2,5,8),legend=levels(iris$Species),cex=2)
```

Figure 26-4 shows a screenshot of the result. You'll see all 25 symbols in a plot we'll create momentarily in Section 26.1.5.

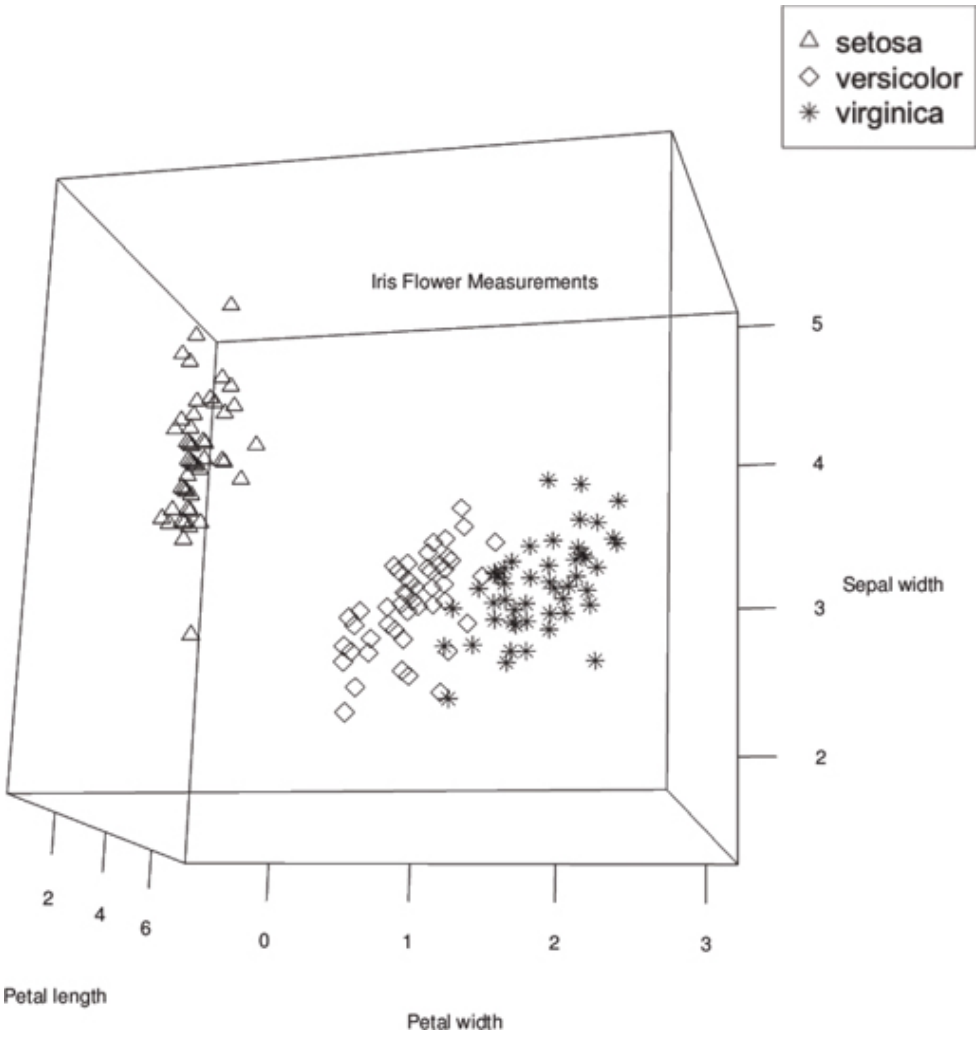


Figure 26-4: Illustrating `pch03d` *for plotting different point characters in an RGL graphic using the* `iris` *data*

26.1.4 3D Text

In a similar vein to drawing different point characters, use the `text3d` function to include what is effectively 2D text within an RGL graphic, reminiscent of the base R `text` function (first seen in Section 7.3).

Close any open RGL devices and, remaining with the `iris` data objects `pwid`, `plen`, and `swid` defined at the beginning of the chapter, start a new plot of the measurements by re-running the same line as before, keeping the plot open:

```
R> plot3d(x=pwid,y=plen,z=swid,type="n",xlab="Petal width",ylab="Petal length",
          zlab="Sepal width",main="Iris Flower Measurements")
```

Consider now the following code, whereby we use `substr` (Section 4.2.4) to abbreviate the species labels to the first three characters and supply these shortened strings directly to the `texts` argument of `text3d`, adding to the previously opened plot:

```
R> spec <- substr(iris$Species,1,3)
R> text3d(x=pwid,y=plen,z=swid,texts=specs)
```

This use of text obviates the need for a legend, given the three species are readily identified by their abbreviations. Figure 26-5 shows a screenshot of the result.

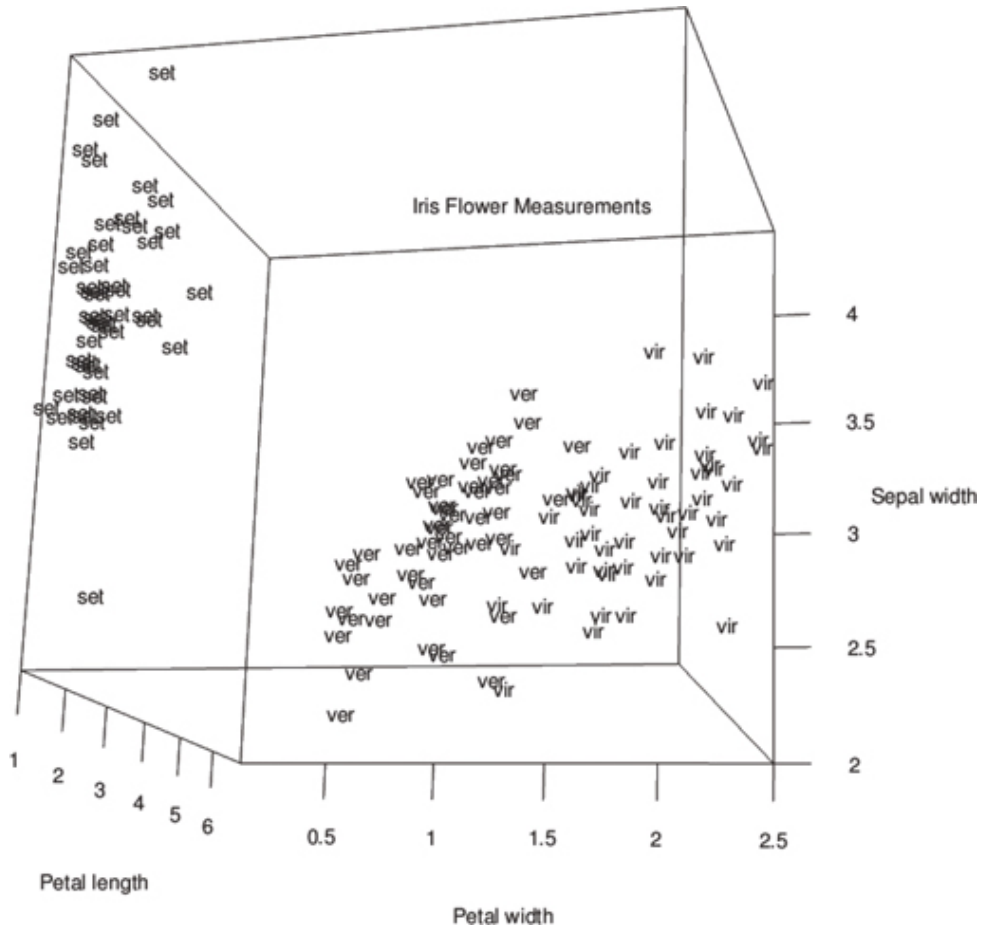


Figure 26-5: Illustrating `text3d` for plotting text in an RGL graphic using the `iris` data

26.1.5 3D Lines, Axes, Grids, and Other Annotations

There are many other `rgl` functions designed to mirror base R functionality when it comes to adding visual features to a 3D graphic. Place plotted lines

with `lines3d` (reminiscent of `lines`) and `segments3d` (the counterpart to `segments`). You can add 3D axes and boxes with `axis3d`, `axes3d`, or `box3d` (mirroring `axis` and `box`); titles and margin text are possible via `title3d` and `mtext3d` (like `title` and `mtext`, respectively); and `grid3d` superimposes grid lines (serving much the same purpose as `grid`).

Let's look at some of these functions by creating a plot similar to the set of plots from Figure 7-5 on page 151, which illustrated the 25 types of point characters in base R graphics. Close any open RGL graphics devices so that the following line creates a new plot (we'll build on this progressively, so keep it open):

```
R> pch03d(x=1:25,y=1:25,z=1:25,pch=1:25,radius=0.75)
```

In Section 26.1.3, you used `pch03d` to add to an existing RGL graphic. When called anew, this command will open a new RGL device and place 25 points (each one corresponding to the uniquely numbered point character via `pch`) along a perfect diagonal line at (x, y, z) integer coordinates from (1, 1, 1) to (25, 25, 25), but no axes will appear.

To resolve this, use `axes3d` to place axis planes along the three dimensions by simply calling the following:

```
R> axes3d()
```

There are various options for controlling axis tick placement and labels (see the documentation at `?axes3d`); the defaults are fine here. Notice that the plot is open-faced in that no overall box has been rendered along all sides. You can execute `box3d()` to add one; I'll leave that out in this example.

Suppose now you want to plot the diagonal line upon which the 25 different point characters sit. A simple call to `lines3d` achieves this; the following call supplies the `x`, `y`, and `z` arguments with the starting and ending x-, y-, and z-coordinates of the desired line, respectively. The line is made slightly thicker than default, and given an attractive sea-green finish:

```
R> lines3d(x=c(1,25),y=c(1,25),z=c(1,25),col="seagreen4",lwd=3)
```

To emphasize the height of each point, let's add some vertical line segments from the bottom reaching up to touch each one by using `segments3d`. Unlike its base R counterpart, `segments3d` doesn't separate the

"from" and "to" coordinates into different arguments (recall the use of `x0`, `y0`, `x1`, and `y1` in `segments` and `arrows` in Section 7.3). Instead, it takes each sequential pair of observations provided to the `x`, `y`, and `z` arguments to be the beginning and end of each line segment, in that order. So, to draw the desired vertical lines on the existing RGL device, you first need to set up these vectors containing consecutive pairs of "from" and "to" locations in the 3D space, for each axis. To that end, consider the following objects:

```
R> xseg <- rep(1:25,each=2)
R> xseg
[1]  1  1  2  2  3  3  4  4  --snip--
R> yseg <- rep(1:25,each=2)
R> yseg
[1]  1  1  2  2  3  3  4  4  --snip--
R> zseg <- rep(1:25,each=2)*c(0,1)
R> zseg
[1]  0  1  0  2  0  3  0  4  --snip--
```

The `rep` command (Section 2.3.2) is used with the `each` argument to replicate each entry of the integer vector `1:25` twice. Both `xseg` and `yseg` are identical, with repeated pairs of values. In the context of `segments3d`, this means neither the starting nor ending position of each segment is going to change in terms of its x-axis or y-axis value. This makes sense because the intention is to draw perfectly vertical lines: The only coordinate that will change from the start to the end of a given segment is the z-axis position. With that in mind, look at `zseg`. This object begins in the same way as `xseg` and `yseg` (in terms of the use of `rep`), but it is subsequently multiplied by the length-2 vector `c(0,1)`. Vector recycling takes effect (refer to Section 2.3.4), meaning the first value of each consecutive pair will be forced to be 0, with the second being merely multiplied by 1 (and therefore remaining unchanged).

Interpreted as collective pairs of values along the x-, y-, and z-directions (look again at the content of the `xseg`, `yseg`, and `zseg` objects), we can now see how `segments3d` will render the segments. The first (x, y, z) line segment will be drawn from (1, 1, 0) to (1, 1, 1), the second from (2, 2, 0) to (2, 2, 2), the third from (3, 3, 0) to (3, 3, 3), and so on. Call the following line to add these segments to the ongoing plot, wherein we also make use of the built-in `rainbow` palette (Section 25.1.2) to add further flair:

```
R> segments3d(x=xseg,y=yseg,z=zseg,col=rainbow(25))
```

Now add a reference grid to the bottom-most plane with the `grid3d` function; its first argument should be a character string of two characters that specifies the precise axis plane upon which to draw (it can also be a vector of length-2 character strings). In the following call, we instruct the function to draw the grid on the lower z-axis plane with the argument `side="z-"` and the optional argument `at=1:25` stipulates the grid is to be drawn at all integer values from 1 through to 25:

```
R> grid3d(side="z-",at=1:25)
```

You can experiment with the `side` argument to see how it is interpreted. The minus designation can alternately be a plus if you wish the grid to be drawn on the upper plane of the specified axis; for example, try using `side="y+"` to place a grid on the upper end of the y-axis. You might also use the `col` argument to change the color of the grid lines to something other than the default gray.

A plot this decisive deserves a title. As a finishing touch, use `title3d` to describe your conquest:

```
R> title3d(main="Veni, vidi, pch03d")
```

Figure 26-6 provides a screenshot of the aftermath.

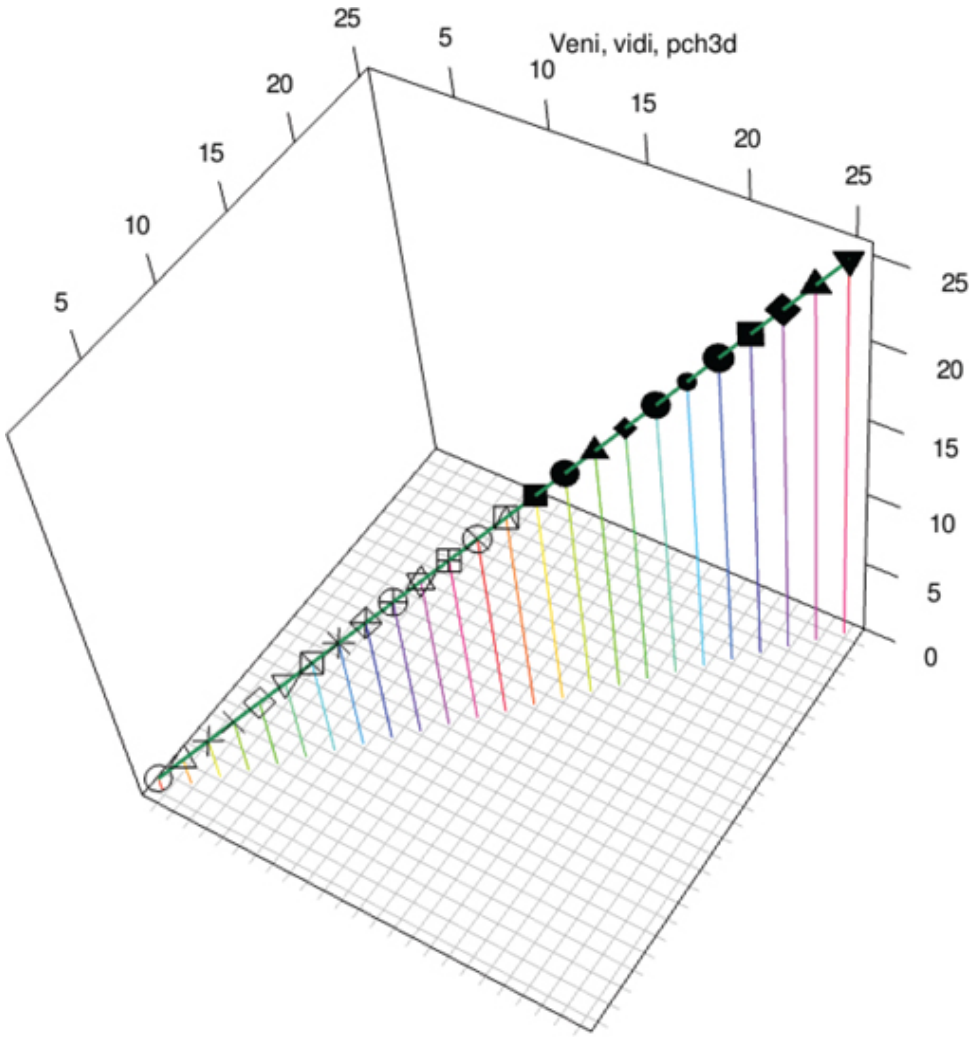


Figure 26-6: Illustrating the effect of various `rgl` functions (`axes3d`, `lines3d`, `segments3d`, `grid3d`, and `title3d`) in showcasing the 25 different point characters available via `pch03d`

Other available drawing tools unique to `rgl` include `triangles3d` and `quads3d`. Think of these as extensions to `segments3d`: Where the latter draws line segments based on pairs of coordinates, `triangles3d` draws triangles based on triplets of 3D coordinates, and `quads3d` draws quadrilaterals based on quadruplets. We won't explore these functions here, but you can do so by typing `?triangles3d` at the R prompt and running the examples at the bottom of the help file. We will, however, focus attention on `persp3d` for `rgl`-based perspective plots come Section 26.2.

26.1.6 Extended Example: A Custom, Interactive 3D Scatterplot

Take a look back at Figure 25-8 on page 731, wherein you used `scatterplot3d` (the single function in a package of the same name) to create a static 3D scatterplot of petal width, petal length, and sepal width, which utilized color to reference the fourth measurement variable present in the `iris` dataset: sepal length. Let's reproduce this plot using `rgl` functionality so that we have an interactive version. You'll need the four objects `pwid`, `plen`, `swid`, and `slen` as created at the beginning of this chapter (Section 26.1) in what follows.

First, re-create the custom palette used in the earlier static plot to set up the colors corresponding to each sepal length observation in `slen`. For simplicity, I've done so with a categorization of 50 colors (see Section 25.1.4), as opposed to the normalization approach used previously:

```
R> slen.pal <- colorRampPalette(c("purple","yellow2","blue"))
R> cols <- slen.pal(50)
R> slen.cols <- cut(slen,breaks=seq(min(slen),max(slen),length=51),
                    include.lowest=TRUE)
```

Then, either close any currently active RGL devices or clear the background of the one in focus by calling `bg3d(color="white")`. The following call to `plot3d` starts the plot with appropriately colored sepal length (`slen`) spheres placed at petal width (`pwid`), petal length (`plen`), and sepal width (`swid`) coordinates on the x-, y-, and z-axes, respectively:

```
R> plot3d(x=pwid,y=plen,z=swid,type="s",radius=0.1,col=cols[slen.cols],
          aspect=c(1,1.75,1),xlab="Petal width",ylab="Petal length",
          zlab="Sepal width")
```

Recall from Section 7.2.6 the effect of locking the aspect ratio in 2D data plots. This idea extends naturally to 3D plots; we use the `aspect` argument in `plot3d` to control it. The `aspect` argument takes a vector of length 3 describing the relative lengths of the x-, y-, and z-axes, in that order. Here, by changing the second entry to `1.75`, you're lengthening the y-axis by that multiplicative factor relative to the others. This creates the stretched-out effect along the y-axis, which is entirely optional and done purely for the purpose of making the plot more closely resemble the static version produced by `scatterplot3d` in Section 25.2.2.

The colors are assigned by vector indexing using the `slen.cols` factor, and `xlab`, `ylab`, and `zlab` tidy up the axis titles. To add a vertical line from the x-y

plane to each observation, you can use the `segments3d` function, as outlined in the creation of Figure 26-6 in Section 26.1.5. Recall `segments3d` interprets each sequential pair of observations provided to the `x`, `y`, and `z` arguments to be the beginning and end of each line segment, in that order. So you first need to set up these vectors to contain a "from" location and a "to" location in the 3D space. Consider the following code:

```
R> xseg <- rep(pwid,each=2)
R> yseg <- rep(plen,each=2)
R> zseg <- rep(swid,each=2)
R> zseg[seq(1,length(zseg),by=2)] <- min(swid)
```

The first two lines set up the vectors for the x- and y-components, `xseg` and `yseg`, respectively, by simply replicating each observation twice. Just as in the previous example, these are easy, since the "from-to" values don't change in the x- or y-coordinate directions. The z-component does change, however. You first create the `zseg` vector by replicating each sepal width value twice. Every second entry of this vector, starting from the second index position (2, 4, 6, and so on), will be your "to" values, with the "from" values taking the index positions 1, 3, 5, . . . , and which are all set to be the minimum recorded sepal width value. Achieve this last step by appropriately indexing the initially created `zseg` object using the square-bracket operator (refer back to Section 2.3.3) and `seq` (Section 2.3.2), filling the specified slots with `min(swid)`.

Together with `xseg` and `yseg`, you'll end up with lines that go from the bottom x-y plane of the plot at `min(swid)` up to the actual `swid` value (the corresponding *z* value of each sphere). A call to `segments3d` places the lines on the plot; to ensure the color of each line matches its corresponding sphere, you also need to replicate each entry of the vector-indexed collection of colors provided by `cols[slen.cols]` twice, which results in a constant "from-to" color:

```
R> segments3d(x=xseg,y=yseg,z=zseg,col=rep(cols[slen.cols],each=2))
```

Next, executing the following line places a reference grid over the lower x-y plane; unlike the use of `grid3d` in the previous example, here we simply leave the grid lines to be drawn at default values:

```
R> grid3d(side="z-")
```

Lastly, you can add a custom, continuous-color legend to the plot to reference the sepal length. The `bgplot3d` function is a more general version of `legend3d`; it allows you to specify any plotting commands you like to define the RGL device background. Let's do so using the `colorlegend` function of the `shape` package; its usage is detailed in Section 25.1.5. Make sure you have the `shape` package loaded and that your RGL device of the scatterplot is sized to your liking. On my machine, I execute the following:

```
R> bgplot3d({plot.new();colorlegend(slen.pal(50),zlim=range(slen),
                                    zval=seq(4.5,7.5,0.5),digit=1,
                                    posx=c(0.91,0.93),posy=c(0.1,0.9),
                                    main="Sepal length")})
```

The `bgplot3d` function can take multiple plotting commands, which you need to provide as a code chunk within braces (`{ }`), with each command or function call separated by a semicolon (;). In this example, the initial call of `plot.new()` silently initializes the background of the RGL device so that you can add the continuous-color legend, prior to execution of `colorlegend` to draw the legend itself (without that call to `plot.new`, `colorlegend` will still work, but a warning will be issued). Figure 26-7 shows the final result, with the scatterplot still rotatable and zoomable with your mouse.

The ability to rotate a 3D scatterplot, and indeed any of the plots you'll see over the next few sections, with simple mouse commands is especially handy when exploring visuals of higher-dimensional data. You're not restricted to a single viewpoint, and you don't need to manually decide on a viewing angle before actually producing the plot. As you've seen, `rgl` functionality also makes it easy to add extra elements to an existing plot, something that's harder to do with `scatterplot3d` or base R perspective plots.

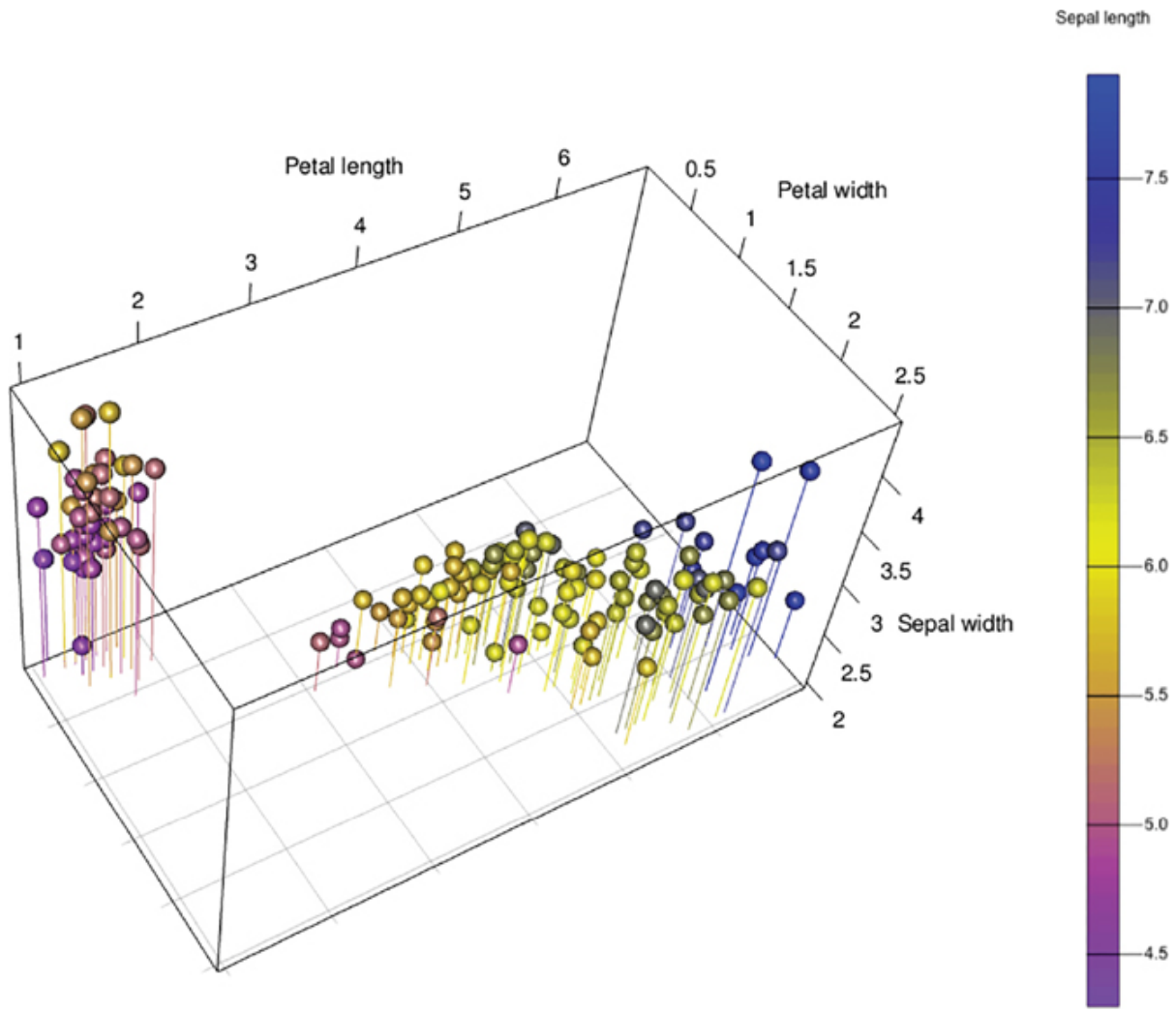


Figure 26-7: An `rgl`-based 3D scatterplot designed to mimic the earlier `scatterplot3d` example, showing the four measurement variables of the `iris` data.

Exercise 26.1

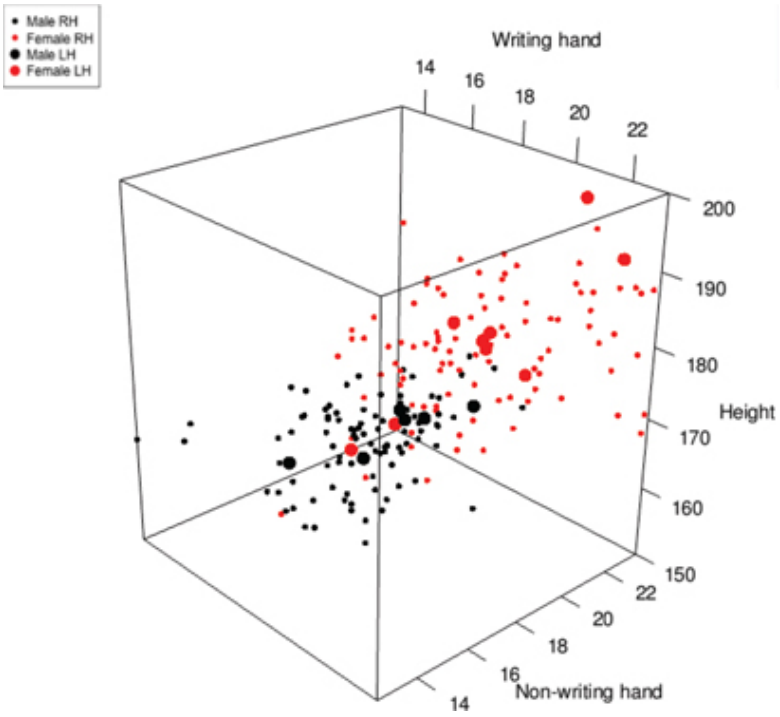
Turn back to the `survey` data frame in the `MASS` package, checking the description of the present variables in the help file `?survey` if necessary. Create a copy of `survey` containing the writing handspan, non-writing handspan, left- or right-handedness, sex, and height columns. Then use `na.omit` to remove any rows of this subsetted data frame that contain missing values (for an example of this, look back at the creation of the `surv` object in Section 25.1.4).

a. Produce a basic interactive 3D point cloud of student height on the z-axis, writing handspan on the x-axis, and nonwriting handspan on the y-axis.

b. Create a more informative version of the scatterplot in (a) that uses color to distinguish between sexes and uses point size to distinguish between left- and right-handed individuals, following these guidelines:

   - Start by plotting only those points that correspond to right-handed individuals. Set the color via vector indexing using the numeric version of the sex factor for right-handed individuals: Females should be black, males red.
   - Set `size=4` for the right-handed individuals and ensure tidy axis labels.
   - Use `points3d` to add the points for left-handed individuals to the existing plot. Note that `points3d` is used in much the same way as `plot3d`, though by default it adds points to the currently active RGL graphic. Colors are to be assigned according to sex in the same way as for the right-handed students; use `size=10`.
   - Ensure the RGL device is manually sized to your liking and add a legend to the top-left corner that references the four types of points: `"Male RH"`, `"Female RH"`, `"Male LH"`, and `"Female LH"`. In setting the legend, use a `pch` value of `19` and use `pt.cex` values of `0.8` and `1.5` for right- and left-handed individuals, respectively.

   For reference, my version of the rotatable 3D scatterplot follows.



In Exercise 25.2 (b), you used `scatterplot3d` to produce a static 3D scatterplot of the built-in `airquality` data. Create a copy of the data frame omitting any rows with `NA` entries by running the following line:

```
R> aq <- na.omit(airquality)
```

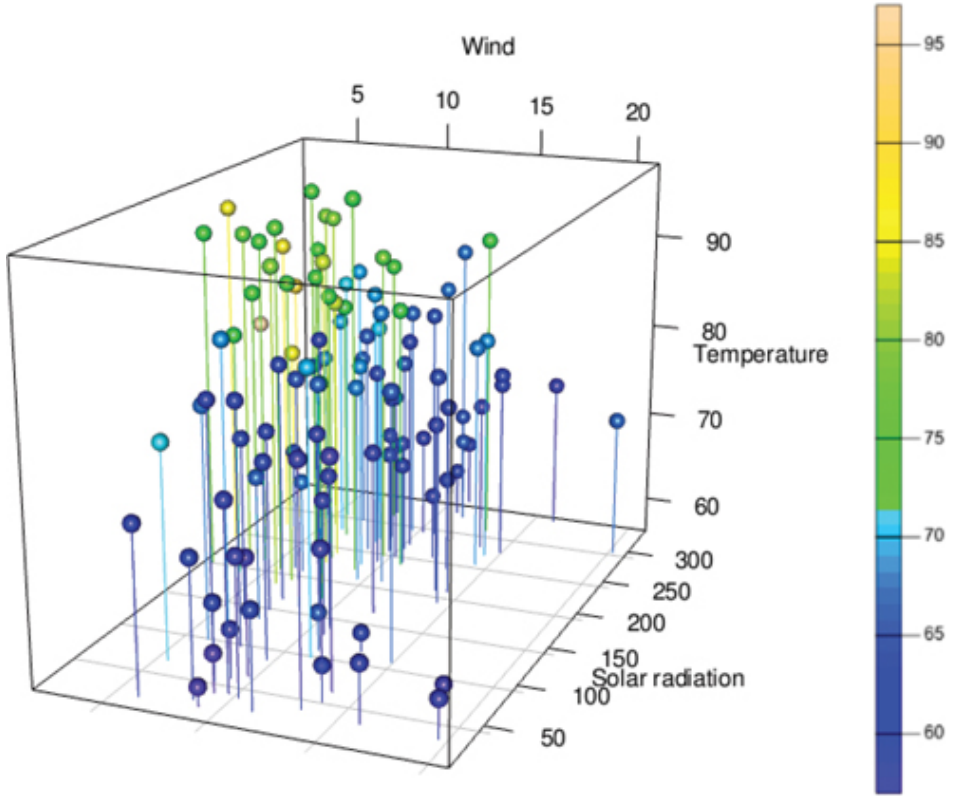
Use the `aq` object for the next task.

c. Create a similar version of the plot from the earlier exercise using `rgl` functionality, displaying wind speed, solar radiation, and temperature on the x-, y-, and z-axes, respectively, according to the following guidelines:

- Set up 50 colors from the built-in `topo.colors` palette. Set up the appropriate color index vector for the ozone values based on the categorization approach (Section 25.1.4).
- Plot the observations as `radius=4` spheres, colored corresponding to ozone value, and modify the aspect ratio so that the y-axis is 1.5 times the length of the other two axes. Provide neat axis titles.

- Use `segments3d` in the same way as in Section 26.1.5 to add correspondingly colored lines, one for each observation, stretching vertically upward from the x-y plane to meet the plotted spheres. Also, place a grid on the lower x-y plane.
- Modify the background of the RGL device, using `colorlegend` from the `shape` package to include a color legend referencing the ozone level; use a sequence of values between 60 and 95, in steps of 5, to label it.

Here's what my result looks like:



26.2 Bivariate Surfaces

Next, you'll look at plotting bivariate surfaces, which are continuous surfaces calculated with respect to a 2D x-y evaluation grid, with `rgl`. You plotted these using base R graphics with the `contour`, `filled .contour`, `image`, and

`persp` commands in Chapter 25. Anything that you can plot using those functions can also be plotted as an interactive perspective plot with the `persp3d` function of `rgl`.

26.2.1 Plotting Basic Perspective Surfaces

As an initial example of plotting bivariate surfaces, take the `mtcars` response surface of mean MPG as a function of horsepower and weight (first investigated in Section 21.5.4 and subsequently used in Chapter 25) as an easy initial example. In Section 25.6.1, you plotted static perspective plots of this surface. The following lines will refit the multiple linear regression model (recall it involves both the main effects and an interactive term between the two predictor variables) and re-create the 20×20 evaluation grid based on the same x- and y-sequences used earlier:

```
R> car.fit <- lm(mpg~hp*wt,data=mtcars)
R> len <- 20
R> hp.seq <- seq(min(mtcars$hp),max(mtcars$hp),length=len)
R> wt.seq <- seq(min(mtcars$wt),max(mtcars$wt),length=len)
R> hp.wt <- expand.grid(hp=hp.seq,wt=wt.seq)
```

To create the surface, predict using the evaluation grid in `hp.wt` as you've done previously, but this time, include the calculation of a 99 percent prediction interval for the raw observations:

```
R> car.pred <- predict(car.fit,newdata=hp.wt,interval="prediction",level=0.99)
```

(You'll use the interval in a later example.)

Then, construct the z-matrix with the same technique used for the creation of the static perspective plot, and draw a green `persp3d` surface with the following two lines:

```
R> car.pred.mat <- matrix(car.pred[,1],nrow=len,ncol=len)
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.mat,col="green")
```

The left-hand side of Figure 26-8 shows a screenshot of the result. Note that it shows the same surface as Figure 25-19 on page 762.

The default lighting and shadowing effect produced by the `persp3d` surface helps with depth perception, similar to the `shade` argument to `persp`.

The main benefit of this version is the mouse-based rotation and zoom interactivity.

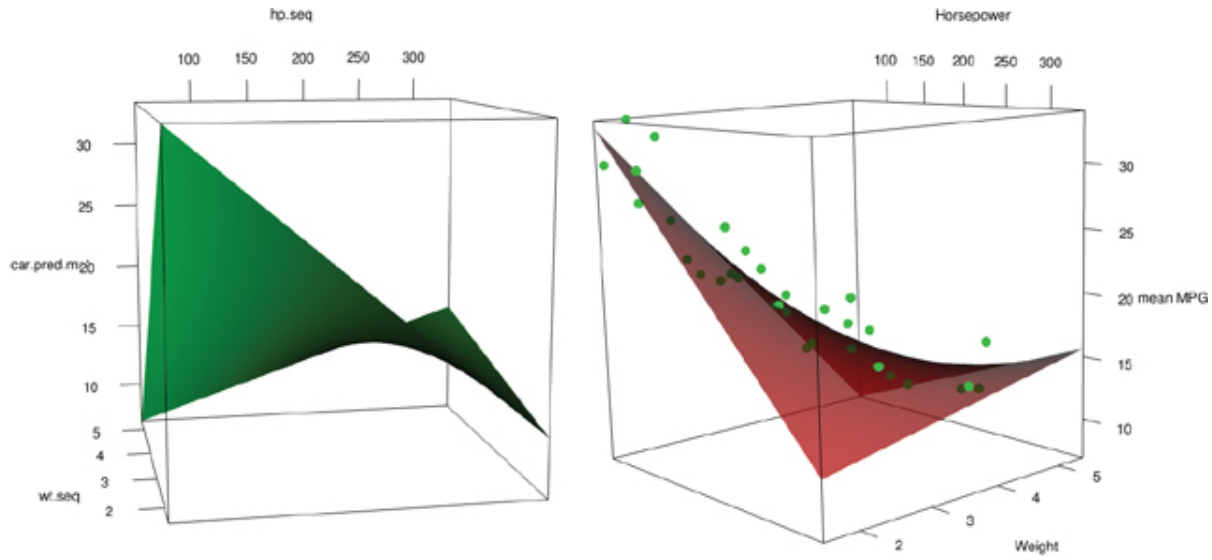


Figure 26-8: Two interactive `persp3d` versions of the `mtcars` response surface. Left: Default appearance, with a green color option. Right: Red, 70 percent opaque surface, with original data superimposed in the 3D space.

26.2.2 Adding Components

Another useful attribute of a `persp3d` plotted surface is the ability to easily add more components, something that's convoluted in base R functionality. We'll continue using the objects created in Section 26.2.1 for the `mtcars` response surface.

Adding Points

As this response surface is based on a model fitted to data on the three variables of horsepower, weight, and MPG, it would be useful to view the raw observations alongside the fitted model. For an `rgl` plot, you can use `points3d` to add points to an existing graphic. Execute the following, resize the RGL device to your liking, and keep the device open:

```
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.mat,col="red",alpha=0.7,
           xlab="Horsepower",ylab="Weight",zlab="mean MPG")
R> points3d(mtcars$hp,mtcars$wt,mtcars$mpg,col="green3",size=10)
```

These two commands plot the predicted mean MPG response surface, this time in red at 70 percent opacity using the optional `alpha` argument, and then add the raw observations to the same image in green, slightly enlarged from their default size. The right-hand side of Figure 26-8 shows a screenshot of this plot. You can now compare the fit of the response surface to the raw data and view it from any angle.

Adding Surfaces

You can also add more perspective surfaces! Let's continue to add to the current plot of the `mtcars` response surface using the `car.pred` object you created for Figure 26-8. Given the use of `predict` in Section 26.2.1, note that the response surface is stored as the first column in `car.pred`; the corresponding lower and upper prediction limits are stored as the second and third columns (flip back to Section 20.4.2 for a discussion of `predict` for linear regression models). To add these prediction bounds to the response surface displayed on the right of Figure 26-8, you first need to store each bounding surface as a z-matrix, both of which will correspond to the same x-y evaluation grid:

```
R> car.pred.lo <- matrix(car.pred[,2],nrow=len,ncol=len)
R> car.pred.up <- matrix(car.pred[,3],nrow=len,ncol=len)
```

Then, keeping the previously created plot of the response surface open, simply call `persp3d` for each of these z-matrices and set the optional `add` argument to `TRUE`, which instructs the `persp3d` function to add to the existing graphic without refreshing the plot:

