

R FOR COLLEGE MATHEMATICS AND STATISTICS

Thomas J. Pfaff

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

R for College Mathematics and Statistics



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

R for College Mathematics and Statistics

Thomas J. Pfaff



CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2019 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20190222

International Standard Book Number-13: 978-0-367-19685-1 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Pfaff, Thomas J., author.
Title: R for college mathematics and statistics / Thomas J. Pfaff.
Other titles: College mathematics and statistics
Description: Boca Raton : CRC Press, Taylor & Francis Group, 2018.
Identifiers: LCCN 2018061710 | ISBN 9780367196851
Subjects: LCSH: Mathematical statistics--Data processing--Problems, exercises, etc. | Mathematics--Study and teaching (Higher)--Data processing. | Mathematical statistics--Study and teaching (Higher)--Data processing. | R (Computer program language)
Classification: LCC QA276.45.R3 P4925 2018 | DDC 519.20285/5133--dc23
LC record available at <https://lccn.loc.gov/2018061710>

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

*To my wife Janice and our four sons:
Liam, Pierce, Owen, and Hugh.
Never stop learning.*



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface	xi
Acknowledgments	xv
1 Getting Started	1
1.1 Importing Data into R	5
2 Functions and Their Graphs	7
2.1 A Piecewise-Defined Function	9
2.2 A Step Function	12
2.3 Polar Coordinates	13
2.4 Parametric Equations	15
2.5 Geometric Definition of a Parabola	16
2.6 Functions that Return a Function	18
2.7 Pythagorean Triples and a Checkerboard Plot	21
2.8 Exercises	28
3 Graphing	29
3.1 Graphing Functions	35
3.2 Scatter Plots	39
3.3 Dot, Pie, and Bar Charts	42
3.3.1 A Look at For Loops	53
3.4 Boxplot with a Stripchart	54
3.5 Histogram	57
3.6 Exercises	61
4 Polynomials	63
4.1 Basic Polynomial Operations	63
4.2 The LCM and GCD of Polynomials	69
4.3 Illustrating Roots of a Degree-Three Polynomial	71
4.4 Creating Pascal's Triangle with Polynomial Coefficients	73
4.5 Calculus with Polynomials	75
4.6 Taylor Polynomial of $\sin(x)$	79
4.7 Legendre Polynomials	81
4.8 Exercises	84

5 Sequences, Series, and Limits	85
5.1 Sequences and Series	86
5.2 The Derivative as a Limit	92
5.3 Recursive Sequences	97
5.4 Exercises	100
6 Calculating Derivatives	103
6.1 Symbolic Differentiation	104
6.2 Finding Maximum, Minimum, and Inflection Points	108
6.3 Graphing a Function and Its Derivative	109
6.4 Graphing a Function with Tangent Lines	112
6.5 Shading the Normal Density Curve Outside the Inflection Points	114
6.6 Exercises	117
7 Riemann Sums and Integration	119
7.1 Riemann Boxes	120
7.2 Numerical Integration	126
7.2.1 Numerical Integration of Iterated Integrals	127
7.3 Area Between Two Curves	128
7.4 Graphing an Antiderivative	132
7.5 Exercises	134
8 Planes, Surfaces, Rotations, and Solids	135
8.1 Interactive: Surface Plots	138
8.2 Interactive: Rotations around the x-axis	141
8.3 Interactive: Geometric Solids	147
8.4 Exercises	150
9 Curve Fitting	151
9.1 Exponential Fit	151
9.2 Polynomial Fit	155
9.3 Log Fit	157
9.4 Logistic Fit	159
9.5 Power Fit	161
9.6 Exercises	164
10 Simulation	165
10.1 A Coin Flip Simulation	165
10.2 An Elevator Problem	168
10.3 A Monty Hall Problem	170
10.4 Chuck-A-Luck	172
10.5 The Buffon Needle Problem	175
10.6 The Deadly Board Game	178
10.7 Exercises	186

11 The Central Limit Theorem and Z-test	189
11.1 A Central Limit Theorem Simulation	189
11.2 Z Test and Interval for One Mean	192
11.3 Z Test and Interval for Two Means	195
11.4 Exercises	197
12 The T-Test	199
12.1 T Test and Intervals for One and Two Means	199
12.1.1 Paired T-Test	205
12.2 Illustrating the Meaning of a Confidence Interval Simulation	206
12.3 Exercises	210
13 Testing Proportions	211
13.1 Tests and Intervals for One and Two Proportions	211
13.2 Illustrating the Meaning of α Simulation	214
13.3 Exercises	217
14 Linear Regression	219
14.1 Multiple Linear Regression	227
14.2 Exercises	230
15 Nonparametric Statistical Tests	231
15.1 Wilcoxon Signed Rank Test for a Single Population	233
15.2 Wilcoxon Rank Sum Test for Independent Groups	234
15.3 Wilcoxon Signed Rank Test for Dependent Data	235
15.4 Spearman's Rank Correlation Coefficient	235
15.5 Kruskal-Wallis One-Way Analysis of Variance	236
15.6 Exercises	237
16 Miscellaneous Statistical Tests	239
16.1 One-way ANOVA	239
16.1.1 Stacking Data	243
16.2 Chi-Square Tests	244
16.3 Testing Standard Deviations	247
16.4 Exercises	252
17 Matrices	253
17.1 Eigenvalues, Eigenvectors and Other Operations	256
17.2 Row Operations	260
17.3 Exercises	262
18 Differential Equations	263
18.1 Newton's Law of Cooling	263
18.2 The Logistic Equation	267
18.3 Predator-Prey Model	273
18.4 Exercises	280

19 Some Discrete Mathematics	281
19.1 Binomial Coefficients, Pascal's Triangle, and a Little Number Theory	281
19.2 Set Theory	283
19.2.1 Venn Diagrams	285
19.2.2 Power Set, Cartesian Product, and Intervals	288
19.2.3 A Cantor Set Example	289
19.3 Graph Theory	291
19.3.1 Creating and Displaying Graphs	291
19.3.2 Random Graphs	299
19.3.3 Some Graph Invariants	302
19.4 Exercises	308
A Loops, Vectors, and Apply	309
B Arctic Sea Ice Data	313
Bibliography	315
Index	317

Preface

My motivation for writing this book is that I firmly believe that R, which is free, should be used consistently in the college classroom, especially in mathematics and statistics. We no longer have to limit ourselves to “nice” functions in calculus classes. We can require reports and homework with professional grade graphs. We can require statistical analysis that includes checking assumptions, graphs, and post hoc tests where appropriate. We can do simulations and experiments in all classes. We are no longer limited to using the graphics in a book. R can be useful for student projects, for creating graphics for teaching, as well as for scholarly work. This list goes on, but in short we now have free software that is an industry standard that can be used to enhance the teaching of mathematics, while preparing our students for professional careers.

R also has the potential to be a cross disciplinary platform. In other words, R is just as useful in a mathematics classroom as a science classroom, and anywhere else a bit of quantitative analysis and graphics is necessary or beneficial. At the same time, using a language like R has the potential to help students learn mathematics due to the need for precision (code won’t run if something is wrong), understanding of symbols and functions, and the logical nature of code.

Currently there are obstacles to faculty learning R. The challenges are twofold. First, there is a significant start-up cost for an instructor to learn R. Yes, there is extensive online documentation and books available, but nothing available really aims to make it as easy as possible for mathematics and statistics instructors. Second, there is little in the way of exercises for teachers to assign. In some cases this isn’t difficult, such as for statistics classes, but in other cases, say calculus, it is more challenging. This book aims to solve both problems by providing targeted examples on how to use R, along with some problem sets aimed at providing ideas for exercises as much as practicing the code in the chapter. This book could be used for a short course for mathematics and statistics students on using R and the exercises will help.

There are often multiple ways to solve a problem in R, as in any language, and I am not an R expert. I generally tried to find the easiest or quickest way to perform a task. Fundamentally, I wrote this book to use R to enhance learning. In my examples, I often chose code that generalize easier as opposed to the shortest code for an example. At other times I chose code that may be more effective for teaching yet less elegant, and where appropriate both teaching code and elegant code. Overall, I tried to present examples with the

R code that might inspire folks to create interesting R related projects for the classroom. Please, take the ideas here and run with them.

Many functions in R have more options than I mention here and I aimed to provide what I think a classroom teacher or students would want or need. The functions or commands demonstrated in this book often do more or have more options than are mentioned in the book. I have chosen to use only base R and also not use RStudio, and whenever possible I avoided using packages, although packages are used in some chapters. If you plan to use this book recognize that there is always more that R can do (there are about 10,000 packages to add on to R and this book uses only a handful) and once you get a feel for R, there are great resources online.

I generally find it easier to learn syntax with meaningful examples. What you will find is that each section has examples to illustrate the use of R. The book does not have to be and is not designed to be used linearly once you go through the first part of the Getting Started chapter and the second part if you need to import data from Excel. I tried to make sure that the basics are in those chapters and that the rest of the chapters can be read as desired. Due to this, the chapters are often repetitive as they are meant to be independent as much as possible. I tried to integrate interesting capabilities of R throughout the book. My thinking was to include just in time code, “gems”, and sprinkle neat ideas throughout where they would make the most sense. There are cases where early chapters are used and in those cases I have clearly pointed to them.