```
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.up,col="cyan",add=TRUE,alpha=0.5)
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.lo,col="cyan",add=TRUE,alpha=0.5)
```

Here, you've also set the color for each additional surface to cyan and set the opacity to 50 percent. You can see a screenshot of the result on the left of Figure 26-9. After rotating it with your mouse, you'll see that all observations fall between the 3D 99 percent prediction interval bounds for this particular model.

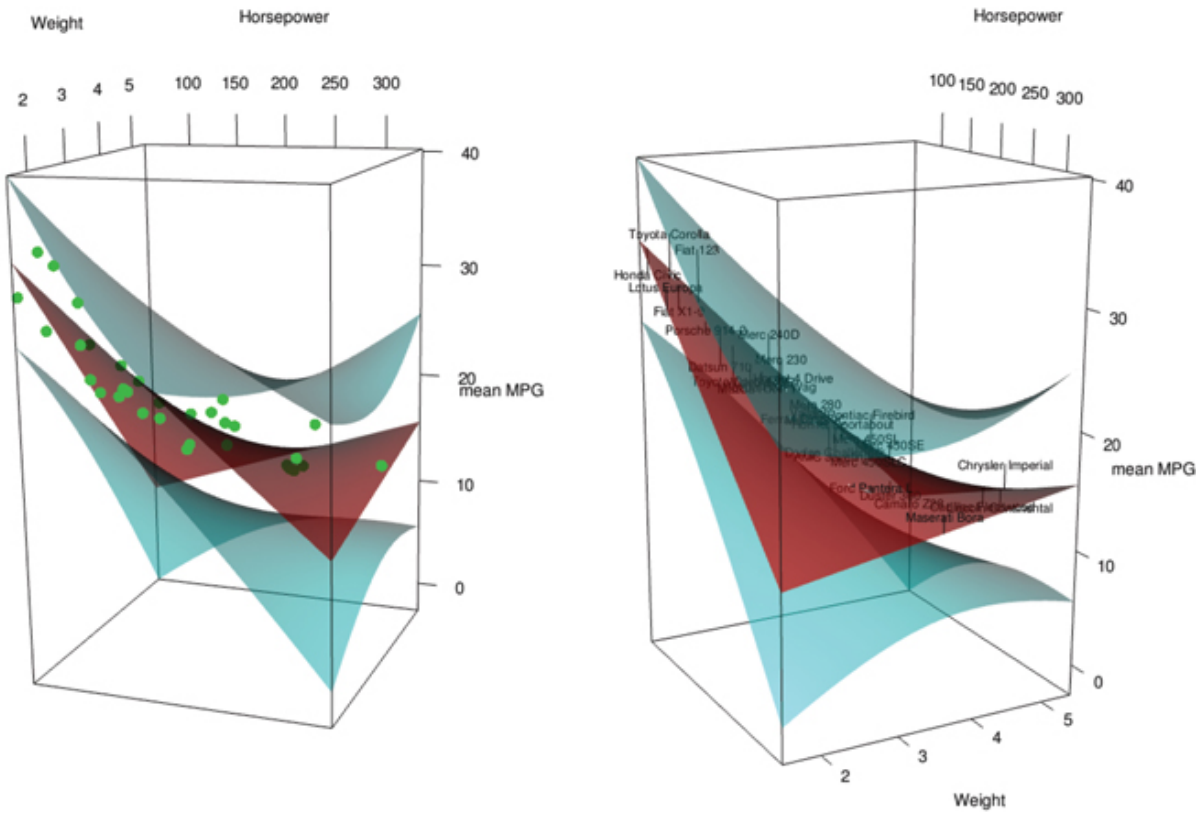


Figure 26-9: Adding further surfaces denoting the 99 percent prediction interval to an existing `persp3d` *plot of the fitted* `mtcars` *model. Left: Green points make up the raw observations. Right: Raw observations are labeled with added text identifying each record, and corresponding line segments mark their residuals.*

Alternatively, you could use `text3d` (illustrated in Section 26.1.4) to label the positions of the raw observations with the row names attribute of the original `mtcars` data frame; this allows you to quickly identify which car is which in the plot itself. In this case, the names are obtained as a vector of character strings using the built-in `rownames` function (Section 5.2.1). Executing the following four lines replots the translucent red response surface of the running example (linear model objects defined in Section 26.2.1), adds the appropriate text at the (x,y,z) coordinates corresponding to that car, and again adds the lower- and upper-bound prediction interval limits as cyan surfaces:

```
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.mat,col="red",alpha=0.7,
           xlab="Horsepower",ylab="Weight",zlab="mean MPG")
R> text3d(x=mtcars$hp,y=mtcars$wt,z=mtcars$mpg,texts=rownames(mtcars),cex=0.75)
```

```
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.up,col="cyan",add=TRUE,alpha=0.5)
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.lo,col="cyan",add=TRUE,alpha=0.5)
```

The specific coordinate associated with each observation is harder to visually discern with text as opposed to the originally plotted green dots, so it makes sense to point their locations out on the fitted surface—and what better way to do so than by using the fitted model residuals? The `segments3d` function is ideal for this purpose, as you know from the 3D scatterplots of the `iris` data. First, you need to set up the "from-to" vectors in the three coordinates (refer to Section 26.1.5 for the initial explanation of `segments3d`'s usage; additional illustration is provided in Section 26.1.6):

```
R> xseg <- rep(mtcars$hp,each=2)
R> yseg <- rep(mtcars$wt,each=2)
R> zseg <- rep(car.fit$fitted.values,each=2)
R> zseg[seq(2,2*nrow(mtcars),2)] <- mtcars$mpg
```

As in the earlier examples, the x- and y-axis values don't change when moving from the "from" location to the "to" location, so these are simply double-replicates of each horsepower and weight entry of the original data frame. You need to instruct the z-axis "from" values to remain as the fitted values of the model (in other words, the actual vertical location of the response surface), and the "to" values are the raw data values of the response variable themselves. Then, assuming your previous plot remains open, a final call to `segments3d` draws on the residuals as standard black line segments for each `text3d`-labeled car:

```
R> segments3d(x=xseg,y=yseg,z=zseg)
```

Take a moment to interact with the final product, a screenshot of which is shown on the right of Figure 26-9.

26.2.3 Coloring by z Value

One advantage of `persp3d` plots is that you can color the surface according to the z values without needing to do anything special. Recall that if you were using the base R `persp` function, coloring by z value would require a minor workaround because you'd need to calculate the relevant vertical position as

the average of four adjacent z-matrix entries that make up each facet (see Section 25.6.2).

Fortunately, this isn't necessary with `persp3d`. Continuing one last time with the `mtcars` response surface, you can set up your desired color palette (refer back to Section 25.1) and assign colors to the entries of the z-matrix themselves without having to average out each set of four adjacent values first. Here, set up a custom palette and categorize each predicted value accordingly via `cut` (the categorization technique of color assignment is detailed in Section 25.1.4):

```
R> blues <- colorRampPalette(c("cyan","navyblue"))
R> blues200 <- blues(200)
R> zm <- car.pred.mat
R> zm.breaks <- seq(min(zm),max(zm),length=201)
R> zm.colors <- cut(zm,breaks=zm.breaks,include.lowest=TRUE)
```

Thereafter, you just need to index the `blues200` vector of colors by `zm.colors` to identify the correct color associated with each entry of the z-matrix, supplying the result directly to the `col` argument of `persp3d`:

```
R> persp3d(x=hp.seq,y=wt.seq,z=car.pred.mat,col=blues200[zm.colors],
           alpha=0.6,xlab="Horsepower",ylab="Weight",zlab="mean MPG")
```

Figure 26-10 shows a screenshot of the result.

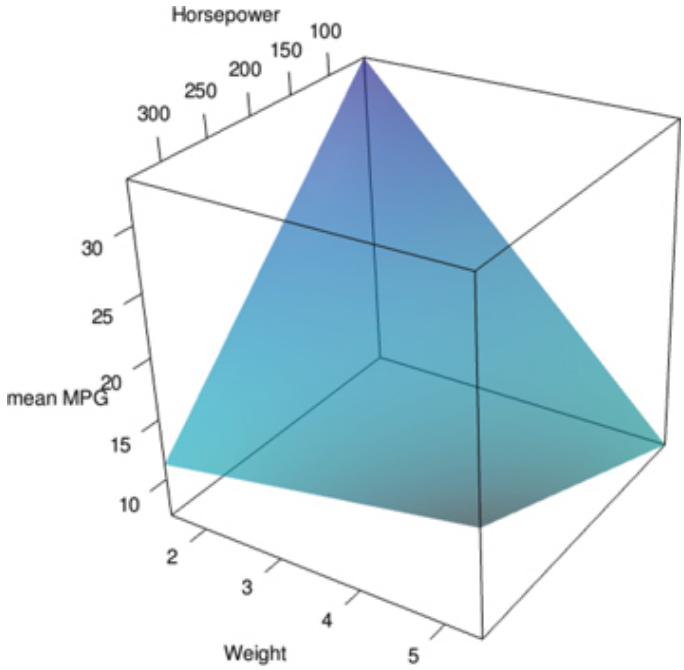


Figure 26-10: A result of direct color assignment corresponding to the z-matrix value in use of `persp3d` *for the* `mtcars` *response surface*

26.2.4 Dealing with the Aspect Ratio

Let's return to the bivariate kernel density estimate of the Chorley–Ribble cancer data, used in Sections 25.5.2 and 25.6.2. Load the `spatstat` package to access the `chorley` data and use the `MASS` package to access the `kde2D` function. Repeated here from Section 25.6.2 for convenience, the following code calculates the KDE surface using `kde2D`, stored as the `$z` component of the `chor.dens.WIN` object:

```
R> chor.WIN <- chorley$window
R> chor.dens.WIN <- kde2d(chorley$x,chorley$y,n=256,
                          lims=c(chor.WIN$xrange,chor.WIN$yrange))
R> chor.xy <- expand.grid(chor.dens.WIN$x,chor.dens.WIN$y)
R> chor.out.mat <- matrix(!inside.owin(x=chor.xy[,1],y=chor.xy[,2],
                                       w=chor.WIN),
                          256,256)
R> chor.dens.WIN$z[chor.out.mat] <- NA
```

With the final line, the code truncates the surface to fall within the polygon that represents the geographical study region by setting all elements of the z-matrix outside that polygon to `NA` (you first saw this in Section 25.5.2).

Then, executing the next few lines of code generates 200 colors from the built-in `rainbow` palette that you used for the previous plot of this kernel estimate (Figure 25-21 on page 767) and categorizes the entries of the truncated z-matrix accordingly:

```
R> zm <- chor.dens.WIN$z
R> rbow <- rainbow(200,start=0,end=5/6)
R> zm.breaks <- seq(min(zm,na.rm=TRUE),max(zm,na.rm=TRUE),length=201)
R> zm.colors <- cut(zm,breaks=zm.breaks,include.lowest=TRUE)
```

The difference here is you don't need to calculate facet averages as you did in Section 25.6.2—`cut` is applied directly to `zm`.

Before calling `persp3d`, remember that you should consider the aspect ratio in the x- and y- coordinate directions since you're dealing with a geographical area (see Sections 7.2.6 and 14.6.4). As you saw in Section 26.1.6, the `aspect` argument in `rgl` functions operates a little differently than the `asp` argument in `plot` or `image` or the `scale`/`expand` arguments in `persp`. In `rgl` plots, including `persp3d`, `aspect` requests a numeric vector of length 3, which defines the relative lengths of the x-, y-, and z-axes, in that order.

By default, `rgl` plots are rendered in a cube, regardless of the numeric width or scale of each axis. As such, we need to ensure axes are sized appropriately when plotting data like that of `chorley`, which demands a one-to-one equivalency of units in the planar (x-y) axes. To determine the appropriate relative scales for these data (or any function based upon them, like the kernel density estimate of interest), you need to calculate the total x-axis and y-axis widths that the study region is defined upon and find their ratio. For the `chorley` data, do this by using the `$xrange` and `$yrange` components of the study region (the `chor.WIN` object), subtracting the lower limit from the upper in each case, giving their relative lengths:

```
R> xd <- chor.WIN$xrange[2]-chor.WIN$xrange[1]
R> xd
[1] 23
R> yd <- chor.WIN$yrange[2]-chor.WIN$yrange[1]
R> yd
[1] 21.38
R> yd/xd
[1] 0.9295652
```

The final ratio of `yd/xd` reveals that you almost have a one-to-one scale, though technically the region is physically shorter in the y-axis, by a factor of around 0.93, than it is in the x-axis.

You can call `persp3d` to plot the Chorley–Ribble KDE surface as follows:

```
R> persp3d(chor.dens.WIN$x,chor.dens.WIN$y,chor.dens.WIN$z,
           col=rbow[zm.colors],aspect=c(1,yd/xd,0.75),
           xlab="Eastings (km)",ylab="Northings (km)",zlab="")
```

Therein, you use `aspect` to stipulate that the x-axis is taken as the reference scale of 1, that the y-axis should be scaled according to a factor of `yd/xd` relative to the x-axis, and that the z-axis should be squashed by a factor of 0.75 relative to the x-axis. Manipulating the y-axis scale via the second entry of `aspect` ensures the preservation of the correct spatial aspect ratio (one unit in the x-axis direction is equal in size to one unit in the y-axis direction). The last value of `aspect` is optional; I chose 0.75 here so that the graphic looks similar to the original `persp` plot in Figure 25-21 on page 767.

Let's finish off the plot by adding a color legend. Ensure you have the `shape` package loaded, resize your RGL device containing the result of the most recent call to `persp3d`, and execute the following command:

```
R> bgplot3d({plot.new();
             colorlegend(col=rbow,zlim=range(chor.dens.WIN$z,na.rm=TRUE),
                         zval=seq(0,0.02,0.0025),main="Density",digit=4,
                         posx=c(0.87,0.9),posy=c(0.2,0.8))})
```

Figure 26-11 shows a screenshot. Remember that `bgplot3d` changes the background of the current RGL device and demands a particular usage; refer to the end of Section 26.1.6. You might want to experiment a little with `posx` and `posy` to find your preferred placement of the color legend.

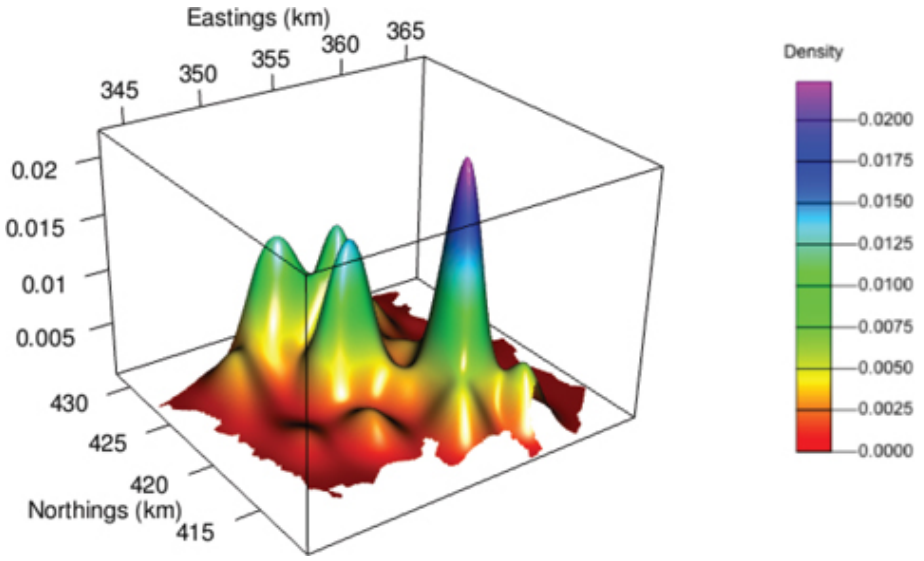


Figure 26-11: An interactive `persp3d` *representation of the Chorley–Ribble kernel density estimate, colored according to z-axis values, with a static color legend*

Exercise 26.2

Return once more to the measurements in the built-in `airquality` data frame. Create a copy of the data frame containing the variables pertaining to temperature, wind speed, ozone level, and month; delete all rows with any missing values by using `na.omit`. You're now going to experiment with an `rgl` visualization of an earlier regression model for mean temperature.

a. Refit the multiple linear model from Exercise 25.4 on page 757, which regressed temperature against the main effects and an interactive effect of wind speed and ozone. Following the same blueprint outlined in several examples throughout Section 26.2, use `expand.grid` and `predict` to construct a z-matrix of the response surface of air temperature; include the estimation of a 95 percent confidence interval for the fitted mean. Then, use `rgl` functionality to produce an interactive 3D plot of the response surface and color it yellow.

b. Using the built-in `topo.colors` palette, replot the response surface, assigning the colors according to the z value (see Section 26.2.3) and setting the opacity at 80 percent. Tidy up the axis titles, resize the RGL device, and leave the plot open.

c. Further enhance the plot from (b) as follows:

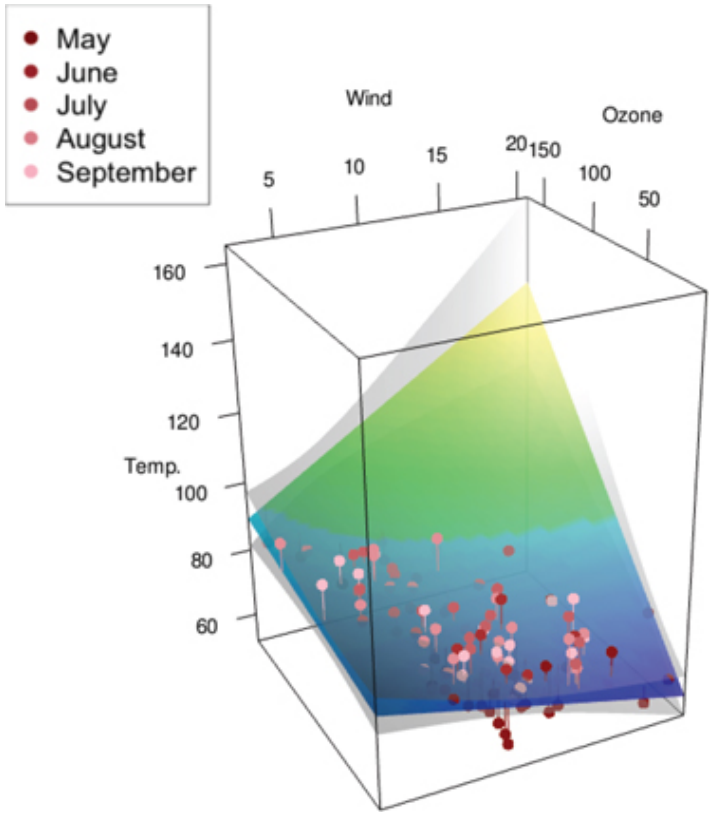
   i. Generate exactly five colors from a custom palette (Section 25.1.3) that goes from `"red4"` to `"pink"` and add the raw wind, ozone, and temperature observations as points to the plot of the response surface. The points should be colored according to month (May through September) using these colors in order. Set `size=8`.

   ii. In the same way as for the right-hand example in Figure 26-9, add vertical lines that denote the residuals of the fitted model to the plot; in other words, each observation should have a vertical line connecting it to the corresponding fitted value of the response surface. These added lines should use the custom palette from earlier to match the color of each data point.

   iii. Add the upper and lower 95 percent confidence limits you stored as part of the model prediction in (a). The added surfaces should both be gray with 50 percent opacity.

   iv. Add a legend to the top-left corner of the interactive plot referencing the five colors of points/lines according to month. Therein, use a `pch` value of `19` and a `cex` value of `2`.

   The result should look like this:



Next, load the `spatstat` package and revisit the `clmfires` dataset. Execute the following lines to restrict attention to only the intentionally lit fires and to obtain the geographical study region:

```
R> fire <- split(clmfires)$intentional
R> firewin <- clmfires$window
```

d. Re-create the static perspective plot from Exercise 25.5 (e) on page 771 as an interactive perspective plot with `persp3d`, based on the following guidelines. You will find the code corresponding to the Chorley–Ribble example discussed throughout Section 26.2.4 particularly useful.

- Use `kde2d` from the `MASS` package to calculate the KDE surface of the `$x`- and `$y`-coordinates of `fire`, truncated to the study region in `firewin`, using a 256×256 evaluation grid (you can simply leave the smoothing bandwidth at its default value).

- Use the built-in color palette `heat.colors` to color the surface according to the z value. Set the opacity to 70 percent.
- Use the axis-specific ranges of the study region to ensure the x-y axes have the correct one-to-one aspect ratio. Reduce the vertical aspect ratio to be 0.6 relative to the x-axis.
- Suppress the z-axis title but add neat `"X"` and `"Y"` titles to the other two axes.

e. Assuming the plot from (d) remains open, make the following enhancements:

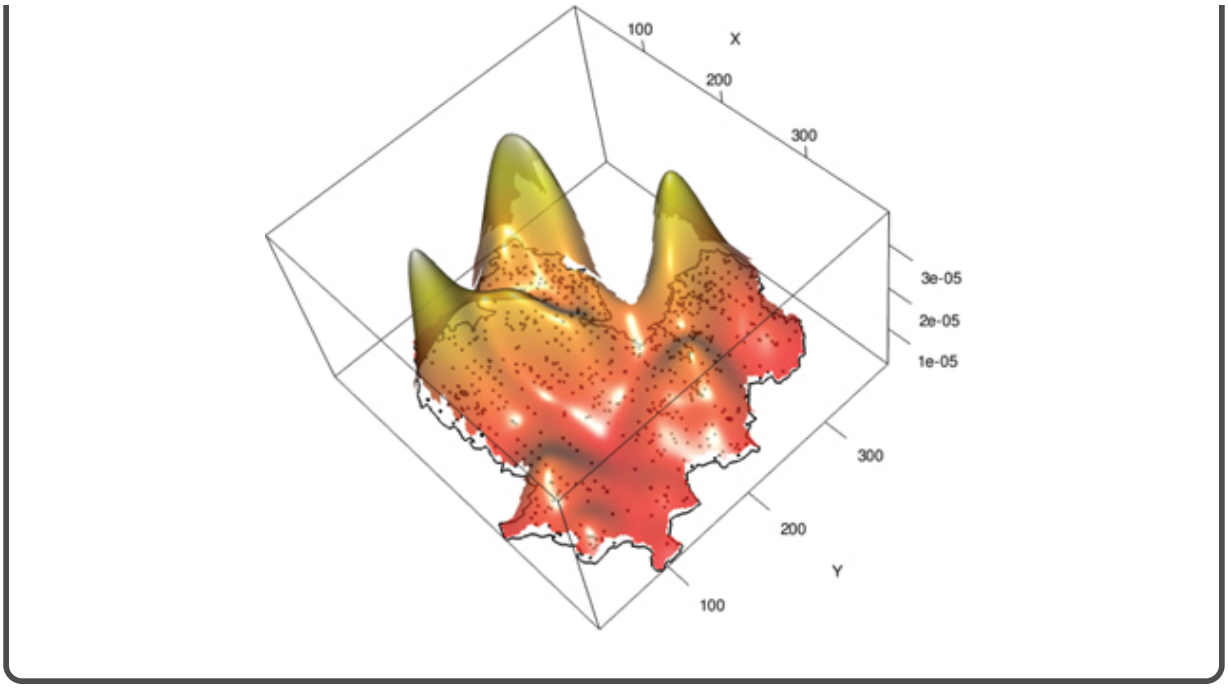
   i. Use `points3d` to add the raw observations so they lie "underneath" the surface itself. To do this, you'll set a constant z value for each data point. A sensible choice is the smallest non-`NA` value of the z-matrix object representing the kernel estimate (the result of the earlier use of `kde2d`), which you can extract using `min` with the optional argument `na.rm=TRUE`.

   ii. Obtain the vectors of the x- and y-coordinates of the irregular polygon that forms the study region by using the `vertices` function of `spatstat` as follows:

```
R> firepoly <- vertices(firewin)
R> fwx <- firepoly$x
R> fwy <- firepoly$y
```

   Supply these two vectors to the appropriate `x` and `y` arguments of the `lines3d` function to add the study region surrounding the superimposed observations. Again, you'll need to specify the z value as the minimum z-matrix value for all drawn lines. Set `lwd=2` for a slightly thicker line than the one drawn by default.

Your production should look something like this:



26.3 Trivariate Surfaces

So far you've looked at bivariate functions of the form $z = f(x, y)$, where your evaluation grid is two-dimensional. In other words, you evaluate the function f from an x value and a y value; the x and y values are plotted on the first two axes and the resulting values, z, are used to plot a third dimension. Next, you'll plot *trivariate* functions, which extends your dimension by one. Think of these functions as $w = f(x, y, z)$. That is, the evaluation grid is itself three-dimensional, and the function f gives you a fourth value, w, to use for plotting the surface.

26.3.1 Evaluation Coordinates in 3D

When dealing with a trivariate mathematical function, you need x, y, and z input values to obtain the result. Rather than a 2D x-y evaluation grid that sits in a square or rectangle, you now have a x-y-z evaluation *lattice* that sits in a cube or some other 3D rectangular prism.

As a perfect first example of a trivariate function, you'll create a color cube of RGB colors, where each point is the result of three values in red, green, and blue (refer to Section 25.1.1 for details). You'll use the three physical axes to reflect the evaluation lattice in the red, green, and blue

values, and the result will be a little sphere plotted with the color corresponding to that coordinate position in the 3D space.

The following code sets up the evaluation lattice in the three coordinate directions:

```
R> reds <- seq(0,255,25)
R> reds
[1]   0  25  50  75 100 125 150 175 200 225 250
R> greens <- seq(0,255,25)
R> blues <- seq(0,255,25)
R> full.rgb <- expand.grid(reds,greens,blues)
R> nrow(full.rgb)
[1] 1331
```

First, you generate equally spaced increasing sequences in the three colors spanning the standard 0 to 255 RGB integer range: The vectors `reds`, `greens`, and `blues` are all identical. Then you use the built-in `expand.grid` function to generate the data frame of all unique color triplets according to these three sequences, resulting in an evaluation lattice of exactly 11^3 = 1,331 specific coordinates. (Note that `expand.grid` works in the same way for higherdimensional evaluation grids as it does for bivariate x-y grids; refer back to Section 25.3.1.)

A subsequent call to `plot3d` places spheres at each 3D evaluation coordinate, setting each sphere's color correspondingly (recall the use of the `rgb` command from Section 25.1.1):

```
R> plot3d(x=full.rgb[,1],y=full.rgb[,2],z=full.rgb[,3],
          col=rgb(full.rgb,maxColorValue=255),type="s",
          size=1.5,xlab="Red",ylab="Green",zlab="Blue")
```

Figure 26-12 shows screenshots of the result from two different angles so you can see how the intensities of the red, green, and blue components of an RGB triplet control the color of each point.

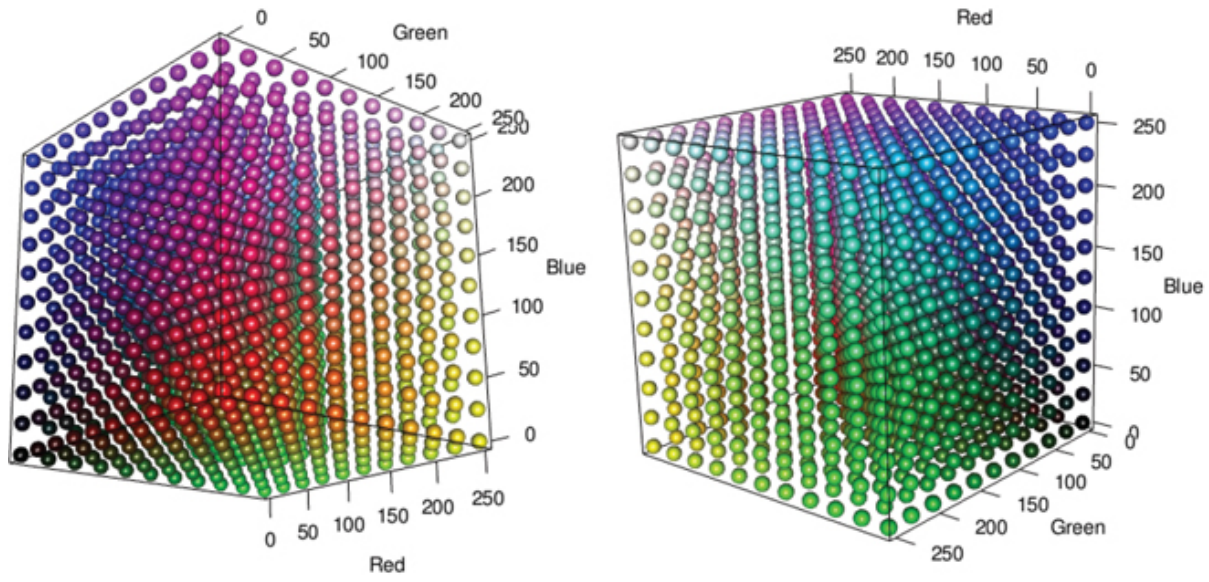


Figure 26-12: An `rgl` color cube, created as spheres whose fourth-dimension result (color itself) is the product of evaluating the trivariate RGB function.

26.3.2 Isosurfaces

Figure 26-12 reveals one of the problems with plotting the individual spheres of points at each 3D evaluation coordinate: It's difficult to see spheres "inside" the 3D prism. This issue also complicates the more general case of visualizing a continuous trivariate function. A solution is to produce a set of *isosurfaces*, which you can think of as a trivariate analog of a contour plot.

With an isosurface, you select a certain level of the values $w = f(x, y, z)$ and join up all the entries of w at that level inside the 3D space to form a shape or blob. These blobs show where in the 3D space the trivariate function takes on the set value. If you then plot these blobs at various levels, you get the aforementioned 3D version of the contour plots you created in Section 25.4.1, showing how the trivariate function fluctuates in value over the three-dimensional domain.

Higher-Dimensional Probability Densities

Cast your mind back to the univariate normal probability density function detailed in Section 16.2.2. First, I'll introduce the idea of higher-dimensional density functions with the bivariate version of the normal

distribution, and then I'll go one step further and use the trivariate version to illustrate isosurface plotting.

To work with multivariate normal distributions, install the `mvtnorm` package with a call to `install.packages("mvtnorm")`. Just like the `rnorm` function for the univariate normal, the `rmvnorm` function can generate random variates from a specified multivariate normal. Once you've installed `mvtnorm`, execute the following code:

```
R> library("mvtnorm")
R> rand2d.norm <- rmvnorm(n=500,mean=c(0,0))
R> plot(rand2d.norm,xlab="x",ylab="y")
```

This produces the plot on the left of Figure 26-13. The `rmvnorm` function generates 500 independent variates from the standard bivariate normal distribution. You center the variates around the coordinate (0, 0) by passing a numeric vector to `mean`. By default, independent standard deviation components of 1 are used in both x-y coordinate directions.

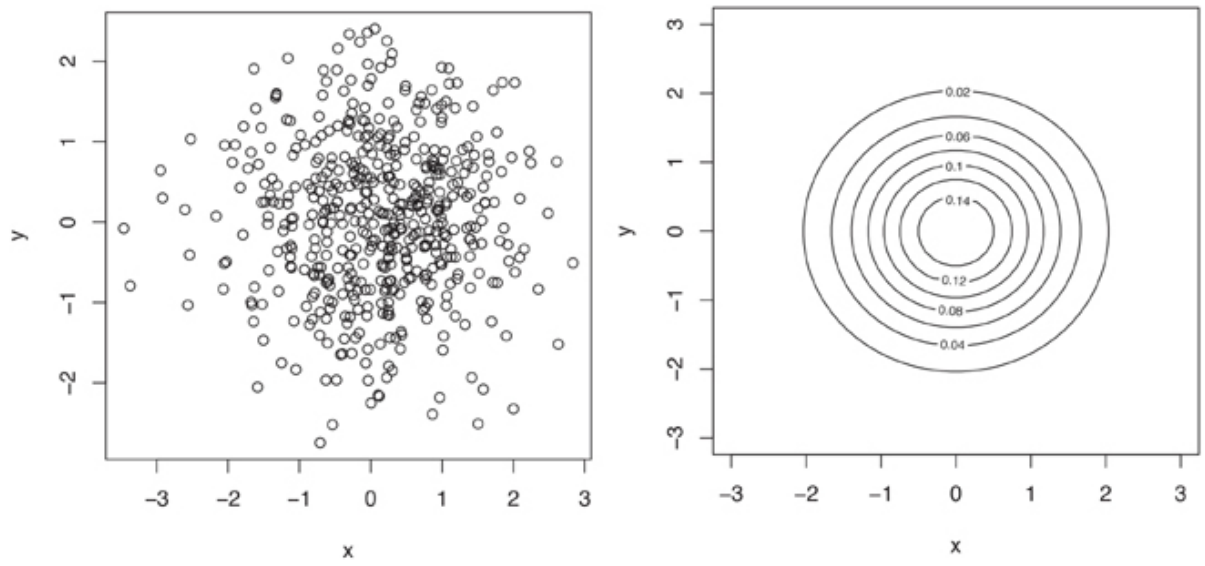


Figure 26-13: Using `mvtnorm` to view randomly generated data and the standard bivariate normal density they came from

To actually view the bivariate density function from which the data points were generated, you need to decide on the x-y evaluation grid and construct the z-matrix as usual by using `expand.grid`. The following code sets up an evenly spaced sequence to use in both coordinate directions and uses

the `dmvnorm` function. Think of this as the multivariate version of `dnorm`; it gives you the density function value at specified coordinates:

```
R> vals <- seq(-3,3,length=50)
R> xy <- expand.grid(vals,vals)
R> z <- matrix(dmvnorm(xy),50,50)
```

Then, you can use `contour` (or `persp` or `persp3d`) to view the density with respect to the defined evaluation grid. The following line produces the plot on the right of Figure 26-13:

```
R> contour(vals,vals,z,xlab="x",ylab="y")
```

With that knowledge, let's now increase the dimension once more: What does a trivariate normal density function look like?

First, let's take a look at some data generated from this density. The following code generates 500 random variates:

```
R> rand3d.norm <- rmvnorm(n=500,mean=c(0,0,0))
R> plot3d(rand3d.norm,xlab="x",ylab="y",zlab="z")
```

This time you've supplied a vector of length 3 as the `mean` argument to `rmvnorm`; the function therefore knows you're requesting the threedimensional version of the distribution. You're telling it that you want the data to come from a trivariate normal, with means of 0, 0, and 0 in each coordinate direction. You can see a screenshot of the `rgl` point cloud of the data produced via the previous call to `plot3d` on the left of Figure 26-14.

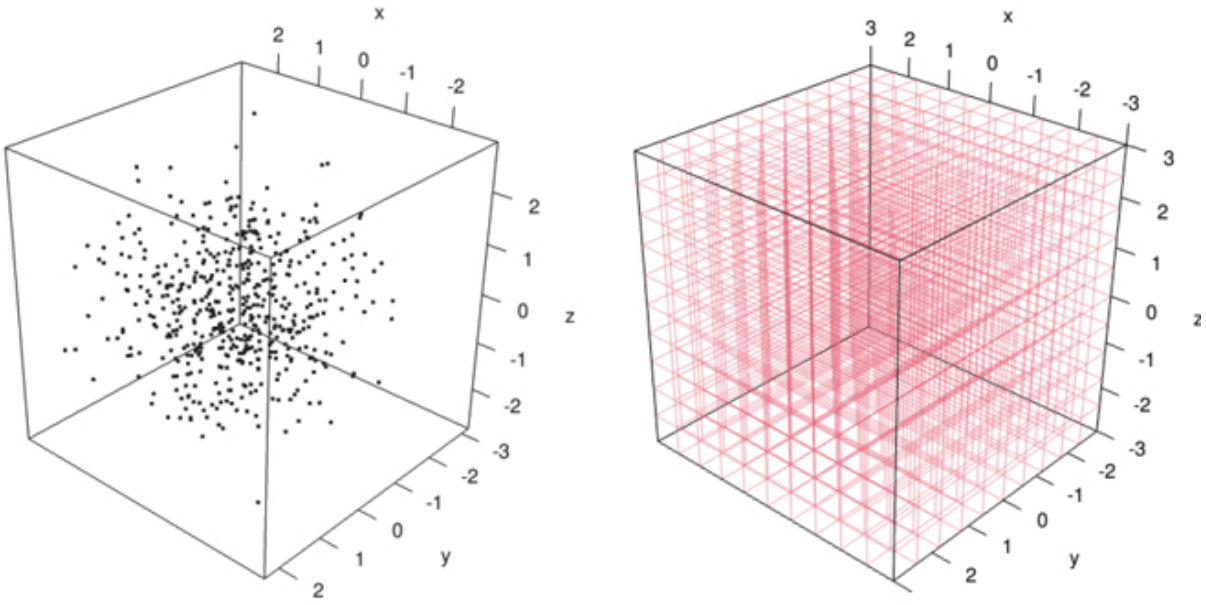


Figure 26-14: Left: Viewing data randomly generated from the standard trivariate normal distribution. Right: A concept of the 3D evaluation lattice upon which the trivariate density function itself will be plotted.

Trivariate Evaluation Lattice

To calculate and display the actual trivariate density function that generated these data, you'll need a 3D evaluation lattice, as noted at the beginning of Section 26.3. Take a look at the plot on the right of Figure 26-14, which shows a 3D 11×11×11 evaluation lattice based on sequences spanning [–3, 3] in x, y, and z. This should give you a clear idea of how increasing the dimension of a continuous function works. Each intersection in the 11×11×11 grid is the 3D equivalent of each intersection of the solid lines in the 2D 6×4 evaluation grid in Figure 25-9 on page 736, and each of the 10^3 mini-3D cubes in this 3D lattice is the 3D equivalent of a 2D facet, as noted in the discussion of Figure 25-20 on page 765. (As with Figure 25-9, you can find the code to plot this 3D lattice on the book's website.)

To plot the result of the trivariate function, you'll need the unique evaluation coordinates of an evaluation lattice. This follows the same principle used in creation of the color cube in the previous section (Figure 26-12). Using `vals`, the sequence of values between –3 and 3 created earlier to plot the bivariate density, the following code produces a data frame of all 50^3 = 125,000 unique 3D evaluation lattice coordinates:

```
R> xyz <- expand.grid(vals,vals,vals)
R> nrow(xyz)
[1] 125000
```

Then, use `dmvnorm` to get the numeric value of the standard trivariate normal just as you did in the bivariate setting. The function automatically knows you're requesting the trivariate version of the density because your data argument `xyz` has three columns:

```
R> w <- array(dmvnorm(xyz),c(50,50,50))
```

Note here that the result is stored appropriately as a 50×50×50 3D array (refer to Section 3.2 for details on `array`). Look at the conceptual diagram of a 3D array, Figure 3-3 on page 51, and compare it to the 3D lattice on the right of Figure 26-14. The trivariate normal values in the object `w` are represented by a 3D block of numbers sitting at each corresponding unique evaluation coordinate in the defined 3D space.

Basic One-Level Isosurface

You can produce an isosurface by using the `contour3d` function, which is part of the `misc3d` package (Feng & Tierney 2008), for use with `rgl` graphics. To use it, you need to decide on the level(s) at which to plot the surface. For densities, a sensible choice is to pick one or more α-level contours (see Section 1.4.3 of the outstanding book by Scott 2015). In brief, for some density f, these levels delineate the $(1 - \alpha) \times 100$ percent "most dense" observations by setting the isosurface to be drawn at positions in the multivariate evaluation lattice that correspond to the density value given by $\alpha \times \max(f)$. For the trivariate standard normal, the maximum value of the density is located at the mean at the coordinate (0, 0, 0), which you can obtain as follows:

```
R> max3d.norm <- dmvnorm(c(0,0,0),mean=c(0,0,0))
R> max3d.norm
[1] 0.06349364
```

You'll use this when producing the next couple of plots.

Install `misc3d`, load it with `library("misc3d")`, and then call `contour3d` using your previously created `vals`, `w`, and `max3d.norm` objects:

```
R> contour3d(x=vals,y=vals,z=vals,f=w,level=0.05*max3d.norm)
```

This produces an isosurface in an RGL device you can rotate and zoom as you desire; you can see a screenshot of the result on the left of Figure 26-15. You supply `contour3d` with the arguments `x`, `y`, and `z` as evenly spaced sequences in the x-, y-, and z-coordinate directions, respectively (in this case, all are equivalent and are defined by the vector `vals`). Supply the corresponding 3D array, defining the entire result of the trivariate function to the `f` argument, and pass the level(s) at which you want to draw the isosurface itself to `level`. Here, I've chosen a small 5 percent α-level, meaning that most observations generated from this density will occur within the visible blob.

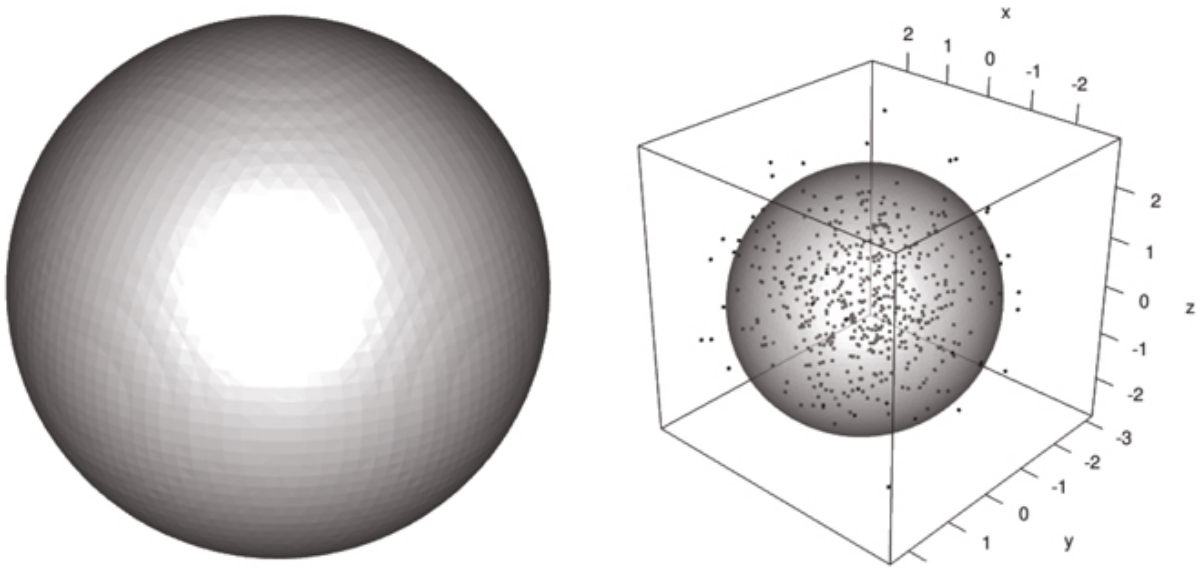


Figure 26-15: An isosurface of the trivariate standard normal density function produced using `contour3d` of the `misc3d` package. Left: A stand-alone plot drawn at an α level of 0.05. Right: Adding the same surface to an existing `rgl` plot of randomly generated trivariate normal observations, at 50 percent opacity.

The plot shows that the shape of the trivariate density is relatively clear based on the plot of the randomly generated data you produced earlier. However, without scale, it's little more than a statistical golf ball. It's often more helpful to view the data alongside the density from which they came, which is also easy to do. The following code re-plots the data in `rand3d.norm` using `plot3d`, and then immediately calls `contour3d` again to draw at the α level of 0.05:

```
R> plot3d(rand3d.norm,xlab="x",ylab="y",zlab="z")
R> contour3d(x=vals,y=vals,z=vals,f=w,level=0.05*max3d.norm,add=TRUE,alpha=0.5)
```

As shown here—and just like the traditional R `contour` function—if you want to use `contour3d` to add to an existing `rgl` plot, you need to explicitly specify `add=TRUE`. You can also use the optional `alpha` argument to adjust opacity, reduced to 50 percent in this example, to "see inside" the density isosurface (don't confuse this argument with the α-level at which the contour is actually drawn, which is an entirely separate decision). A screenshot of the result of the preceding code is given on the right of Figure 26-15.

Multiple Level Control with Color and Opacity

Adjusting opacity is especially useful when you want to plot the isosurface at multiple α-levels at once. Color is also useful in this way, as a variable that can represent a fourth dimension without adding an additional physical axis to the graph.

To view the trivariate normal density at multiple levels, remain with the existing objects and consider the plot produced by executing the following code:

```
R> plot3d(rand3d.norm,xlab="x",ylab="y",zlab="z")
R> contour3d(x=vals,y=vals,z=vals,f=w,
             level=c(0.05,0.2,0.6,0.95)*max3d.norm,
             color=c("pink","green","blue","red"),
             alpha=c(0.1,0.2,0.4,0.9),add=TRUE)
```

Figure 26-16 shows a screenshot of the result. Here, you re-plot the 500 randomly generated trivariate normal observations, and another call to `contour3d` now draws contours at four specific α-levels of the trivariate density: 0.05, 0.2, 0.6, and 0.95. You use the optional `color` argument to render these in pink, green, blue, and red, respectively, and progressively increase the opacity of each level with the `alpha` argument.

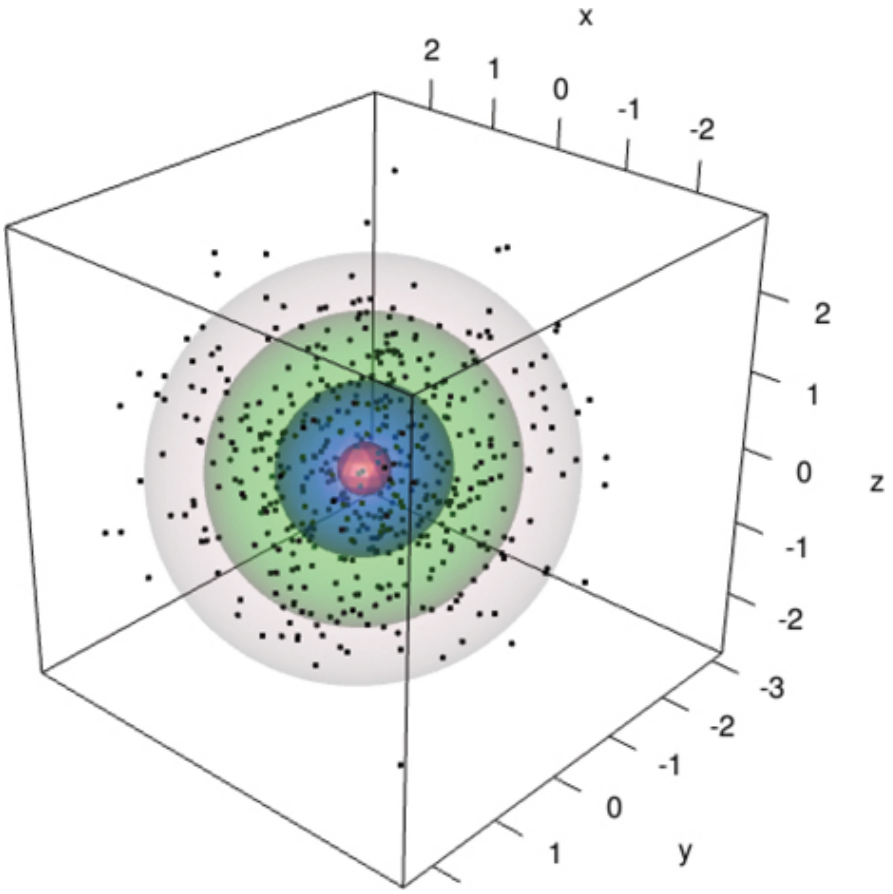


Figure 26-16: Isosurfaces of the trivariate normal density plotted at four levels over the randomly generated observations. Color and opacity are used to distinguish between the different numeric levels of the plotted function.

By inspecting the resulting graphic, you should be able gauge the increase in the density of the points in the 3D space for this distribution in a similar way as you'd use standard 2D contours to appraise the distribution of bivariate observations.

26.3.3 Example: Nonparametric Trivariate Density

For an extended example using real data, look once more at the built-in `quakes` data frame, which includes the spatial location, magnitude, and depth of 1,000 seismic events. In Section 25.4.1, you constructed bivariate kernel density estimates of the 2D longitude-latitude spatial coordinates, using the `MASS` function `kde2D`. As noted there, kernel estimation extends naturally to higher dimensions. The goal now is to calculate and visualize a density estimate of the same spatial earthquake data, but this time do so based on the trivariate coordinates of longitude, latitude, *and* depth, in 3D space.

Raw Data

First, let's plot the raw data. The following code creates a copy of the `quakes` data, extracting those three variables and rendering `depth` negative. I do this so that, when plotted, earthquake depth corresponds to moving *down* the vertical axis to give the impression of depth below sea level:

```
R> quak <- quakes[,c("long","lat","depth")]
R> quak$depth <- -quak$depth
```

Before we actually create the plot, remember we must be mindful of setting an appropriate aspect ratio for geographic data (Section 14.6.4). With `rgl` functionality, the behavior of `plot3d` means we must also ensure the initial cube in which the data are plotted is sized appropriately in the x- and y-axis directions. As discussed in Section 26.2.4, this requires calculating the relative spans of the observed data in the two planar coordinates. In this example, we can do so by simply subtracting the smallest longitude value from the largest (giving the x-axis size), and doing the same for the latitude values (the y-axis size):

```
R> xd <- max(quak$long)-min(quak$long)
R> yd <- max(quak$lat)-min(quak$lat)
R> aspy <- yd/xd
R> aspy
[1] 1.240873
```

The result given as `aspy` reflects the size of the recorded data along the vertical axis, relative to the size of the horizontal axis.

Following details in Section 26.2.4, `aspy` would be the correct value to use inside the `aspect` argument of `plot3d` to scale the y-axis if we sought to treat a unit change in latitude as equivalent to a unit change in longitude in terms of spatial distance. However, we know that this is typically not appropriate for longitude-latitude data, and that an adjustment is needed; for the `quakes` data, it was previously determined that a factor of 1.069 is an appropriate scaling for latitude relative to longitude (Section 14.6.4). As such, we may create a point cloud of the raw data with the following:

```
R> plot3d(x=quak$long,y=quak$lat,z=quak$depth,aspect=c(1,1.069*aspy,1),
          xlab="Longitude",ylab="Latitude",zlab="Depth")
```

Figure 26-17 shows a screenshot of the result. The use of `aspy` in the second position of the `aspect` argument accounts for the relative sizes of the span of the latitude values versus the span of the longitude values, and subsequent multiplication by 1.069 ensures the planar (x-y) margins of the data are approximately geographically accurate. The third entry of `aspect`, which corresponds to the depth variable along the z-axis, is left unaltered at 1 (its visible size is therefore equal to that of the x-axis).

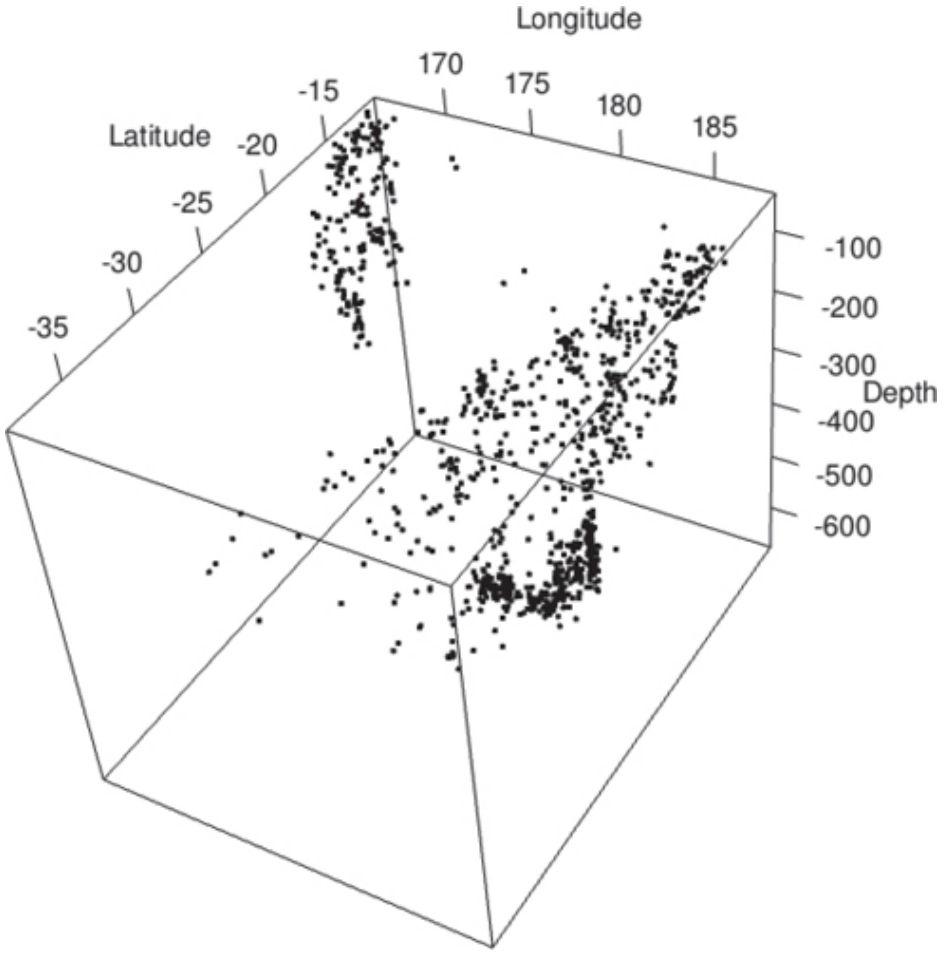


Figure 26-17: Viewing the 3D spatial dispersion of the earthquake occurrences—latitude, longitude, and depth—after finding an appropriate aspect ratio in the planar (x-y) coordinate directions

If you rotate the resulting plot so that you're looking down directly from the top with a bird's-eye view, you'll recognize the 2D spatial patterning that you've plotted already (with the correct aspect ratio preserved); see, for example, Figure 13-1 (page 307) or Figure 25-12 (page 742).

The 3D Estimate Calculation

The evaluation lattice for this kernel estimate will be defined by the entire 3D space in which the longitude, latitude, and (negative) depth data reside, exactly as Section 26.3.2 illustrates. To actually calculate the trivariate KDE surface for the `quak` data, you'll use the impressive functionality of the contributed package `ks` (Duong 2007, 2024, Chacón & Duong 2018). Install the package and load it with a call to `library("ks")`. The `kde` function within the `ks` package allows you to use kernel smoothing to estimate the probability density of 1D through 6D data.

The first argument you supply to `kde` is your data in the form of a matrix or data frame, with the tag `x`. Note that the order of the columns in your data object matters when using `kde`. When called as follows with `quak`, and taking into account the extraction order of the three variables in the earlier code that created `quak`, the x-, y-, and z-coordinate axes in the resulting 3D kernel estimate will correspond to longitude, latitude, and depth, respectively:

```
R> quak.dens3d <- kde(x=quak,gridsize=c(64,64,64),compute.cont=TRUE)
```

This matches the way the data are displayed in Figure 26-17. The `gridsize` argument specifies the lattice resolution in each axis. In this example, I've settled on a 64×64×64 lattice; by default, `kde` chooses the range of evaluation in each coordinate direction so that it's slightly wider than the observed data. Finally, to plot the result, it's useful to also specify the argument `compute.cont=TRUE`; I'll discuss the reason for this in a moment. There are a number of other aspects of the estimate you can tweak with optional arguments, such as the all-important smoothing bandwidth. These have defaults that are sensible in a wide range of scenarios, and have thus been left as-is in the current example. For further information, see the documentation at `?kde`.

The returned object, `quak.dens3d`, has several components. The 3D estimate is provided as an appropriately sized array as the `$estimate` member; execution of the following line confirms it matches the desired lattice resolution:

```
R> dim(quak.dens3d$estimate)
[1] 64 64 64
```

The `$eval.points` component of `quak.dens3d` holds a list (Section 5.1) whose members are the specific evaluation coordinates, which in this case are equally spaced sequences in each of the three axes. The number of members reflects the dimension of the problem, and their order corresponds to the specific axis. You can extract them with the following lines:

```
R> x.latt <- quak.dens3d$eval.points[[1]]
R> y.latt <- quak.dens3d$eval.points[[2]]
R> z.latt <- quak.dens3d$eval.points[[3]]
```

If you print these vectors to your console screen, you'll see that each is a vector of length 64, with `x.latt`, `y.latt`, and `z.latt` corresponding to the variables matching the order of the columns in the data frame `quak`.

Isosurface Level Selection

The selection of the level to display depends on the range of values that make up the result of the trivariate function itself. When you choose `compute.cont=TRUE` in the call to `kde`, you're automatically provided with a collection of appropriate levels. These are returned in the component `$cont` as a numeric vector with a length of exactly 99, representing each integer between 1 percent and 99 percent.

Internally, these levels are calculated by working out the result of the trivariate function at the location of each of the originally observed data points and then using `quantile` to obtain all the integer-valued percentiles (from 99 percent to 1 percent) of these density values (for a refresher on quantiles, refer to Section 13.2.4). These are returned in decreasing order; in other words, `quak.dens3d$cont[1]` corresponds to the 99th percentile, and `quak.dens3d$cont[99]` is the 1st percentile.

Though these values are obtained differently from the α-levels you experimented with when plotting the trivariate normal density (Section 26.3.3), you essentially end up with the same interpretation when visualizing the result; these values allow you to interpret the isosurfaces in terms of estimated observation "denseness." For example, extract the lower quartile (or the 25th percentile) with the following:

```
R> quak.dens3d$cont[75]
      25%
```

```
2.567628e-05
```

This provides the value of the trivariate density function that is estimated to separate the most spatially diffuse 25 percent of the observations from the rest (in other words, so that the resulting blobs encapsulate the most spatially dense 75 percent of the data).

Ensure you have both the `rgl` and `misc3d` packages loaded. When you execute the following code, it first replots the raw `quak` data and then adds the corresponding isosurface using the lower point-wise density quartile—the value reported previously—as the desired level:

```
R> plot3d(x=quak$long,y=quak$lat,z=quak$depth,aspect=c(1,1.069*aspy,1),
          xlab="Longitude",ylab="Latitude",zlab="Depth")
R> contour3d(x=x.latt,y=y.latt,z=z.latt,f=quak.dens3d$estimate,
             color="blue",level=quak.dens3d$cont[75],add=TRUE)
```

You can see the result on the left of Figure 26-18, where the blobs (colored blue by the code) represent the 3D contour at the specified level. Higher levels of the trivariate function—that is, more densely grouped points—are "inside" these blobs, obscured from view. To see inside the isosurfaces, adjust the opacity with the `alpha` argument.

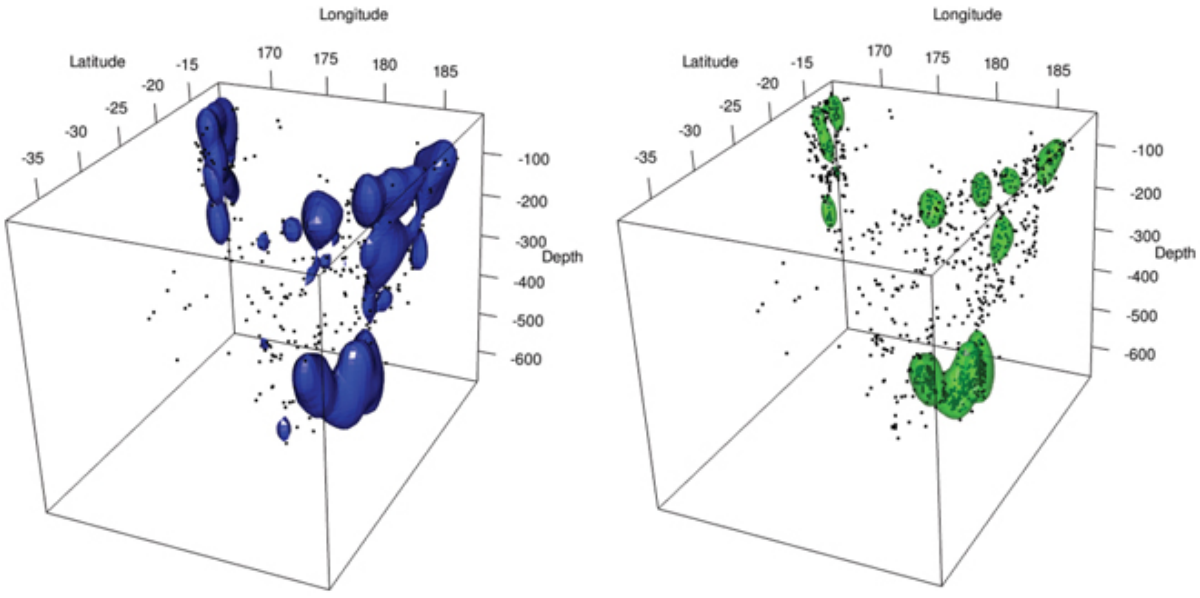


Figure 26-18: Isosurfaces of a trivariate kernel density estimate drawn at point-specific density quantiles, based on use of the contributed `kde` and `contour3d` functions (`ks` and `misc3d` packages, respectively)

Let's test this with a different density level, in this case the median value that delineates the observations associated with the lower and upper 50 percent of estimated density values:

```
R> quak.dens3d$cont[50]
        50%
4.890472e-05
```

Rerun the immediately preceding call to `plot3d` to replot the raw `quak` data. Then call `contour3d` to produce the result on the right of Figure 26-18, allowing you to see through the green blobs:

```
R> contour3d(x=x.latt,y=y.latt,z=z.latt,f=quak.dens3d$estimate,
             color="green",level=quak.dens3d$cont[50],add=TRUE,alpha=0.5)
```

Lastly, you'll highlight the top 80 percent of the most densely aggregated observations using multiple levels. Execute the following:

```
R> qlevels <- quak.dens3d$cont[c(80,60,40,20)]
R> qlevels
        20%          40%          60%          80%
2.260448e-5 3.692132e-05 6.086846e-05 1.258873e-04
```

This obtains four levels: the quantiles above which the 80 percent, 60 percent, 40 percent, and 20 percent most densely clustered observations are identified. Set up a couple of vectors to control the color and opacity of each increasing level of denseness accordingly:

```
R> qcols <- c("yellow","orange","red","red4")
R> qalpha <- c(0.2,0.3,0.4,0.5)
```

The corresponding sequences of colors (`qcols`) and levels of opacity (`qalpha`) reveal that the isosurface will darken in color and become more opaque as the density increases.

One final time, replot the raw `quak` data using the `plot3d` shown previously. Then it's simply a matter of supplying your isosurface values, colors, and opacity levels to each appropriate argument in `contour3d`:

```
R> contour3d(x=x.latt,y=y.latt,z=z.latt,f=quak.dens3d$estimate,
             color=qcols,level=qlevels,add=TRUE,alpha=qalpha)
```

Figure 26-19 shows screenshots of the resulting graphic of varying perspective and zoom. Increasing levels of denseness are reflected by isosurfaces of darkening yellow-to-red color and increased opacity.

You can see that the tightest grouping of earthquakes occurs quite deep and toward the eastern edge of the 3D spatial prism (the visible "threechamber" density blob is a well-known feature of these particular data).

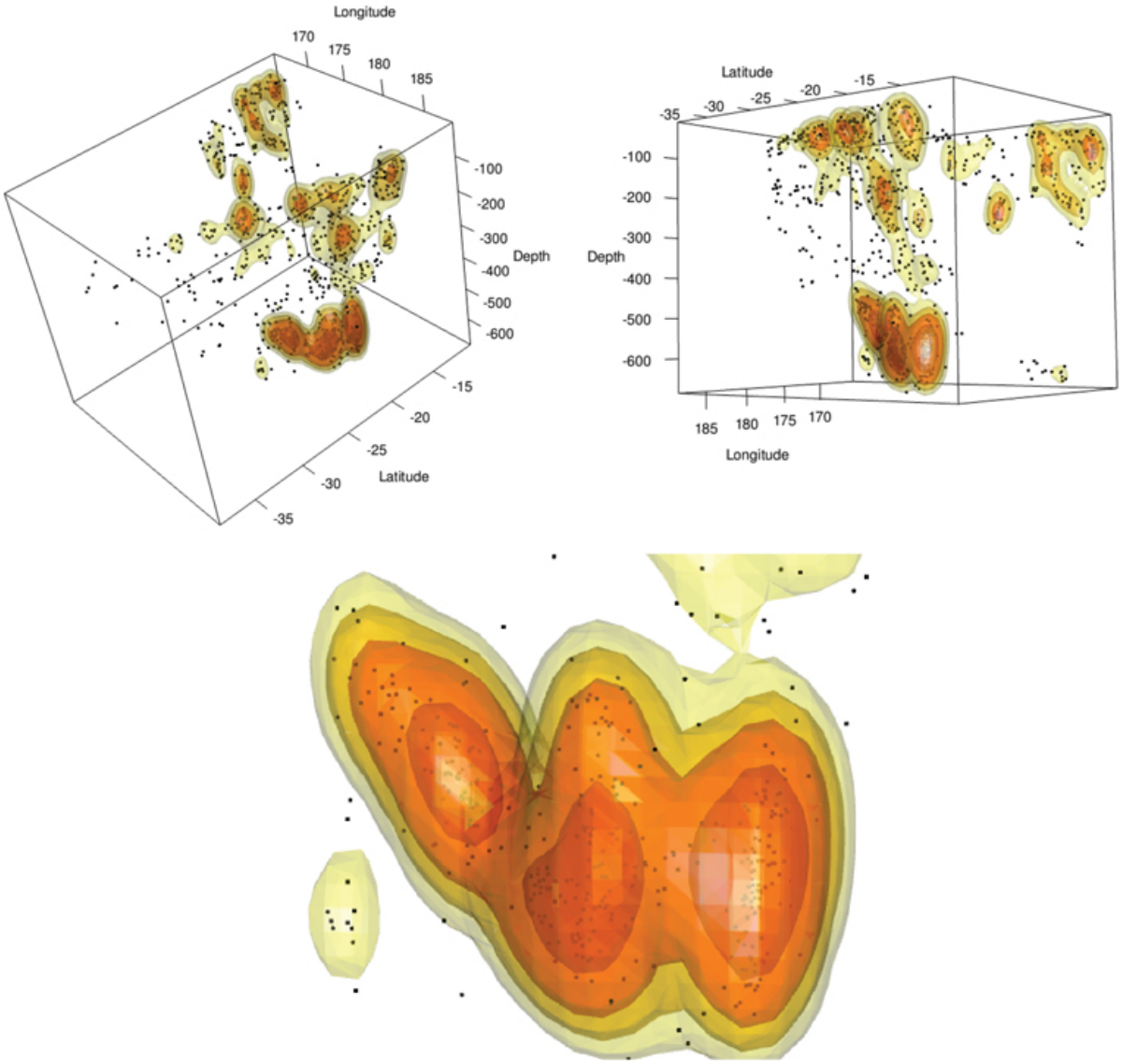


Figure 26-19: Three screenshots of the trivariate kernel density estimate of the earthquake observations, taken from varying angles and with different levels of zoom.

26.4 Handling Parametric Equations

In most of this chapter's examples so far, the surfaces are directly defined by the coordinates of a regular evaluation grid or lattice, but there are situations where the final axis you want to visualize is *not* a function of some evaluation grid. This occurs quite naturally when you simply want to draw familiar geometric shapes, but it also extends to more complicated situations in mathematics.

In this section, you'll plot from a collection of parametric equations, which together define the shape or surface of interest. This section will assume you're familiar with the fundamental trigonometric functions sine and cosine, as well as the conversion of angles from degrees to radians, since R deals with the latter by default (these concepts have popped up briefly in Section 14.6.4). That said, I'll walk you through the relevant calculations and R code as needed.

26.4.1 Simple Loci

Using mathematical terminology, a *locus* (plural *loci*) is a set of points that satisfy, and are hence defined by, a particular set of equations. In a programming sense, these equations govern how individual numeric elements of the resulting objects are calculated, which you can then easily plot using familiar R functions.

NOTE

When discussing loci here, any mention of 2D or 3D space refers to Euclidean space, which is the standard way to interpret coordinates in the x-, y-, and z-axes.

2D Circle

Let's start with a simple example: a 2D circle. To find any point on a circle, you need to know the circle's center coordinate and its radius, and you need to provide a specific angle at which to look (typically taken to be relative to a perfectly horizontal line). Any planar 2D point (x, y) that lies on a circle can be expressed with the following pair of equations if you take the center

to be at the coordinate (a, b), with a fixed radius of $r > 0$, looking at the angle θ:

$$x = a + r\ \cos(\theta) \quad \text{and} \quad y = b + r\ \sin(\theta) \quad (26.1)$$

If θ is expressed in degrees, you can assume $0 \leq \theta < 360$. To convert from degrees to radians (as required by R), you must multiply by $\pi/180$; let $\theta' = \theta\pi/180$ represent the corresponding conversion such that $0 \leq \theta < 360$ implies $0 \leq \theta' < 2\pi$.

To draw a circle based on the equations in (26.1), first decide on a radius, then decide on a center point, and then generate the corresponding values of x and y for a set of values that satisfy the stated constraints on θ. Consider the following code:

```
R> radius <- 3
R> a <- 1
R> b <- -4.4
R> angle <- 0:360*(pi/180)
R> x <- a+radius*cos(angle)
R> y <- b+radius*sin(angle)
R> plot(x,y,ann=FALSE,asp=1)
R> abline(v=a)
R> abline(h=b)
```

The circle will have a radius of 3 and be centered at (1, –4.4). Given the sequence defined as `angle`, note the plot will place a point at each integer angle from 0 to 360 degrees; here, I've allowed the upper limit to be equal to exactly 360 to fully complete the rotation. In the same line, you convert this sequence to radians with multiplication by $\pi/180$ in order to use the built-in R functions `cos` and `sin`. The geometric value of pi (π = 3.1415 . . .) is held within the ready-to-use R object `pi`, as noted previously in Section 14.6.4 (see the help file `?Constants`). The last three lines execute the plot, shown in Figure 26-20.

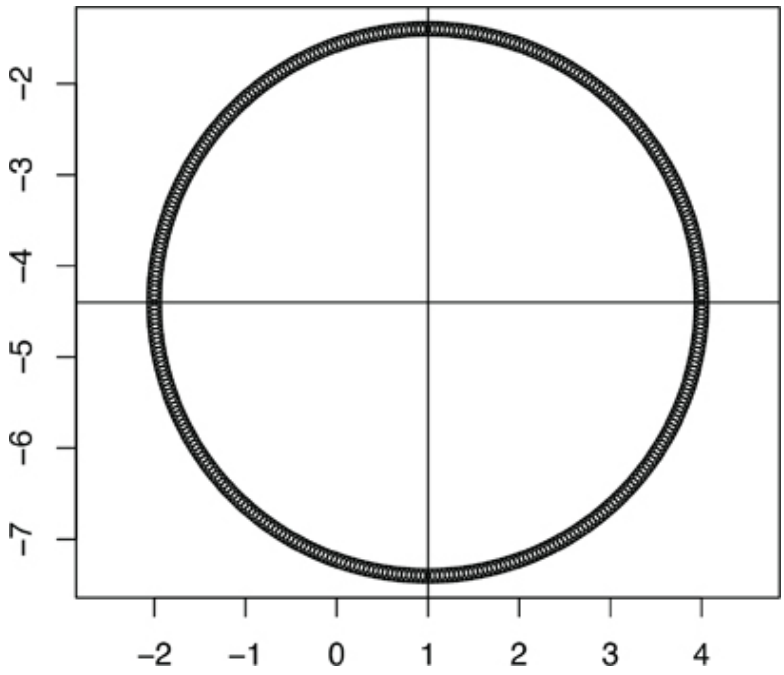


Figure 26-20: Drawing a 2D circle in R with center (1,–4.4) and radius 3, following the relevant parametric equations of the locus

The key takeaway here is that y isn't calculated as a direct result of x in the same way that you might obtain a fine, evenly spaced increasing sequence in x and then evaluate y when plotting, for example, a linear regression model. Rather, the equations in (26.1) jointly define the rules of the locus in the 2D space.

3D Cylinder

You plot surfaces with three dimensions similarly, only now your equations set up the rules for all satisfying points in the x-, y-, and z-axes.

For an example, note that points lying on a hollow cylinder can be defined by the following equations:

$$x = r\ \cos(\theta), \quad y = r\ \sin(\theta), \quad \text{and}\ \ z = z \quad (26.2)$$

To actually plot points that satisfy these rules, you need to decide on a fixed radius r, recognize that $0 \leq \theta < 360$ (in degrees), and define the appropriate constraints on z (which controls the length of the cylinder in terms of start point and end point. For instance, you could decide upon some maximum length, $h > 0$, and stipulate $0 \leq z \leq h$. Note that to generate

vectors for x and y, you need to first set up numeric sequences spanning the possible values of θ. Consider the following code:

```
R> r <- 3
R> h <- 10
R> zseq <- 0:h
R> theta.rad <- 0:360*(pi/180)
```

These lines show a radius of 3 set as `r` and a maximum cylinder length of 10 as `h`. The sequence in z is set up as the 11 integer values from 0 to 10 in `zseq`; this will allow you to place points on the locus at each of these defined z values. The sequence for θ is set up as `theta.rad` (again, note the necessary conversion to radians via the multiplicative factor of $\pi/180$). Then, you need all unique combinations of these parameter values to get all relevant (x, y, z) coordinates for plotting. You can do this using `expand.grid` (Section 25.3.1):

```
R> ztheta <- expand.grid(zseq,theta.rad)
R> nrow(ztheta)
[1] 3971
```

Calling `nrow` on the result shows that you now have $11 \times 361 = 3{,}971$ unique height-angle values. Now you're able to generate the values for x, y, and z as defined by Equation (26.2). You could use a `for` loop (Section 10.2), cycling through each row of `ztheta`, but it's neater to use implicit looping in `apply` (refer back to Section 10.4.1 for details):

```
R> x <- apply(ztheta,1,function(vec) r*cos(vec[2]))
R> y <- apply(ztheta,1,function(vec) r*sin(vec[2]))
R> z <- apply(ztheta,1,function(vec) vec[1])
```

Note here that anonymous functions (see Section 11.3.2) are used to operate on the two-element height-angle (in that order) vectors that make up each row of `ztheta`.

You can use `persp3d` from the `rgl` package to plot this kind of parametrically defined surface, but in a slightly different way than in earlier sections of this chapter. The calculated `x`, `y`, and `z` components must now *all* be supplied as identically sized, appropriately arranged matrices. This is because there's no longer an evenly spaced evaluation grid in the x- and y-coordinate directions; together with the z values, the x and y values have all

been defined through application of Equation (26.2). Instead, you effectively have a *latent* evaluation grid defined by the unique combinations of parameter values (height z and angle θ' in this case).

The matrices in all of the x-, y-, and z-coordinates are given by the three 11×361 matrices filled with `x`, `y`, and `z` in the typical column-wise fashion:

```
R> xm <- matrix(x,length(zseq),length(theta.rad))
```

```
R> xm <- matrix(x,length(zseq),length(theta.rad))
R> ym <- matrix(y,length(zseq),length(theta.rad))
R> zm <- matrix(z,length(zseq),length(theta.rad))
```

At this point, it's worth introducing the built-in `outer` function, which takes a sequence of values in two variables and produces all unique combinations of values, computes the result at each combination, and then returns the results as a matrix, thus doing the three tasks just completed by `expand.grid`, `apply`, and `matrix` in one go. With this approach, you could've alternatively created `xm`, `ym`, and `zm` directly from `zseq` and `theta.rad` by simply calling the following:

```
R> xm <- outer(zseq,theta.rad,function(z,t) r*cos(t))
R> ym <- outer(zseq,theta.rad,function(z,t) r*sin(t))
R> zm <- outer(zseq,theta.rad,function(z,t) z)
```

The only difference here is that the anonymous function provided as the third argument must be explicitly defined in terms of two separate arguments that represent the values of the necessary height and angle parameters.

However you chose to obtain `xm`, `ym`, and `zm`, it's now just a matter of calling `persp3d` with these coordinate matrices. Make an additional call to `points3d` to emphasize the precise evaluation points that are returned in those matrices:

```
R> persp3d(x=xm,y=ym,z=zm,col="red")
R> points3d(x=xm,y=ym,z=zm)
```

The result is shown on the left of Figure 26-21.

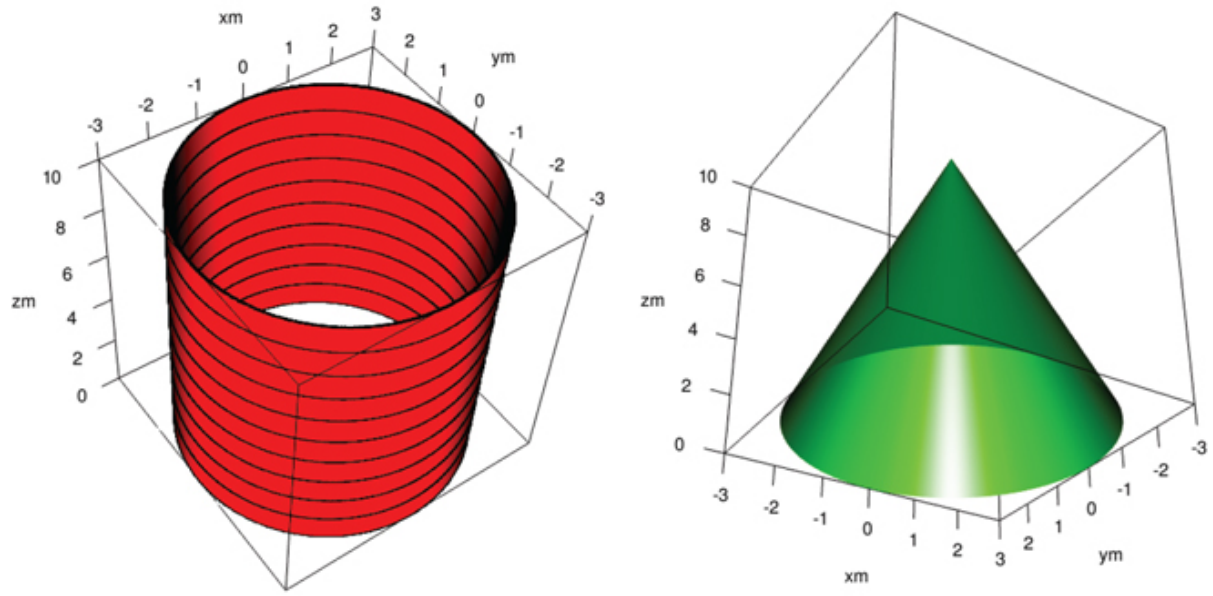


Figure 26-21: Drawing a cylinder and a cone using `persp3d`*, with matrix arguments in all three coordinate directions. The loci are defined by the corresponding parametric equations. The black rings visible on the cylinder represent the actual evaluation points stored in the required matrices* `xm`*,* `ym`*, and* `zm`*.*

3D Cone

Once you understand the procedure involved with setting up the x-, y-, and z-coordinate matrices, you can display virtually any 3D shape or surface with ease. Taking r, h, and θ to be the base radius, maximum height from zero, and angle, respectively, a cone follows these equations:

$$x = \frac{h-z}{h} r \cos(\theta), \quad y = \frac{h-z}{h} r \sin(\theta), \quad \text{and} \quad z = z \quad (26.3)$$

Using the same objects `r`, `h`, `zseq`, and `theta.rad` defined previously for the cylinder example, the following code alters the anonymous functions in the previous use of `outer` to reflect Equation (26.3):

```
R> xm <- outer(zseq,theta,function(z,t) (h-z)/h*r*cos(t))
R> ym <- outer(zseq,theta,function(z,t) (h-z)/h*r*sin(t))
R> zm <- outer(zseq,theta,function(z,t) z)
R> persp3d(x=xm,y=ym,z=zm,col="green")
```

The right of Figure 26-21 shows a screenshot the result.

26.4.2 Mathematical Abstractions

Many areas of mathematics, applied mathematical modeling, and statistics utilize properties of high-dimensional shapes. To round off this chapter, and indeed the book, let's employ `rgl` to take a look at a couple of famous abstractions using skills from Section 26.4.1.

Möbius Strip

A classic example of an abstraction is the *Möbius strip*, a continuous surface that has only one side and one edge. It can be expressed using the following equations:

$$x = F(\nu, \theta)\ \cos\ \theta, \quad x = F(\nu, \theta)\ \sin\ \theta, \quad \text{and} \quad z = \tfrac{\nu}{2} \sin\left(\tfrac{\theta}{2}\right) \qquad (26.4)$$

Here, $-1 \leq v \leq 1$, and the following is true:

$$F(\nu, \theta) = 1 + \tfrac{\nu}{2}\ \cos\left(\tfrac{\theta}{2}\right)$$

The parameter v controls the position of the point along the width of the strip, and θ controls the rotation angle.

You can draw the strip in the same way as the cylinder and cone from earlier. First, set up the sequences over the possible values of v and θ, done here at a resolution of 200 each:

```
R> res <- 200
R> vseq <- seq(-1,1,length=res)
R> theta.rad <- seq(0,2*pi,length=res)
```

Next, use `outer` to obtain the 200×200 matrices in each of the x-, y-, and z-coordinates as per Equation (26.4):

```
R> xm <- outer(vseq,theta.rad,function(v,t) (1+v/2*cos(t/2))*cos(t))
R> ym <- outer(vseq,theta.rad,function(v,t) (1+v/2*cos(t/2))*sin(t))
R> zm <- outer(vseq,theta.red,function(v,t) v/2*sin(t/2))
```

Then, a quick call to `plot3d` from the `rgl` package will show you the 40,000 locations, based on the defined `vseq` and `theta.rad` sequences, that lie on the Möbius strip. The result of the following line

```
R> plot3d(x=xm,y=ym,z=zm)
```

is shown on the left of Figure 26-22 (you may need to enlarge the RGL device to see the individual points more clearly).

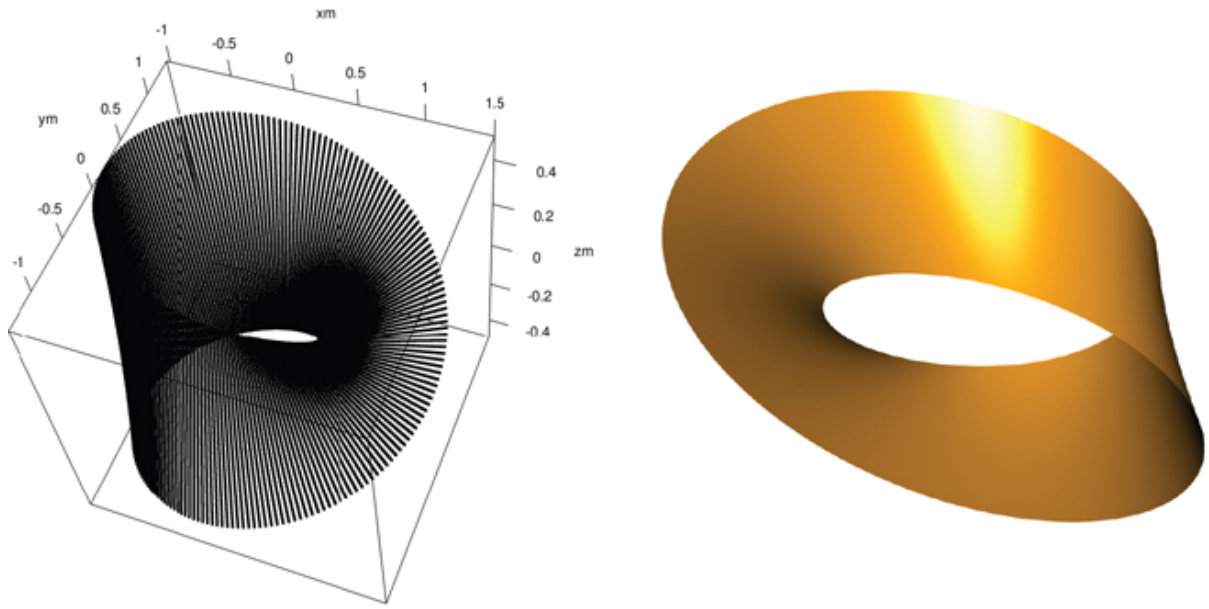


Figure 26-22: Drawing a Möbius strip in R. Left: Specifically calculated points on the strip, visualized with `plot3d`. Right: The surface formed by joining up the points via `persp3d`.

Let's display the strip as a continuous surface using `persp3d` to fully appreciate the one-side/ one-edge phenomenon. The screenshot on the right of Figure 26-22 shows the result of the following code:

```
R> persp3d(x=xm,y=ym,z=zm,col="orange",axes=FALSE,aspect=c(1,1,0.25),
           xlab="",ylab="",zlab="")
```

Note the use of `axes` to suppress the default box and axes, the use of `aspect` to squash the z-axis, and the use of empty strings to remove the default axis titles denoting `xm`, `ym`, and `zm`.

You could also use color in a more interesting way to emphasize the wraparound nature of a Möbius strip. Taking inspiration from a similar collection of colors defined in Section 25.1.3, create the following custom palette:

```
R> patriot.colors <- colorRampPalette(c("red4","red","white","blue",
                                        "white","red","red4"))
```

This palette has been specifically generated to go from a dark red to white to blue, then wrap back around from the blue back to white and the

dark red. This is needed for the way in which colors from `patriot.colors` will be assigned, in a point-wise fashion, to the plotted strip.

The color vector for plotting the surface will need to be of length $200^2 = 40{,}000$ given the preset value of `res` (governing the length of both `vseq` and `theta`). To fill the vector, execute the following:

```
R> patcols <- patriot.colors(2*res-1)
R> stripcols <- rep(NA,res^2)
R> for(i in 0:(res-1)){
+   stripcols[1:res+res*i] <- patcols[1:res+i]
+ }
```

The first line generates exactly 399 colors from `patriot.colors`, and the second sets up a vector of the required length, which will store the assigned colors (`stripcols`). The `for` loop ensures that at the first iteration, the elements `1` through `200` in `stripcols` will be assigned colors `1` through `200` from `patcols`; at the second iteration, the elements `201` through `400` will be assigned colors `2` through `201` from `patcols`, and so on. This gives the colors their wraparound appearance.

To properly understand the `for` loop, first look at the order of the arguments as supplied to the previous calls to `outer` used to produce the initial Möbius strip plot in Figure 26-22. Having `vseq` first and `theta` second implies that each column of 200 in the resulting matrices corresponds to a span from –1 to 1 in v, which refers to moving from one end of one of the lines of points to the other—that is, up across the width of the strip.

By using an index variable `i` from `0` to `199` (inclusive), the loop assigns each consecutive block of 200 elements in `stripcols` (increased at every iteration via `+res*i`) the 200 elements from the 399 `patcols` by collectively moving forward exactly one element (increased at every iteration via `+i`). This changes the color from red to white to blue in the plotted lines of points in the early stages of the loop, but as it progresses, working its way around the strip, the specific span of that palette is incrementally shifted until it goes from blue to white to red as you rotate around to the last few plotted lines of points. The effect is a smooth change in color as you alter both v and θ. You can see the result of the following in Figure 26-23:

```
R> persp3d(x=xm,y=ym,z=zm,col=stripcols,axes=FALSE,aspect=c(1,1,0.25),
           xlab="",ylab="",zlab="")
```

Given it's an `rgl` graphic, you can still rotate and zoom the plot as normal. You can also experiment with different values of `aspect`; changing these axis-specific scalings will have an effect on the overall appearance of the result.

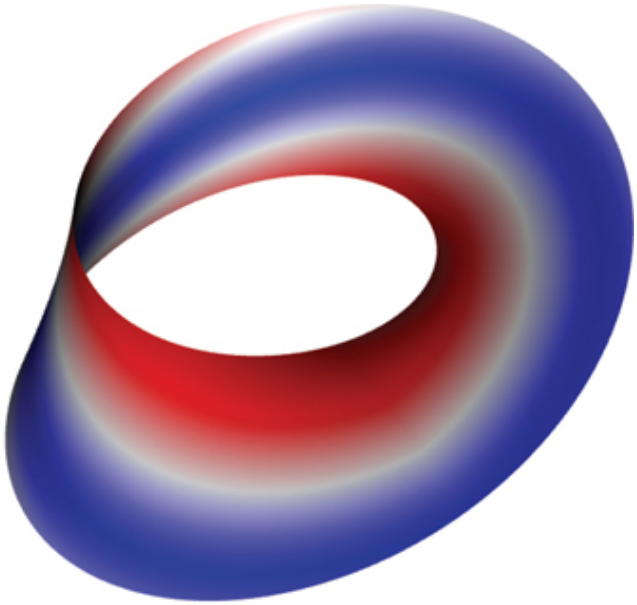


Figure 26-23: A patriotic Möbius strip, created with careful construction of an appropriate color vector.

Torus

Another shape often expressed in 3D space is a *ring torus* (plural *tori*), the classic topological "shape with one hole" that resembles, for lack of a better word, a donut. Such a torus may be parameterized with the following equations:

$$x = F(\theta_2; \alpha, \beta) \cos\ \theta_1, \quad x = F(\theta_2; \alpha, \beta)\ \sin\ \theta_1, \quad \text{and}\ \ z = \alpha\ \sin\ \theta_2 \quad (26.5)$$

Here, $0 \leq \theta_1 < 2\pi$ and the same for θ_2 (assuming angles measured in radians). Furthermore,

$$F(\theta_2;\ \alpha, \beta) = \beta + \alpha \cos\ \theta_2$$

The fixed values α and β control the radius of the "tube" (in other words, the relative thickness of the donut) and the overall size of the torus in terms of distance from the middle of the hole to the middle of the tube. Provided $\alpha < \beta$, the equations in (26.5) give you the classic ring torus shape; you can get different kinds of tori by relaxing that condition on α and β.

Setting $\alpha = 1$ and $\beta = 2$, the following code uses the `theta.rad` object defined earlier for the Möbius strip to compute the matrices in the x-, y-, and z-coordinate directions as per Equation (26.5):

```
R> alpha <- 1
R> beta <- 2
R> xm <- outer(theta.rad,theta.rad,function(t1,t2) (beta+alpha*cos(t2))*cos(t1))
R> ym <- outer(theta.rad,theta.rad,function(t1,t2) (beta+alpha*cos(t2))*sin(t1))
R> zm <- outer(theta.rad,theta.rad,function(t1,t2) alpha*sin(t2))
```

Refer to Section 26.4.1 to remind yourself of the usage of `outer` if necessary.

Then, this line reveals the calculated points of the torus:

```
R> plot3d(x=xm,y=ym,z=zm)
```

And the next line gives you the final appearance of the continuous surface:

```
R> persp3d(x=xm,y=ym,z=zm,col="seagreen4",axes=FALSE,xlab="",ylab="",zlab="")
```

Figure 26-24 shows screenshots of both (if you can't see the individual points for the left-hand plot, try making the RGL device larger).

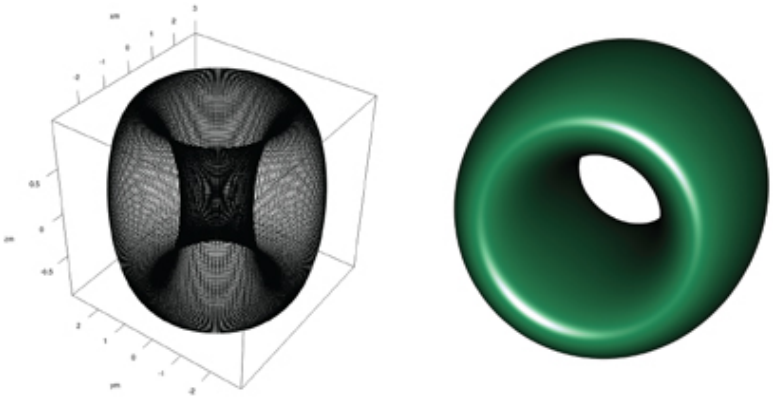


Figure 26-24: Drawing a ring torus in R. Left: Specifically defined points on the surface, visualized with `plot3d`. Right: The shape formed by joining up the points on the left using `persp3d`.

Earlier, you used a specifically constructed color vector to color the Möbius strip, but you can assign color on any such surface by identifying the specific points in the defined matrices that you want to exert control over. Since it's the last example in the book, let's have a little fun with the current mathematical donut to illustrate this point-wise indexing.

First, let's make the dough itself look authentic. The following line sets up the vector of length 200^2 = 40,000 (remember this is appropriate given the length of `theta.rad`, specified as the value of the `res` object at the beginning of this section) to store the colors you're going to use. Initially, every element is set as a dough-colored tan.

```
R> donutcols <- rep("tan",res^2)
```

Next, add some icing. Looking at the distribution of the points, displayed in the plot on the left of Figure 26-24, shows that the top half of plotted locations on the surface of this torus are at z-coordinates greater than zero. With this, you can overwrite the relevant elements of `donutcols` with the following line:

```
R> donutcols[as.vector(zm)>0] <- "pink"
```

Lastly, any premium donut should have sprinkles. You need a mechanism to identify random locations on the top half of the surface and color them appropriately. To do this, you can use the built-in `sample` function to randomly select a subset of elements from an existing vector. If you have the integers 1 through 10, for example, and you want to randomly select four, execute the following:

```
R> sample(x=1:10,size=4)
[1] 8 9 2 6
```

Here, the argument `x` takes the vector from which to select a sample, and `size` takes the number of elements you want to draw from that vector. (Note that you're likely to get a different set of four random numbers when executing this line.)

With that knowledge, use the following code to distribute your sprinkles:

```
R> sprinkles <- c("blue","green","red","violet","yellow")
R> donutcols[sample(x=which(as.vector(zm)>0),size=300)] <- sprinkles
```

This sets up five distinct sprinkle colors; then randomly selects 300 locations, strictly from locations on the iced surface area; and finally assigns to them the five colors. The nature of vector recycling (Section 2.3.4) means there will be exactly 60 of each color sprinkle, randomly placed on the top half of the torus. You can add more sprinkles by increasing `size`, though given the number of colors, you should ensure `size` remains evenly divisible by 5. Again, because `sample` is randomly sampling the sprinkle locations, the specific placement of your own sprinkles will differ.

The following call completes the visual treat:

```
R> persp3d(xm,ym,zm,col=donutcols,aspect=c(1,1,0.4),axes=FALSE,
           xlab="",ylab="",zlab="")
```

The result is shown in Figure 26-25.

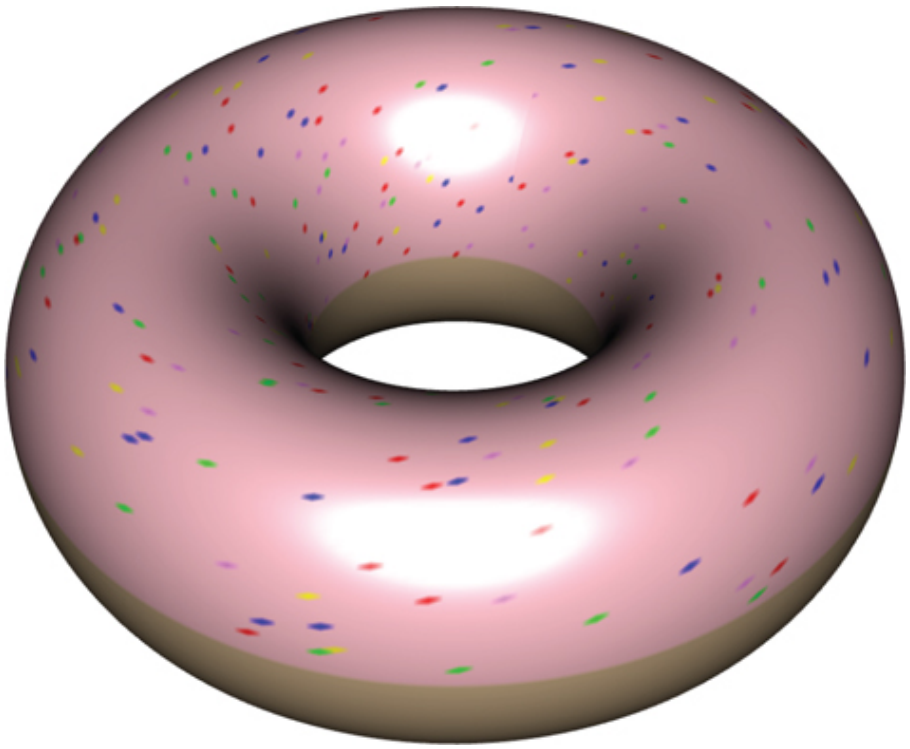


Figure 26-25: A delicious mathematical donut; coloring of the torus surface is achieved by identification of corresponding positions in a color vector and subsequent element replacement.

You can find a more serious (and technical) visualization of a ring torus, in its role as a convenient computational structure used for generating specially defined high-dimensional random normal variates, in Davies & Bryant (2013).

Exercise 26.3

Ensure you have the functionality of the `mvtnorm`, `rgl`, `misc3d`, and `ks` packages available in your current R session. By specifying different covariance matrices, you can control how the different components of a multivariate normal random variable relate to one another, which affects the appearance of the distribution itself. In the standard trivariate normal density, for example, the elements (x, y, z) are independent of one another. Executing the following code will generate 1,000 observations from a non-standard trivariate normal distribution where the three elements are related to one another in a specific way:

```
R> covmat <- matrix(c(1,0.8,0.4,0.8,1,0.6,0.4,0.6,1),3,3)
R> rand3d.norm <- rmvnorm(1000,mean=c(0,0,0),sigma=covmat)
```

Note that the covariance matrix `covmat` is supplied to the optional `sigma` and that the mean of this collection of points remains centered at (0, 0, 0).

a. Plot the generated data as an interactive 3D point cloud, with simple axis titles `"x"`, `"y"`, and `"z"`. The points should form an elliptical shape, in contrast to the spherical shape in the standard trivariate normal in Figures 26-15 and 26-16. Keep the plot open.

b. Using a 50×50×50 evaluation lattice between –3 and 3 in each of the three axes, calculate this particular trivariate normal density function using `dmvnorm` and store it as an appropriately sized array. Note that you'll also need to set `sigma=covmat` in any use of `dmvnorm`. Calculate the maximum value of the density and

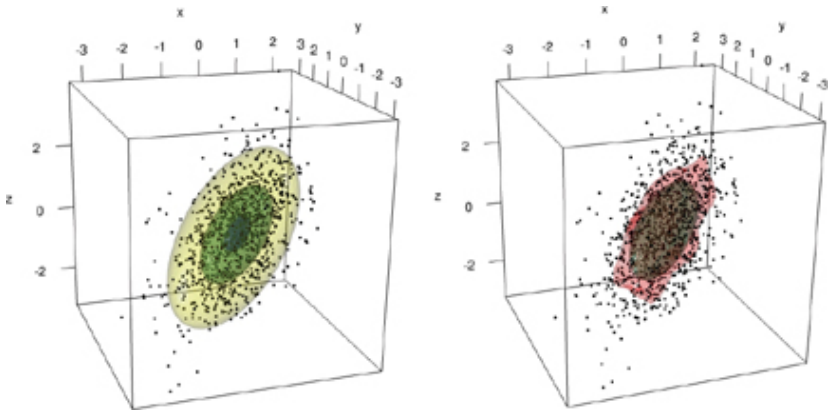
use this to superimpose upon the point cloud isosurfaces at three specific α levels: 0.1, 0.5, and 0.9. Color the isosurfaces `"yellow"`, `"seagreen4"`, and `"navyblue"` and set them at 20 percent, 40 percent, and 60 percent opacity, respectively. You will find the code that led to the production of Figure 26-16 in Section 26.3.2 helpful for this.

c. Now use the `kde` function (from the `ks` package) to calculate a trivariate kernel estimate of the density based on the 1,000 generated observations (follow the example in Section 26.3.3 for assistance). Ensure that the returned object contains the vector of 99 sensible contour levels. Replot the point cloud of (a) in a new RGL device and then make two separate calls to `contour3d` (from the `misc3d` package) as follows, the goal being to compare the shape of the theoretical distribution to that of the kernel-estimated version:

    i. The first call to `contour3d` should superimpose the theoretical contour at an α level of 0.5, that which you obtained in (b). Use the same color and opacity as you did in (b).

    ii. The second call to `contour3d` should draw an isosurface at the 50th percentile as estimated from the point-specific kernel density estimate. Make this one red and reduce the opacity to 20 percent.

    For reference, my results from (b) and (c) are represented by the following screenshots on the left and right, respectively (note that the appearance of your KDE isosurface will vary slightly because of the random generation of the 1,000 data points, which dictate the final estimate).



The `MASS` package has another dataset you've not yet met. The `Boston` data frame object contains a number of descriptive observations concerning house prices in suburbs of Boston, Massachusetts, in the 1970s (Harrison & Rubinfeld 1978). Ensure the `MASS` package is loaded and inspect the help file `?Boston` to learn about the present variables.

d. Identify the variables for average number of rooms, percentage of lower-socioeconomic-status dwellings, and median value. Experiment with the visualization of a 3D scatterplot as follows:

   i. Use `rgl` functionality to plot the three variables, with rooms, status, and value on the x-, y-, and z-axes, respectively; supply tidy axis titles. The data points should be plotted as gray spheres, with `radius=0.4`. Keep the plot open.

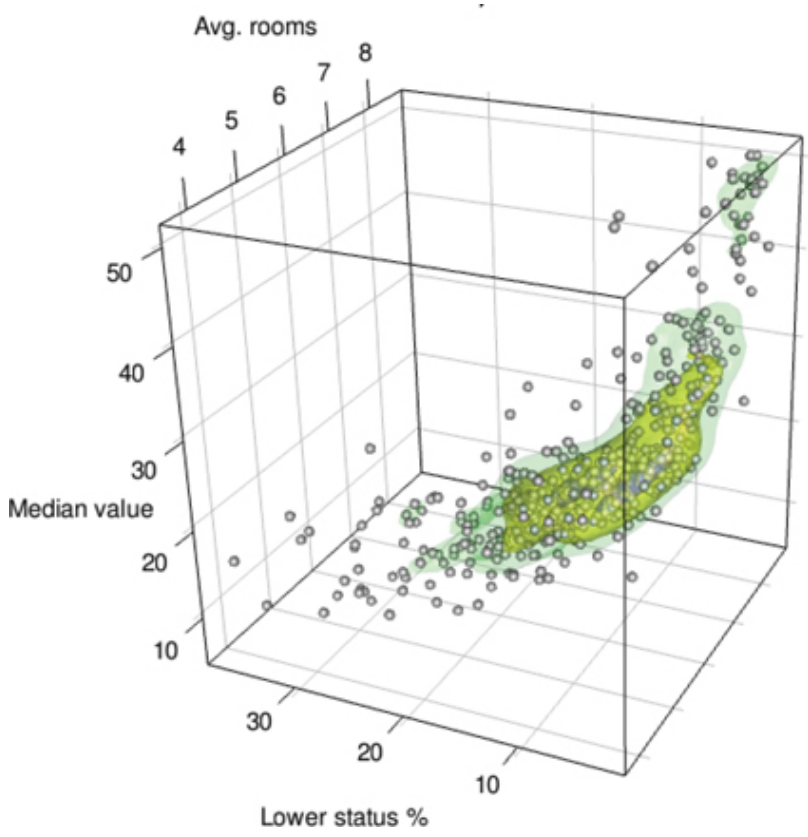
   ii. Use `kde` from the `ks` package to estimate the trivariate density function of these data. Base the estimate on a 64×64×64 evaluation lattice; ensure the 99 integer percentiles of the observation-specific density levels are returned. Superimpose isosurface contours delineating the 75 percent, 50 percent, and 10 percent "most dense" observations using green, yellow, and blue. Set the opacity at 10 percent, 40 percent, and 50 percent, respectively.

   iii. Finally, add reference grids along the three planes identified at the lower z-, upper x-, and upper y-axis

locations.

e. Interpret the final plot from (d). For example, what values of the present variables tend to characterize the most common types of houses in Boston's suburbs?

For reference, here's what my result looks like:

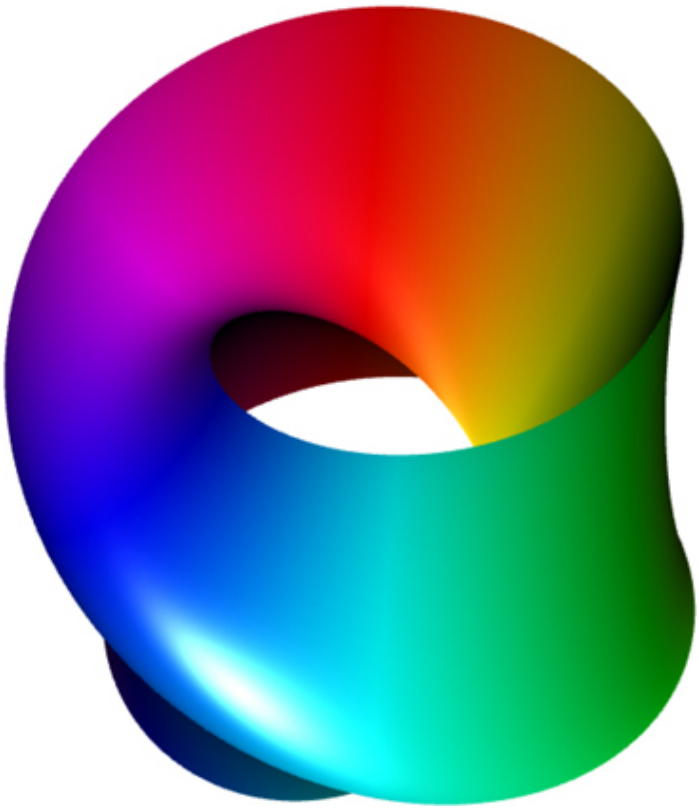


The umbilic torus is another interesting 3D shape in mathematics; a version of this can be defined by the following parametric equations:

$$\begin{aligned} x &= \sin(\theta)\, F(\theta, \phi) \\ y &= \cos(\theta)\, F(\theta, \phi) \\ z &= \sin(\theta/3 - 2\phi) + 2\, \sin(\theta/3 + \phi) \end{aligned}$$

In these equations, $F(\theta, \phi) = 7 + \cos(\theta/3 - 2\phi) + 2\cos(\theta/3 + \phi)$, and you allow for both $-\pi \le \theta \le \pi$ and $-\pi \le \phi \le \pi$. Follow along with the torus example in Section 26.4.2 to help complete this final task.

f. Using the same sequence of length 1,000 for both θ and ϕ (that is, 1,000 values between $-\pi$ and π), plus 1,000 colors generated from the built-in `rainbow` palette simply assigned directly to the `col` argument when calling `persp3d`, produce an interactive 3D plot of the umbilic torus. Suppress the box, axes, and axis titles. In viewing the object from different perspectives, note that, much like the Möbius strip you plotted earlier, this shape has only one edge. Your result should resemble the following:



Important Code in This Chapter

Function/operator	Brief description	First occurrence
`plot3d`	Interactive 3D point cloud	Section 26.1.1, p. 774

Function/operator	Brief description	First occurrence
`legend3d`	Add RGL device legend	Section 26.1.2, p. 776
`bg3d`	Reset RGL device background	Section 26.1.2, p. 777
`pch03d`	Plot classic `pch` characters in 3D	Section 26.1.3, p. 777
`text3d`	Add 3D text	Section 26.1.4, p. 779
`axes3d`	Add 3D axes	Section 26.1.5, p. 780
`lines3d`	Add 3D lines	Section 26.1.5, p. 780
`segments3d`	Add 3D line segments	Section 26.1.5, p. 781
`grid3d`	Add grid to axis plane	Section 26.1.5, p. 781
`title3d`	Add 3D main/axis titles	Section 26.1.5, p. 781
`bgplot3d`	Replot RGL device background	Section 26.1.6, p. 784
`persp3d`	Interactive perspective plot	Section 26.2.1, p. 788
`points3d`	Add 3D points	Section 26.2.2, p. 789
`rmvnorm` (package `mvtnorm`)	Multivariate normal variates	Section 26.3.2, p. 800
`dmvnorm` (package `mvtnorm`)	Multivariate normal density	Section 26.3.2, p. 800
`contour3d` (package `misc3d`)	Draw isosurface	Section 26.3.2, p. 802
`kde` (package `ks`)	Multivariate kernel estimation	Section 26.3.3, p. 806
`pi`	Geometric value π	Section 26.4.1, p. 811

Function/operator	Brief description	First occurrence
`sin`, `cos`	Sine and cosine	Section 26.4.1, p. 811
`outer`	Outer array product	Section 26.4.1, p. 813
`sample`	Random sample from vector	Section 26.4.2, p. 819

A
INSTALLING R AND CONTRIBUTED PACKAGES



This appendix provides more detail on how to install R and its contributed packages. R is available through CRAN (the Comprehensive R Archive Network), which you can access through the R website at *https://www.r-project.org*. I'll cover only the fundamentals here, but you'll find a substantial amount of information in the R FAQ at *https://CRAN.R-project.org/doc/manuals/* (Hornik & the R Core Team 2024). This should be
your first port of call if you need help installing R and its packages. Installation of R itself is dealt with in Section 2 of the FAQ, and installation of add-on packages is covered in Section 5.

A.1 Downloading and Installing R

Once at the R website, click **CRAN mirror** in the welcome text or **CRAN** under the Download heading on the left, as shown in Figure A-1, and a page will load that asks you to select a CRAN mirror site.

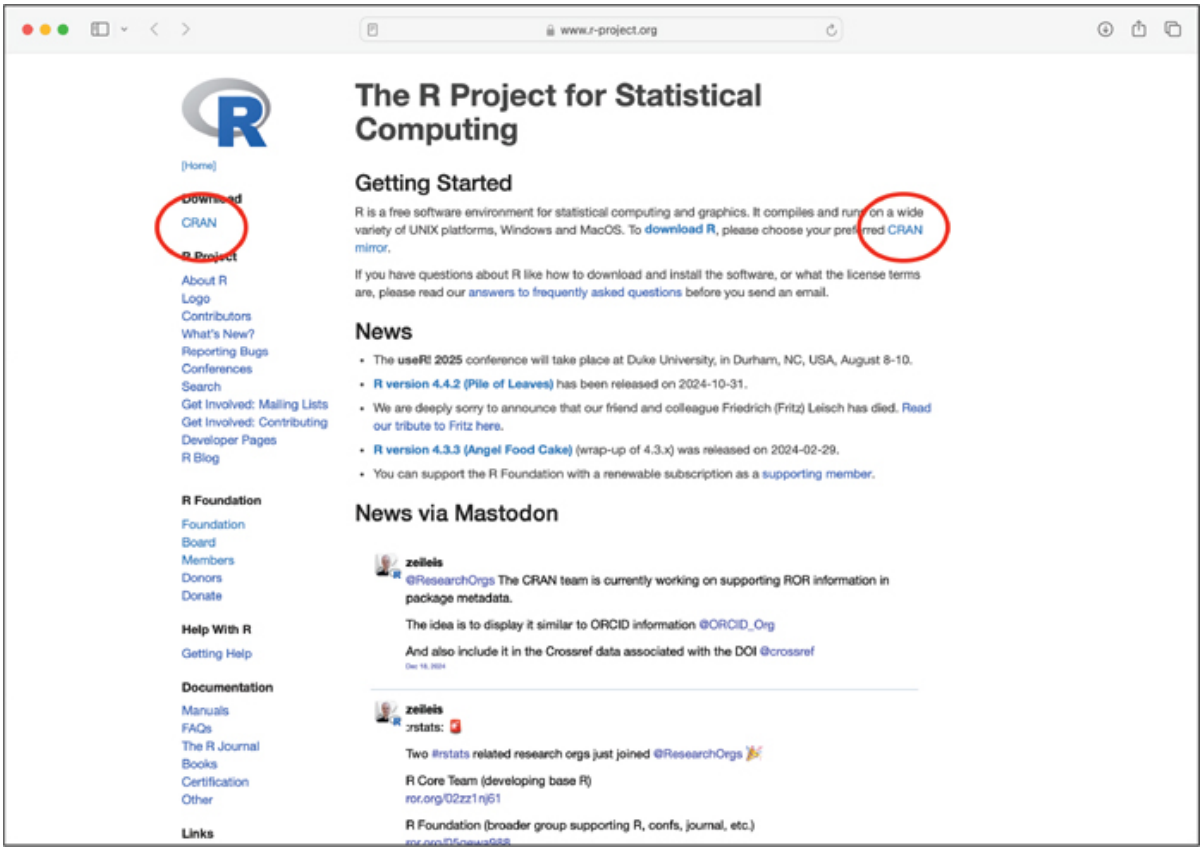


Figure A-1: The R home page

Any mirror will work, but best practice is to pick one close to your geographical location (mine is hosted at the University of Auckland in New Zealand) and click the link. Figure A-2 shows what you can expect to see.

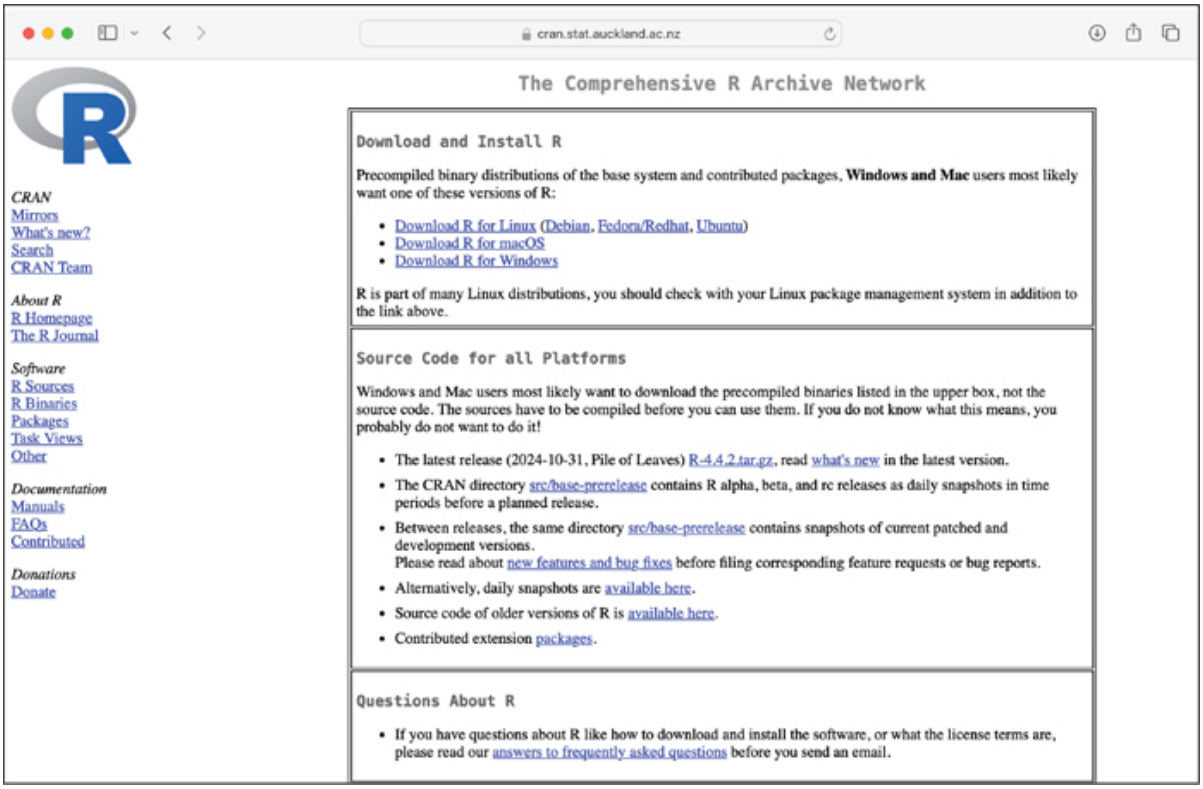


Figure A-2: A CRAN mirror site. This is where you'll find the various download links.

Looking at the topmost section (Download and Install R), identify the link corresponding to your operating system. The procedure to install the program is very much the same you would follow for any other application. The following points offer further OS-specific comments:

- If you're a Windows user, click the Windows link, and from that page choose the installation file (the binary executable) for the `base` distribution. Double-click the executable file and follow the instructions in the installation wizard. Windows users who ultimately wish to write their own R packages will also need to install Rtools, an addon application also listed on this download page.
- If you're a Mac user, click the macOS link and you'll be taken to a page with the packaged binary files. At the time of writing, there are still two versions available: one for older Macs with an Intel CPU, and the other for the newer ARM-based machines (Macs

with Apple's own chips, for example M1, M2, and so on). If you're not sure whether you have Intel or Apple silicon, navigate to the menu bar on your desktop, click ▶ **About This Mac** and look at the entry for Processor or Chip. Download the correct file for your operating system. Once the download is complete, doubleclick it to launch the installer; follow the instructions therein. I also recommend getting the XQuartz window system, freely downloadable from *https://xquartz.org*, which provides support for additional graphics devices.

- Upon clicking the corresponding link, Linux users will be taken to a subdirectory with folders named after operating systems such as Debian or Ubuntu. Click the link relevant to you and you'll be taken to a page providing step-by-step command line instructions for installing R.

A.2 Using Packages

R *packages* (or *libraries*) are collections of code that hold data and functionality used in R. It's essential to get comfortable with loading these libraries to access certain features and commands.

You can think of there being three flavors of R package. Those that make up the core functionality of the software are included with the installation and are automatically loaded when you open R. Also, a typical R installation will include a handful of *recommended* packages that aren't automatically loaded. Finally, a huge collection of user-contributed packages—more than 22,000 at the time of writing—vastly extend R's functionality.

A.2.1 Base Packages

The *base packages* provide the essential syntax and commands for programming, computing, and graphics production, as well as the built-in datasets, basic arithmetic, and statistical functionality, and they are

immediately available when you start R. At the time of writing, there are 14:

```
base       compiler   datasets   grDevices   graphics   grid    methods
parallel   splines    stats      stats4      tlctk      tools   utils
```

You'll find a brief description of each base package in Section 5.1.1 of the aforementioned R FAQ.

A.2.2 Recommended Packages

At the time of writing, there are 15 *recommended packages*, as noted in Section 5.1.2 of the R FAQ. These packages are included with any standard R installation, and they extend the functionality of the base packages to include slightly more specialized (yet still ubiquitous) statistical methods and computational tools. In this book, you'll be using only `MASS` and `boot` from the following list:

```
KernSmooth   MASS      Matrix   boot   class   cluster
codetools    foreign   lattice  mgcv   nlme    nnet
Rpart        spatial   survival
```

These recommended packages *aren't* loaded automatically. To access functions or datasets from these packages, call `library` to load the packages manually. For example, to access the datasets available as part of `MASS`, execute the following at the prompt:

```
R> library("MASS")
```

Some packages provide a short welcome message upon loading, and R always informs you of any masking (see Section 12.3.1) that has occurred.

When you close your current R session, the package will also close, so you'll need to reload it when you want to use it in another instance of R. If you no longer need a package in any given session and want to unload it to, for example, avoid any potential masking issues, use `detach` as follows:

```
R> detach("package:MASS",unload=TRUE)
```

You can find topics and technical details concerning package loading and unloading in Sections 9.1 and 12.3.1 of this book.

A.2.3 Contributed Packages

In addition to the built-in and recommended packages, there's a massive collection of user-contributed packages available through CRAN, serving all kinds of purposes and applications in statistics and mathematics, data science, computing, and graphics. From any CRAN mirror site, the Packages link on the left of the page (see Figure A-2) will take you to a page with links that provide up-to-date lists of all the available packages on CRAN. You'll also find the useful CRAN Task Views web page, which is a collection of subject-specific articles giving an overview of relevant packages, as Figure A-3 shows. This is a great way to get familiar with the types of specialized analyses that are possible in R.

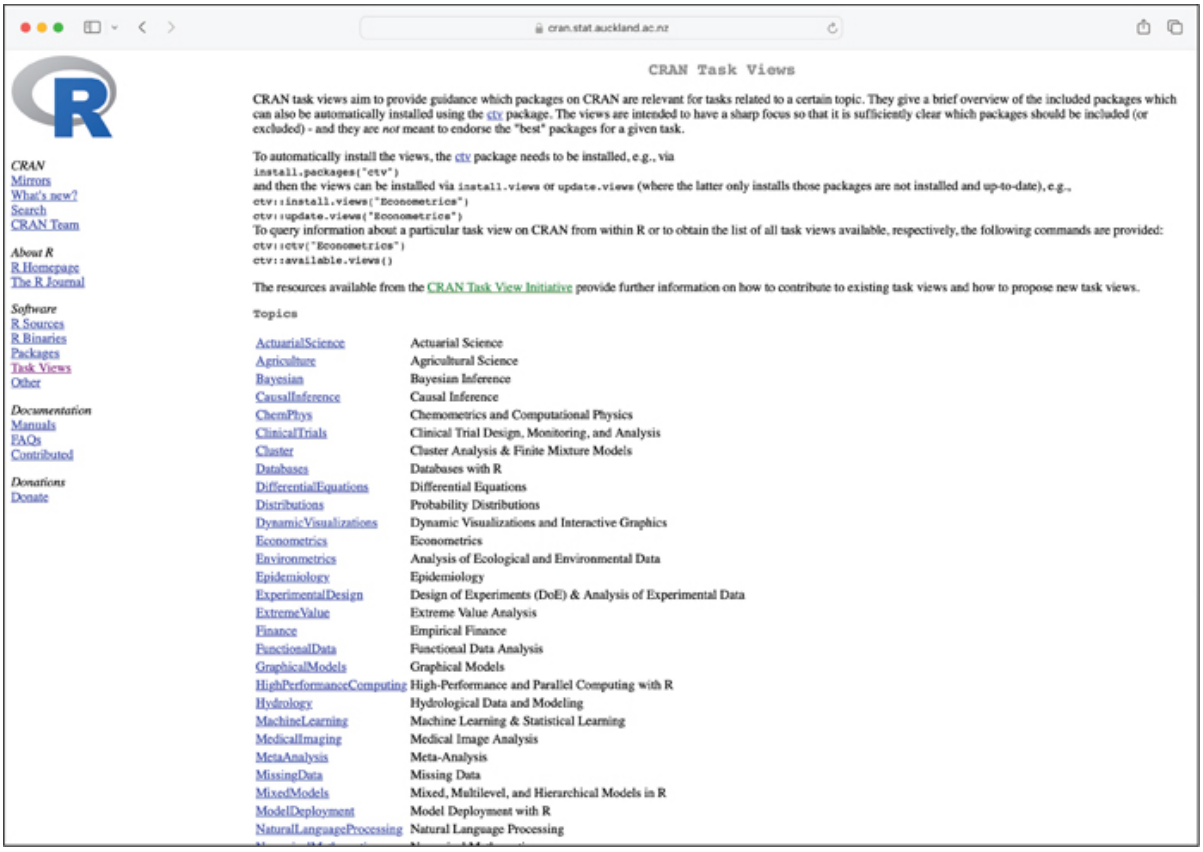


Figure A-3: The CRAN Task Views web page. Each article discusses prominent CRAN packages in use in the field.

Because of the sheer number of available packages, R doesn't include them all upon its installation, and as a researcher you'll only ever be interested in a relatively small subset of methods at any one time.

In this book, you'll make use of a handful of contributed packages and their dependencies. Some are used to access certain datasets or objects, and others are used for their unique functionality or to illustrate statistical methodology. They are listed here:

```
car            faraway   GGally    ggplot2   ggvis
gridExtra      ks        misc3d    mvtnorm   rgl
scatterplot3d  shape     spatstat  tseries   vioplot
```

Accessing any contributed package requires an internet connection to first download and install it. Packages are generally less than a few megabytes in size. Once a package is installed, you then load it with the

usual call to `library` to access the relevant functionality. For the packages listed previously, you'll be prompted to do this when necessary in the relevant parts of the book.

You'll now look at a few ways to perform a package download and installation in the base R application, using the `ks` package (Duong 2007, 2024) as an example. The RStudio editor offers an additional way do this; see Section B.2.2.

NOTE

Contributed R packages tend to be of good quality in terms of correctness, speed and efficiency, and user-friendliness. Although there are fundamental compatibility checks that must be passed before a submitted package is made available on CRAN, passing them isn't in isolation a sign of the overall quality and usability of a given package. You can only really gauge that by using the package, studying its documentation, and seeking out any related publications.

Finding Packages on CRAN

Every R package on CRAN has its own standard web page, providing the direct links to the downloadable files and important information about the package. Locate the package name on one of the lists on CRAN and click it, or do a quick Google search. Using the `ks` package as the example, we'd search for *ks r cran*. Figure A-4 shows the top of the web page for `ks`.

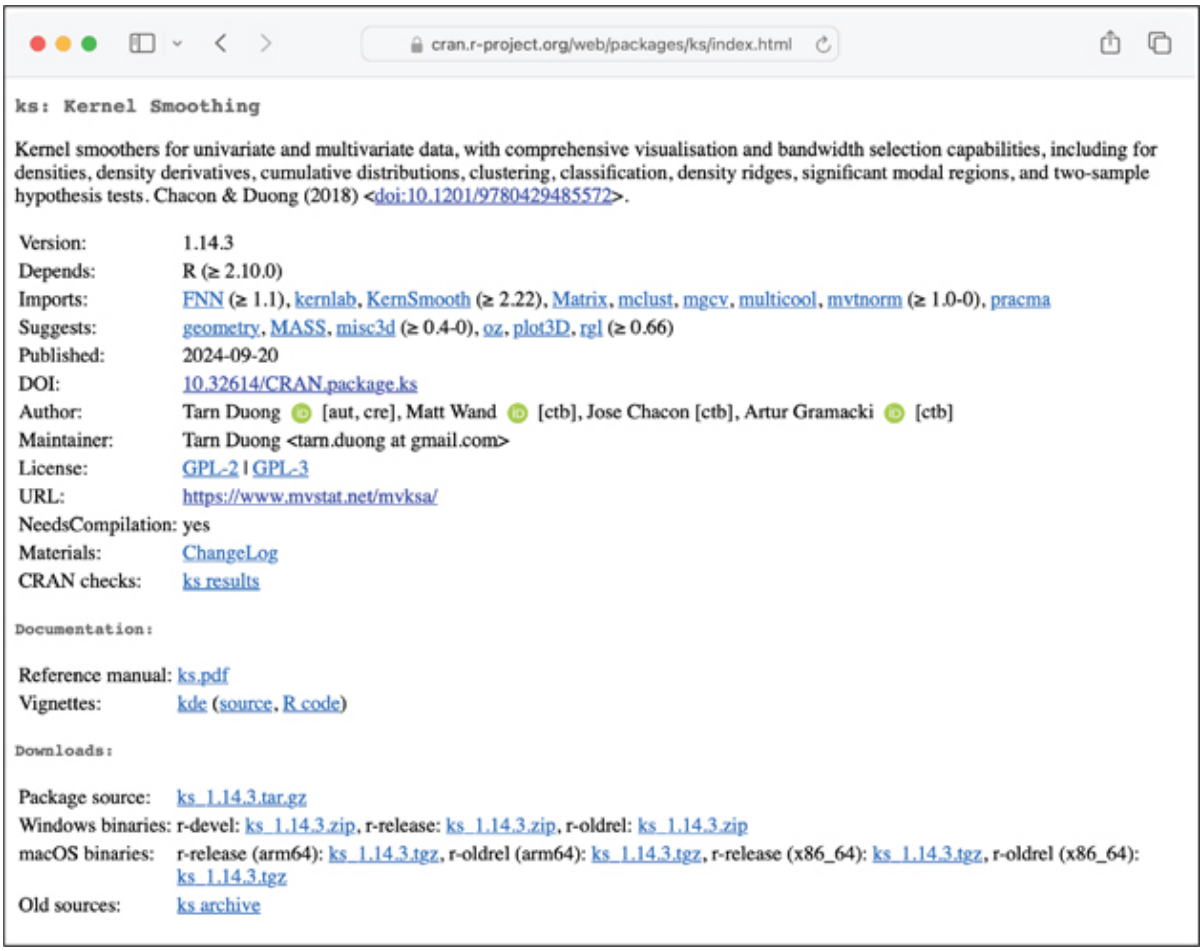


Figure A-4: Descriptive information on the CRAN web page for the `ks` package

Along with basic information such as the version number and the maintainer's name and contact information, you'll see the Depends and the Imports fields. These detail the R version requirement as well as any additional packages that the package you're interested in relies on in order to be successfully installed and used. The Suggests field lists other packages that can enhance or complement the desired package but aren't strictly necessary for successful installation.

Figure A-4 shows that your R version needs to be at least 2.10.0 to install the current version of `ks` at the time of writing. It also shows that `ks` requires other packages be available, including `FNN`, `kernlab`, and `KernSmooth`. The `ks` package also suggests a couple of potentially useful additional packages, such as `misc3d` and `rgl`.

If you install an R package directly from CRAN, as is typical, any required additional packages that you don't already have installed will also be downloaded and installed automatically by default. Complementary packages under the Suggests field are not automatically installed.

Installing Packages at the Prompt

The quickest way to download and install a contributed R package is to use the `install.packages` command directly from the R prompt, being sure to specify the desired package name as a character string (that is, in quotes). Remember that package names are case-sensitive. I see the following on my Mac:

```
R> install.packages("ks")
--- Please select a CRAN mirror for use in this session ---

--snip--
```

Upon executing a call to `install.packages`, you may first be asked to choose a CRAN mirror. I found the New Zealand mirror, as shown in Figure A-5, and clicked Ok. Once you've done this, your selected mirror will remain set as the go-to site until you reset it; see Section A.4.1.

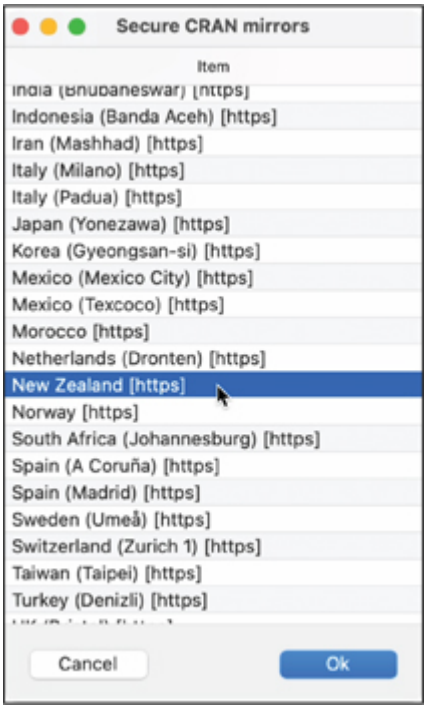


Figure A-5: Selecting my local CRAN mirror

After you click **Ok**, you'll see additional output detailing the download and installation progress of any dependent packages (in the snipped output).

Some additional notes are warranted:

- You need to install a package only once. It will save to your local R installation, available to be loaded with a call to `library` in any subsequent R session.
- You might be prompted to use or create a local folder on your computer to store installed packages. This is done to ensure R knows where to get packages from when they're requested with `library`. Agreeing to this means you have a user-specific library of packages, which is generally a good idea.
- There are a number of optional arguments for `install.packages`; see the help file given by a call to `?install.packages` at the prompt. For example, to specify a CRAN mirror in the console, supply the relevant URL as a character string to the `repos` argument; to

prevent dependencies from installing, use the `dependencies` argument.

- It is generally recommended that you run `update.packages()` (see Section A.3) prior to using `install.packages`. This updates any previously installed package dependencies to their latest versions before you install the target package.
- You can also install R packages "from source"—that is, from the uncompiled code, which may have more recent updates than the pre-compiled binary version of the package. If you're a macOS user, recent versions of R will ask you in the console output whether you'd like to download packages from source for packages with versions available that are more recent than the precompiled binary version. To do so, you need certain command line tools installed on your system; if the download fails, you can stick with the binary version by answering `n` ("no") at the `download from source` prompt.

Installing Packages with the GUI

The basic R graphical user interface (GUI) gives you the option to download and install contributed packages from the console using menu items. Here, you'll look briefly at the Windows and macOS versions.

On Windows, click **Packages ▶ Install package(s) . . .** , shown on the left of Figure A-6. After selecting a CRAN mirror, a tall window will open, listing all available packages in alphabetical order. Scroll to select the package you're interested in. You can see my selection of `ks` on the right of Figure A-6. Click **OK** to download and install the package and any dependencies.

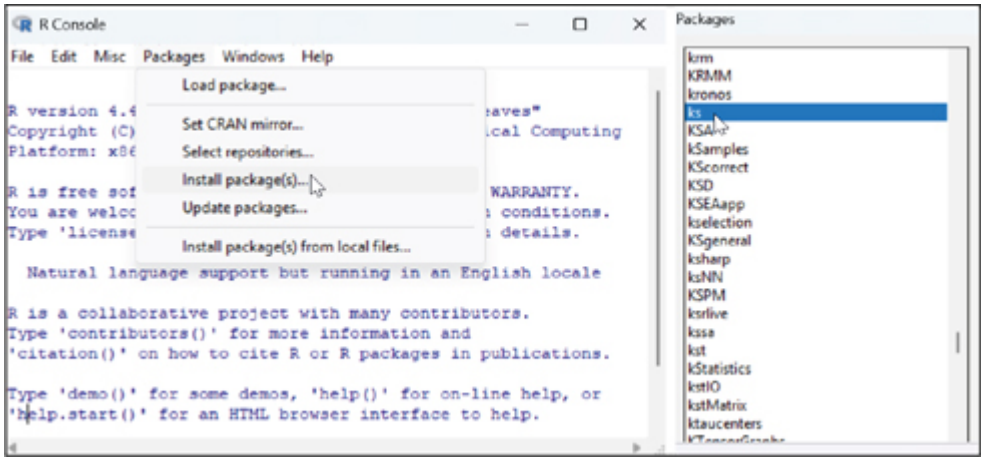


Figure A-6: Initiating download and installation of a contributed R package (and any missing dependencies automatically) via GUI menus in Windows

On a Mac, click **Packages & Data ▶ Package Installer** in the macOS menu bar, as shown on the top of Figure A-7. When the package installer opens, click **Get List** to bring up the table of available packages (you may be asked to select a CRAN mirror if there isn't already a default mirror set). Select the package(s) you want (there's a text-based search option near the top of the window to help with this), being sure to select the **Install Dependencies** box near the bottom of the installer *before* clicking **Install Selected**. This is shown on the bottom of Figure A-7; R will then download and install everything it needs. Note that the options on the bottom left of the installer allow you to choose exactly where the installed packages will be stored; if you're a non-admin user, you may need to create a user-specific library, as mentioned earlier.

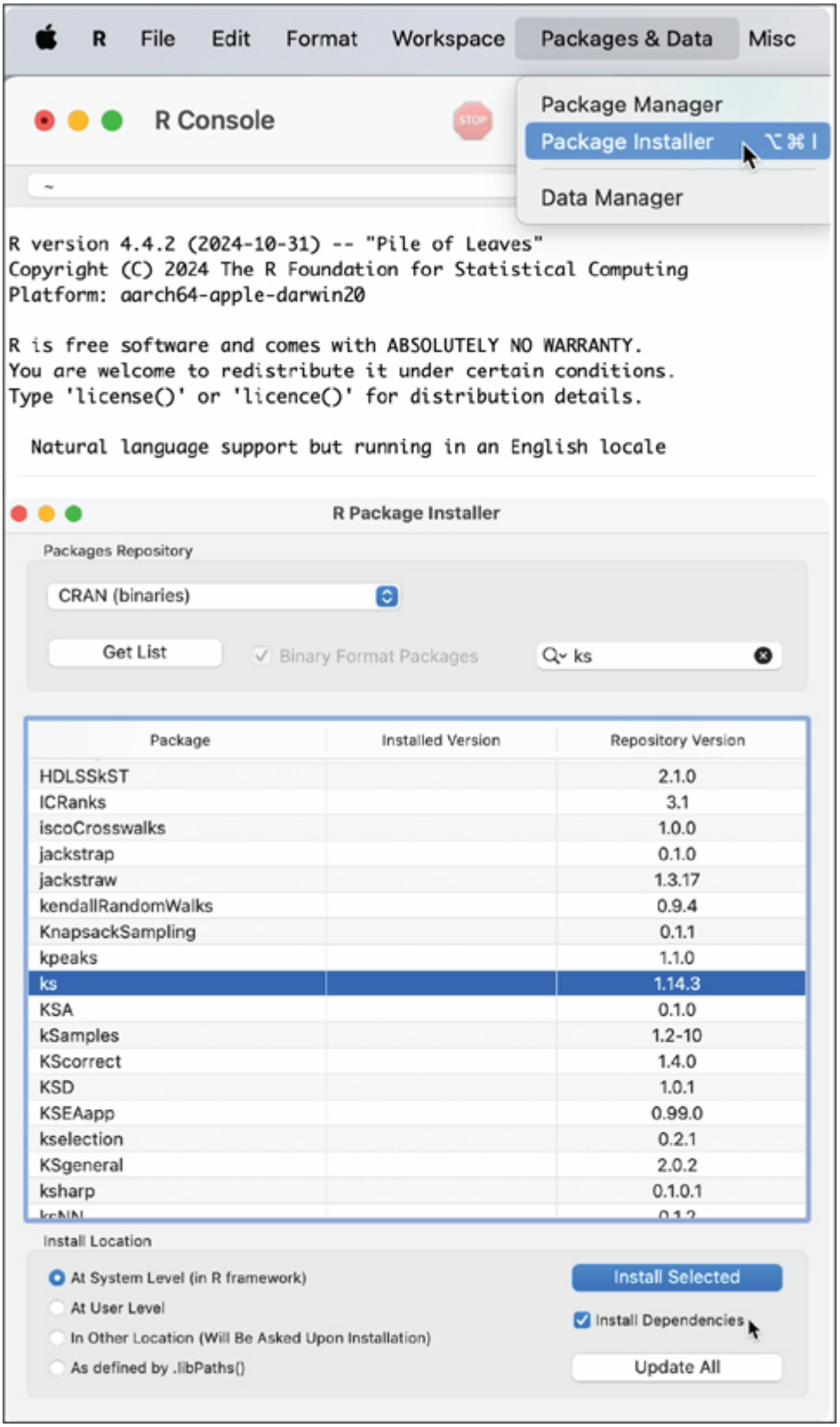
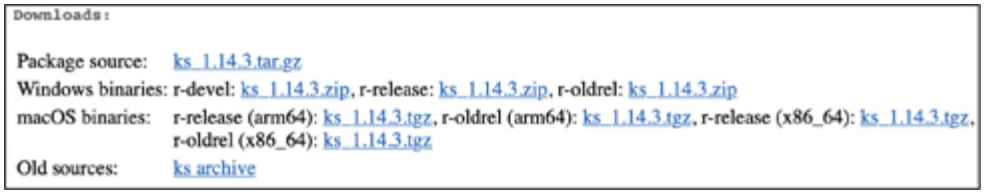

R File Edit Format Workspace Packages & Data Misc
Package Manager
Package Installer ⌥⌘I
Data Manager
R Console
R version 4.4.2 (2024-10-31) -- "Pile of Leaves"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R Package Installer
Packages Repository
CRAN (binaries)
Get List
Binary Format Packages
ks
Package
Installed Version
Repository Version
HDLSSkST 2.1.0
ICRanks 3.1
iscoCrosswalks 1.0.0
jackstrap 0.1.0
jackstraw 1.3.17
kendallRandomWalks 0.9.4
KnapsackSampling 0.1.1
kpeaks 1.1.0
ks 1.14.3
KSA 0.1.0
kSamples 1.2-10
KScorrect 1.4.0
KSD 1.0.1
KSEAapp 0.99.0
kselection 0.2.1
KSgeneral 2.0.2
ksharp 0.1.0.1
Install Location
At System Level (in R framework)
At User Level
In Other Location (Will Be Asked Upon Installation)
As defined by .libPaths()
Install Selected
Install Dependencies
Update All

Figure A-7: Initiating download and installation of a contributed R package (and any missing dependencies) using the GUI-based package installer in macOS

Installing Packages Using Local Files

Installing packages using local files is handy if, for example, you're working with packages in development or you want to share a packaged set of functions that isn't available on CRAN.

To illustrate this, we'll work with the locally saved file of the `ks` package. You can download the package file from CRAN through your internet browser in the same way you'd download anything else, store it on your local drive, and then direct R to install using the target file. On the CRAN web page for `ks`, you'll find a Downloads section (the relevant snippet is shown in Figure A-8). For Linux, choose the *Package source*. For Windows and macOS, choose the corresponding *.zip* or *.tgz* file labeled *r-release* (in the case of macOS, you'll additionally need to choose either the Intel or ARMbased version depending on your hardware). The *r-oldrel* and *r-devel* versions should be used only if you're experiencing compatibility issues. The "Old sources" link holds archived source files for old versions.


Downloads:

Package source:	ks_1.14.3.tar.gz
Windows binaries:	r-devel: ks_1.14.3.zip, r-release: ks_1.14.3.zip, r-oldrel: ks_1.14.3.zip
macOS binaries:	r-release (arm64): ks_1.14.3.tgz, r-oldrel (arm64): ks_1.14.3.tgz, r-release (x86_64): ks_1.14.3.tgz, r-oldrel (x86_64): ks_1.14.3.tgz
Old sources:	ks archive

Figure A-8: The Downloads section of the CRAN web page for `ks`

On Windows, select **Packages ▶ Install package(s) from local files . . .** (visible below the Install package(s) . . . option shown in Figure A-6). This will open a file browser so you can navigate to the downloaded *.zip* file for the package; R will do the rest.

On macOS, once you've downloaded the *.tgz* file, select **Packages & Data ▶ Package Installer**. Use the drop-down menu at the top of the installer to select **Local Binary Package**, as shown in Figure A-9. To open a file browser to find the local file, you need to click **Install . . .** at

the bottom of the installer. R will take it from there (note that this should still work if your downloaded *.tgz* file has been automatically converted to a *.tar* file).

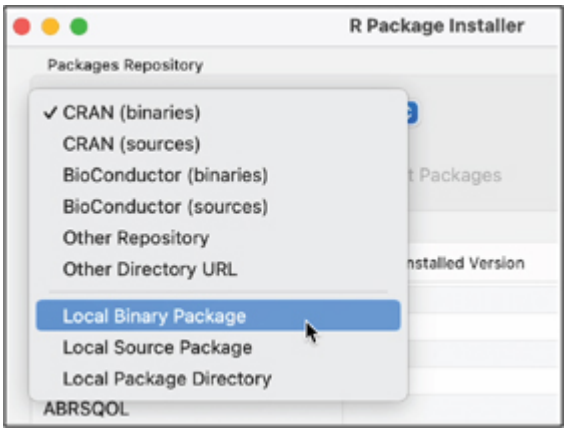


Figure A-9: Using the package installer in the base R GUI on macOS to install an R package from a local file

It's important to know that this method *does not* automatically install any dependencies; it can't because it's operating solely on the locally saved file for the selected package. You can install the package dependencies in the same way, but remember you'll also need to ensure you install any dependencies of those, and so on; be sure to check the Depends and Imports fields on the CRAN package web pages, as noted earlier.

As such, it's far easier to use `install.packages` directly from the R prompt, or through the GUI, to automate this process whenever possible. You need to do local file installations only if the automatic methods fail for some reason, if you're installing a package that isn't on any readily accessible repository like CRAN (see Section A.4.2), or if the package isn't directly available for your operating system and you have some other workarounds in place.

A.3 Updating R and Installed Packages

There are roughly four new releases of R per year that address functionality, compatibility issues, and bug fixes. It's a good idea to keep

up-to-date with these new releases. The R project home page and any CRAN mirror site will tell you the latest release, and you can execute `news()` at the R prompt for details on what's new.

Contributed R packages are also frequently updated and new package files uploaded to CRAN. Packages you've installed won't auto-update, so it's up to you to update them. It's difficult to predict how often such updates will be released, since it's completely up to the maintainers, but it's worth checking for new versions of the packages you've installed every few months or, at the very least, when you upgrade your version of R.

Checking for package updates is easy. A simple call to `update.packages()` at the prompt, with no arguments, will systematically look through your installed packages and flag any that have more recent versions available.

For example, on my current installation, executing the following tells me that a later version of `MASS` is available (along with several other packages not shown in this snipped output):

```
R> update.packages()

--snip--

MASS :
 Version 7.3-61 installed in
    /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/library
 Version 7.3-64 available at https://cran.stat.auckland.ac.nz
Update? (Yes/no/cancel) Y

--snip--
```

Enter `Y` to download the updated package from CRAN. If there are several packages with updates available, R will ask you one at a time whether you'd like to update, and you have to enter `Y` (or `n` or `c`) for each.

You can also perform package updates using the R GUI menus (or manually with local file installation). In Windows, select **Packages ▶ Update packages . . .** to open a list of available updates to your current packages. On macOS, a column in the populated table in the package installer (which appears after clicking **Get List**) provides information on the version of every package you currently have installed, as well as the version currently on CRAN, giving you the option to install the more recent version. There's also the Update All button, which is what you'd typically use.

A.4 Using Other Mirrors and Repositories

You have several options if you find yourself wanting to change the CRAN mirror associated with your typical package installation process or, indeed, change the target repository itself to one other than CRAN.

A.4.1 Switching CRAN Mirror

You'll rarely need to change your CRAN mirror, but you might want to if, say, your usual mirror site is inaccessible for some reason or you want to use R from a different location. To query your currently set repository, call `getOption` with `"repos"`:

```
R> getOption("repos")
                              CRAN
"https://cran.stat.auckland.ac.nz"
```

To change this to, say, the mirror at the University of Melbourne, simply assign the new URL to the `repos` component in a call to `options` as follows:

```
R> options(repos="https://cran.ms.unimelb.edu.au")
```

Any subsequent usage of `install.packages` or `update.packages` will now use this Australian mirror for downloads.

A.4.2 Reviewing Other Package Repositories

CRAN isn't the only repository of R packages. There are several others, including Bioconductor (see *https://www.bioconductor.org*), Omegahat (see *https://www.omegahat.net*), and R-Forge (see *https://www.rforge.net*), and there are several more. These repositories tend to deal with different subjects. Bioconductor, for example, hosts packages that deal with DNA microarray and other genomic analysis methodologies; Omegahat hosts packages that focus on web- and Java-based applications.

In terms of general statistical analyses, CRAN is the go-to repository for most users. To learn more about the other repositories, visit the associated websites.

A.5 Citing and Writing Packages

It's important that the work put into R and its packages is appropriately recognized when the software is used in, for example, data analysis as part of your research projects. Indeed, when you get to the stage where you're thinking about writing your own packages, it pays to be aware of the following.

A.5.1 Citing R and Contributed Packages

For citing R or its packages, the `citation` command produces the relevant output:

```
R> citation()
To cite R in publications use:

  R Core Team (2024). _R: A Language and Environment for Statistical
  Computing_. R Foundation for Statistical Computing, Vienna, Austria.
  <https://www.R-project.org/>.

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {R: A Language and Environment for Statistical Computing},
    author = {{R Core Team}},
```

```
    organization = {R Foundation for Statistical Computing},
    address = {Vienna, Austria},
    year = {2024},
    url = {https://www.R-project.org/},
  }

We have invested a lot of time and effort in creating R, please cite it
when using it for data analysis. See also 'citation("pkgname")' for
citing R packages.
```

Note that LATEX users are conveniently catered to via automatically generated BIBTEX entries.

You can also cite individual packages if these have been instrumental in completing a particular piece of work. Here's an example:

```
R> citation("MASS")
To cite the MASS package in publications use:

  Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics
  with S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0