Chapters begin with basic examples and move towards more complicated code with the goal of demonstrating engaging ideas to inspire. Don’t be intimidated by examples later in chapters. For instance, there is a big difference between the first and last example in the chapter on functions and their graphs. Chapters don’t need to be read in their entirety and if you get to the point where it is too much then move on to something else. It is easy to go overboard with examples, and in some places it could be argued that I did, but I think there is value in all of the examples. It is also the case that some graphs are not visually appealing. In these instances, the graph was designed to illustrate what can be done and is possibly exaggerated.

To make good use of this book you must play. In other words, change values that are used in the book to see what they really do. Don’t worry; you won’t break R. You should also constantly look at the variables created. For example, if the book used the assignment `x=1:10`, then after invoking the command you should enter `x` to see what `x` represents. Keep in mind that copy and paste are your friends. If you want more information about a function type `?functionname`, such as `?plot`, into the console to call the R documentation on the function, which will open in a browser, or simply google R and the command name. If you are looking for more, consider `ggplot2`, `dplyr`, `plotly`, `tidyverse`, find a package that matches your interests or explore r-bloggers.com.

Using the Book

Before beginning keep in mind that there is a steep curve in learning any language, especially if you have never coded. Be patient; recognize that you will need to keep track of parentheses, commas, and every detail of syntax. Throughout the book R functions in the text as well as user created functions are generally **in this font** but options and variables within functions are left as normal text. The code is written to display color, but the book is in black and white. You will have to run the code to see it in full color. Each chapter ends with a quick code review as a reference. In general, code should have comments to help read the code (in R comments are preceded by `#`), but comments are left out in the examples here because it would make the code boxes longer and cluttered while also being redundant given the explanation around each code box. In the code spaces are included only when necessary, but of course you can and should add space in the code for readability.

I have made some choices that serious R users would likely find offensive. As mentioned, I have left out comments. I have also chosen to use `=` for assignment instead of `<-`. In mathematics and statistics classes I think `A=2` is easier to understand than `A<-2`, but at some point you might consider moving to `<-`. I have typically favored for loops over using an apply function. I think this matches undergraduate mathematics better (see appendix A Loops, Vectors, and Apply). In general, I made choices to complement the mathematics or statistics classroom over efficiency or elegance. I use function and command interchangeably when referring to R code and note that R provides endless examples of functions that aren't real valued.

I recommend that you start with the first part of the Getting Started chapter and then follow that up with at least the first few sections of the functions and their graphs chapter. I suggest creating a new folder on your computer for your R scripts (or documents in a Mac) and naming your files something useful so they are easily found for reuse. You want to avoid reinventing the wheel when coding.

I have chosen not to use RStudio in this book or in my own classes for various reasons. On the other hand, I do not discourage the use of RStudio and you might find it helpful. RStudio does make importing data in spreadsheets easier and if you need to do that frequently then consider using RStudio. In the Getting Started chapter I make some references to using RStudio. One other advantage of using RStudio is that .R files will open automatically in RStudio, say as an attached file in an email, whereas they don't with just R.

In examples that use randomness I sometimes set a seed and sometimes I don't. Each chapter ends with a code review specific to that chapter and so what I say about an R function at the end of one chapter may differ from

another chapter. Finally, I have a website with the sea ice data and all the code in the book. You can copy and paste the code into R to see the output in color, but note that sometimes typing your code helps in the learning process.
<http://sustainabilitymath.org/r-book/>

Acknowledgments

I'm grateful to Stan Seltzer whose suggestions improved the first draft of this book. Wade Pickren, the director of the Center for Faculty Excellence at Ithaca College, has provided support and encouragement. The creators and contributors that made R and keep R going need to be recognized for their time, effort, and expertise. The fact that R is freely available is amazing. Lastly, much of what I have learned about R has been by way of Google. I need to thank and recognize the many people that have posted information about R, in particular those that have posted on Stack Overflow and R-bloggers. I appreciate all of you.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

1

Getting Started

If you do not have R on your desktop then download it now from <https://www.r-project.org/>. When you click on download R, you will first be asked to choose a CRAN mirror, a location to download the software from, and then will be to a page to choose between Windows, Mac, or Linux. Following the Download R for Windows link will take you to the page for Windows that has a download R for the first time link. After that link you will be directed to a page with a download link at the top. Following the Download R for (Mac) OS X link takes you to a download R for Mac OS X page. Click on the first link on the left under the Latest release header. Linux users can follow the Download R for Linux link. In all cases you can search for videos if you are having difficulties loading R.

When you open R you will see the R Console box with > prompt. Take a few moments and play with R as a calculator with * as multiplication and ^ for exponents. For example we calculate $2^6 \cdot 3^6 \cdot 43$.

R Code

```
> 2^6*3^6*43
```

```
[1] 123456
```

Throughout this text these boxes will represent the R code and its output. Now that you are comfortable using the R console we want to stop using it as our primary input method as scripts are preferred. Open up a new script (document for Macs) by going to File -> New script. A new window opens with the name Untitled - R Editor. Save this file to a location of your choosing and name it FirstScript and note that it is a .R filetype. The R Editor allows you to type your commands and edit them before you run them in the R console. The file can also be saved for future reference and reuse.

Working in the R Editor

- Text following # will be ignored by R. This is how you add comments to your commands so that you can easily read them later.
- Ctrl + R (Command Return with a Mac) will send the current line in the editor to the console. Similarly, any highlighted commands in the editor will be sent to the console when you press Ctrl + R.

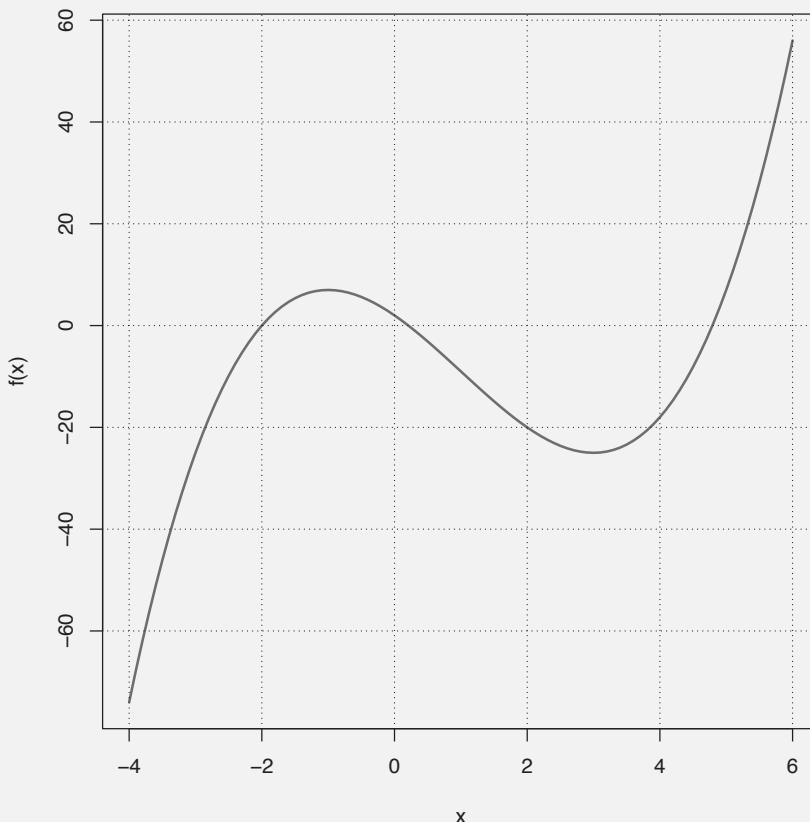
- A + sign in the R console means that R is expecting something more. For example you might be missing a parenthesis or bracket. To get out and back to the > prompt use the Esc key.
- Recognize that your toolbar and submenu options depend on which window, console or editor, is selected.

Here are two quick examples to get started, but first a few conventions of this book. R code is taken from the console and so lines of code are preceded by >. If a line does not have a > then the line above was broken and started a new line to fit the page. It is meant to be part of the single line started above and preceded by >. A line that starts with a + is, in fact, a separate line of code, but is part of a larger collection of code. The code in this book uses color, but to see the color you will need to run the code as this book is in black and white.

The first example, graphs the function $f(x) = x^3 - 3x^2 - 9x + 2$ on the interval $[-4, 6]$. To do this, we first define the function. The syntax sets f equal to function(x), a function with variable x, with the expression defined enclosed in braces. Note that * is necessary to represent multiplication. Entering 9x will generate an error as it must be 9*x. The **curve** function here has five input elements separated by a comma. The first is the function to be graphed. The next two define the x-axis interval for the graph. The first three entries are not optional and must be in this order. The next two elements are optional and can be in any order. We set lwd=2 for the line width and col to red, in quotes, for the color of the graph. A grid is added with **grid**. The first two elements, NULL, set the grid lines to the tick marks given by the graph created with **curve**. There are other options here that can be used. The last entry sets the color of the grid lines, with a default of gray. To add another function to the graph, **curve** can be used again, but the option add=TRUE must be included. To learn more about **curve** and **grid** run ?curve or ?grid, which will open a webpage with details about the functions. In general, a question mark followed by a command will open an R documentation page with details about the command.

R Code

```
> f=function(x){x^3-3*x^2-9*x+2}
> curve(f,-4,6,lwd=2,col="red")
> grid(NULL,NULL,col="black")
```



The next example begins by generating a random data set of size 150 from a random normal distribution with mean 0 and standard deviation 1. The data returned by **rnorm** is set to the variable **example**. At this point **example** is a vector of length 150. We use **summary** to return the five number summary plus the mean of the data set **example**. Note, that the data set is randomly generated so results will differ. Use **set.seed(42)** to get the same results. We can use **sd(example)** for the standard deviation and **boxplot(data)** to create a boxplot of the data.