A BibTeX entry for LaTeX users is

  @Book{,
    title = {Modern Applied Statistics with S},
    author = {W. N. Venables and B. D. Ripley},
    publisher = {Springer},
    edition = {Fourth},
    address = {New York},
    year = {2002},
    note = {ISBN 0-387-95457-0},
    url = {https://www.stats.ox.ac.uk/pub/MASS4/},
  }
```

A.5.2 Writing Your Own Packages

Once you become an R expert, you may find yourself with a suite of functions, datasets, and objects that others might find useful or that you

use often enough to warrant packaging them up in a standardized, easily loadable format. There's certainly no obligation to submit your packages to CRAN or any other repository, but if you do aim to do so, note that there are rather strict requirements to assure users of the stability and compatibility of your package.

If you're interested in constructing your own installable R package, see the official *Writing R Extensions* manual, accessible on any CRAN mirror site by clicking the Manuals link under Documentation on the left of the home page (see Figure A-2). You may also want to seek out the book by Wickham & Bryan (2023), which provides useful instruction on the R package-writing process and associated dos and don'ts.

B
WORKING WITH RSTUDIO



Although the base R application is all you typically need to unleash the full suite of available functionality, the bare-bones appearance of the console and code editor can be off-putting to some, especially beginners. One of the best integrated development environments (IDEs) designed specifically to enhance day-to-day use of the R language is RStudio, developed by Posit, a commercial software and data science company (Posit team 2024). This appendix provides an overview of RStudio and its most commonly used tools.

Like R itself, you can use the desktop version of RStudio on Windows, macOS, and Linux systems. It is free to download and use as an individual, with paid business/enterprise versions also available. Before installing RStudio, you must first have the base R language installed, as described in Appendix A (macOS users will also want XQuartz; see Section A.1). Then, you can download RStudio from the official website at *https://posit.co/downloads/*.

The RStudio website also hosts a variety of useful support articles and links, as well as instructions for various special enhancements, some

of which are noted in Section B.2. If you need help with RStudio, the user guide at *https://docs.posit.co/ide/user/* should be your first port of call.

B.1 Basic Layout and Usage

The RStudio IDE is split into four panes, and you can customize the content and layout to suit your preferences. When you first open it, you'll only see the R console; the editor itself will appear when you open or create an R script. Figure B-1 shows my setup, which differs slightly from the default appearance. In it, I'm playing with the `ggplot2` faceting examples from Section 24.3.2, using the book's source code file for Part V.

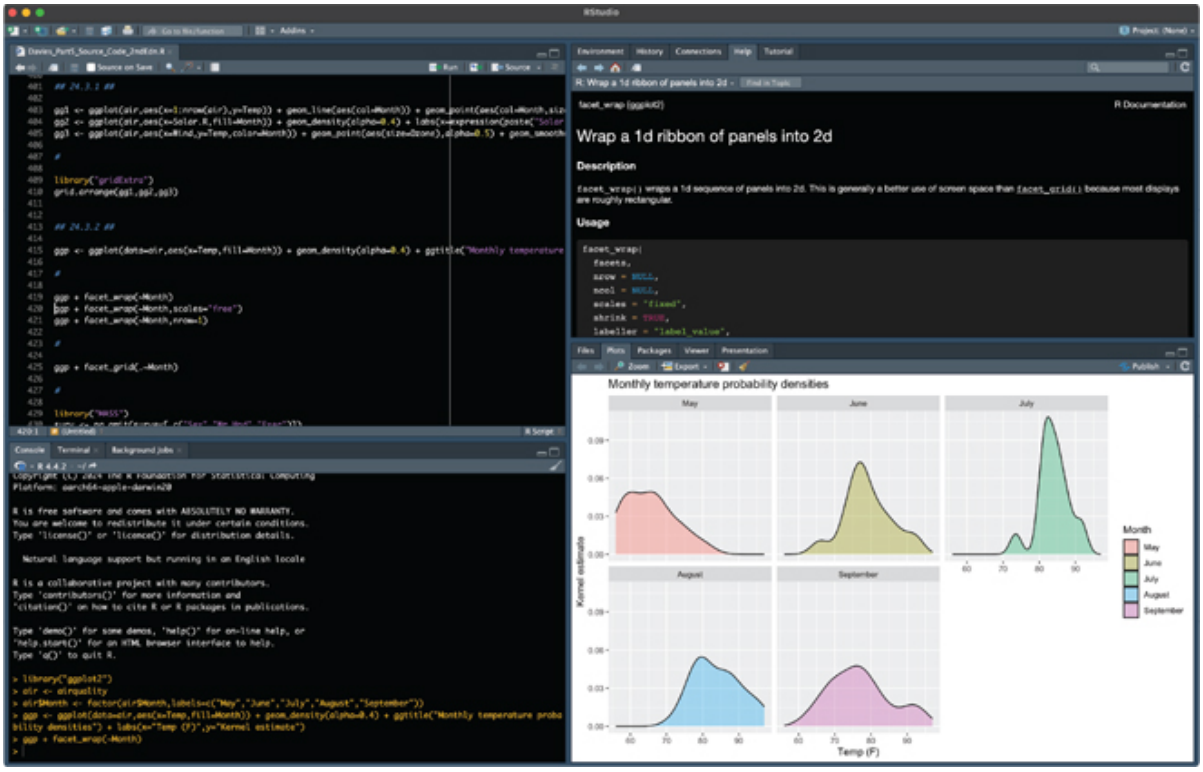


Figure B-1: RStudio in action. The four panes can be arranged and suppressed as you like; here, you can see the code editor (top left), the console (bottom left), the help pages (top right), and the graphics viewer (bottom right). The panes on the right also have additional tabs to choose from.

You write R code in the built-in editor and execute it in the console; the shortcut "send code to console" keystrokes are CTRL-ENTER in

Windows and ⌘-RETURN on a Mac. The textual output appears in the console as usual.

B.1.1 Editor Features and Appearance Options

One of the most useful features of the RStudio editor is the color-themed code highlighting and bracket matching. This makes for an easier coding experience than in the base R editor, particularly when writing long lines of code. There are also autocomplete options that pop up as you're typing in either the editor or the console. You can see an example of this in Figure B-2.



Figure B-2: RStudio's autocomplete feature includes hints about each option.

These features and more can be enabled, disabled, and customized using the RStudio options (select **Tools ▶ Global Options . . .** on both Windows and macOS); you can see the Code and Appearance sections of these options in Figure B-3. In the right-hand image, you can see my selection of a dark visual theme; while the default RStudio installation comes with a selection of built-in themes, you can also find a wealth of downloadable options on the internet, using **Add . . .** to add them to the list.

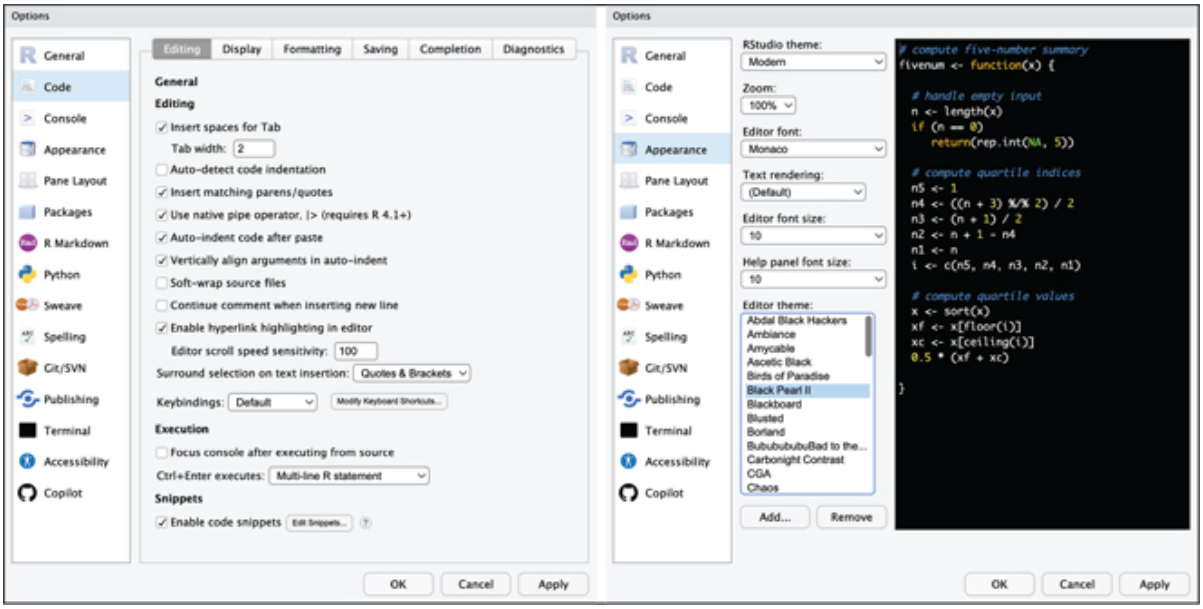


Figure B-3: The options panes for code editing (left) and appearance (right)

B.1.2 Pane Customization

Customizing the arrangement and content of the four RStudio panes to suit your own preferences is easy. You'll always have a pane for code editing (also called the *source*) and another for the console, but you can set up a number of additional tabs to be displayed on the two remaining utility panes. These include a file browser that you can use to search for and open R scripts on your local machine, plot and document viewers, standard R function help files, and a package installer.

You can configure your utility panes with the drop-down menus and checkboxes in the Pane Layout section of the RStudio options. Figure B-4 shows my current settings; I've made a few changes, including moving the help files to the top right utility pane, with the graphical displays underneath, since I often want to refer to function documentation while experimenting with my plots.

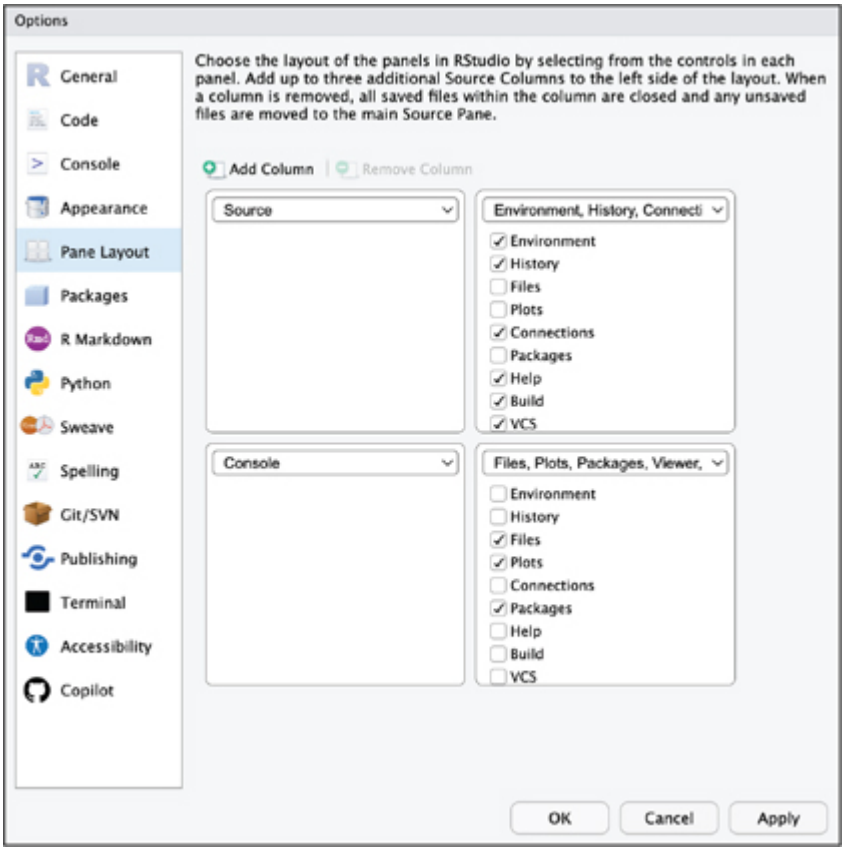


Figure B-4: Pane arrangement and content options

B.2 Auxiliary Tools and Features

RStudio gives you access to a number of handy tools to use in conjunction with R, some of which I'll briefly highlight here. For more information on a particular feature, check the supporting documentation at *https://docs.posit.co/ide/user/*.

B.2.1 Projects

RStudio *projects* assist with development and file management when you're working on more complicated endeavors. In these situations, you're typically working with multiple script files, you might want to save separate R workspaces, or you might have certain R `options` set to specific or nondefault values. RStudio facilitates this process so you don't have to set all of that up with each new R session.

At the top right of the RStudio window, you'll see a Project: (None) button. Clicking **Project: (None) ▶ New Project . . .** (or alternatively, **File ▶ New Project . . .**) will open the New Project Wizard; as shown in Figure B-5, you can choose to generate a new folder for the project or create the project in an existing folder. There are a handful of different project templates to choose from based on what you're aiming to do; the basic, empty project template is given simply as **New Project**. If generating a new folder at the same time, you'll also be asked to name it (if you choose to create a project in an existing folder, this automatically names the project after the existing folder name).

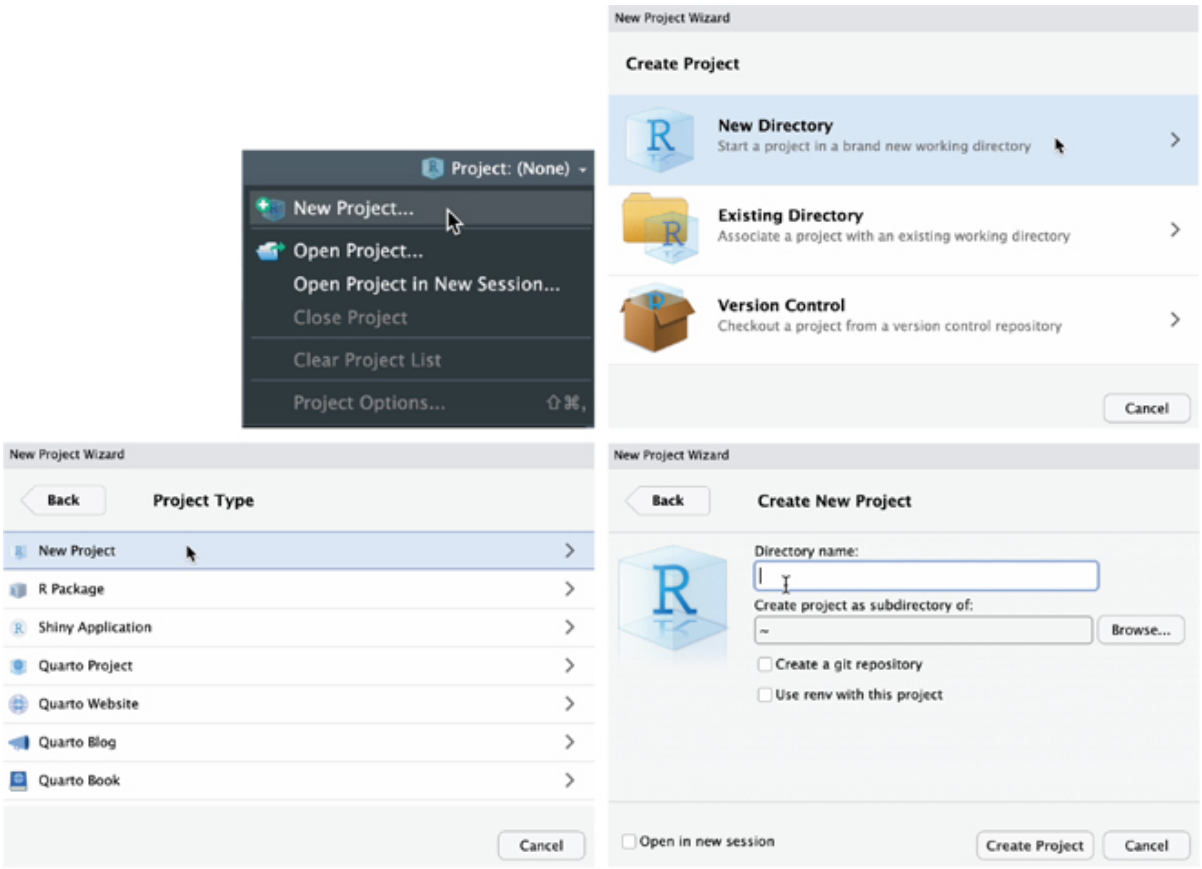


Figure B-5: RStudio project menus; setting up an empty project in a new directory

Creating a new R project using the basic template does the following useful things:

- Sets the working directory as the project folder

- Saves, by default, the R workspace, history, and all *.R* source files in said folder
- Creates a *.Rproj* file, which can be used to open a saved project at a later date, and stores RStudio options set specifically for that project

The other types of project template do similar things. For example, choosing **R Package** as the template will prepare the specific folder structure required by R packages. When you're working on a specific project, its name will appear in place of "(None)" on the **Project: (None)** button.

B.2.2 Package Manager

RStudio provides a package manager to facilitate the downloading, installation, updating, and loading of contributed packages. You'll find it in the Packages tab on your chosen utility pane. It lists only those packages that you currently have installed, along with their version numbers, and you can tick the checkbox next to each package name to load it (equivalent to using `library` at the R console prompt) or untick the checkbox to unload it (equivalent to using `detach`).

Mine appears in Figure B-6. I've just selected the box for the `car` package, which automatically executes the relevant call to `library` in the console. Notice that its dependency `carData` is automatically loaded as well (and its checkbox becomes selected).

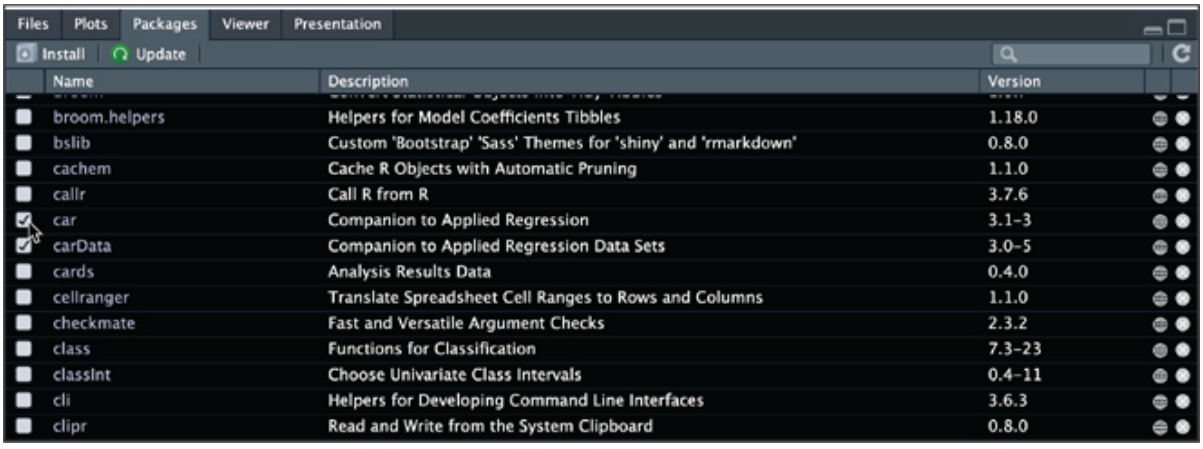


Figure B-6: The RStudio package manager, showing `car` being loaded by clicking its checkbox

The figure also shows the Install and Update buttons for packages. To install a new package, click **Install** and start typing the name of the package you want in the field. RStudio will give you options as you type, as shown for the `ks` package on the left of Figure B-7. Ensure **Install dependencies** is selected in order to automatically install any additionally required packages.

To update packages, click **Update** to open the dialog box on the right of Figure B-7; here, you can choose to update either individual packages or all of them by clicking **Select All**.

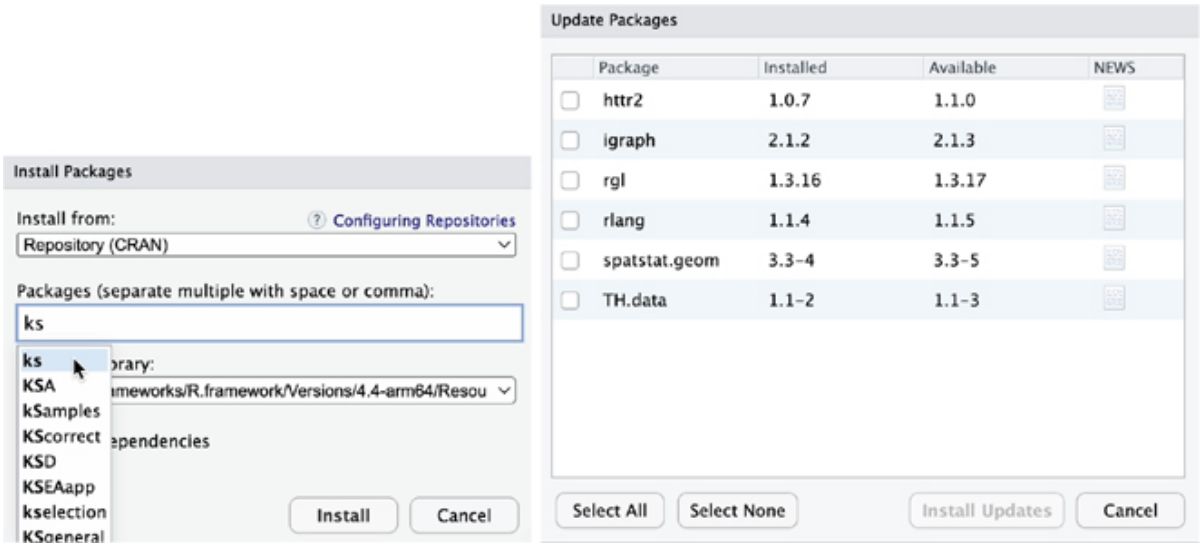


Figure B-7: Package installation and update features in RStudio

Naturally, you can still use the `install.packages`, `update.packages`, and `library` commands directly from the console prompt within RStudio if you prefer. RStudio uses its own CRAN mirror by default, which generally provides good performance no matter where you are. Go to the Packages section of Tools ▶ Global Options . . . to change this.

B.2.3 Support for Profiling

In Section 12.2.3, we examined the R profiler, a tool for measuring execution time and investigating computational bottlenecks in your

code. This requires you to first invoke the profiling tool (via the `Rprof` command), execute your function or code, and terminate the profiler. You then need to use the `summaryRprof` command to interrogate the results, which in their most basic form are presented as tables of output in the R console.

RStudio makes use of the R profiler very convenient, combining it with the for-purpose visualization tools in the `profvis` package (Wickham et al. 2024). Use **Profile ▶ Profile Selected Line(s)** to automatically generate a summary of the profiler's report.

Figure B-8 shows an example using a highlighted code chunk from the original illustration of the R profiler in Section 12.2.3 (see the source code file for Part II of this book). Shown is the menu selection, the graphical byline summary of computational expense (the Flame Graph tab), and the table of results in the call stack of functions (the Data tab).

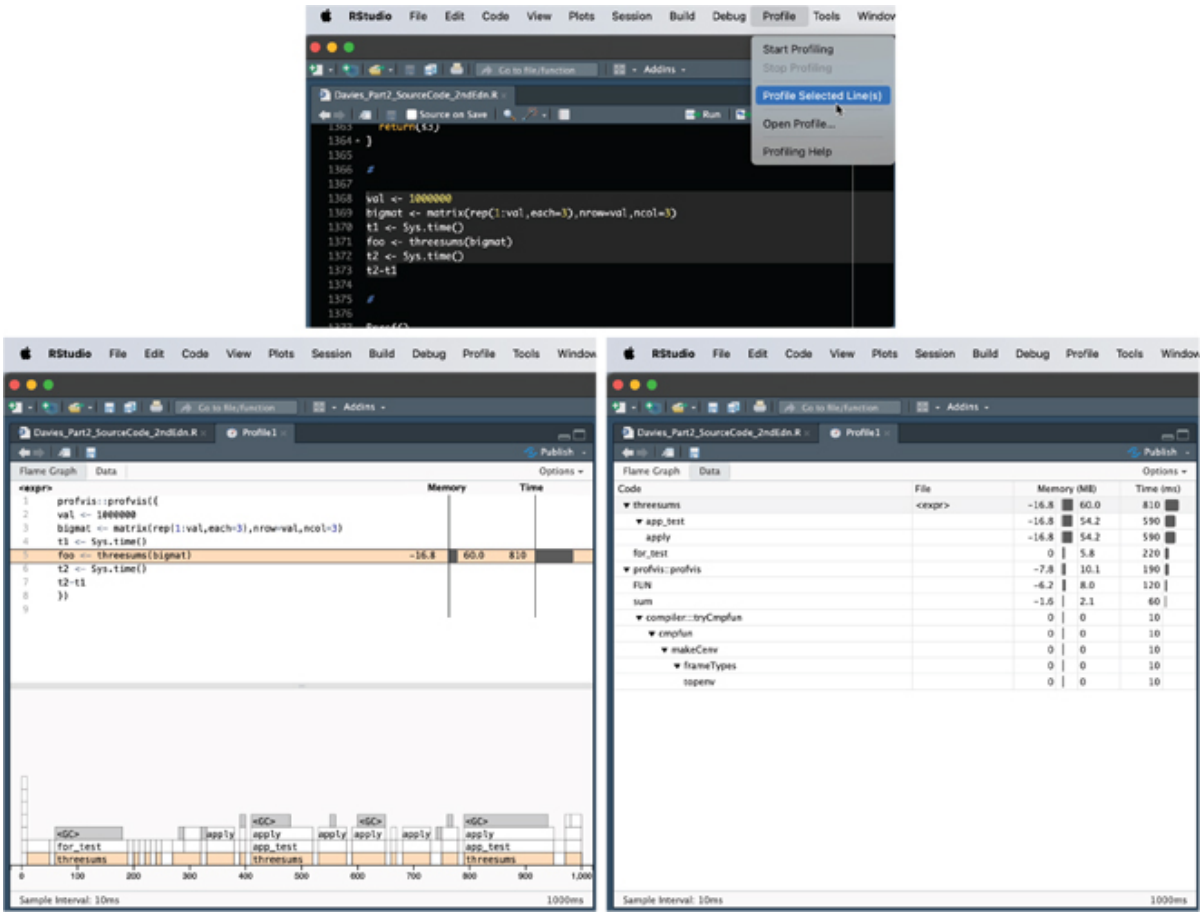


Figure B-8: Using the R profiler in RStudio

B.2.4 Support for Debugging

RStudio also has built-in tools for code debugging. Debugging strategies usually involve being able to "pause" your code at a specific point to inspect your objects and function values in a given "live" state. Specific techniques are best left to more advanced texts such as *The Art of Debugging* (Matloff & Salzman 2008) and *The Art of R Programming* (Matloff 2011); see also Chapter 22 of *Advanced R* (Wickham 2019). I mention it here since the tools available in RStudio provide more convenient, higher-level support for debugging than base R commands alone.

You might like to learn more once you've started writing programs composed of multiple interlinked R functions. With respect to R and RStudio in particular, see the introductory article by Jonathan McPherson on the support website at *https://support.posit.co/hc/en-us/articles/205612627-Debugging-with-the-RStudio-IDE*.

B.3 Markup, Document, and Graphics Tools

When writing up reports on a project or tutorials for particular analyses, researchers often use a markup language. One of the best-known markup languages, particularly in the sciences, is LATEX; it facilitates a unified approach to typesetting, formatting, and layout of technical documents. There are specialized packages that incorporate R code into the compilation of these documents, as well as a growing set of other markup languages used for much the same end goal. These are in turn incorporated into RStudio, allowing you to create dynamic documents that make use of R code and graphics without switching between different applications.

You'll need a TEX installation on your computer to get the most out of the following tools, which you can find at *https://www.latex-project.org*.

B.3.1 Sweave

Sweave (Leisch 2002) was arguably the first markup language to become popular with R; its functionality is included with any standard R installation. Sweave follows typical LATEX markup rules; in your document, you declare special fields called *chunks*, in which you write R code and instruct any corresponding output to be displayed; the output can include both console text and graphics. When you compile the Sweave file (which has a *.Rnw* extension), the R code fields are sent to R for live evaluation, with the results appearing in the specified places of the finished product.

To start a new document, choose **File ▶ New File ▶ R Sweave**, as shown in Figure B-9. At the time of writing, you can find the Sweave user manual at *https://stat.ethz.ch/R-manual/R-devel/library/utils/doc/Sweave.pdf*, and there are a number of tutorials and examples that can be found around the internet.

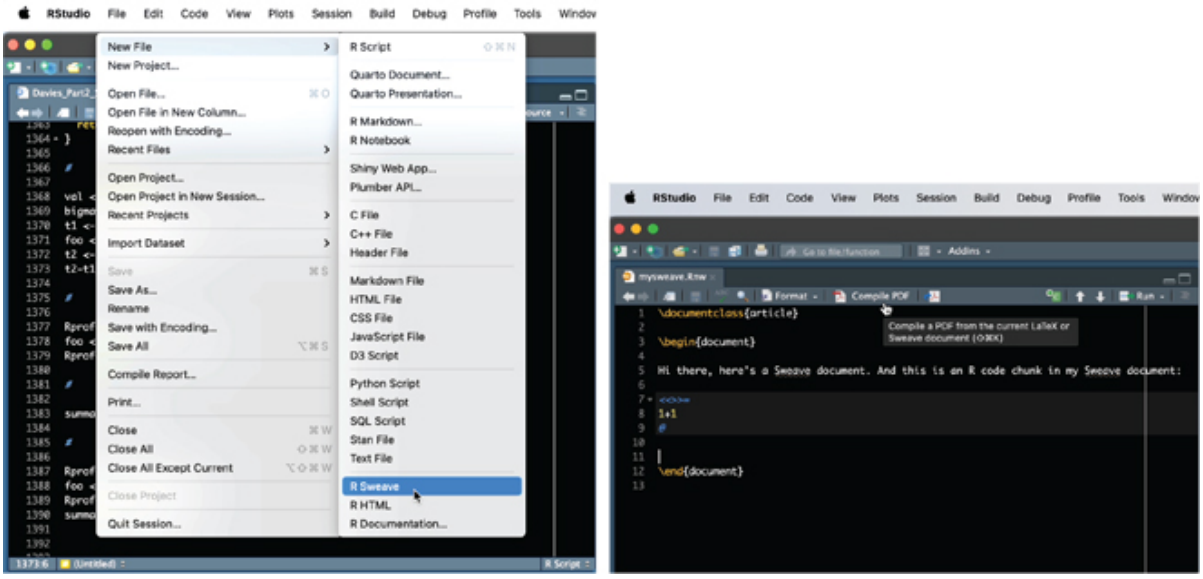


Figure B-9: Menu item for a new Sweave document in RStudio (left) and the view of an example Sweave document in the editor (right)

On the right of Figure B-9 is an example Sweave document (see the downloadable resources for this book) named *mysweave.Rnw*. As this screenshot shows, you use the **Compile PDF** button to render the result. For this to work, a LATEX installation is required on your machine.

B.3.2 knitr

`knitr` (Xie 2015, 2024) is an R package designed as an extension to Sweave, with some additional features that make document creation easier and more flexible. You can choose between the classic Sweave or `knitr` as the default document "weaver" for *.Rnw* files, which you can control by going to the Sweave section of **Tools ▶ Global Options . . .** (see Figure B-10).

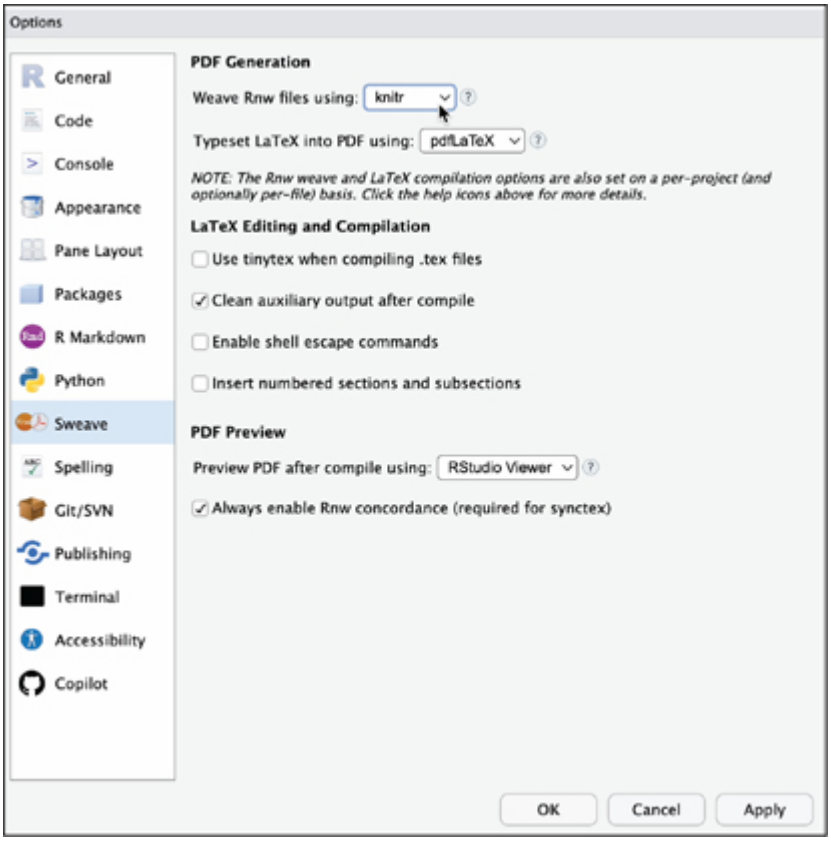


Figure B-10: Choosing the default document weaver of your markup document in the Sweave tab of RStudio's global options

To learn more about Sweave and `knitr` with respect to RStudio, see *https://support.posit.co/hc/en-us/articles/200552056-Using-Sweave-and-knitr*.

B.3.3 R Markdown

R Markdown (Allaire et al. 2024, Xie et al. 2018) is another dynamic document creation tool, downloadable from CRAN as the `rmarkdown` package. Although it uses `knitr` to compile documents, one of the objectives of R Markdown is to minimize the need to learn complicated markup languages like LATEX, and as a result, its syntax is generally acknowledged as being simpler than that of Sweave. Working from a *.Rmd* source file, you can create a variety of output document types, such as PDF (this format requires a LATEX installation on your machine), HTML, and Word.

Click **File ▶ New File ▶ R Markdown . . .**, as shown on the top left of Figure B-11, to start a new R Markdown document; this opens the New R Markdown dialog on the top right of Figure B-11. Here, you can choose the appropriate document type for your needs, and then you're provided with a basic template in the RStudio editor; the template for a basic document is shown on the bottom of Figure B-11. The template tells you how to compile a document (by clicking **Knit**) and even points you toward the R Markdown home page at *https://rmarkdown.rstudio.com*, which you should certainly investigate if you're interested in learning more. The R Markdown developers have detailed, beginner-friendly guides on this website, such as *https://rmarkdown.rstudio.com/lesson-1.html*.

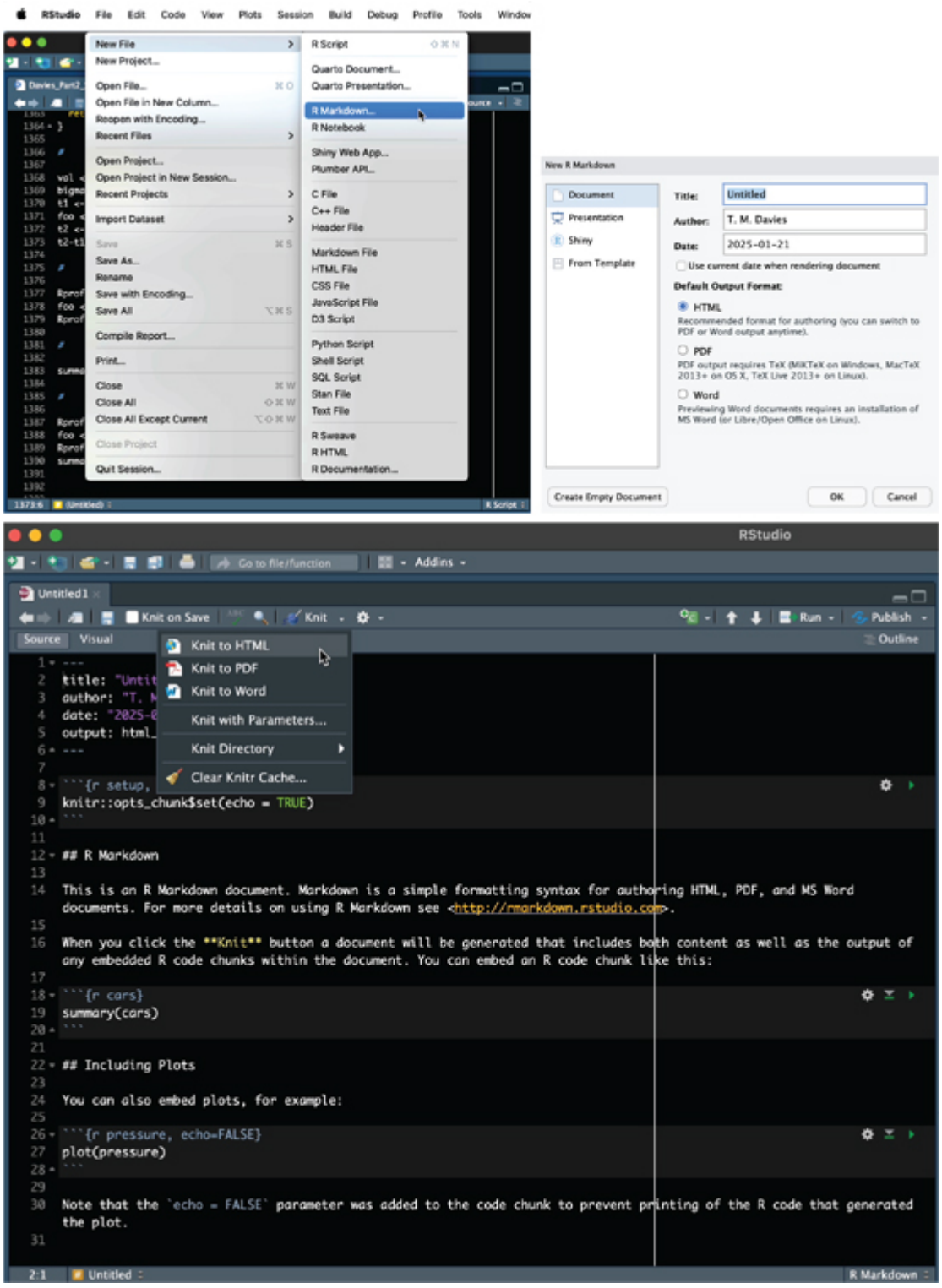


Figure B-11: Starting a new R Markdown document in RStudio. Templates relevant to your chosen document type are automatically provided.

B.3.4 Quarto

Quarto (*https://quarto.org*) is a relatively new tool from the Posit team. It is very similar to R Markdown, using `knitr` to compile or render documents, and as such, it is designed to be largely backward-compatible (that is, it will be able to compile most *.Rmd* files originally composed in R Markdown asis). Quarto is intended as a more general markup framework than R Markdown, providing support for not just R, but multiple data science languages such as Python. Your editable document looks a little different, as does the syntax used to declare code chunks, but the overall outcome is much the same. Quarto documents use a *.qmd* extension.

In the RStudio IDE, clicking **File ▶ New File ▶ Quarto Document . . .** (Figure B-12, top left) will open a dialog requesting basic details about your document and intended output format (Figure B-12, top right), much like that for R Markdown documents. When you've made your choices, RStudio will open a template corresponding to the details you provided.

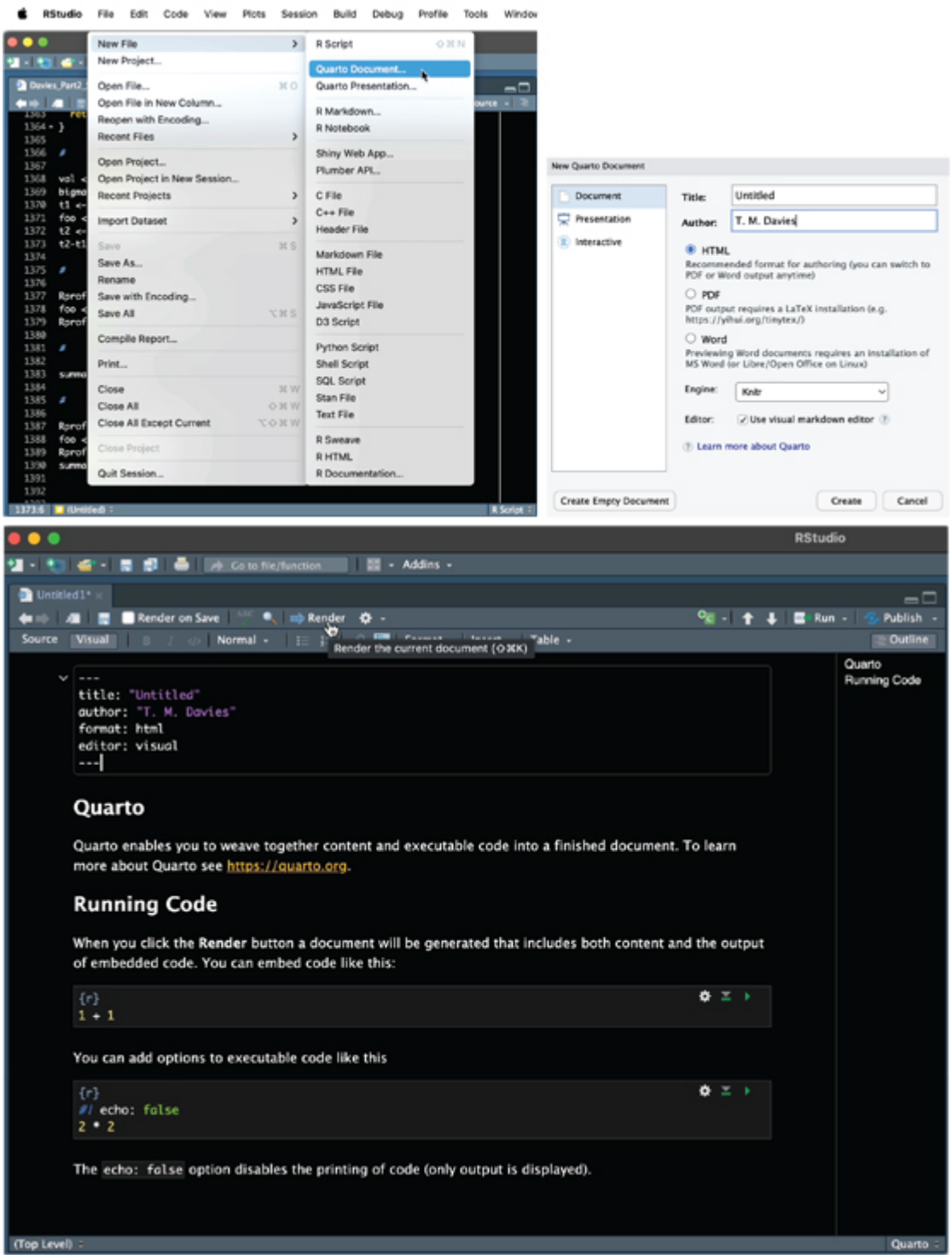


Figure B-12: Starting a new Quarto document in RStudio. Templates relevant to your chosen document type are automatically provided.

You produce the formatted document by clicking **Render** (you'll be prompted to first save your *.qmd* file if you've not already done so). This

is shown on the bottom of Figure B-12. Like R Markdown files, you can control finer details of the formatting of the final document by editing the declarations in the preamble at the top of the script.

If you're at the point where you want to start writing dynamic documents to combine your R code with text in a report or presentation style, I recommend getting to grips with Quarto (or R Markdown). Indeed, notice the presence of Quarto project templates in Figure B-5. When you're ready to take the plunge, there are plenty of examples and tutorials on the aforementioned Quarto website; see *https://quarto.org/docs/get-started/hello/rstudio.html*.

B.3.5 Shiny

Shiny is a framework for creating interactive web applications, often based around data visualization. Shiny apps require you to have an R session running behind the scenes, which drives the plots as the user interacts with the application in a web browser. The R package `shiny` (Chang et al. 2024) provides the required functionality.

Like other tools associated with the RStudio IDE, Shiny is a high-level framework intended to be friendly to both users and developers. Navigating to **File ▶ New File ▶ Shiny Web App . . .** will present you with a basic template. If you're interested in sharing your data, statistical models and analyses, or graphics, visit *https://shiny.posit.co* in the first instance. The development team has put a tremendous amount of work into creating comprehensive tutorials and examples to help you get started.

I have found Shiny useful in presenting results in academic research projects. In Davies et al. (2022), for example, we fitted a spatial model to a complex geochemical dataset. The sheer volume of results was simply too great to present in the paper itself, and instead we created a Shiny app so readers could peruse the results on demand, without needing to open R and interrogate large data structures. You can visit this app, which runs an R session on a server at my university department, at *https://www.stats.otago.ac.nz/nzgeochem*.

Once you're comfortable with Shiny, you can even combine its capabilities with R Markdown or Quarto to create interactive documents; notice the Shiny document option in the top right image of Figure B-11 and the Interactive document option in the top right of Figure B-12.

REFERENCES

Adler, D., Kelly, S.T., Elliott, T., and Adamson, J. (2022). *vioplot: violin plot*. *https://github.com/TomKellyGenetics/vioplot*, R package version 0.4.0.

Allaire, J., Xie, Y., Dervieux, C., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2024). *rmarkdown: Dynamic Documents for R*. *https://github.com/rstudio/rmarkdown*, R package version 2.29.

Anderson, E. (1935). "The irises of the Gaspe Peninsula." *Bulletin of the American Iris Society*, **59**, 2–5.

Atkinson, A. (1985). *Plots, Transformations and Regression*. Oxford University Press, UK.

Auguie, B. (2017). *gridExtra: Miscellaneous Functions for "Grid" Graphics*. *https://cran.r-project.org/web/packages/gridExtra/index.html*, R package version 2.3.

Bache, S.M. and Wickham, H. (2022). *magrittr: A ForwardPipe Operator for R*. *https://cran.r-project.org/web/packages/magrittr/index.html*, R package version 2.0.3.

Baddeley, A., Rubak, E., and Turner, R. (2015). *Spatial Point Patterns: Methodology and Applications with R*. Chapman and Hall/CRC, Boca Raton, USA.

Baddeley, A. and Turner, R. (2005). "spatstat: An R package for analyzing spatial point patterns." *Journal of Statistical Software*, **12**(6), 1–42.

Banerjee, S. and Roy, A. (2014). *Linear Algebra and Matrix Analysis for Statistics*. Chapman & Hall, USA.

Becker, R., Chambers, J., and Wilks, A. (1988). *The New S Language*. Chapman & Hall, UK.

Bollen, K. and Jackman, R. (1990). "Regression diagnostics: An expository treatment of outliers and influential cases." In J. Fox and J.S. Long (eds.), "Modern Methods of Data Analysis," Sage, USA.

Canty, A. and Ripley, B. (2024). *boot: Bootstrap R (SPlus) Functions*. *https://cran.r-project.org/web/packages/boot/index.html*, R package version 1.3-31.

Chacón, J. and Duong, T. (2018). *Multivariate Kernel Smoothing and Its Applications*. Chapman & Hall, USA.

Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., and Borges, B. (2024). *shiny: Web Application Framework for R. https://cran.r-project.org/web/packages/shiny/index.html*, R package version 1.10.0.

Chatterjee, S., Hadi, A., and Price, B. (2000). *Regression Analysis by Example*. Wiley, USA, 3rd edition.

Chu, S. (2001). "Pricing the C's of diamond stones." *Journal of Statistics Education*, **9**(2). *http://jse.amstat.org/v9n2/datasets.chu.html*.

Comber, L. and Brunsdon, C. (2021). *Geographical Data Science and Spatial Data Analysis: An Introduction in R*. Sage, London, UK.

Corder, G. and Foreman, D. (2014). *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, New Jersey, USA, 2nd edition.

Cox, D. and Snell, E. (1981). *Applied Statistics: Principles and Examples*. Chapman & Hall, USA.

Davies, T., Banerjee, S., Martin, A., and Turnbull, R. (2022). "A nearest-neighbour Gaussian process spatial factor model for censored, multi-depth geochemical data." *Journal of the Royal Statistical Society Series C (Applied Statistics)*, **71**, 1014–1043.

Davies, T. and Bryant, D. (2013). "On circulant embedding for Gaussian random fields in R." *Journal of Statistical Software*, **55**(9), 1–21.

Davison, A. and Hinkley, D. (1997). *Bootstrap Methods and Their Applications*. Cambridge University Press, UK.

Dickey, D.A. and Arnold, J.T. (1995). "Teaching statistics with data of historic significance: Galileo's gravity and motion experiments." *Journal of Statistics Education*, **3**(1). *https://jse.amstat.org/v3n1/datasets.dickey.html*.

Diggle, P. (1990). "A point process modelling approach to raised incidence of a rare phenomenon in the vicinity of a prespecified point."

Dobson, A. (1983). *An Introduction to Statistical Modelling*. Chapman & Hall, UK.

Duong, T. (2007). "ks: Kernel density estimation and kernel discriminant analysis for multivariate data in R." *Journal of Statistical Software*, **21**(7), 1–16.

Duong, T. (2024). *ks: Kernel Smoothing*. *https://cran.r-project.org/web/packages/ks/index.html*, R package version 1.14.3.

Faraway, J. (2014). *Linear Models with R*. Chapman & Hall, USA, 2nd edition.

Feng, D. and Tierney, L. (2008). "Computing and displaying isosurfaces in R." *Journal of Statistical Software*, **28**(1), 1–24.

Fisher, J. (1976). "Homicide in Detroit: The role of firearms." *Criminology*, **14**, 387–400.

Fisher, R. (1936). "The use of multiple measurements in taxonomic problems." *Annals of Eugenics*, **7**(2), 179–188.

Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*. Sage, USA, 2nd edition.

Golub, G. and Van Loan, C. (1989). *Matrix Computations*. Johns Hopkins University Press, UK, 2nd edition.

Grolemund, G. and Wickham, H. (2011). "Dates and times made easy with lubridate." *Journal of Statistical Software*, **40**(3), 1–25. *https://www.jstatsoft.org/article/view/v040i03*.

Gupta, S. (2012). "How R Searches and Finds Stuff." *https://blog.thatbuthow.com/how-r-searches-and-finds-stuff/*.

Hand, D., Daly, F., McConway, K., Lunn, D., and Ostrowski, E. (eds.) (1994). *A Handbook of Small Data Sets*. Chapman & Hall, USA.

Härdle, W. (1990). *Applied Nonparametric Regression*. Cambridge University Press, USA.

Harraway, J. (1995). *Regression Methods Applied*. University of Otago Press, New Zealand.

Harrison, D. and Rubinfeld, D. (1978). "Hedonic prices and the demand for clean air." *Journal of Environmental Economics and*

Management, **5**, 81–102.

Henderson, H. and Velleman, P. (1981). "Building multiple regression models interactively." *Biometrics*, **37**, 391–411.

Hornik, K. and the R Core Team (2024). "R FAQ." *https://cran.r-project.org/doc/manuals/R-FAQ.html*.

Hu, Y. and Scarrott, C. (2018). "evmix: An R package for extreme value mixture modeling, threshold estimation and boundary corrected kernel density estimation." *Journal of Statistical Software*, **84**(5), 1–27.

Kessler, F. and Battersby, S. (2019). *Working with Map Projections: A Guide to their Selection*. CRC Press, Boca Raton, USA.

Kruschke, J. (2010). *Doing Bayesian Data Analysis: A Tutorial with R and BUGS*. Academic Press/Elsevier, USA.

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Springer, USA.

Kurt, W. (2019). *Bayesian Statistics the Fun Way: Understanding Statistics and Probability with Star Wars, LEGO, and Rubber Ducks*. No Starch Press, USA.

Kusnierczyk, W. (2012). *rbenchmark: Benchmarking routine for R. https://cran.r-project.org/web/packages/rbenchmark/index.html*, R package version 1.0.0.

Leisch, F. (2002). "Dynamic generation of statistical reports using literate data analysis." In W. Härdle and B. Rönz (eds.), "Compstat 2002 —Proceedings in Computational Statistics," Physika Verlag, Germany.

Lemon, J. (2006). "Plotrix: a package in the red light district of R." *R-News*, **6**(4), 8–12.

Ligges, U. and Fox, J. (2008). "R Help Desk: How can I avoid this loop or make it faster?" *R News*, **8**(1), 46–50.

Ligges, U. and Mächler, M. (2003). "Scatterplot3d—an R package for visualizing multivariate data." *Journal of Statistical Software*, **8**(11), 1–20.

Lock, R. (1993). "1993 new car data." *Journal of Statistics Education*, **1**(1).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, USA.

Matloff, N. and Salzman, P. (2008). *The Art of Debugging: With GDB, DDD, and Eclipse*. No Starch Press, USA.

McNeil, D. (1977). *Interactive Data Analysis*. Wiley, USA.

Mersmann, O. (2024). *microbenchmark: Accurate Timing Functions*. *https://cran.r-project.org/web/packages/microbenchmark/index.html*, R package version 1.5.0.

Mirai Solutions GmbH (2024). *XLConnect: Excel Connector for R*. *https://cran.r-project.org/web/packages/XLConnect/index.html.*, R package version 1.1.0.

Montgomery, D., Peck, E., and Vining, G. (2012). *Introduction to Linear Regression Analysis*. Wiley, USA, 5th edition.

Murdoch, D. and Adler, D. (2024). *rgl: 3D Visualization Using OpenGL*. *https://cran.r-project.org/web/packages/rgl/index.html*, R package version 1.3.1.

Murrell, P. (2018). *R Graphics*. Chapman & Hall, USA, 3rd edition.

Murrell, P. and Ihaka, R. (2000). "An approach to providing mathematical annotation in plots." *Journal of Computational and Graphical Statistics*, **9**, 582–599.

Neuwirth, E. (2022). *RColorBrewer: ColorBrewer Palettes*. *https://cran.r-project.org/web/packages/RColorBrewer/index.html*, R package version 1.1 3.

Ooms, J., James, D., DebRoy, S., Wickham, H., and Horner, J. (2024). *RMySQL: Database Interface and 'MySQL' Driver for R*. *https://cran.r-project.org/web/packages/RMySQL/index.html*, R package version 0.10.29.

Pebesma, E. and Bivand, R. (2023). *Spatial Data Science: With Applications in R*. Chapman and Hall/CRC, Boca Raton, USA.

Peng, R. (2022). *R Programming for Data Science*. bookdown.org, USA. *https://bookdown.org/rdpeng/rprogdatascience/*, Online e-book.

Posit team (2024). *RStudio: Integrated Development Environment for R*. Posit Software, PBC, Boston, MA. *https://www.posit.co*.

R Core Team (2025). *foreign: Read Data Stored by 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', . . .* *https://cran.r-project.org/web/packages/foreign/index.html*, R package version 0.8 88.

Reinhart, A. (2015). *Statistics Done Wrong: The Woefully Complete Guide*. No Starch Press, USA.

Ripley, B. and Lapsley, M. (2024). *RODBC: ODBC Database Access*. *https://cran.r-project.org/web/packages/RODBC/index.html*, R package version 1.3 26.

Royston, P. (1982). "Algorithm AS 181: The W test for Normality." *Applied Statistics*, **31**, 176–180.

Schloerke, B., Cook, D., Larmarange, J., Briatte, F., Marbach, M., Thoen, E., Elberg, A., and Crowley, J. (2024). *GGally: Extension to 'ggplot2'*. *https://cran.r-project.org/web/packages/GGally/index.html*, R package version 2.2.1.

Schorling, J., Roach, J., Siegel, M., Baturka, N., Hunt, D., Guterbock, T., and Stewart, H. (1997). "A trial of church based smoking cessation interventions for rural African Americans." *Preventive Medicine*, **26**, 92–101.

Scott, D. (2015). *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley, USA, 2nd edition.

Snee, R. (1974). "Graphical display of twoway contingency tables." *The American Statistician*, **28**, 9–12.

Soetaert, K. (2024). *shape: Functions for Plotting Graphical Shapes, Colors*. *https://cran.r-project.org/web/packages/shape/index.html*, R package version 1.4.6.1.

Sterne, J. and Smith, G. (2001). "Sifting the evidence—what's wrong with significance tests?" *British Medical Journal*, **322**(7280), 226–231.

Takezawa, K. (2006). *Introduction to Nonparametric Regression*. Wiley, USA.

Tippett, L. (1950). *Technological Applications of Statistics*. Wiley, USA.

Tong, H., Thanoon, B., and Gudmundsson, G. (1985). "Threshold time series modelling of two Icelandic riverflow systems." *Water Resources Bulletin*, **21**, 651–661.

Trapletti, A. and Hornik, K. (2024). *tseries: Time Series Analysis and Computational Finance*. *https://cran.r-project.org/web/packages/tseries/index.html*, R package version 0.1058.

Umlauf, N., Klein, N., Simon, T., and Zeileis, A. (2021). "bamlss: A Lego toolbox for flexible Bayesian regression (and beyond)." *Journal of Statistical Software*, **100**(4), 1–53.

Urbanek, S. (2022). *RJDBC: Provides Access to Databases Through the JDBC Interface*. *https://cran.r-project.org/web/packages/RJDBC/index.html*, R package version 0.2-10.

Venables, W. and Ripley, B. (2002). *Modern Applied Statistics with S*. Springer, USA, 4th edition.

Wand, M. and Jones, M. (1995). *Kernel Smoothing*. Chapman & Hall, USA.

Warnes, G.R., Gorjanc, G., Magnusson, A., Andronic, L., Rogers, J., MacQueen, D., and Korosec, A. (2024). *gdata: Various R Programming Tools for Data Manipulation*. *https://cran.r-project.org/web/packages/gdata/index.html*, R package version 3.0.1.

Wei, T. and Simko, V. (2024). *R package 'corrplot': Visualization of a Correlation Matrix*. *https://github.com/taiyun/corrplot*, (Version 0.95).

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer, USA.

Wickham, H. (2019). *Advanced R*. Chapman & Hall/CRC, USA, 2nd edition.

Wickham, H. and Bryan, J. (2023). *R Packages: Organize, Test, Document, and Share Your Code*. O'Reilly Media, USA, 2nd edition.

Wickham, H., Çetinkaya-Rundel, M., and Grolemund, G. (2023). *R for Data Science: Import, Tidy, Transform, Visualize and Model Data*. O'Reilly, USA, 2nd edition.

Wickham, H. and Chang, W. (2024). *ggvis: Interactive Grammar of Graphics*. *https://cran.r-project.org/web/packages/ggvis/index.html*, R package version 0.4.9.

Wickham, H., Chang, W., Luraschi, J., and Mastny, T. (2024). *profvis: Interactive Visualizations for Profiling R Code*. *https://cran.r-project.org/web/packages/profvis/index.html*, R package version 0.4.0.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer, USA, 2nd edition.

Willems, J., Saunders, J., Hunt, D., and Schorling, J. (1997). "Prevalence of coronary heart disease risk factors among rural blacks: A community based study." *Southern Medical Journal*, **90**, 814–820.

Woosley, A. and McIntyre, A. (1996). *Mimbres Mogollon Archaeology*. University of New Mexico Press, USA.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman & Hall/CRC, USA, 2nd edition.

Xie, Y. (2024). *knitr: A General Purpose Package for Dynamic Report Generation in R*. *https://yihui.org/knitr/*, R package version 1.49.

Xie, Y., Allaire, J., and Grolemund, G. (2018). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida. *https://bookdown.org/yihui/rmarkdown/*.

Zeileis, A., Fisher, J.C., Hornik, K., Ihaka, R., McWhite, C.D., Murrell, P., Stauffer, R., and Wilke, C.O. (2020). "colorspace: A toolbox for manipulating and assessing colors and palettes." *Journal of Statistical Software*, **96**(1), 1–49.

INDEX

Symbols

`+` (addition) operator
    arithmetic, 15, 35
    matrices, 48, 57
`&` (AND element-wise) operator, 64–66, 102
`&&` (AND single comparison) operator, 64–66, 102
`%~~%` (approximately equal to) expression string, 678
`<-` (arrow notation), 19–20, 35
`\` (backslash) escape sequence character, 8, 74, 77–78, 102
`{ }` (brace) characters
    `for` loops, 223
    `if` statements
        nested, 216–217
        stand-alone, 210–212
    mathematical expressions, 675
    multiple commands, 746, 784
    `while` loops, 232
`:` (colon) operator
    formatting times, 97
    interaction, 586–587, 597
    sequences, 23, 27, 29, 31, 35, 499
`#` (comment), 7, 14
`*` (cross-factor) operator, 590, 595, 597
`/` (division) operator, 15, 35
`$` (dollar) operator, 107, 114, 118

\\ (double backslash) operator
escape sequence, 77
filepath specification, 8, 174
:: (double colon) operator, 294–295, 300
" (double quotation mark) character
strings, 73, 78, 102
[[ ]] (double square brackets) characters, 104, 118
. (ellipsis; dot-dot-dot notation) character, 203–204, 264–269, 276, 768
= (equal sign) character
assigning objects, 19–20, 35
variable mapping, 703–704
== (equal to) operator, 61, 102
\" escape sequence, 77
/ (forward slash) operator
filepath specification, 8, 78, 173
formatting dates, 94
> (greater than) operator, 61, 102
>= (greater than or equal to) operator, 61, 102
? (help) function, 10, 14
< (less than) operator, 61, 102
<= (less than or equal to) operator, 61, 102
· (matrix multiplication) symbol, 558
%*% (matrix product) operator
linear regression, 558–559, 630, 678
matrices, 49, 57
* (multiplication) operator, 15, 35, 47, 57
\n escape sequence, 77, 149, 657, 672
! (NOT) operator, 65, 102
!= (not equal to) operator, 61, 102
| (OR element-wise) operator, 64–66, 102

|| (OR single comparison) operator, 64–66, 102, 212
|> (pipe) operator (base R), 205–207, 702
%>% (pipe) operator (magrittr package), 207, 702
^ (power) operator, 15, 675
?? (search help) function, 12, 14
[ ] (square-bracket) characters, 27–28, 35, 70, 105, 118
- (subtraction) operator
    arithmetic, 15, 35
    matrices, 48, 57
\t escape sequence, 77
~ (tilde) formula operator, 352
:= (universal assignment) operator, 703–704, 708

A

abline function
    adding lines to plots, 154, 157, 168
    bin width and customization, 341
    linear regression model diagnostics, 629
    multiple linear regression, 564
    simple linear regression, 525–526
abs function, 316, 332
absolute values, 316–317, 468
Access databases, Microsoft, 179
add1 function, 605–608, 614–615, 643
add argument, 743
add_axis command, 705, 708
add_legend command, 704, 708
addition
    arithmetic, 15–16

matrices, 47–48
addition (+) operator
arithmetic, 15, 35
matrices, 48, 57
adjustcolor function, 688, 708, 723
adjusted R^2 statistic, 620
aes function, 161, 168
aesthetic mappings, 161
data visualization and graphics, 685, 689, 694
geoms, 164–167
aggregate function, 512, 519
AIC (Akaike's "An Information Criterion")
alternatives to, 620
stepwise selection, 613–618
airquality dataset, 690–692, 695
all function, 63–64, 102, 106
alpha argument, 722–723, 803–804
alternative argument, 461, 470, 475
alternative hypotheses, 456
testing
categorical variables, 481
means, 459–460, 464
proportions, 474
Type I errors, 490
Type II errors, 493
AND element-wise (&) operator, 64–66, 102
Anderson–Darling test, 506
AND single comparison (&&) operator, 64–66, 102
ann argument, 667, 683
anonymous (disposable) functions, 272

arguments, 266
calculating proportions and percentages, 314
parametric equations, 813–814
ANOVA (analysis of variance), 503–518
Kruskal–Wallis test, 516–518
one-way, 503–511
between-group variability, 507
categorical predictors and, 551–552
equality of variances, 504, 506
hypotheses and diagnostic checking, 504–506
independence, 504–505
normality, 504–505
table construction, 506–509
within-group variability, 507–508
two-way, 511–515
interaction plots, 514
main effects and interactions, 511–515
suite of hypotheses, 511–512
`anova` function, 603, 643
`any` function, 63–64, 102
`aov` function, 508–509, 513, 519
`apply` function, 238–240, 249
anonymous functions, 272
implicit loops, 242
measuring computational expense, 288, 291
parametric equations, 813
variants of, 240–241
approximately equal to (`%~~%`) expression string, 678
`args` function, 202, 207
arguments, 10, 258–269. *See also names of specific arguments*
argument matching conditions, 200–204

ellipses, 203–204, 264–269
exact, 200–201
mixed, 203
partial, 201–202
positional, 202–203
checking for missing, 263–264
graphical parameters, 147
help files for, 10
lazy evaluation, 258–261
setting defaults, 261–262
supplementary, 204
arithmetic
operators, 15, 35
order of operations, 16–17
arithmetic mean. *See* mean
`array` function, 52, 56–57
arrays, 37, 51–57
creating, 52–53
deleting and overwriting elements in, 56
extractions, 53–56
subsets, 53–55
arrow notation (`<-`), 19–20, 35
`arrows` function
adding arrows to plots, 154, 158, 168
clipping, 658–659
`as.Date` function, 93, 97–98, 102
as-dot coercion functions, 87, 137–142. *See also names of specific as-dot functions*
`as.integer` function, 279–280
`as.logical` function, 138–139
`as.matrix` function, 140

`as.numeric` function
categorical predictors, 539
coercion, 138–139
color indexing, 717
date-time operations, 96, 99
hypotheses and diagnostic checking, 506
interval interpretation, 532
measures of spread, 322
numeric representation of factors, 86–87, 102
`asp` (aspect ratio) function, 147–148, 153–154, 168
`aspect` argument, 783, 816
aspect ratio
bivariate surfaces, 792–794
plotting, 152–154
`as.vector` function, 139–140
asymmetric distributions, 388–389, 450–451
`attach` function, 298–299
`attr` function, 131–132, 143
`attributes` function, 130–131, 143, 300
attributes of objects, 130–132
axes
customization, 668–670
labels, 148–149
point clouds, 779–782
`axes3d` function, 779–780, 825
`axes` argument, 667, 683, 745–746
`axis3d` function, 779
`axis` function, 668, 683

B

backslash (\) escape sequence character, 8, 74, 77–78, 102
backward selection (backward elimination), 609–614
`bamlss` package, 722
bandwidth (smoothing parameter; window width)
    density plots, 344–346, 348
    nonparametric bivariate density, 741
    nonparametric trivariate density, 807
    smooth density estimates, 691
    violin plots, 355
`barplot` function, 334–335, 370, 378
bar plots, 333–339
    creation, 334–336
    cumulative probability distributions, 378
    dodged, 335–337
    in `ggplot2`, 336–337
    stacked, 335
baseline (no-change) hypotheses, 456
base R GUI (graphical user interface), 4–5
    bracket-matching feature, 6
    code-coloring feature, 6
    console and editor panes, 5–6
    editor pane in, 5–6
    graphics device, 145, 146
    installing packages, 834–836
    preferences pane, 6
    progress bar, 286–287
    shortcuts, 7
Bayesian Information Criterion (BIC), 620
`%b` date marker, 93–94, 102
`%B` date marker, 93–94, 102
Bell Laboratories, xxxi

Bernoulli distribution, 394–395
beta distribution, 431
`bg3d` function, 777, 782, 825
`bgplot3d` function, 784, 794, 825
bias, 445
bimodal distributions, 388–389
binary (dichotomous) variables, 538–542
    Bernoulli distribution, 394
    dummy coding, 542–544
    linear regression model of, 540–541
    predictions from, 541–542
binomial coefficient, 395
binomial distribution, 395–400
    `dbinom` function, 397–398
    negative, 404
    `pbinom` function, 398
    `qbinom` function, 399
    `rbinom` function, 399
Bioconductor, 839
bivariate surfaces, 733–736, 788–797
    adding components
        points, 789
        surfaces, 789–791
    aspect ratio, 792–794
    coloring by *z* value, 792
    constructing evaluation grid, 733–735
    plotting perspective surfaces, 788–789
    z-matrices
        conceptualizing, 735–736
        constructing, 735
`bmp` function, 180, 189

body code (function body), 252
Bonferroni correction, 492–493
`boot` package, 594
boundary correction, 345, 348
`box3d` function, 779–780
`box` function, 655, 667–668, 683
`boxplot` function, 350–351, 370
box plots (box-and-whisker plots), 350–353
    in `ggplot2`, 352–353
    side-by-side, 351–352
    stand-alone, 350–351
    violin plots vs., 354
    whiskers, 351
braces. *See* `{ }` characters
`break` function, 243–244, 249, 254
`breaks` argument, 90, 340–341
`bty` argument, 667, 683
built-in (ready-to-use) functions, 9. *See also names of specific built-in functions*
`byrow` argument, 38, 202–203

C

`car` package, 183–184, 295, 297
case sensitivity, 3, 8
categorical predictors, 538–553
    binary variables, 538–542, 540
        baseline/reference value and additive effects, 538–539
        linear regression model of, 540–541

predictions from, 541–542
changing reference level, 547–548
checking for equivalence with oneway ANOVA, 551–552
multilevel variables, 542–547
dummy coding, 542–544
linear regression model of, 544–546
predictions from, 546–547
categorical variables, 304
testing, 480
single variables, 481–484
two variables, 484–489
treating as numeric categorical predictors, 548–551
`cat` function, 74–75, 102, 137, 280
`cbind` function
calculating leverage, 629–630
chi-squared tests of independence, 487
column bindings, 39–40, 57
column insertions, 113–114
`ceiling` function, 316, 332
centrality, measures of, 309–314
mean, 309
median, 309
mode, 310
central limit theorem (CLT)
confidence intervals, 446–447
hypothesis tests, 471
sampling distributions, 437
`cex` (character expansion) argument, 147–148, 168, 673
`c` function
factors, 89
variadic behavior, 203–204

vector creation, 21, 23, 28, 35, 56
character sensitivity, 3
character strings, 72–80
concatenation, 74–77
escape sequences, 77–78
pattern matching, 78–79
string creation, 73–74
substrings and matching, 78–80
`ChickWeight` dataset
condition object modification, 234
`for` loops, 224–225
help file, 170
`chickwts` data frame
ANOVA, 503–506, 508
probability density functions, 416
summary statistics, 311–315, 319, 322
variables, 305
`chisq.test` function, 483–484, 488, 502
chi-squared distributions, 426–428
description and appearance, 426–427
implementation, 427–428
tests
of distribution, 480–484
of independence, 480, 484–488
Chorley–Ribble (`chorley`) cancer
dataset, 752–757, 792–794
circles, parametric equations for, 811–812
CIs. *See* confidence intervals
`citation` command, 840
citing R and contributed packages, 840–841
classes, 132–136

data structures, 134–135
inheriting from multiple, 135–136
stand-alone vectors, 133–134
`class` function, 133–135, 143
classical (frequentist) probability, 372
clipping, 657–659
`close` function, 286, 300
CLT. *See* central limit theorem
`cm.colors` function, 714, 772
coding
code execution, 7
conventions and styles, xxxvi–xxxvii
coefficient of determination (R^2), 528, 530, 602
adjusted, 620
`coef` function, 527, 553
coercion, 137–142
as-dot functions, 87, 138
concatenation and, 75
factors, 139
implicit vs. explicit, 137–138
multiple, 138
object classes and structures, 139–140
`col2rgb` function, 711–712, 772
`col` argument
box customization, 667–668
facet color, 764
setting point/line color, 147–151, 168
3D plots, 775
`collapse` argument, 77
collinearity (multicollinearity), 582, 640–643

correlated predictors, 641–643
warning signs, 640–641
`colMeans` function, 206
`colnames` function, 111, 118
colon. *See* : operator
color, 709–728
basic plotting, 149–151
bivariate surfaces, 792
hue-chroma-luminance, 725
hue-saturation-value, 725
indexing
via categorization, 716–718
via normalization, 718–720
isosurfaces, 803–804
legends, 720–722
opacity, 722–724
palettes
built-in, 714–715
custom, 715–716
in point clouds, 775–778
RGB, 710–713
`color` argument, 163, 165, 168, 804
`colorlegend` function
legends, 720–722, 724, 772
pixel images, 750–751
3D scatterplots, 731, 784
`color.legend` function, 722
`colorRamp` function, 718–719, 772
`colorRampPalette` function, 715, 744, 772
`colors` function, 149, 710, 772
`colorspace` package, 725

`colSums` function
chi-squared tests of independence, 486
implicit looping, 239
measuring computational expense, 288, 300
pipes, 206
comments, 7
complements of events, 375
complexity, 599
goodness-of-fit vs., 599–601
guidelines, 600–601
principle of parsimony, 600
Comprehensive R Archive Network. *See* CRAN
concatenation of strings, 74–77
conditional probability, 373
cones, parametric equations for, 814–815
confidence intervals (CIs), 435, 446–453
independent, identically distributed observations, 436
interpretation of, 451
intervals for a mean, 446–449
mean heights example, 533
multiple linear regression, 565
prediction intervals vs., 531–532
proportions, 449–450
quantities with asymmetric sampling distributions, 450–451
standard error, 436
`confint` function, 530, 553, 565
`conf.level` argument, 461, 475
`conflicts` function, 296, 300
confounding, 556
contingency tables, 485
continuous random variables, 376–377, 380–388

concept of, 380
cumulative probability distributions, 385–386
mean and variance, 386–388
probability density functions, 381
continuous variables, 304–305
`contour3d` function, 802–803, 825
`contour` function, 737–739, 741, 743, 772
contour plots, 736–749
color-filled contours, 743–746
contour lines, 737–740
nonparametric bivariate density, 740–743
parametric response surface, 738–740
topographical map, 737–738
control flow mechanisms, 209–249
`break` function, 243–244
`for` loops, 223–231
via index or value, 226–228
nesting, 228–231
`if` statements, 209–223
element-wise checks with `ifelse`, 214–215
`else` declaration and, 213–214
nested and stacked, 216–218
stand-alone, 210–213
`switch` function, 218–223
implicit looping, 238–243
`apply` function, 238–240
`lapply` function, 241
`sapply` function, 241
`tapply` function, 240
whether to use, 242–243
`next` function, 244–246

`repeat` statements, 246–247
`while` loops, 232–238
condition object modification, 233–238
conventions and styles, xxxvi–xxxviii
coding, xxxvi–xxxvii
exercises, xxxviii
math and equation references, xxxvii–xxxviii
Cook's distance
calculating, 631–634
graphically combining with leverage and residuals, 635–638
`cooks.distance` function, 632–633, 643
`coord_flip` function, 336–337, 370
coordinate-based plots (plots of spatial coordinates), 365–369
in `ggplot2`, 369
longitude and latitude, 365–369
pre-existing distance-based units, 365
coordinates. *See also* scatterplots
placing ad hoc annotations on, 661–662
point-and-click interaction, 659–665
retrieving silently, 659–660
3D evaluation, 798–799
visualizing selected, 660–661
`coord_sf` function, 369
`cor` function, 325–328, 332
corrected AIC (AIC_c), 620
correlation, 325–328
`corrplot` package, 722
`cos` function, 366, 811, 825
covariance, 324–325, 328–329
covariate profiles, 568, 584

cov function, 325–328, 332
CRAN (Comprehensive R Archive Network)
    contributed packages, 179, 362, 720, 830–837
    downloading and installing R, 3, 827–829
    finding packages on, 832–833
    mirror site, 827–828
    switching mirror, 839
criterion-based (information-based) model selection methods, 613
cross-factor (*) operator, 590, 595, 597
.*csv* files, 171, 176
cumsum function, 378, 391
cumulative probability distributions
    of continuous random variables, 385–386
    of discrete random variables, 377–378
cut function, 90, 102, 766, 792
cylinders, parametric equations for, 811–814

D

data argument
    ANOVA table construction, 508
    matrix creation, 38
    partial argument matching, 201
    scoping, 195–196
database management systems (DBMSs), 179
data.frame function, 110–112, 118, 203–204
data frames, 109–118
    combinations and column insertions, 113–114
    constructing, 110–112
    logical record subsets, 115–116
    masking, 297–299

subsetting, 112–113, 116–118
`data` function, 171–172, 189, 193
datasets. *See also names of specific datasets*
R-ready, 169–172
built-in, 170–171
contributed, 171–172
writing out, 179–180
`datasets` package, 170, 304
data structures, 103–118
arrays, 37, 51–57
creating, 52–53
extractions, 53–56
subsets, 53–55
classes and, 134–135
data frames, 109–118
combinations and column insertions, 113–114
constructing, 110–112
logical record subsets, 115–116
subsetting, 112–113, 116–118
lists, 103–109
creating and accessing components, 104–105
naming members, 106–107
nesting, 107–109
matrices, 37–51
creating, 38–40
operations and algebra, 45–51
subsetting, 40–45
vectors, 21–35
creating, 21–22
element extraction, 26–31
lengths, 26

repetition, 24–25
sequences, 23–24
sorting, 25
subsetting, 27–30
vector-oriented behavior, 31–35
data types, 59–102
characters, 72–80
concatenation, 74–77
escape sequences, 77–78
string creation, 73–74
substrings and matching, 78–80
dates and times, 92–101
creating and formatting dates, 93–95
incorporating time, 97–101
operations with dates, 95–97
factors, 81–92
collapsing, 87–88
combining, 89–90
creating from continuous values, 90–92
defining and ordering levels, 83–85
dropping categories, 86
identifying categories, 81–83
numeric representation of, 86–87
logical values, 59–72
logical operators, 64–67
numeric representation of, 67–68
relational operators, 60–64
subsetting and extracting with, 68–72
TRUE and FALSE, 60
data visualization, 333–370
aesthetic mappings, 685, 689, 694

bar plots, 333–337
creating, 334–336
in ggplot2, 336–337
box plots, 350–353
in ggplot2, 352–353
side-by-side, 351–352
stand-alone, 350–351
density plots, 343–350
bandwidths and customization, 344–345
construction, 344
in ggplot2, 347–350
overlaying different distributions, 345–347
histograms, 339–343
bin width and customization, 341
example of, 340
in ggplot2, 341–343
pie charts, 337–339
plots of spatial coordinates, 365–369
longitude and latitude, 365–369
pre-existing distance-based units, 365
scatterplots and coordinate-based plots, 357–369
in ggplot2, 361–365, 369
matrices of, 359–361
single, 358–359
violin plots, 353–357
design and interpretation, 354–355
in ggplot2, 356–357
side-by-side, 355
dates and times, 92–101
creating and formatting dates, 93–95
incorporating time, 97–101
operations with dates, 95–97

.dat files, 179
`dbeta` function, 431
`dbinom` function, 396–398, 432
DBMSs (database management systems), 179
`dchisq` function, 427–428, 432
`%d` date marker, 93–94, 102
debugging, 280
    in RStudio, 849–850
decimal (floating-point) numbers, 134
degrees of freedom (df), 419, 508–509
`density` function, 344–346, 348, 370, 690
density plots, 343–350
    bandwidths and customization, 344–345
    construction, 344
    in `ggplot2`, 347–350
    noise in, 345
    overlaying different distributions, 345–347
dependent/paired samples testing, 468–471
design matrices, 558
`detach` function, 297, 299–300, 830
`dev.new` command, 648–649, 677, 683
`dev.off` function, 181–182, 189, 650, 683
`dev.set` command, 649, 683
`dexp` function, 423–424, 432
`df` function, 429–430, 432
`d`-functions, 393
`dgamma` function, 431
`dgeom` function, 404
`dget` function, 186–189
`dhyper` function, 404

`diabetes` dataset, 586–587, 603–604, 697
`diag` command, 42, 44, 46–47, 57
dichotomous variables. *See* binary variables
`digit` argument, 722
`digits` argument, 315
`dim` argument, 52, 56
`dim` function
    data frame construction, 111, 118
    matrix dimensions, 40, 49, 57
    perspective plots, 766
`dimnames` function
    argument matching, 200–201, 203
    array dimensions, 131–132, 143
    data frame construction, 112, 118
discrete random variables, 376–380
    cumulative probability distributions, 378
    mean and variance, 379–380
discrete variables, 304
disposable functions. *See* anonymous functions
`disp` variable, 573, 575, 578
distributions of probability, 393–432
    Bernoulli, 394–395
    binomial, 395–400
    chi-squared, 426–428
    exponential, 422–426
    F-distribution, 428–430
    normal, 410–419
    Poisson, 400–404
    shape in, 388–391
    Student's *t*-distribution, 419–422
    uniform, 404–410

division (/) operator, 15, 35
dmultinom function, 404
dmvnorm function, 800, 802, 825
dnbinom function, 404
dnorm function, 412, 432
document tools, 850–854
    knitr package, 851
    Quarto, 852–854
    R Markdown, 851–852
    Shiny, 854
    Sweave, 850–851
dollar ($) operator, 107, 114, 118
dot-dot-dot notation (...), 203–204, 264–269, 276, 768
dot functions. *See also names of specific dot functions*
    as-dot, 87, 137–142
    is-dot, 136–137
double backslash (\\) operator
    escape sequence, 77
    filepath specification, 8, 174
double colon (::) operator, 294–295, 300
double quotation mark (") character
    filepaths, 8
    strings, 73, 78, 102
double square brackets ([[ ]]) characters, 104, 118
downloading and installing R, 827–837
    base packages, 829–830
    contributed packages, 830–837
        finding on CRAN, 832–833
        installing at prompt, 833–834
        installing using GUI, 834–836
        installing using local files, 836–837

from CRAN, 3
recommended packages, 830
using packages, 829
dpois function, 402–403, 432
dput command, 186–189
drop1 function, 643
backward selection, 609–612
stepwise AIC selection, 614–615
droplevels function, 85, 102, 235
dt function, 420–421, 432
dummy coding, 542–544
artificial nature of, 544
multilevel variables, 542–543
orthogonal contrasts, 547
dunif function, 406–408, 432
dweibull function, 431

E

e (Euler's number), 17
each argument, 780
element-wise (vector-oriented)
behavior, 31–35
element-wise multiplication, 47
ellipsis (.) operator, 203–204, 264–269, 276, 768
else statements, 213–214, 249
nesting/stacking, 216–217
switch function vs., 218–219
e-notation, 18–19
environment function, 197, 207

environments in scoping rules, 194–196
global, 194
local/execution, 195–196
package environments and namespaces, 194–195
EPS files, storing plots as, 181
equal sign (=) character
assigning objects, 19–20, 35
variable mapping, 703–704
equal to (==) operator, 61, 102
equation reference conventions and styles, xxxvii–xxxviii
errors in hypothesis testing, 489–490
statistical power, 497–501
Type I, 490–493
Type II, 493–497
escape sequences, 77–78
Euler's number (*e*), 17
events, 372–375
complement of, 375
dependent vs. independent, 373
intersection of, 372–374
union of, 374
`evmix` package, 345
exact argument matching, 200–201
Excel spreadsheet workbooks, 176–177
exception handling, 277–285
in body of function, 281–283
errors and warnings, 278–280
suppressing warning messages, 283–285
`try` statements, 280–285
execution environments, 195–196
exiting R, 13

`expand.grid` function
contour plots, 739
evaluation grid construction, 734–735, 772
parametric equations, 813
trivariate surfaces, 798, 800
`exp` command, 17, 35, 425
exponential distribution, 422–426
`dexp` function, 423–424
family, 425
`pexp` function, 424
`qexp` function, 425
`rexp` function, 425
exponentials, 17–18
`EXPR` argument, 219
`expression` function, 672–674, 683
extended regular expressions, 72
external datafiles, reading in, 172–179
spreadsheet workbooks, 176–177
table format, 172–176
web-based files, 177–179
extraneous (nuisance) variables, 556
extrapolation vs. interpolation, 536

F

`facet_grid` command, 696–699, 708
facets, 691
coloring, 764–767
in `ggplot2`, 695–701
single categorical variables, 695–698
two categorical variables, 698–699

`facet_wrap` command, 695, 708
`factor` function, 84–85, 102, 547
factors, 81–92
    collapsing, 87–88
    combining, 89–90
    creating from continuous values, 90–92
    defining and ordering levels, 83–85
    dropping categories, 86
    identifying categories, 81–83
    numeric representation of, 86–87
`faithful` object, 354
`FALSE` (`F`) logical value, 60–63, 102
    `else` statements, 213
    `if` statements, 212
    reserved names, 198–199
`family` argument, 670–671, 673, 683
`faraway` package
    facets, 697
    masking, 295, 297
    multiple linear regression, 586
    partial *F*-test, 603–604
F-distribution, 428–430
Fibonacci sequence, 246, 252–255, 264–265, 273, 279
`file.choose` command, 174, 180, 189
filepaths, 8
    specification, 78, 173–174
files, reading and writing, 169–189
    external datafiles, 172–179
        spreadsheet workbooks, 176–177
        table format, 172–176
        web-based files, 177–179

object read/write operations, 184–188
determining suitability, 187–188
`dput` and `dget`, 186–187
`save` and `load`, 185–186
`saveRDS` and `readRDS`, 185–186
R-ready datasets, 169–172
built-in, 170–171
contributed, 171–172
writing out data files and plots, 179–184
datasets, 179–180
plots and graphics files, 180–184
`fill` argument, 353
`filled.contour` function, 743–744, 772
`fitted` function, 527, 631, 643
`fitted.values` function, 631, 643
fitting
linear models with `lm` command, 524–526
logarithmic transformations, 578–581
polynomial transformations, 572–577
five-number summary, 318–319, 323, 332, 350–353, 531
floating-point (decimal) numbers, 134
`F` logical value. *See* `FALSE` logical value
`floor` function, 316, 332
`font` parameter, 671, 673, 683
fonts, 670–672
`foreign` package, 179
`for` loops, 223–231, 249
`apply` function vs., 238–239
`break` function and, 244
via index or value, 226–228
lazy evaluation, 259

Möbius strips, 817
nesting, 228–231
`next` function and, 245
perspective plots, 768
`repeat` statements and, 247
`try` statements and, 282
`format` argument, 93
`format` function, 95, 102
formula notation (symbolic notation), 522
forward selection (forward elimination), 605–609
forward slash (/) operator
filepath specification, 8, 78, 173
formatting dates, 94
`frac` parameter, 674, 683
frequentist (classical) probability, 372
F-test
omnibus, 552, 566–568, 640
partial on nested models, 601–604
`FUN` argument, 239–240, 272, 313–314, 512
function body (body code), 252
`function` command, 251–258
creating functions, 252, 272, 276
adding arguments, 254
returning results, 254–256
`return` statement, 256–258
functions. *See also* arguments
argument matching, 200–204
ellipses, 203–204
exact, 200–201
mixed, 203
partial, 201–202

positional, 202–203
calling, 193–207
documentation, 10–12
exception handling, 277–285
errors and warnings, 278–280
`try` statements, 280–285
`function` command, 251–258
creating functions, 252
`return` statement, 256–258
help files for, 10, 12
masking, 293–299
data frame variable distinction, 297–299
function and object distinction, 293–296
pipes, 205–207
syntax and examples, 205–207
whether to use, 207
progress and timing, 286–293
measuring completion time, 287–288
profiling code, 288–293
tracking progress, 286–287
scoping rules, 193–199
global environments, 194
local/execution environments, 195–196
package environments and namespaces, 194–195
reserved and protected names, 198–199
search path, 196–198
specialized, 269–276
anonymous, 272
helpers, 269–272
recursive, 273–276
writing, 251–276

G

`gal` dataset, 625
Galileo, 625
gamma distribution, 431
Gaussian distribution. *See* normal distribution
generic functions, 147
geographic projections, 365–367
`geom_bar` function, 336, 370
`geom_boxplot` function, 352–353, 370
`geom_density` function, 347–348, 370, 690–691
geometric distribution, 404
`geom_histogram` function, 341–342, 370
`geom_hline` function, 166, 168
`geom_line` function, 166–168, 694
`geom_point` function, 162, 168, 686
    aesthetic mappings, 164–166
    facets, 699
    independent plots, 694
    scatterplots, 361–362
    smoothing and shading, 687, 689
geoms (geometric modifiers), 162–164, 685
    aesthetic mapping with, 164–167
    setting appearance constants with, 162–163
`geom_segment` function, 166–168
`geom_smooth` function, 687, 689, 699, 708
`geom_violin` function, 356, 370
`geom_vline` function, 343, 370
`getOption` function, 839
`getwd` function, 8, 14

GGally package, 362
gghist function, 342
ggpairs function, 362, 370
ggplot2 package, 160–167, 685–708
ggplot function, 160–163, 166, 168
    ad hoc object read/write operations, 184–185
    aesthetic mapping with geoms, 164–167
    bar plots, 336–337
    box plots, 352–353
    density plots, 347–350
    graphics, 685–686
    histograms, 341–343
    multiple plots and variable-mapped facets
        facets mapped to categorical variables, 695–701
        independent plots, 692–694
    pie charts, 338
    plotting example, 160–162
    scatterplots and coordinate-based plots, 361–365, 369
    setting appearance constants with geoms, 162–163
    smoothing and shading, 686–691
        adding LOESS trends, 687–689
        constructing smooth density estimates, 689–691
    violin plots, 356–357
    writing graphics to file, 182–183
ggsave function, 182–183, 189
ggtitle function, 162, 168, 337, 691
ggvis package, 701–708
global environments, 194
goodness-of-fit
    complexity vs., 599–601
        guidelines for, 600–601

principle of parsimony, 600
GOF tests, 480–484
graphical parameters, 147–154
aspect ratio, 152–154
automatic plot types, 148
color, 149–151
controlling multiple plots in one device, 651
custom region limits, 151–152
line and point appearances, 151
style and suppression in base R, 665–667
title and axis labels, 148–149
graphical user interface. *See* base R GUI
graphics devices, 648–654
closing, 650–651
controlling multiple plots in one, 651–654
setting `mfrow`/`mfcol` parameter, 651–652
using `layout` function, 652–654
drivers, 647–648
opening manually, 648–649
switching between, 649–650
graphics files, 180–184
`graphics` package, 195
graphics tools
Shiny, 854
Sweave, 850–851
`gray.colors` function, 714–715, 772
`gray` function, 714, 772
greater than (`>`) operator, 61, 102
greater than or equal to (`>=`) operator, 61, 102
Greek symbols, 672–673
`grep` function, 78, 102

grid3d function, 779, 781, 784, 825
grid.arrange function, 692, 695, 697, 708
gridExtra package, 692, 697
grid function, 678, 683
grids
    evaluation type, 733–735
    in point clouds, 779–782
    points and pixels, 749–752
gridsize argument, 807
gsub function, 79, 102
GUI. *See* base R GUI

H

hash mark (#) character, 7, 14
hatvalues function, 630, 643
HCL (hue-chroma-luminance) specification, 725
hcl.colors function, 714, 725, 750, 772
hcl function, 725
hcl.pals function, 714
head command, 111, 118, 205–206
heat.colors function, 714, 725, 772
help (?) function, 10, 14
helper functions, 269–272
    externally defined, 270–271
    internally defined, 271–272
help files, 10–12
help function, 10, 14
help.search function, 12, 14
heteroscedasticity, 622–623

higher-dimensional plots, 709–772
bivariate surfaces, 733–736
contour plots, 736–749
perspective plots, 761–772
pixel images, 749–760
representing and using color in, 709–728
3D scatterplots, 728–733
high-level programming languages, 4
`highlight.3d` argument, 730
`hist` function, 340–341, 370
histograms, 339–343
bin width and customization, 341
example of, 340
in `ggplot2`, 341–343
violin plots vs., 354
homoscedasticity, 623
Hornik, K., 171
`hsv` function, 725
`%H` time marker, 98, 102
hue-chroma-luminance (HCL) specification, 725
hue-saturation-value (HSV) specification, 725
hypergeometric distribution, 404
hypothesis testing, 455–503
categorical variables, 480–489
`chisq.test` function, 483–484, 488
chi-squared test of distribution, 481–484
chi-squared test of independence, 485–488
single categorical variables, 481–484
two categorical variables, 484–489
components of, 455–458
hypotheses, 456

p-value, 455–457
significance level, 457–458
test statistic, 457
criticisms of, 458
errors, 489–490
statistical power, 498, 501
Type I, 490–493
Type II, 493–497
means, 458–472
one-sample *t*-test, 459–461
paired/dependent samples, 468–471
single means, 459–462
testing, 459–460, 464
`t.test` function, 461–462
two means, 462–472
unpaired/independent samples, 463–468
proportions, 473–480
one-sample *Z*-test, 474–476
`prop.test` function, 475–476, 478
single, 473–476
two, 476–480
two-sample *Z*- test, 476–478
significance level in, 457–458
statistical power, 497–502
power curves, 500–501
simulation, 498–500
test statistic in, 457

I

`ice.river` dataset, 171

IDE (integrated development environment). *See* RStudio
identity matrix, 46
`ifelse` function, 214–215, 249
`if` statements, 209–223, 249
    element-wise checks with `ifelse`, 214–215
    ellipses, 265
    `else` statements, 213–214
    `missing` function and, 263
    nesting/stacking, 216–218
    stand-alone statements, 210–213
    `switch` function vs., 218–220
`I` function, 575, 597
`image` function, 749–750, 764, 772
implicit looping, 238–243
    `apply` function, 242
    `lapply` function, 241
    `sapply` function, 241
    `tapply` function, 240
    whether to use, 242–243
`include.lowest` argument, 91
independent, identically distributed (iid) observations, 436
independent/unpaired samples testing
    pooled variance, 465–468
    unpooled variance, 463–465
`INDEX` argument, 313, 512
indexes, 26–28
    converting logical flag vectors into numeric index vectors, 69
    lazy evaluation, 259
    repeating, 29
    retrieving components from lists, 104
    sequences of, 29

vectors of, 29–30
infinite loops, 232
infinity, 120–122
inflections (saddles), 578
`Inf` object, 120–122, 143, 198, 214
`-Inf` object, 120–122, 143, 198, 461
information-based (criterion-based) model selection methods, 613
inheritance, 132, 135–136
`inherits` function, 136
`input_checkbox` command, 704
`input_numeric` command, 704
`input_radiobuttons` command, 704, 708
`input_select` command, 704
`input_slider` command, 702, 706, 708
`input_text` command, 704
`inside.owin` function, 756, 772
installing R. *See* downloading and installing R
`install.packages` function, 9–10, 14, 839, 848
instructors, recommendations to, xxxix–xl
integrated development environment (IDE). *See* RStudio
integration, 384
`interaction.plot` function, 514–515, 519
interaction plots, 514–515
interactive effects
concept of, 584–586
higher-order interactions, 593–594
one categorical, one continuous predictor, 586–589
two categorical predictors, 590–591
two continuous predictors, 591–593
intercept and slope parameters, 157, 524–525

interpolation vs. extrapolation, 536
intersection of two events, 373–374
intervals
    interpretation, 532–534
        confidence intervals, 533
        prediction intervals, 533–534
    for means, 446–449
    plots, 534–536
    for proportions, 449–450
inversion of matrices, 49–51
IQ (intelligence quotient), 466–468
IQR (interquartile range), 321–323
`IQR` (interquartile range) command, 322–323
`iris` dataset
    point clouds, 773–779, 782–783
    scatterplots, 358–361, 729–732
`is.data.frame` function, 136–137
is-dot (object-checking) functions, 136–137. *See also names of specific is-dot functions*
`is.finite` function, 121, 143, 244
`is.infinite` function, 121, 143
`is.integer` function, 136–137
`is.logical` function, 137
`is.mat` function, 227–228
`is.matrix` function, 136–137, 228
`is.na` function, 125, 143
`is.nan` function, 123, 143
`is.null` function, 128, 143
`is.numeric` function, 134, 136–137, 279
isosurfaces, 799–804

higher-dimensional probability densities, 799–801
multiple level control with color and opacity, 803–804
nonparametric trivariate density, 804–810
one-level, 802–803
trivariate evaluation lattice, 801–802
`is.vector` function, 136–137
`italic` function, 674, 683

J

`jitter` function, 320, 332
jittering, 320
Journal of Statistics Education, 177
.jpeg files, 180–181
`jpeg` function, 180–181, 189

K

`kde2d` function, 772
bivariate surfaces, 792–793
nonparametric bivariate density, 740–741, 753, 755
nonparametric trivariate density, 804
`kde` function, 806–807, 825
Kendall's correlation coefficient, 328
kernel density estimation (KDE)
density plots, 344
nonparametric bivariate density, 740–742, 746, 753–755
smooth density estimates, 689–691
trivariate surfaces, 806–807
kernel weight, 344

keywords (reserved names), 198–199
`knitr` package, 851
`kruskal.test` function, 517, 519
Kruskal–Wallis test, 516–518
    example and interpretation, 516–518
    hypotheses, 516
`ks` package, 9–10, 806, 832–833

L

labels. *See* text and labels
`labels` argument, 91
`labs` function, 691
`lapply` function, 241, 249
`las` parameter, 669–670, 683
`lat.asp` function, 366–367
`layer_histogram` command, 702, 708
`layer_points` command, 704, 708
`layer_smooths` command, 705–706, 708
`layout` function, 652–654, 683
`layout.show` function, 653
lazy evaluation, 258–261
least-squares regression (line of best fit), 526–528
`legend3d` function, 775–777, 825
`legend` function, 154, 158–159, 168, 341
legends
    color, 720–722, 724, 772
    point clouds, 775–778
`length` function, 82
    lists, 104

vectors, 22, 26–29, 35
less than (<) operator, 61, 102
less than or equal to (<=) operator, 61, 102
`letters` function, 734, 772
`levels` function, 82, 102, 131
Levene's test, 506
leverage
calculating, 629–630
graphically combining with Cook's distance and residuals, 635–638
in linear regression, 627–630
lexical scoping, 196
libraries, 9. *See also names of specific libraries*; packages
`library` function, 9, 14, 830–831, 848
linear regression, 599–643
collinearity, 640–643
goodness-of-fit vs. complexity, 599–601
influence in, 627–629
multiple, 555–597
residual diagnostics, 620–640
selection algorithms, 601–618
simple, 521–553
line of best fit (least-squares regression), 526–528
lines
in plotting, 151, 155–156
in point clouds, 779–782
`lines3d` function, 779–780, 825
`lines` function, 154, 158, 168, 577
`linetype` function, 163, 168
line type function. *See* `lty` function

line width (`lwd`) function, 147, 151, 168, 667–668
`list.files` function, 173, 189
`list` function
    ellipses, 266
    list creation, 104, 106, 118
    variadic behavior, 203–204
lists of objects, 103–109
    creating and accessing components, 104–105
    named lists, 88
    naming members, 106–107
    nesting, 107–109
    slicing, 105
literal regular expressions, 72
`lm` command
    multiple linear regression, 560
    simple linear regression, 525–526, 540, 548–553
`load` command, 13, 185–189
local environments, 195–196
local files, installing contributed packages using, 836–837
locally estimated scatterplot smoothing (LOESS), 687–689, 694, 705–706, 723
locally weighted scatterplot smoothing (LOWESS), 687
`locator` command, 659–661, 683, 712
loci in parametric equations, 811–815
    2D circles, 811–812
    3D cones, 814–815
    3D cylinders, 811–814
`loess` function, 688–689, 708
logarithmic transformations, 578–582
    fitting, 579–581
    plotting fit, 581–582

logarithms, 17–18, 579
`log` function, 17–18, 35
logical operators, 65
logical values (logicals), 25, 59–72
    logical flag vectors, 68–69
    logical operators, 64–67
    numeric representation of, 67–68
    relational operators, 60–64
    subsetting and extracting with, 68–72
    subsetting data frames with, 115–116
    `TRUE` and `FALSE`, 60
`longlat.conv` function, 367
loops, 223
    `break`, 243–244
    `for`, 223–231
        via index or value, 226–228
        nesting, 228–231
    implicit looping, 238–243
        `apply` function overview, 238–240
        `apply` function variants, 240–241
        whether to use, 242–243
    infinite, 232
    `next`, 244–246
    `repeat`, 246–247
    rotating perspective plots with, 767–769
    `while`, 232–238
        condition object modification, 233–238
lower-tailed tests, 456
LOWESS (locally weighted scatterplot smoothing), 687
low-level programming languages, 4
`ls` command

ad hoc object read/write operations, 184–185
assigning objects, 20, 35
scoping, 194–195, 207
`lty` (line type) function
box customization, 667–668
plotting, 147–148, 151, 168
`lty.hplot` argument, 730
`lubridate` package, 92
lurking variables, 556
`lwd` (line width) function, 147, 151, 168, 667–668

M

`magrittr` package, 207, 702
`main` argument, 147, 168, 730, 774
Mann–Whitney U test (Wilcoxon ranksum test), 471, 516
`mapping` function, 161, 168
`mapply` function, 241
`MARGIN` argument, 239
margins and regions, 654–659
clipping, 657–659
customizing limits, 151–152
figure region, 655
outer region/margins, 655
plot region, 655
spacing, 655–657
markup tools
Quarto, 852–854
R Markdown, 851–852
Sweave, 850–851

`mar` parameter, 655, 683
masking, 198, 293–299, 722
data frame variable distinction, 297–299
function and object distinction, 293–296
clashing package objects, 295–296
unmounting packages, 296
`MASS` package, 295–297
`mat` argument, 259–260
mathematics, 15–19
abstractions, 815–824
Möbius strips, 815–817
tori, 817–820
arithmetic, 15–17
conventions and styles, xxxvii–xxxviii
e-notation, 18–19
expressions, 673–675
logarithms and exponentials, 17–18
matrices, 37–51
creating, 38–40
dimensions, 40
filling direction, 38–39
naming rows and columns, 39