R Code

```
> example=rnorm(150,0,1)
> summary(example)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.99309	-0.61249	-0.04096	-0.02874	0.64116	2.70189

For an example of a t-test, we test the data example with an alternative of $\mu < 0$. The function **t.test()** is used. The first entry is the data and is not optional. The other three inputs set the value of μ_0 to 0, the alternative hypothesis to less than, and the confidence level to 0.90. The default μ_0 is 0 so this isn't necessary here. The default alternative is two-sided and the options for alternative are two.sided, less, and greater and must be in quotes. The default confidence level is 0.95. To learn more about these functions run `?t.test` or `?boxplot`, for example.

R Code

```
> t.test(example,mu=0,alternative="less",
conf.level=0.90)
```

One Sample t-test

```
data: example
t = -0.35007, df = 149, p-value = 0.3634
alternative hypothesis: true mean is less than 0
90 percent confidence interval:
-Inf 0.07694166
sample estimates:
mean of x
-0.02874049
```

At this point, you should feel free to explore the chapters. The next section is specifically for importing data into R that you may not need at the moment. Depending on your interests, two good chapters to turn to next are [Chapter 2](#), Functions and Their Graphs or one of the basic statistics chapters, [Chapters 11, 12, or 13](#). One last tip: Some chapters use a package. A package can be thought of as an add-on to R that does something more than the base distribution. There are over 10,000 packages for R. To use a package, it must first be downloaded, which only has to be done once. To download a package, make sure the console window is highlighted and use the package menu along the top. You will first need to select as CRAN location, as is done when loading R, and then select the package. Search load package in R if there are any difficulties. When you want to use a particular package run **library(name)**, which you will see in this book anytime a package is used.

1.1 Importing Data into R

One of the strengths of R is statistical computing, but entering large data sets directly into R is not a common practice. Typically, for colleges mathematics and statistics courses, your data will come from a spreadsheet and you will want to import that data into R. The example below assumes that a csv file has been created, likely from Excel, from the Arctic Sea Ice data from Appendix B.

Importing a csv File

- Make sure your csv file is free of commas and apostrophes within cells, has only one tab, and you have deleted unnecessary cells. Also, your first cell cannot be named ID.
- With the Console window selected go to File → Change dir... and select the folder that contains the csv file. In a Mac, Change Dir is under Misc.
- Enter the command `DataManager=read.table("File Name.csv", header=TRUE, sep=",")`. Here DataManager gives the data a name within the R environment. Choose a meaningful name. Note header=TRUE tells R that you have column names, which is common. You can set this to FALSE. Tip: Copy and paste your file name into R to avoid typos.
- Use the command `names(DataManager)` to see your column names. This also serves as a quick check on importing your data, although it is by no means a guarantee.
- Use `head(DataManager, n=x)` to see the first x rows of data.

Here is an example using the Arctic Ice data. The **read.table** function imports the file. If the file does not have headers then set header to FALSE or better add headers. The imported file will be a data frame and we set it to ICE. We use **names(ICE)** to see the header names. Notice that R inserts a . for spaces in the column names. The listing of column names in the console may differ as R will fill the space based on the current size of the window, but it will not resize the output if you change the window size. If you would like to see the data for a particular column then use `Ice$column-name`. For example `Ice$Years.after.1970` will return the years after 1970 column of data. For the precocious, **plot(year, March.Extent.in.MSK)** will produce a scatter plot with R default settings. More details will be provided in [Chapter 3](#).

R Code

```
> Ice=read.table("Arctic-Ice-Data-R.csv",header=TRUE,  
sep=",")  
> names(Ice)  
  
[1] "year"           "Years.after.1970"  
[3] "March.Extent.in.MSK" "June.Extent.in.MSK"  
[5] "September.Extent.in.MSK"
```

Note: If you are using RStudio, then there is a button, Import Dataset, to import csv and other files. RStudio will not replace spaces with a period. In this case, the syntax to reference a column of data is `Ice$'Years after 1970'`. If there are no spaces in the title then the single quotes are not needed. One solution here is to change the headers in the csv file and replace the spaces with a period.

If there is an error you will see it in the console. In the next example, the `rm(Ice)` command was used to remove the file `Ice` from the workspace before providing this example. The file name is incorrect in the `read.table` command yielding the error. Two common errors in importing files are leaving out the `.csv` in the title and not changing the directory (see box on Importing Data into R).

R Code

```
> rm(Ice)  
> Ice=read.table("Ice-Data-R.csv",header=TRUE,sep=",")  
  
Error in file(file, "rt") : cannot open the connection  
In addition: Warning message:  
In file(file, "rt") :  
  cannot open file 'Ice-Data-R.csv':  
  No such file or directory
```

2

Functions and Their Graphs

We define and use functions throughout this book, and in this chapter we focus on functions and their associated graphs. We provide examples of commonly used mathematical functions: piecewise, step, parametric, and polar. We broaden the use of **function** by using the geometric definition to obtain a parabola, discuss functions that return functions, and create a function that returns Pythagorean triples. In each case we graph the function; we keep our graphs basic leaving chapter 3 for further details on graphing, except for our last example where we create a checkerboard graph.

The command for creating a function is **function(){}{}**, where the variable(s) are listed inside the parenthesis and the function is defined within the braces. In our first example, we define the function f to be $x^2\sin(x)$ and evaluate it at 3 with f(3). There are a number of predefined functions such as **abs**, **sqrt**, the trigonometric functions, hyperbolic functions, **log** for the natural log, **log10**, **log2**, and the exponential function **exp**. So, for example, **sin(x)** is available to use in our definition of f. Note that * must be used for multiplication as we cannot simply juxtapose objects.

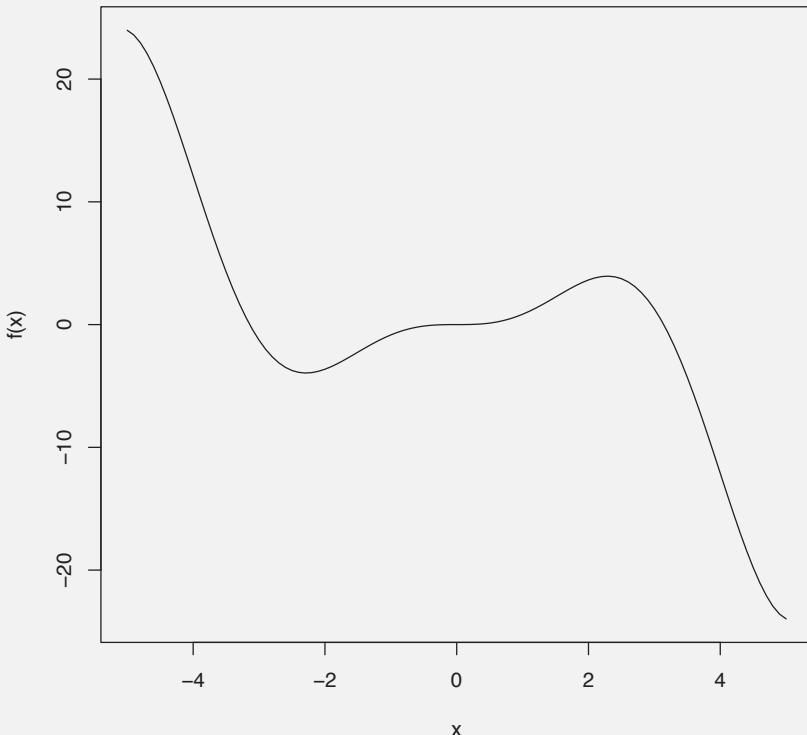
R Code

```
> f=function(x){x^2*sin(x)}  
> f(3)  
  
[1] 1.27008
```

We can plot our function with **curve**. The first three arguments must be the function, the lower value for the independent variable, and the upper value for the dependent variable. The default range for the dependent variable is selected based on the minimum and maximum of the function on the given interval. There are numerous options, such as **ylim** for the *y* limits, **lwd** for the width (i.e., thickness) of the curve, **col** for the color of the curve, **xlab** and **ylab** for labeling the axis, and **lty** for the type of line (e.g., dashed, dotted).

R Code

```
> curve(f,-5,5)
```



Our next two examples illustrate functions of two and three variables. The first returns the area of a rectangle given the length and width. The second returns the area of a triangle given the length of the three sides. Note that in the second example we define the variable p within the braces of the function making it easier to define the function. As we shall see later in the chapter, we are not limited to one or two lines of code within those curly brackets. In using function, the function will return the value on the last line inside the curly brackets. We will see in examples below that we use **return** to make explicit the value that the function will return.

R Code

```
> rectangle.area=function(l,w){l*w}  
> rectangle.area(4,6)
```

```
[1] 24
```

R Code

```
> area.triangle.sides=function(a,b,c){p=(a+b+c)/2;
  sqrt((p)*(p-a)*(p-b)*(p-c))} 
> area.triangle.sides(3,4,5)
```

```
[1] 6
```

2.1 A Piecewise-Defined Function

A piecewise function can be defined using an **ifelse** statement within **function**. We define a function that returns $\sin(x)$ when $x < 2$ and otherwise returns x^2 . When we evaluate our function at $-\pi/4$ using $-\text{pi}/4$ we get -0.707. On the other hand, 25 is returned when we execute **f.piecewise(5)**. If a piecewise function has more than two pieces then we can nest the **ifelse** statement.

R Code

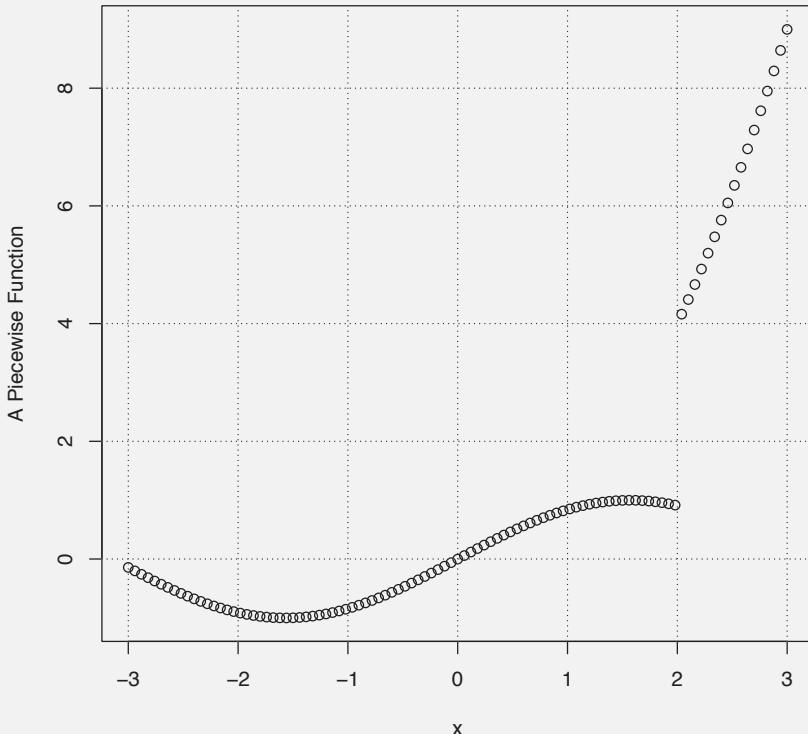
```
> f.piecewise=function(x){ifelse(x<2,sin(x),x^2)} 
> f.piecewise(-pi/4)
```

```
[1] -0.7071068
```

Graphing a piecewise function can be tricky. If simply plotting points to see any jumps is acceptable, then we can use **curve**, provide the x range, and use type set to p for points. If we do this but set type to "l" (or remove type="p" since "l" is the default), then a line will connect the points across the jump. Here we labeled the y -axis with **ylab** and added a grid. By using **NULL** for the first two elements of **grid**, R will put grid lines at the tick marks set by **curve**. The default color is a gray that is often faint, so we set **col** to black.

R Code

```
> curve(f.piecewise,-3,3,type="p",
  ylab="A Piecewise Function")
> grid(NULL,NULL,col="black")
```



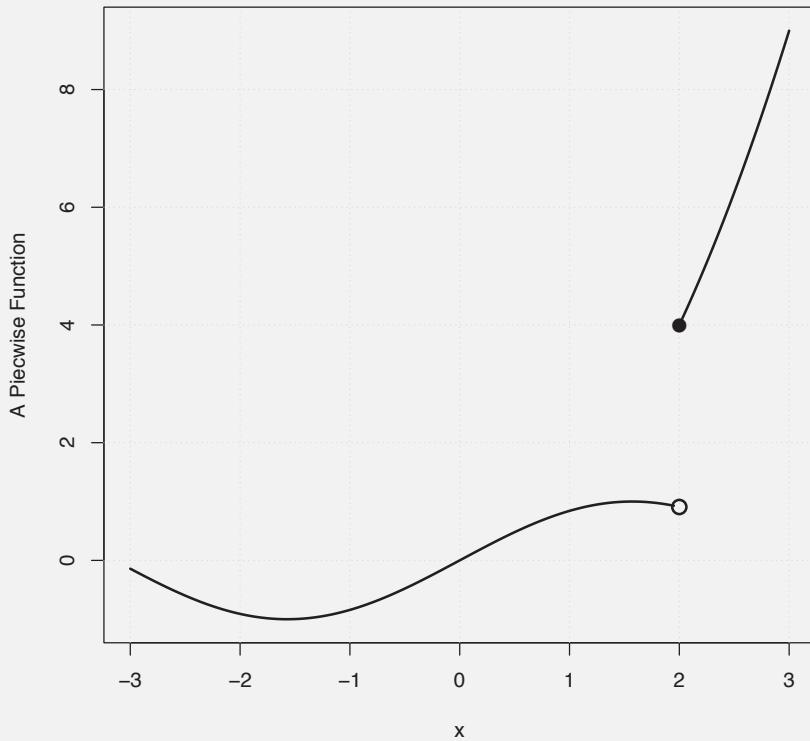
Our next example creates the same graph but with open and closed dots at the jump. We begin by using the **curve** function as above but we set the type to “n”, which doesn’t graph the curve. This sets up a graph frame so that we can then add the two parts of the piecewise function. Note that we labeled the y -axis and add a grid as in the last example.

The first part of the piecewise function is added to the graph with **curve**. A key element of **curve** is that it will only graph the function on the given x range regardless of the range of x in the graph that exists. In this case we chose a range from -3 to 1.95 , where a bit of trial and error was used so that the function stopped at the edge of the open circle added later in the code. We use `add=TRUE` so that the function is added to the current graph as opposed to creating a new graph. We also set the line width to 2 . The next use of **curve** is the same with the exception of the range for x . Our last two lines add points to the graph. The first use of **points** places an open circle (`pch=1`) at $(2, \sin(2))$. The point is scaled with `cex=1.5`, or 1.5 times the default size. The

next use of **points** sets the point character, pch=16, to a solid circle located at (2,f.piecewise(2)), which is again scaled with cex=1.5.

R Code

```
> curve(f.piecewise,-3,3,type="n",
+       ylab="A Piecewise Function")
> grid(NULL,NULL,col="black")
> curve(f.piecewise,-3,1.95,add=TRUE,lwd=2)
> curve(f.piecewise,2,3,add=TRUE,lwd=2)
> points(2,sin(2),pch=1,cex=1.5,lwd=2)
> points(2,f.piecewise(2),pch=16,cex=1.5)
```



2.2 A Step Function

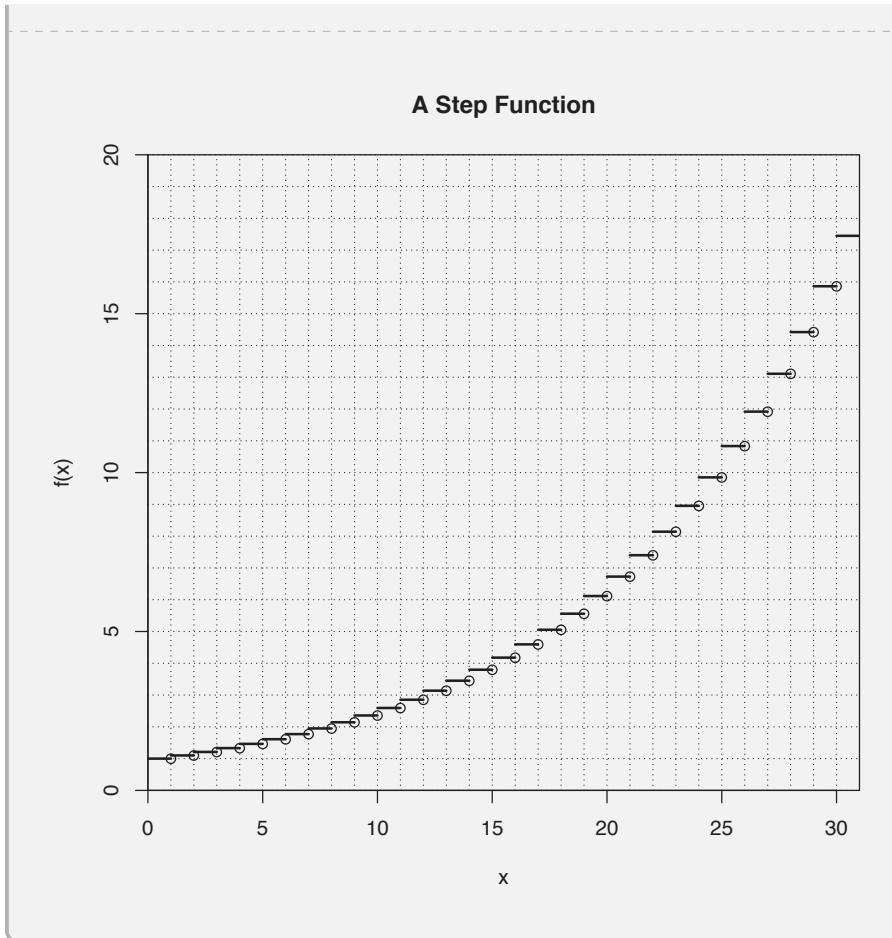
In this example we use a step function to represent the amount of money in an account that starts with \$1 and earns 10% interest compounded yearly for 30 years. We will use the **stepfun** function and the key is that it requires the locations of the jumps and the value of the function at each jump. We use the colon function to define the values for x , the location of the jumps. Here $1:30$ returns the integers from 1 to 30. In general $a:b$ creates a sequence with unit steps starting at a and ending without exceeding b .

We create the sequence of function values with **sapply**. Note that we need one more function value than “jump locations” to account for the first jump. The function **sapply** will evaluate the given function, 1.10^x at each value of x , 0 to **max(x)**, and return those values as the sequence y . The first four values of y are 1.000000, 1.100000, 1.210000, and 1.331000. We use **max(x)**, the maximum of the sequence x , so that if we change the last value in x we don’t have to change it in other places in the code. We define the function **step** to be the result of the step function created by x and y ; **step(x,y,right=TRUE)**, with the option right set to TRUE, which determines if the intervals should be closed on the right. In other words, right set to TRUE corresponds to the function being right continuous. The default is FALSE. The function **step** can be evaluated so, for example, $\text{step}(11.25) = 2.853117$.

The **plot** function recognizes step functions created with **stepfun**. The x range is set from 0 to the **max(x)+1** with **xlim**. Similarly, the y range is set from 0 to 20, which allows for a little space above the last step beyond what the default (leaving out **ylim**) would produce. The default graph frame when using **plot** provides 4% of space on each side of the axis. In this case our axis would start slightly below 0 in both x and y . As a matter of style, we remove this extra space with **xaxs="i"** and **yaxs="i"**. We do not want vertical lines drawn at the jumps and so **verticals=FALSE**, and a title is added with **main**. Note that the last step extends to infinity and is cut off by our choice of the range for x . There is an option in **plot** to remove the open circles; **do.points=FALSE**. In the previous examples grid lines were added with **grid**. Another way to add grid lines is to use **abline**, which adds horizontal and vertical lines at the values set to **h** and **v**, respectively. The line type of 3 is dotted, with values for **lty** ranging from 1 to 6.

R Code