row and column bindings, 39–40
operations and algebra, 45–51
addition and subtraction, 47–48
identity matrix, 46
inversion, 49–51
multiplication, 48–49
scalar multiple, 47
transposition, 45–46
singular, 49

subsets, 40–45
extracting rows, columns, and diagonals, 41–42
omitting and overwriting, 42–45
`matrix` command
argument matching, 201–202
attributes, 131
`if` statements, 212
matrix creation, 38, 56–57
scoping, 193, 195
matrix inversion (`solve`) function, 50, 57, 558, 630
matrix multiplication (·) symbol, 558
matrix notation, 557–558
matrix product (`%*%`) operator
linear regression, 558–559, 630, 678
matrices, 49, 57
matrix transposition (`t`) function, 46, 57, 558, 630
`maxColorValue` argument, 722–723
`max` function, 310–312, 332
`%m` date marker, 93, 94, 102
mean (arithmetic mean), 309
intervals for, 446–449
measures of centrality, 309
sampling distributions for, 436–441, 438
standard deviation known, 437
standard deviation unknown, 437
temperatures example, 437–441
testing, 458–472
single mean, 459–462
two means, 462–472
mean (expectation; expected value)
continuous random variables, 386–388

discrete random variables, 379–380
`mean` argument, 801
`mean` function, 310–312, 314, 332
mean squared error (MSE) effect, 508–509
mean squared group (MSG) effect, 508
median, 309
`median` function, 310–312, 332
member references, 104
`mfcol` parameter, 652–654, 683
`mfrow` parameter, 651–654, 683
`mgp` parameter, 669–670, 683
`microbenchmark` package, 292
Microsoft Access databases, 179
Microsoft Excel spreadsheet workbooks, 176–177
`min` function, 310–312, 332
mirrors, 839
`misc3d` package, 802, 808
`missing` function, 263–264, 276
Möbius strips, 815–817
modality in distributions, 388–391
mode, 310
model-based inference, 528
model selection algorithms, 601–618
backward selection, 609–613
forward selection, 605–609
partial *F*-test, 601–604
stepwise AIC selection, 613–618
`month.abb` object, 94, 102
`month.name` object, 94, 102
motivation in interactive terms, 584–586

`mtcars` dataset
bar plots, 334–335
bivariate surfaces, 788
clipping, 657–658
contour plots, 738–739, 744
Cook's distance, 633–634
density plots, 344–348
histograms, 340
inspecting and interpreting residuals, 623
interactive effects, 591–593
numeric transformation, 573–576, 579–582
perspective plots, 761
pipes, 205
stepwise AIC selection, 614–618
`mtext` function, 657, 672, 678, 683
`%M` time marker, 98, 102
multicollinearity. *See* collinearity
multidimensional arrays, 37, 51–57
creating, 52–53
deleting and overwriting elements in, 56
extractions, 53–56
subsets, 53–55
multilevel variables, 542–547
dummy coding, 542–543
linear regression model of, 544–546
predictions from, 546–547
multi-modal distributions, 388–389
multinomial distribution, 404
multiple linear regression, 555–597
interactive terms, 584–597
concept and motivation, 584–586

higher-order interactions, 593–595
one categorical, one continuous predictor, 586–589
two categorical predictors, 590–591
two continuous predictors, 591–593
numeric transformation, 572–584
logarithmic transformations, 578–582
polynomial transformations, 572–578
partial effects in, 563–564
R implementation and interpretation, 560–571
adding predictors, 560–563
computing omnibus *F*-test, 566–568
finding confidence intervals, 565
interpreting partial effects, 563–564
model selection, 561–562
predicting from multiple linear model, 568–569
visualizing the model, 564–565
terminology, 556
theory, 556–559
example of, 558–559
extending simple model to multiple model, 557
representing in matrix notation, 557–558
multiple testing problem, 492
multiplication (*) operator, 15, 35, 47, 57
multivariate data, 306–307
`mvtnorm` package, 800

N

`NA` (not available) notation, 124–126, 143, 198, 215
named lists, 88
`names` function, 106–107, 118, 131, 227

NaN (not a number) notation, 122–124, 143, 198–199
na.omit function, 126, 143, 686, 696, 717
na.rm argument, 312, 766
natural log, 17
nchar function, 73, 102
ncol function
    arguments, 201, 259
    data frame construction, 111, 118
    matrix creation, 38, 40, 57
negative binomial distribution, 404
newdata argument, 535, 541–542, 581
newline (\n) escape sequence, 77, 149, 657, 672
news function, 838
next function, 244–246, 249
no-change (baseline) hypotheses, 456
nominal categorical variables, 304–305
nonparametric bivariate density
    Chorley–Ribble data, 752–757
    earthquake data, 740–743
nonparametric smoothing, 687
nonparametric trivariate density, 804–810
    isosurface level selection, 806–810
    raw data, 805–806
    3D estimate calculation, 806
normal (Gaussian) distribution, 410–419
    dnorm function, 412
    example of, 418–419
    pnorm function, 412–415
    qnorm function, 415–417
    rnorm function, 417–418

normality, assessing, 626–627
normalization, color indexing via, 718–720
`normalize` function, 718, 730
NOT (`!`) operator, 65, 102
not a number (`NaN`) notation, 122–124, 143, 198–199
not available (`NA`) notation, 124–126, 143, 198, 215
not equal to (`!=`) operator, 61, 102
`nrow` function
    arguments, 201, 259
    data frame construction, 111, 118
    matrix creation, 38, 40, 57
    nonparametric bivariate density, 756
`nuclear` dataset
    backward selection, 609–613
    forward selection, 605–608
    higher-order interactions, 594–595
nuisance (extraneous) variables, 556
null hypotheses, 456
`NULL` value, 127–129, 143, 198, 290
numerical overflow and underflow, 121
numeric variables, 304

O

object-checking (is-dot) functions, 136–137. *See also names of specific is-dot functions*
object-oriented programming, 132
objects
    as-dot coercion functions, 87, 137–142
    assigning, 19–21

attributes, 130–132
classes, 132–136
data structures, 134–135
inheriting from multiple, 135–136
stand-alone vectors, 133–134
deleting, 20–21
is-dot functions, 136–137
masking, 293–296
clashing package objects, 295–296
unmounting packages, 296
naming, 20
read/write operations, 184–188
determining suitability, 187–188
`dput` and `dget`, 186–187
`save` and `load`, 185–186
`saveRDS` and `readRDS`, 185–186
observation window (`owin`) object class, 753, 756
`OlsonNames` function, 98, 102
`oma` parameter, 655, 683
Omegahat, 839
omnibus *F*-test, 552, 566–568, 640
one-sided hypotheses, 456
one-way ANOVA. *See* one-way *under* ANOVA
opacity, 722–724, 803–804
OpenGL, 773
`options` function, 6, 14
`ordered` argument, 84, 547
ordinal categorical variables, 304
OR element-wise (`|`) operator, 64–66, 102
origin coordinate, 367

OR single comparison (||) operator, 64–66, 102, 212
orthogonal contrasts, 547
outer function, 813, 825
outliers, 329–332, 627–630
owin (observation window) object class, 753, 756

P

packages (libraries), 9–10. *See also names of specific packages*
    base, 829–830
    citing R and contributed, 840–841
    clashing objects in, 295–296
    contributed, 830–837
        finding on CRAN, 832–833
        installing at prompt, 833–834
        installing using GUI, 834–836
        installing using local files, 836–837
    environments and namespace, 194–196
    installing, 9–10, 833–837
    loading, 9
    recommended, 830
    unmounting, 296
    updating, 10, 838
    using, 829
    writing, 841
paired/dependent samples testing, 468–471
pairs function, 360, 370
palette function, 710–711, 772
palettes, 710, 714
    built-in color, 714–715, 772
    custom color, 715–716

parameters. *See also* graphical parameters
in simple linear regression model, 524
statistics vs., 307–308
parametric equations, 810–824
loci, 811–815
2D circles, 811–812
3D cones, 814–815
3D cylinders, 811–814
mathematical abstractions, 815–824
Möbius strips, 815–817
tori, 817–820
parametric response surface, 738–740
parenthesis. *See* `( )` characters
`par` function, 651–655, 683
parsimony principle, 600
partial argument matching, 201–202
`paste` function, 74–76, 102, 137–138, 674
`pbeta` function, 431
`pbinom` function, 396, 398, 432
`pch` (point character) parameter, 147–148, 151, 168
`pch03d` function, 777–778, 780, 825
`pchisq` function, 427–428, 432
`pcol` function, 712–713
PDF files, storing plots as, 181–182
`pdf` function, 181–182, 189, 648
Pearson's product-moment correlation coefficient, 325–328
PEMDAS mnemonic, 16
percentages, calculating, 313–314
percentiles, 318
Perl regular expressions, 72

persp3d function, 788–793, 813, 816, 825
perspective plots (wireframe plots), 761–772
    coloring facets, 764–767
    interpreting and adjusting angles, 761–764
    rotating with loops, 767–769
persp function, 761–763, 768, 772
persprot function, 768–769
pexp function, 424, 432
pf function, 429–430, 432
p-functions, 393, 431
pgamma function, 431
pgeom function, 404
pharmaceuticals, interactive effects between, 585, 591–593
phi argument, 761–763, 767–768
phyper function, 404
pie charts, 337–339, 338
pie function, 337, 370
pi function, 811, 825
pipe (|>) operator, 205–207, 702
pipe (%>%) operator, 207, 702
pipes, 205–207
PIs. *See* prediction intervals
pixel images, 749–760
    grid points and pixels, 749–752
    nonparametric bivariate density estimate, 752–757
    surface truncation and empty pixels, 752–760
planar point pattern (ppp) objects, 752
plot3d function, 825
    color and legends, 776
    points and spheres, 774–775

scatterplot example, 783
trivariate surfaces, 798, 803, 805, 809
plot customization, 647–683
graphics devices, 648–654
closing, 650–651
controlling multiple plots in one, 651–654
`layout` function for multiple plots in one, 652–654
manually opening new, 648–649
`mfrow`/`mfcol` parameters for multiple plots in one, 651–652
switching between devices, 649–650
point-and-click coordinate interaction, 659–665
placing ad hoc annotations, 661–662
retrieving coordinates silently, 659–660
visualizing selected coordinates, 660–661
regions and margins, 654–659
clipping, 657–659
custom spacing, 656–657
default spacing, 655–656
figure region, 655
outer region/margins, 655
plot region, 655
scatterplot example, 675–679
text and label notation, 670–675
font, 670–672
Greek symbols, 672–673
mathematical expressions, 673–675
traditional R plots, 665–670
axis customization, 668–670
box customization, 667–668
graphical parameters for style and suppression, 665–667

`plot` function, 145–148, 168. *See also* plots and plotting
adding points, lines, and text, 154–160
density plots, 344–346
ellipses, 204
linear regression model diagnostics, 624, 643
simple linear regression, 522
visual controls and graphical parameters, 147–154
aspect ratio, 152–154
automatic plot types, 148
color, 149–151
custom region limits, 151–152
line and point appearances, 151
title and axis labels, 148–149
`plotmath` mode, 672
`plotrix` package, 722
plots and plotting, 145–168, 773–825. *See also* plot customization
adding points, lines, and text, 154–160
bivariate surfaces, 733–736, 788–797
adding components, 789–791
aspect ratio, 792–794
coloring by z value, 792
constructing evaluation grid, 733–735
plotting perspective surfaces, 788–789
z-matrices, 735–736
color, 709–728
built-in palettes, 714–715
custom palettes, 715–716
indexing via categorization, 716–718
indexing via normalization, 718–720
legends, 720–722
opacity, 722–724

RGB alternatives, 725
RGB hexadecimal codes, 710–713
contour plots, 736–749
color-filled contours, 743–746
contour lines, 737–743
coordinate-based, 365–369
in `ggplot2`, 369
longitude and latitude, 365–369
pre-existing distance-based units, 365
`ggplot2` package, 160–167
aesthetic mapping with geoms, 164–167
displaying multiple plots, 691–701
plotting toy example, 160–162
setting appearance constants with geoms, 162
parametric equations, 810–824
perspective plots, 761–772
coloring facets, 764–767
interpreting and adjusting angles, 761–764
rotating with loops, 767–769
pixel images, 749–760
grid points and pixels, 749–752
nonparametric bivariate density estimate, 752–757
surface truncation and empty pixels, 752–760
point clouds, 773–787
color and legends, 775–778
customized interactive 3D scatterplot, 782–785
3D cloud points and spheres, 774–775
3D lines, axes, grids, and other annotations, 779–782
3D point characters, 777–778
3D text, 778–779
3D scatterplots, 728–733

customized interactive, 782–785
syntax, 729
visual enhancements, 730–732
trivariate surfaces, 798–810
evaluation coordinates in 3D, 798–799
isosurfaces, 799–804
nonparametric trivariate density, 804–810
visual controls and graphical parameters, 147–154
writing out, 180–184
`pnbinom` function, 404
`png` function, 180, 189, 648
`pnorm` function, 412–415, 418, 432, 444
point character (`pch`) parameter, 147–148, 151, 168
point clouds, 773–787
color and legends, 775–778
customized interactive 3D scatterplot, 782–785
3D cloud points and spheres, 774–775
3D lines, axes, grids, and other annotations, 779–782
3D point characters, 777–778
3D text, 778–779
`points3d` function, 775, 789, 825
`points` function
adding points to existing plots, 154, 157–158, 168
interval plots, 534
linear regression model diagnostics, 629
Poisson distribution, 400–404
`dpois` function, 402–403
`ppois` function, 402–403
`qpois` function, 403
`rpois` function, 403
`polygon` function, 391

continuous random variables, 384
probability density functions, 407, 414–415
sampling distributions, 439
polynomial contrasts, 547
polynomial curvature, 572–573
polynomial transformations, 572–578
fitting, 573–577
plotting, 577–578
pitfalls of, 578
pooled estimate of the variance, 466
pooled variance testing, 465–468
Posit, 843
positional argument matching, 202–203
`postscript` function, 181, 189
power curves of statistical power, 500–501
`power.tester` function, 499–500
`ppois` function, 402–403, 432
`ppp` (planar point pattern) objects, 752
`predict` command, 533–535, 541–542, 553, 577
prediction intervals (PIs), 532
confidence intervals vs., 531–532
for individual observations, 533–534
predictions
from binary variables, 541–542
from fitted simple linear models, 531–538
confidence interval vs. prediction interval, 531–532
interpolation vs. extrapolation, 536
interval interpretation, 532–534
interval plots, 534–536
from multilevel variables, 546–547

from multiple linear models, 568–569
`print` function, 162, 280
probability, 371–391
complement, 375–376
conditional, 373
dependent vs. independent events, 373
distributions of, 393–432
Bernoulli, 394–395
binomial, 395–400
chi-squared, 426–428
exponential, 422–426
F-distribution, 428–430
normal, 410–419
Poisson, 400–404
Student's *t*-distribution, 419–422
uniform, 404–410
events and, 372–373
frequentist, 372
intersection of events, 373–374
random variables
continuous, 376–377, 380–388
discrete, 376–380
probability distributions and, 376–391
shape, skew, and modality in, 388–391
realizations, 376–377
union of events, 374
probability density functions, 381–385, 404–431
chi-squared distribution, 426–428
exponential distribution, 422–425
F- distribution, 428–430
higher-dimensional, 799–801

normal distribution, 410–419
Student's *t*-distribution, 419–422
uniform distribution, 404–410
probability mass functions, 377, 393–404
Bernoulli distribution, 394–395
binomial distribution, 395–400
Poisson distribution, 400–404
`proc.time` function, 288
`prod` function, 34–35, 312
profiling
progress and timing, 288–293
measuring completion time, 287–288
tracking, 286–287
in RStudio, 848–849
`profvis` package, 292
proportions, 313–314
intervals for, 449–450
sampling distributions for, 441–444
testing, 473–480
single proportions, 473–476
two proportions, 476–478
`prop.test` function, 475–476, 478, 502
protected names, 198–199
`pt` function, 421–422, 432
`punif` function, 408, 432
p-value, 455–457
`pweibull` function, 431

Q

qbeta function, 431
qbinom function, 396, 399, 432
qchisq function, 427–428, 432
qexp function, 425, 432
qf function, 429–430, 432
q function, 13–14
q-functions, 393, 431
qgamma function, 431
qgeom function, 404
qhyper function, 404
qnbinom function, 404
qnorm function, 415–418, 432
qplot function, 161, 336
qpois function, 403
qqlined function, 417
qqline function, 416, 432
qqnorm function, 416–417, 432
qt function, 420–421, 432
quads3d function, 782
quakes dataset
    box plots, 350–352
    color opacity, 723
    contour plots, 745
    graphics devices, 648–650
    longitude and latitude, 366–368
    perspective plots, 768
    summary statistics, 311, 319, 322, 327
    trivariate surfaces, 805–809
    univariate and multivariate data, 306
    violin plots, 355–356

`quantile` function, 318–319, 332
quantile-quantile (QQ) plots
    assessing normality, 626
    Kruskal–Wallis test, 517
    normal distribution, 415–418
    one-way ANOVA, 506
quantiles, 318
`quarters` function, 95, 102
Quarto, 852–854
`qunif` function, 408–409, 432
`qweibull` function, 431

R

radians, 366
`radius` argument, 775
`rainbow` function
    built-in palettes, 714, 772
    coloring facets, 766
    hue-saturation-value, 725
    nonparametric bivariate density, 753
    opacity, 722
    3D lines, 781
random error, 445
random variables, 376–391
    continuous, 376–377, 380–388
        concept, 380
        cumulative probability distributions, 385–386
        mean and variance, 386–388
        probability density functions, 381
    discrete, 376–380

cumulative probability distributions, 378
mean and variance, 379–380
realizations, 376–377
shape, skew, and modality, 388–391
randomly generated realizations, 399
`range` function, 310–312, 332
raw data, describing, 303–308
categorical variables, 304
examples, 305
numeric variables, 304
parameters vs. statistics, 307–308
univariate and multivariate data, 306–307
`rbenchmark` package, 292
`rbeta` function, 431
`rbind` function, 39–40, 57, 113–114
`rbinom` function, 396, 399, 432
`rchisq` function, 427–428, 432
`RColorBrewer` package, 714, 725
R Core Team, xxxii
.RData extension, 13
`read.csv` command, 176–177, 189
`readLines` function, 177
`readRDS` command, 185–186, 188–189
`read.table` command, 173–176, 179, 189
ready-to-use functions, 9. *See also names of specific built-in functions*
realizations, 376–377, 399
recursive functions, 273–276
red-green-blue (RGB) values
alternatives to, 725
hexadecimal color codes, 710–713

triplets, 710
reference categories/levels, 85
regions and margins, 654–659
clipping, 657–659
customizing limits, 151–152
figure region, 655
outer region/margins, 655
plot region, 655
spacing, 655–657
regression coefficient significance tests, 529–530
relational operators, 60–64
`relevel` function, 85, 102, 547–548
`repeat` statements, 246–247, 249, 254
repetitions, 24–25
`rep` function, 22, 24–25, 29, 35, 780
`replacement` argument, 79
reserved names (keywords), 198–199
`resid` function, 527
residuals
assumptions, 523–524
diagnostics, 506, 620–640
assessing normality, 626–627
calculating Cook's distance, 631–634
calculating leverage, 629–630
combining residuals, leverage, and Cook's distance, 635–638
illustrating outliers, leverage, and influence, 627–630
inspecting and interpreting, 622–626
illustrating, 526–528
residual sum of squares (RSS), 557

response surfaces, 593
`return` statements, 252, 256–258, 276
`rev` function, 688, 708
`rexp` function, 425, 432
`rf` function, 429–430
R-Forge, 839
`r`-functions, 393
`rgamma` function, 431
`rgb` function, 711–712, 718, 722–723, 772
RGB values. *See* red-green-blue values
`rgeom` function, 404
`rgl` package, 773–825
    bivariate surfaces, 788–797
        adding points, 789
        adding surfaces, 789–791
        aspect ratio, 792–794
        coloring by *z* value, 792
        plotting perspective surfaces, 788–789
    parametric equations, 810–824
    point clouds, 773–787
        color and legends, 775–778
        custom, interactive 3D scatterplot, 782–785
        points and spheres, 774–775
        3D lines, axes, grids, and annotations, 779–782
        3D point characters, 777–778
        3D text, 778–779
    trivariate surfaces, 798–810
        evaluation coordinates in 3D, 798–799
        isosurfaces, 799–804
        nonparametric trivariate density example, 804–810
.Rhistory file, 13

rhyper function, 404
ring tori, 817–820
RJDBC package, 179
R language, 827–841. *See also* base R GUI
citing, 840–841
downloading and installing, 9–10, 827–837
base packages, 829–830
contributed packages, 830–837
finding packages on CRAN, 832–833
packages at prompt, 833–834
packages using GUI, 834–836
packages using local files, 836–837
recommended packages, 830
FAQs, 827
history of, xxxi–xxxii
home page, 828
mirrors and repositories, 839
switching CRAN mirror, 839
updating, 10, 838
using packages, 829
writing packages, 841
R Markdown, 851–852
rmarkdown package, 851
rm command, 20–21, 35, 199
rmultinom function, 404
rmvnorm function, 800–801, 825
RMySQL package, 179
rnbinom function, 404
rnorm function, 417–418, 432
robustness, 331
RODBC package, 179

`round` function, 315, 332, 397
`[row, column]` operator, 40, 57, 112, 118
`row.names` argument, 180
`rownames` function, 111, 118, 791
`rowSums` function
    chi-squared tests of independence, 486
    implicit looping, 239
    measuring computational expense, 288–289, 300
    profiling, 291
`rpois` function, 403, 432
`Rprof` function, 289–290, 300
`R>` prompt, xxxvi, 6
RSS (residual sum of squares), 557
`rstandard` function, 627, 643
RStudio, 4–5, 843–854
    downloading, 843
    `ggvis` package, 701
    graphics devices, 146
    layout and usage, 844–846
        editor features and appearance options, 844–845
        pane customization, 845–846
    markup, document, and graphics tools, 850–854
        `knitr` package, 851
        Quarto, 852–854
        R Markdown, 851–852
        Shiny, 854
        Sweave, 850–851
    shortcuts, 7
    package manager, 847–848
    tools and features, 846–850
        package manager, 847–848

projects, 846–847
support for debugging, 849–850
support for profiling, 848–849
user guide, 844
working directory, 7–9
`rt` function, 422, 432
`runif` function, 409, 432
`rweibull` function, 431

S

S3 classing structure, 133
S4 classing structure, 133
saddles (inflections), 578
`sample` function, 819, 825
sampling distributions, 435–445
for finite-sized samples, 445
for a sample mean, 436–441
standard deviation known, 437
standard deviation unknown, 437
temperature example, 437–441
for a sample proportion, 441–444
`sapply` function, 241, 249, 259, 266
`save` command, 13, 185–189
`saveRDS` function, 185–186, 188–189
saving work, 13–14
scalar multiple value, 47
`scale_fill_grey` function, 336, 370
`scale_linetype_manual` function, 343, 370
`scale_shape_manual` function, 361, 370

scale_x_discrete function, 336, 370
scale_y_continuous function, 336, 370
scan function, 177
scatterplot3d function, 728–730, 772, 782
scatterplots, 357–369
    covariance and correlation, 325–328
    customizing example, 675–679
    in ggplot2, 361–365
    matrix of, 359–361
    single plot, 358–359
    3D, 728–733
        customized interactive, 782–785
        syntax, 729
        visual enhancements, 730–732
scatter.smooth function, 688
scope argument
    forward selection, 606, 609
    stepwise AIC selection, 614
scoping, 193–199
    environments, 194–196
        global, 194
        local/execution, 195–196
        package environments and namespaces, 194–195
    lexical, 196
    reserved and protected names, 198–199
    search path, 196–198
scripts, 6
    saving as plain text files, 14
sd (standard deviation) command, 322, 332
search function, 196, 207, 294
search help (??) function, 12, 14

`segments3d` function, 779–781, 783–784, 791, 825
`segments` function
    adding lines to plots, 154, 157, 168
    interval plots, 534
    probability density functions, 407
    simple linear regression, 527
semicolon (;) character, 746, 784
separation (+) operator, 513, 560
`sep` argument, 75–76, 173, 178
`seq` function, 22–24, 35, 197, 341
sequences, creating, 23–24
serialization, 186–188
`setTxtProgressBar` function, 286, 300
`setwd` function, 8, 14, 173
`shade` argument, 763
shading. *See* smoothing and shading
`shape` argument, 163, 168
`shape` package, 720, 722, 750
`shapiro.test` function, 627, 643
Shapiro–Wilk test, 506, 627
Shiny, 854
`side` argument, 781
`silent` argument, 283
simple linear regression, 521–553
    categorical predictors, 538–553
        binary variables, 538–542
        changing reference level, 547–548
        checking for equivalence with one-way ANOVA, 551–552
        multilevel variables, 542–547

treating categorical variables as numeric, 548–551
concepts, 523–528
defining models, 523–524
estimating intercept and slope parameters, 524–525
fitting linear models with `lm` command, 525–526
illustrating residuals, 526–528
explanatory variable, 521, 523
extending to multiple linear regression, 557
linear relationship example, 521–523
predictions, 531–538
confidence interval vs. prediction interval, 531–532
interpolation vs. extrapolation, 536
interval interpretation, 532–534
interval plots, 534–536
response variable, 521, 523
statistical inference, 528–531
coefficient of determination, 530
regression coefficient significance tests, 529–530
`summary` command, 528–531
`sin` function, 811, 825
singular matrices, 49
`size` argument, 163, 168
skew, 340, 388–391
in distributions, 319, 388–391
`skip` argument, 179, 768
S language, xxxi
sleep, pausing execution in R via, 286–287
`sleep` argument, 768
slope and intercept parameters, 157, 524–525
smoothing and shading, 686–691
adding LOESS trends, 687–689

constructing smooth density estimates, 689–691
smoothing parameter. *See* bandwidth
`solve` (matrix inversion) function, 50, 57, 558, 630
`sort` function, 22, 25, 35
sorting, 25
spacing
customizing margins, 656–657
default margins, 655–656
`span` argument, 689
spatial coordinates, plots of. *See* coordinate-based plots
`spatstat` packages, 752–753, 756
Spearman's correlation coefficient, 328
special values, 119–130
infinity, 120–122
`NA`, 124–126
`NaN`, 122–124
`NULL`, 127–130
`spheres3d` function, 775
`split.screen` function, 654
spread, measures of, 319–324
example, 319–320
interquartile range, 321
R implementation, 322–324
standard deviation, 321
variance, 321
spreadsheet workbooks, 176–177
`sqrt` function
categorical predictors, 552
square roots, 16, 35
variance and standard deviation, 322, 380

square-bracket ([ ]) characters, 27–28, 35, 70, 105, 118
square roots, 16
`srt` argument, 678, 683
SS (sum-of-squares) value, 508–509
standard deviation, 321–322
standard deviation (`sd`) command, 322, 332
standard normal (Gaussian) distribution, 411
statistical inference, 528–531
    coefficient of determination, 528, 530
    five-number summary, 531
    regression coefficient significance tests, 529–530
    residual standard error, 530
    `summary` command, 528–531
statistical power, 497–502
    power curves, 500–501
    simulation, 498–500
statistics, 303–332
    describing raw data, 303–308
        categorical variables, 304
        examples, 305
        numeric variables, 304
        parameters vs. statistics, 307–308
        univariate and multivariate data, 306–307
    parameters vs., 307–308
    summarizing data, 309–332
        counts, percentages, and proportions, 313–314
        measures of centrality, 309–313
        measures of spread, 319–324
        outliers, 329–332
        quantiles, percentiles, and five-number summary, 318

relationships between numeric variables, 324–328
rounding, floors, ceilings, and absolute values, 315–317
`step` function, 615, 643
`%S` time marker, 98, 102
stochastic mechanisms, 376
`stop` command, 278, 300
stopping conditions, 273
`str` function, 110, 118
strings
concatenating, 74–77
creating, 73–74
`stringsAsFactors` argument, 175
`strptime` function, 98–99, 102
students, recommendations to, xxxviii–xxxix
Student's *t*-distribution, 419–422
definition and degrees of freedom, 419–420
`dt` function, 420–421
`pt` function, 421–422
`qt` function, 420
`rt` function, 422
`sub` function, 79, 102
subscript (`[ ]`) parameter, 675, 683
`subset` function, 116–118, 206
subsetting
arrays, 53–55
data frames, 112–118
with logicals, 68–72, 115–116
matrices, 40–45
extracting rows, columns, and diagonals, 41–42
omitting and overwriting, 42–45

vectors, 26–31
`substr` function, 78–79, 102
substrings, 78–80
subtraction (-) operator
arithmetic, 15, 35
matrices, 48, 57
`sum` function, 34–35, 294–295, 312–314
`summary` function
ANOVA table construction, 508–509
five-number summary, 318–319, 323, 332
inspecting and interpreting residuals, 622
multiple linear regression, 562
simple linear regression, 528–531, 545, 551, 553
`summaryRprof` function, 290–291, 300, 849
summary statistics, 309–332
counts, percentages, and proportions, 313–314
measures of centrality, 309–313
mean, 309
median, 309
mode, 310
R implementation, 310–313
measures of spread, 319–324
interquartile range, 321
R implementation, 322–324
standard deviation, 321
variance, 321
outliers, 329–332
quantiles, percentiles, and five-number summary, 318
relationships between numeric variables, 324–328
correlation, 325
covariance, 324–325

R implementation, 325–328
rounding, floors, ceilings, and absolute values, 315–317
sum-of-squares (SS) value, 508–509
`sum` parameter, 674, 683
superscript (^) parameter, 674, 683
`suppressWarnings` function, 283–284
surface truncation, 752–760
`survey` dataset
collinearity, 641–642
color indexing, 717
controlling multiple plots in one graphics device, 653
faceting, 696
`ggvis` graphics, 702
Kruskal–Wallis test, 516–517, 685–686
multiple linear regression, 560
partial *F*-test, 602
point-and-click coordinate interaction, 661
simple linear regression, 521–523, 539, 544, 548
Sweave, 850–851
`switch` function, 218–220, 249
symbolic notation (formula notation), 522
symmetric distributions, 388–389
`Sys.sleep` function, 286, 300
systematic error, 445
`Sys.time` function, 97, 102, 287–288, 300
`Sys.timezone` function, 98

T

`t` (matrix transposition) function, 46, 57, 558, 630

`\t` (tab) escape sequence, 77
table-format files, 172–176
    delimiters in, 172
    headers in, 171–172
    missing values in, 172
`table` function, 84, 102, 310–314, 332
`tail` command, 111, 118
`tapply` function, 240, 249
    categorical predictors, 539–540
    measures of centrality, 312–314
    one-way ANOVA, 504
    two-way ANOVA, 512
`tcl` parameter, 669, 683
`terrain.colors` function, 714, 772
text and labels, 670–675
    annotations in point clouds, 779–782
    color legends, 720–722
    font, 670–672
    Greek symbols, 672–673
    mathematical expressions, 673–675
    in point clouds, 778–779
    title and axis using `plot` function, 148–149
`text3d` function, 778–779, 790–791, 825
`text` function
    adding text to plots, 154, 158, 168
    clipping, 658–659
    fonts, 671
    linear regression model diagnostics, 629
`theme_bw` function, 336–337, 370
`theta` argument, 761–763, 767–768
3D plots, 773–825

bivariate surfaces, 788–797
adding components, 789–791
aspect ratio, 792–794
coloring by *z* value, 792
plotting perspective surfaces, 788–789
z-matrices, 736
parametric equations, 811–814
point clouds, 773–787
color and legends, 775–778
customized interactive 3D scatterplot, 782–785
3D cloud points and spheres, 774–775
3D lines, axes, grids, and other annotations, 779–782
3D point characters, 777–778
3D text, 778–779
scatterplots, 728–733
syntax, 729
visual enhancements, 730–732
trivariate surfaces, 798–810
evaluation coordinates in 3D, 798–799
isosurfaces, 799–804
nonparametric trivariate density, 804–810
`ticktype` argument, 763
`tiff` function, 180, 189
tilde (~) operator, 351–352, 370, 560, 611, 702
time and timing. *See* dates and times; progress and timing *under* profiling
`title3d` function, 781, 825
`title` function, 657, 673, 678, 683
title labels, 148–149
`T` logical value. *See* `TRUE` logical value

`topo.colors` function, 714, 772
topographical maps, 737–738
tori, 817–820
transposing matrices, 45–46
`triangles3d` function, 782
trimodal distributions, 388–389
trivariate surfaces, 798–810
    evaluation coordinates in 3D, 798–799
    isosurfaces, 799–804
        higher-dimensional probability densities, 799–801
        multiple-level control with color and opacity, 803–804
        one-level, 802–803
        trivariate evaluation lattice, 801–802
    nonparametric trivariate density, 804–810
        isosurface level selection, 806–810
        raw data, 805–806
        3D estimate calculation, 806
`TRUE` (`T`) logical value, 60–63, 102
    `else` statements, 213
    `if` statements, 210, 212
    reserved names, 198–199
`tryCatch` function, 283
`try` statements, 280–285, 300
    in body of function, 281–283
    suppressing warning messages, 283–285
`tseries` package, 171
`t.test` function, 455–456, 461–462, 464–465, 470, 502
two-sample *t*-test (Welch's *t*-test), 464
two-sided (two-tailed) tests, 456
two-way ANOVA, 511–515

interaction plots, 514
main effects and interactions, 511–515
suite of hypotheses, 511–512
.txt files, 171
`txtProgressBar` function, 286, 300
`type` argument, 147–148, 168, 775
Type I errors in hypothesis testing, 490
Type II errors in hypothesis testing, 490, 493
sample size, 496
statistical power, 497–498
variability of measurements, 495
`typeI.tester` function, 491–492
`typeII.tester` function, 494–495, 499
`typeof` function, 135

U

uniform distribution, 404–410,
`dunif` function, 406–408
`punif` function, 408
`qunif` function, 408–409
`runif` function, 409
unimodal distributions, 388, 389
union of two events, 374
univariate data, 306–307
universal assignment (`:=`) operator, 703–704, 708
University of Auckland, xxxi
University of Otago, xxxix
unpaired/independent samples testing
pooled variance, 465–468

unpooled variance, 463–465
update function, 643
backward selection, 609, 611
forward selection, 605, 607–608
update.packages function, 9–10, 14, 834, 837–839, 848
updating R and installed packages, 10, 838
upper-tailed tests, 456

V

vapply function, 241
var (variance) command, 322, 332
variables, 303
binary, 538–542
categorical
nominal, 304–305
ordinal, 304
continuous, 304–305
discrete, 304
multilevel, 542–547
nuisance, 556
numeric, 304–305
random, 376–391
continuous, 376–377, 380–388
discrete, 376–380
variadic behavior, 203–204
variance, 321
continuous random variables, 386–388
discrete random variables, 379–380
variance (var) command, 322, 332
vector-oriented (vectorized; elementwise) behavior, 31–35

vectors, 21–35
    classes and, 133–134
    creating, 21–22
    element extraction, 26–31
    indexes, 29–30
    lengths, 26
    recycling, 32, 44, 62
    repetition, 24–25
    sequences, 23–24
    sorting, 25
    subsetting, 27–30
    vector-oriented behavior, 31–35
Venn diagrams
    complement of events, 375
    intersection of events, 373
    union of events, 374
violin plots, 353–357
    design and interpretation, 354–355
    in `ggplot2`, 356–357
    histograms and box plots vs., 354
    side-by-side, 355
`vioplot` function, 354–355, 370
visual controls and graphical parameters, 147–154
    aspect ratio, 152–154
    color, 150
    line and point appearances, 151
`volcano` dataset, 737

W

`warning` command, 278, 300

warnings
in exception handling, 278–280
suppressing with `try` statements, 283–285
`warpbreaks` dataset, 512–513, 590–591
web-based files, 177–179
web resources, xl
`weekdays` function, 95, 102
Weibull distribution, 431
Welch–Satterthwaite equation, 464
Welch's *t*-test (two-sample *t*-test), 464
`which` function, 102
creating and formatting dates, 94–95
factors, 82
logical subsetting and extraction, 69–71
`NaN` notation, 123
`while` loops, 232–238, 249
`break` function and, 244
condition object modification, 233–238
`repeat` statements and, 247
whitespace, 178
Wilcoxon rank-sum test (Mann–Whitney U test), 471, 516
`wilcox.test` function, 471
Wilson score interval, 476
window width. *See* bandwidth
wireframe plots. *See* perspective plots
workflow, 5–12
code execution, 7
comments, 7
console and editor panes, 5–6
help files and function documentation, 10–12

packages, 9–10
installing, 9–10
loading, 9
updating, 10
working directory, 7–9
changing default, 8–9
checking location of, 8
workspace images, 13
workspaces, 13
`write.table` function, 179–180, 189

X

`X` argument, 239
`xaxs` parameter, 666, 683, 739
`xaxt` parameter, 667
`xlab` argument, 147
axis labels, 149, 162, 168
perspective plots, 763
point clouds, 774
3D scatterplots, 730, 783
`XLConnect` package, 176
`xlim` (xlimits) argument, 147–148
color legends, 721
custom region limits, 151–152, 168
nonparametric bivariate density, 755
plot customization, 666
.xlsx (*.xls*) files, 176
`xpd` parameter, 657–659, 683
XQuartz window system, 829
`xseg` argument, 780–781

Y

`yaxs` parameter, 666, 683, 739
`yaxt` parameter, 667
`%y` date marker, 93–94, 102
`%Y` date marker, 93–94, 102
`yintercept` argument, 166, 168
`ylab` argument, 147
    axis labels, 149, 162, 168
    perspective plots, 763
    point clouds, 774
    3D scatterplots, 730, 783
`ylim` (ylimits) argument, 147–148
    color legends, 721
    custom region limits, 151–152, 168
    nonparametric bivariate density, 755
    plot customization, 666
`yseg` argument, 780–781

Z

`zlab` argument, 730, 763, 774, 783
`zlim` argument, 720
z-matrices, 733
    conceptualizing, 735–736
    constructing, 735
Z-tests
    one-sample, 474–476
    two-sample, 476–478