```
> x=1:30
> y=sapply(0:max(x),function(x)(1.10^x))
> step=stepfun(x,y,right = TRUE)
> plot(step,xlim=c(0,max(x)+1),ylim=c(0,20),lwd=2,xaxs=
"i",yaxs="i",verticals=FALSE,main="A Step Function")
> abline(h=0:20,v=1:30,lty=3)
```



2.3 Polar Coordinates

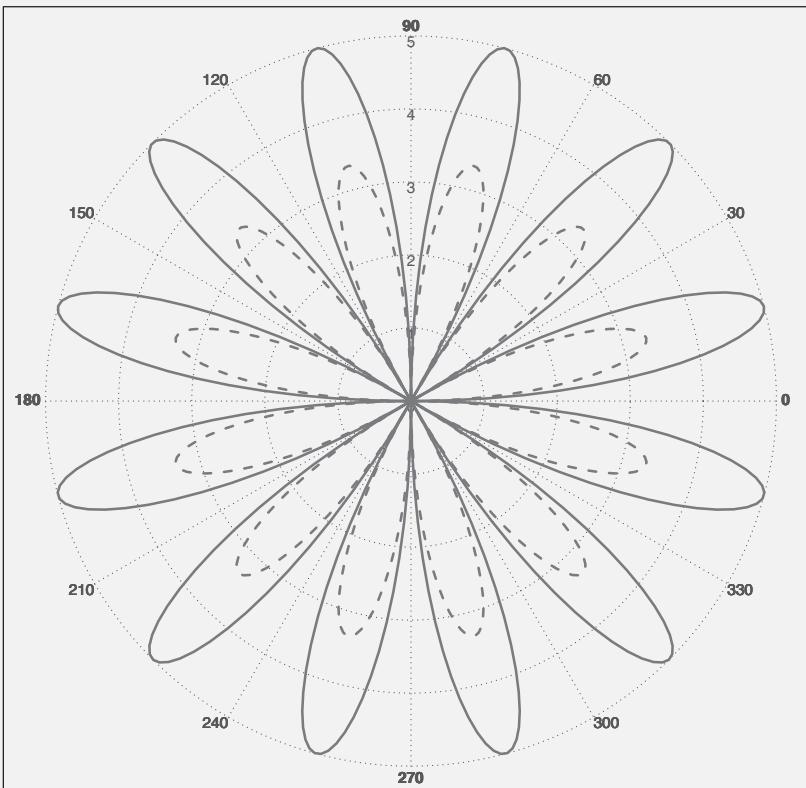
The `pracma` (Practical Numerical Math Functions) package “Provides a large number of functions from numerical analysis and linear algebra, numerical optimization, differential equations, time series, plus some well-known special mathematical functions. Uses ‘MATLAB’ function names where appropriate to simplify porting [3].” In particular, the package has a function to graph in polar coordinates. To do this we need a sequence of input values in radians and a function. We use `seq` to create a sequence of values from 0 to 2π of length 360, which is set to `t`. The function `f` is defined in terms of `t` to be

$5 \sin(6t)$. Before creating the graph with **polar** we set the bottom, left, top, and right, margins to 1, 1, 2, and 1 line, respectively.

R Code

```
> library(pracma)
> t=seq(0,2*pi, length=360)
> f=function(t){5*sin(6*t)}
> par(mar=c(1,1,2,1))
> polar(t,f(t),grcol="gray20",bxcol="black",col="purple",
lwd=2,main=expression(5*sin(6*theta)~ "and"~ ~
2(5*sin(6*theta))/3))
> polar(t,2*f(t)/3,col="purple",lty=2,lwd=2,add=TRUE)
```

$5\sin(6\theta)$ and $2(5\sin(6\theta))/3$



The first two arguments of **polar** are **t** and **f(t)**. The polar grid color, **grcol**, is set to gray20 and can be set to FALSE to be suppressed. A black box is placed around the graph with **bxcol**, which can also be set to FALSE.

The graph is colored purple and the line width is 2. A title is created with **expression** and set to main. Within expression we need to use * to juxtapose objects, quotes around text, and double tildas with a space between them to create space between the formulas and the text. We add the graph of $2(5\sin(6t))^3$ with **polar** by including the argument add=TRUE. The global arguments such as grcol, bcol, and main are left out, but we do color the graph purple and use a dashed line type by setting lty=2.

2.4 Parametric Equations

The butterfly curve by Temple H. Fay is our choice to demonstrate defining and graphing parametric equations. The first two lines define the equations for **x** and **y** as a function of the parameter **t**. We then define the function **Butterfly** as a function of **t** that returns the values of **x(t)** and **y(t)** as we see in **Butterfly(2*pi)**. We chose to define **x** and **y** outside the function so that we can use them for graphing and it is easier to read, but we could have imbedded them within the definition of the Butterfly function.

R Code

```
> x=function(t){sin(t)*(exp(cos(t))-2*cos(4*t)-
  (sin(t/12))^5)}
> y=function(t){cos(t)*(exp(cos(t))-2*cos(4*t)-
  (sin(t/12))^5)}
> Butterfly=function(t){return(c(x(t),y(t)))}
> Butterfly(2*pi)
```

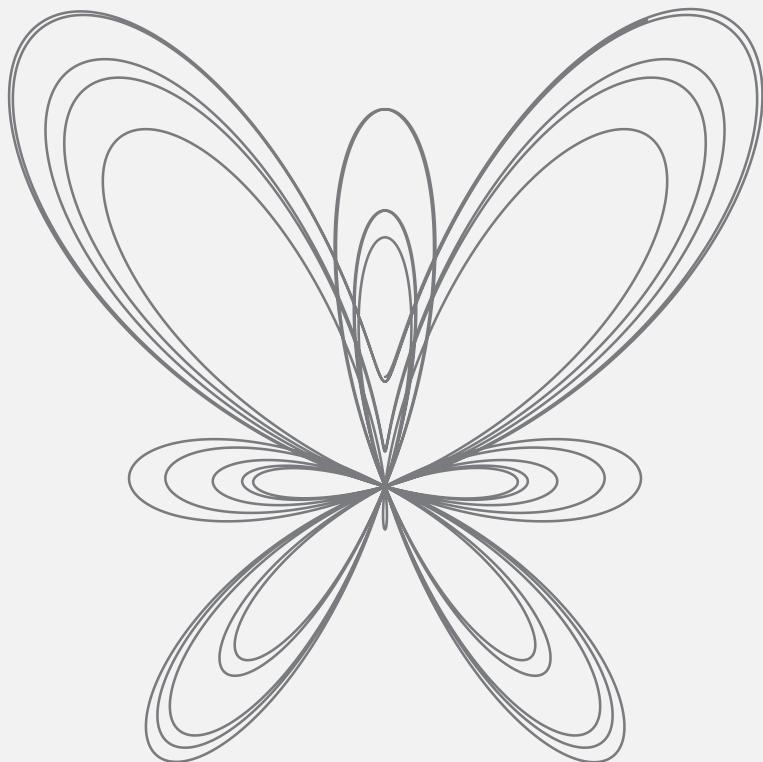
```
[1] -1.682687e-16 6.870318e-01
```

We graph the function by generating a set of **x** and **y** values, but first we set the margins to one line on all four sides with **par(mar=c(1,1,1,1))**. The default margin settings are 5.1, 4.1, 4.1, and 2.1 for the bottom, left, top, and right margins, respectively, which would leave the butterfly off center. A sequence of values from 0 to 2π in steps of 0.001 is defined by **seq()** and set to **t.values**. We use **plot** to graph the set of **x** and **y** values from **x(t.values)** and **y(t.values)**. We use type 1 for lines, color the graph purple, and set the line width to 2. The default axis labels and axes are removed by setting **ann** and **axes** to FALSE.

R Code

```
> par(mar=c(1,1,1,1))
> t.values=seq(0,12*pi, 0.001)
```

```
> plot(x(t.values),y(t.values),type="l",col="purple",
lwd=2,ann=FALSE,axes=FALSE)
```



2.5 Geometric Definition of a Parabola

A parabola can be described geometrically as the set of points that are at an equal distance from a point and a line and we can use this to derive an algebraic expression for it. Our goal here is to define and graph a function $f(x)$ that returns the value of y such that (x, y) is at an equal distance from the point $(0, 1)$ and the x -axis. We first call the package `rootSolve` which has a function, **uniroot.all**, that will return roots of a function. We define the variable `y.max` and set it to 20. This will be used for the upper limit of the

interval where **uniroot.all** will search for roots. It is defined here to make it easier to change as necessary. We define the function **g** as a function of **x** that will return the value of **y** that is at an equal distance between the point $(0, 1)$ and the x -axis. Within the braces of **g** we define the function **f** as the difference of $\sqrt{x^2 + (1 - y)^2}$ and **y**. The function **uniroot.all** will return all roots of the function **f** on the given interval from 0 to **y.max**. The function **g** returns this value. As a check, we see that **g(0)** is, in fact, 0.5.

R Code

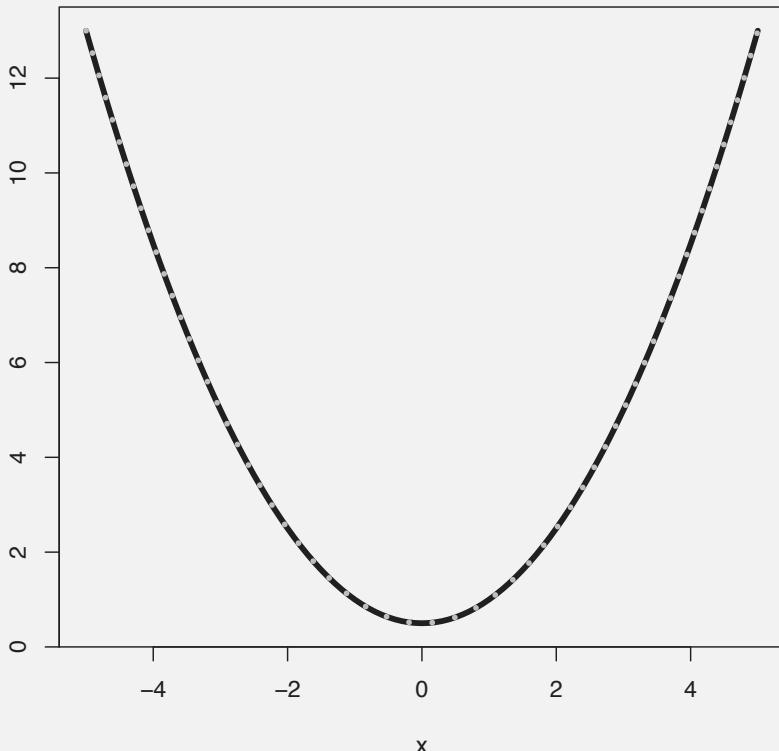
```
> library(rootSolve)
> y.max=20
> g=function(x){f=function(y){sqrt(x^2+(1-y)^2)-y}
+ return(uniroot.all(f,c(0,y.max)))}
> g(0)

[1] 0.5
```

Before graphing our function we have two issues. First, we set the margins with **par(mar=c(5,3,3,2))** and note that if we are following the text the margins are currently set to 1, 1, 1, and 1. If we just started R for this example, then the margins are set to the defaults of 5.1, 5.1, 4.1, and 2.1. Second, there is a vectorization issue with the function **g** and so we define **h** to be **Vectorize(g)**, and we will graph **h**. To begin learning about vectorization, along with other issues, and R see http://www.burns-stat.com/pages/Tutor/R_inferno.pdf. We graph **h** with **curve** from -5 to 5, with a line width of 4 and remove the y label by setting **ylab** to empty quotes. For comparison, we graph $(x^2 + 1)/2$ over the same interval. The line type is set to 3 (dotted), line width to 4, and color to gray70. Setting **add=TRUE** adds this to the previous graph instead of creating a new graph frame. In this example a line width of 4 was used to help see the dotted gray70 graph on top of the black curve.

R Code

```
> h=Vectorize(g)
> par(mar=c(5,3,3,2))
> curve(h,-5,5, lwd=4, ylab="")
> curve((x^2+1)/2,-5,5,add=TRUE,lty=3,lwd=4,
col="gray70")
```



2.6 Functions that Return a Function

Functions can be defined to return a function, and we provide two such examples. The first example will define a function that returns the reflection of a function about the origin. In our second example, our function will return an exponential function with the given parameters, that will allow us to graph a family of functions.

The **Reflect.origin** function is a function of h that returns the function of x , $-h(-x)$. As an example, we define the function f to be $x^5 - 3x^3 - 3x + 2$ and check to see if $f(2)$ equals **Reflect.origin(f)(2)**. It does not, which proves that f is not odd.

R Code

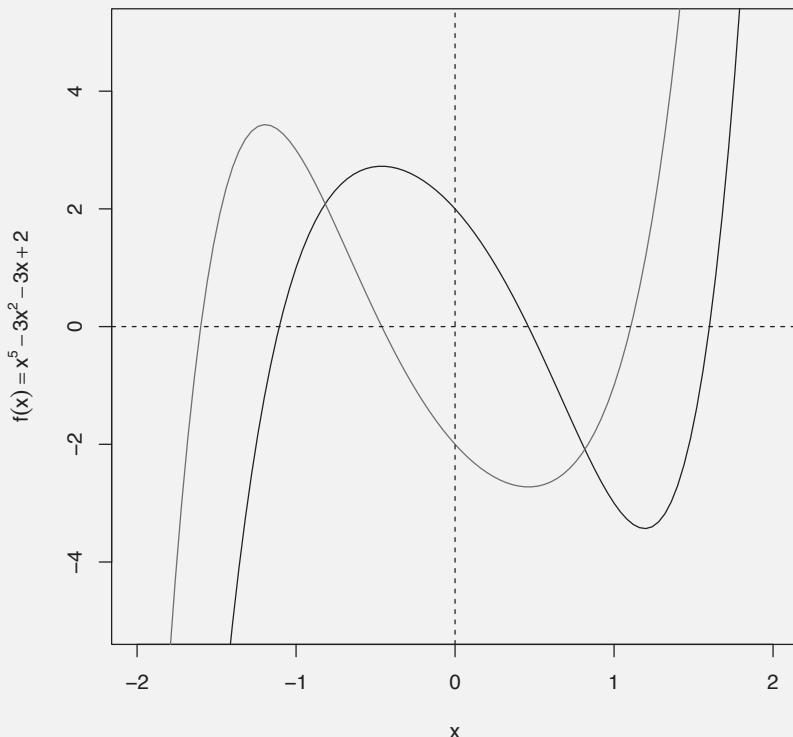
```
> Reflect.origin=function(h)
{return(function(x){-h(-x)})}
> f=function(x){x^5-3*x^2-3*x+2}
> f(2)==Reflect.origin(f)(2)
```

```
[1] FALSE
```

We now graph **f** and its reflection about the origin. We use **par** to add an extra line beyond the defaults on the left-hand side; otherwise the superscripts are too close to the edge of the *y*-axis label. We use **curve** to graph **f** from -2 to 2 , set the *y*-axis to -5 to 5 , label the *y*-axis with the expression of the function, and add a title with **main**. We add the graph of its reflection about the origin by graphing **Reflect.origin(f)(x)** from -2 to 2 and color it red. We set **add=TRUE** to add it to the current graph frame. For perspective **abline** places a vertical line at $x = 0$ with **v=0** and a horizontal line at $y = 0$ with **h=0**. We set the line type to 2 for a dashed line.

R Code

```
> par(mar=c(5.1,5.1,4.1,2.1))
> curve(f,-2,2,ylim=c(-5,5),ylab=expression(f(x) ==
x^5-3*x^2-3*x+2),main="Graph of f(x) and its
Reflection (in red) about the Origin")
> curve(Reflect.origin(f)(x),-2,2,col="red",add=TRUE)
> abline(v=0,h=0,lty=2)
```

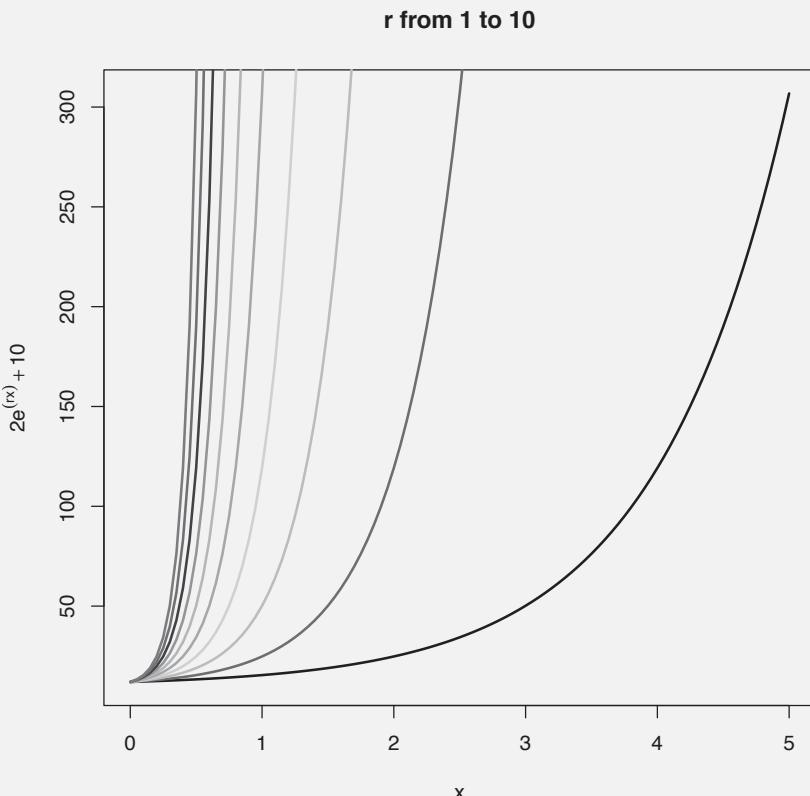
Graph of $f(x)$ and its Reflection (in red) about the Origin

The next example defines the function **E** with variables *a*, *b*, and *c*, while returning the function $ae^{bx} + c$. We will graph a collection of these functions with different values of *b*. We create an initial graph with **curve** displaying **E(2,1,10)** from 0 to 5 with a line width of 2. We label the *y*-axis with the expression $2e^{rx} + 10$ and add a title with **main**. Next we begin a for loop where *r* ranges from 2 to 10. For each value of *r* **curve** will add **E(2,r,10)** to the graph, *add=TRUE*, with a line width of 2. The function **rainbow(9)** creates a vector of 9 colors ranging across the colors of **rainbow** and for each value of *r* the graph will be colored with the *r*-1 color of this vector; **rainbow(9)[r-1]**. Note that the for loop begins and ends with a brace.

R Code

```
> E=function(a,b,c) {return(function(x){a*exp(b*x)+c})}
> curve(E(2,1,10)(x),0,5,lwd=2,ylab=
```

```
expression(2*e^(r*x)+10),main="r from 1 to 10")
> for(r in 2:10){curve(E(2,r,10)(x),add=TRUE,lwd=2,
col=rainbow(9)[r-1])}
```



2.7 Pythagorean Triples and a Checkerboard Plot

The graph for our last example is significantly more involved than the previous examples, but the function is not. We define **Euclid1** as a function of n and m and it will return the Pythagorean triple generated by $m^2 - n^2$, $2mn$, and $m^2 + n^2$. Within the function we initialize the vector triple as a numeric vector with length 3. We then set **triple[1]**, **triple[2]**, and **triple[3]** to $m^2 - n^2$, $2mn$, and $m^2 + n^2$, respectively. The function returns the vector triple. For example

Euclid1(2,3) returns the vector 5, 12, 13. Note that the function as written assumes that the first input is the smaller of the two as **Euclid1(3,2)** returns -5, 12, and 13. One way to deal with this is to have the function return “error” if the first input is larger, and this is done with the function **Euclid2**.

R Code

```
> Euclid1=function(n,m){triple=numeric(3)
+ triple[1]=m^2-n^2
+ triple[2]=2*m*n
+ triple[3]=m^2+n^2
+ return(triple)}
> Euclid1(2,3)
```

```
[1] 5 12 13
```

The key difference between **Euclid1** and **Euclid2** is that the latter has an if-then-else statement. If ($m > n$) then we do exactly what was done for **Euclid1**. In fact, we could have used the function **Euclid1** within **Euclid2**, but we chose to keep them independent. On the other hand, if $m > n$ is not true, then the function returns “error”. With this function **Euclid2(3,2)** returns “error”. Note that each part of the if and else statements is encased in braces and the last two braces represent closing the else portion and then closing the function definition.

R Code

```
> Euclid2=function(n,m){
+ if (m > n){
+ triple=numeric(3)
+ triple[1]=m^2-n^2
+ triple[2]=2*m*n
+ triple[3]=m^2+n^2
+ return(triple)
+ } else {return("error")}}
> Euclid2(3,2)
```

```
[1] "error"
```

We now proceed to create a checkerboard plot of Pythagorean triples. Each row of the plot darkens the three squares that represent a given triple. To get there we have to create a matrix of 1s, representing triples, and 0s. Our first step is to create a matrix of n and m values to be evaluated by **Euclid1**. The columns of this matrix will be filled by n and m . We define n to be the list of numbers 1, 1, 2, 1, 2, 3, ..., 1, 2, 3, 4, 5, 6 and m to be 2, 3, 3, ..., 7, 7, 7, 7, 7, where **rep(i,j)** repeats the value i , j times. We could use for loops to create these, which would be more efficient, but possibly less clear. We use **matrix**

to define A where **c(n,m)** concatenates the vectors n and m, and that list of values fills the matrix A with two columns by column, byrow=FALSE. The first five rows of A and both columns are shown with **A[1:5,]**.

R Code

```
> n=c(1:1,1:2,1:3,1:4,1:5,1:6)
> m=c(rep(2,1),rep(3,2),rep(4,3),rep(5,4),rep(6,5),
rep(7,6))
> A=matrix(c(n,m),ncol=2,byrow=FALSE)
> A[1:5,]
```

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

The **mapply** function evaluates Euclid1 with each row of A. Note that A[,1] is column 1 and **A[,2]** is column 2. The output is a 3 by 21 matrix where each column is a Pythagorean triple. We illustrate the first five columns with **Triple[,1:5]** where the empty first argument in **Triples** returns all rows.

R Code

```
> Triples=mapply(Euclid1, A[,1], A[,2])
> Triples[,1:5]
```

```
[,1] [,2] [,3] [,4] [,5]
[1,]    3    8    5   15   12
[2,]    4    6   12    8   16
[3,]    5   10   13   17   20
```

Our second-to-last construction before producing the plot creates a matrix whose first column is the order of the triple and the second is the associated triple. In this case **rep(1:21,each=3)** creates the vector 1, 1, 1, 2, 2, 2, ..., 21, 21, 21 and **c(Triples)** stacks the columns of **Triples**. These two vectors are put together with **cbind** or column bind. The first 10 rows and both columns are returned with **Two.Columns[1:10,]**.

R Code

```
> Two.Columns=cbind(rep(1:21,each=3),c(Triples))
> Two.Columns[1:10,]
```

```
[,1] [,2]
[1,] 1 3
[2,] 1 4
[3,] 1 5
[4,] 2 8
[5,] 2 6
[6,] 2 10
[7,] 3 5
[8,] 3 12
[9,] 3 13
[10,] 4 15
```

Finally, we create the 21 by 85 matrix B with all 0s as we have 21 triples and the largest value is 85. The second line is key in that each row of Two.Columns is read as an (i,j) location of B and is replaced with a 1. The first two rows and 10 columns are returned by B[1:2,1:10]. Notice, for example that in the first row there is a 1 under 3,4, and 5. Similarly, compare the second row with the output of Two.Columns[1:10,] above.

R Code

```
> B=matrix(0,ncol=85,nrow=21)
> B[Two.Columns]=1
> B[1:2,1:10]
```

```
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 0 0 1 1 1 0 0 0 0 0
[2,] 0 0 0 0 0 1 0 1 0 1
```

The beginning of creating our graph sets the bottom, left, top and right, margins to 3,3,4, and 2, respectively with **par(mar=c(3,3,4,2))**. The title of the graph is complicated enough that we define it separately and set it to Title. Here, we use **expression**, for mathematical objects, and then **atop** to stack our two lines. The first line is created by **paste** where text is in quotes. The second part of **atop** is all text and in quotes. The **image** function creates the checkerboard plot where each column of the matrix is plotted along a row, and hence the need to transpose B with **t(B)**. The **image** function creates the plot from 0 to 1 and so we remove the axes with **axes=FALSE** and we will create our own in subsequent lines. The color scheme is set with **col** and the title is added with the main argument.

We set corners to the four corners of the graph frame with **par("usr")**. To create a grid, **abline** is used with the vertical lines set to the sequence generated by **seq(corners[1],corners[2],len=86)**. Note that we need 86 lines for 85 cells. Similarly, **h=seq(corners[3],corners[4],len=22)** creates the horizontal lines.

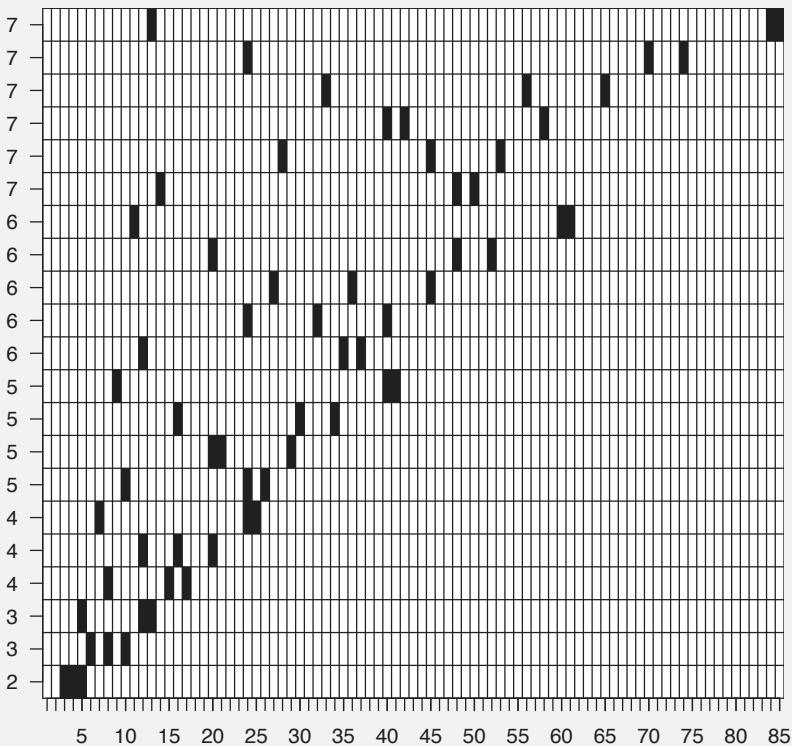
We now use **axis** three times. The first use adds 85 tick marks along the x -axis. Here the first argument 1 is for the x -axis, the second is where to place the tick marks, and the third leaves off any labels with label=FALSE. The next line places the values 5 through 85, **seq(5,85,by=5)** set to label, at the locations listed by **seq(start.axis,1,length=17)**, where start.axis is the fifth tick mark found by **seq(0,1,length=85)[5]**. Again, the first argument of 1 is for the x -axis. The last use of **axis** is for the y -axis, first argument 2, and labels the 21 locations from 0 to 1, **seq(0,1,length=21)**, with the values from the vector m created earlier. The last argument, las=2, orients the labels so that they are horizontal.

R Code

```
> par(mar=c(3,3,4,2))
> Title=expression(atop(paste("Pythagorean Triples
Generated by ",m^2-n^2," , ",2*m*n, ", and ",m^2+n^2,"."),
"The y-axis displays m with the values of n taken from 1
to m-1."))
> image(t(B),axes=FALSE, col=c("white","black"),
main=Title)
> corners=par("usr")
> abline(v=seq(corners[1],corners[2],len=86),
h=seq(corners[3],corners[4],len=22))
> axis(1,at=seq(0,1,length=85),label=FALSE)
> start.axis=seq(0,1,length=85)[5]
> axis(1,at=seq(start.axis,1,length=17),
label=seq(5,85,by=5))
> axis(2,at=seq(0,1,length=21),label=m,las=2)
```

Pythagorean Triples Generated by $m^2 - n^2$, $2mn$, and $m^2 + n^2$.

The y-axis displays m with the values of n taken from 1 to $m-1$.



Code Review for the Functions and Their Graphs Chapter

abline(h=v1,v=v2) adds horizontal lines at the points given by vector $v1$ and vertical lines at the points given by vector $v2$. Execute ?abline for optional graphing parameters.

axis(i,at=label=) adds tick marks to the graph at the location given by at with labels given by labels. The values of i are 1, 2, 3, and 4, for the bottom, left, top, and right axis, respectively.

curve(f,a,b) graphs the function f from a to b . When the argument add=TRUE is included the function is added to the current graph. There are numerous options. Execute ?curve for optional graphing parameters.

function(x₁, x₂, ..., x_n) {expression} defines a function of

x_1, x_2, \dots, x_n with expression. The function will return the last line of expression and for clarity use return().

grid($n1, n2$) adds a grid to the current graph with $n1$ and $n2$ cells in the x and y direction, respectively. The arguments can be set to NULL in which case lines are placed at the current tick marks of the graph. Two useful options are col, for the color of the grid lines, and lty, for the type of grid lines.

ifelse(condition, expression1, expression2) returns expression1 if condition is true or else expression2 is returned.

image(A) is used here to create a checkerboard plot where A is a matrix of 0s and 1s. The columns of A are associated with rows on the checkerboard. Use col=c("color1", "color2") for the colors of the grids.

par(mar=c(i, j, k, l)) sets the bottom, left, top, and right margins to i , j , k , and l lines, respectively.

par("usr") returns a vector x_1, x_2, y_1 , and y_2 representing the corners of the current graph.

points(x, y) adds a point at (x, y) to the graph. Vectors can be used for x and y to add multiple points. Execute ?points for optional graphing parameters.

polar($t, f(t)$) returns a polar plot of $f(t)$, where t is given in radians. Uses the pracma package. Allows for most optional graphing parameters, with grcol the grid color and bxcoll the box color specific to polar().

rainbow(n) returns a vector of n colors spaced across the colors of a rainbow.

stepfun($v1, v2$) returns a step function with jumps listed by vector $v1$ and jumps defined by the vector $v2$. Note vector $v2$ must be one longer than vector $v1$.

which.min(v) returns the location of the minimum value of vector v , with the first location being returned if there are ties.

2.8 Exercises

1. Define and graph a piecewise function that is x^2 on $(-\infty, 0]$, $\cos x$ on $(0, 2\pi)$, and 1 on $[2\pi, \infty)$ on the domain $[-10, 10]$.
2. Define and graph a step function to represent each of the three scenarios. (1) \$10 invested at 5% interest, compounded annually, for 25 years. (2) Scenario (1) but add an extra \$10 to the account at year 10. (3) Scenario (1) but add \$1 to the account each year.
3. Define and graph parametric equations for a unit circle.
4. Using polar coordinates create a graph of a “rose” with two sets of three petals, with each set a different color. Hints: $a \cos b\theta$ and $a \sin b\theta$.
5. Define and graph a hyperbola based on the geometric definition similar to the graph of a parabola in [Section 2.5](#).
6. Create two functions, one that returns a horizontal shift of a function and one that returns a vertical shift of a function. Create graphs to illustrate each.
7. Similar to the exponential example in [Section 2.6](#), illustrate how changing each of a , b , and c affects the graph of $ax^2 + bx + c$.
8. Create a Euclid3 function that will take a pair of integers in any order and return a Pythagorean triple or “error” if the integers are equal.
9. The checkerboard plot created in [Section 2.7](#) is dependent on the order of the n, m pairs of integers. In this case the values of m in the matrix A were nondecreasing. Following the example, create the plot with the same value of n and m , but have the n values nondecreasing.

3

Graphing

The primary and most general function for graphing is **plot** and there are special functions for statistics related graphing commands such as **barplot**, **boxplot**, **dotchart**, **hist**, and **pie**. The options that control the elements of **plot** generally work for the other graphing functions and so it makes sense to begin with **plot**. We note that **curve** is another functions for graphing that may be a more natural choice for mathematicians as it is used to graph real valued functions, which was used extensively in [Chapter 2](#). Our first set of examples will provide an orientation to the **plot** command and the graphing frame. The goal is to orientate us to the graphing frame, the margin, and the outer margin. One could get away with skipping these three examples and referring back to them as needed. On the other hand, we illustrate a multitude of options while demonstrating the structure R has in place for creating exceptional graphs.

The example begins by defining the two vectors `x0` and `y0`, which are the coordinates of the four corners of the blue dashed square. The beginning of the **plot** function has the vectors `x0` and `y0` to provide *x* and *y* coordinates for plotting. The point (1, 1) is listed twice in the vectors because plot will draw lines from one point to the next and without the repeated (1, 1) the bottom line of the square would be missing. The plot type is `o`, which plots the points and also connects them. The next two options, `pch` and `cex`, define the type of points to plot and the size with `cex=3` increasing the point size by 300%. Similarly, `lty` and `lwd` define the type of line and the thickness. The options `xlim=c(0,5)`, `ylim=c(0,5)` set the axis ranges, while `xlab` and `ylab` set axis captions and `main` provides a title for the graph. Lastly, we set `col="blue"` so that the graph is in blue. Note how text items are in quotations. The axis tick marks and numbering are created by default.

R Code

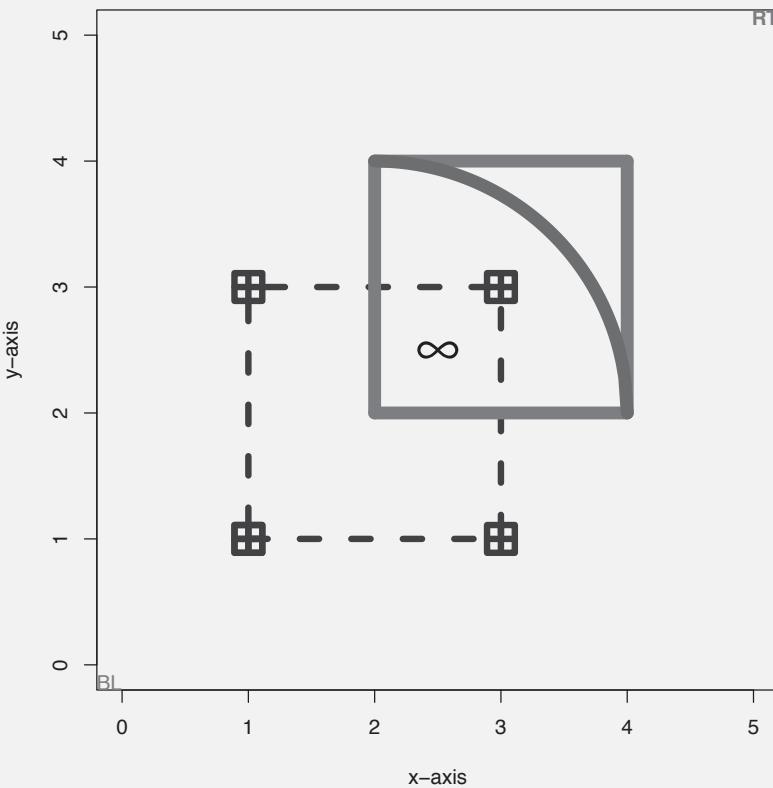
```
> x0=c(1,1,3,3,1)
> y0=c(1,3,3,1,1)
> plot(x0,y0,type="o",pch=12,cex=3,lwd=5,lty=2,
+ xlim=c(0,5),ylim=c(0,5),xlab="x-axis",ylab=
+ "y-axis",col="blue",main="An Example")
> x1=c(2,2,4,4)
> y1=c(2,4,4,2)
```

```

> x2=c(2,4,4,2)
> y2=c(4,4,2,2)
> segments(x1,y1,x2,y2,lwd=10,col="darkorchid")
> curve(2+sqrt(4-(x-2)^2),2,4,col="red",lwd=10,
add=TRUE)
> p=par("usr")
> text(p[1], p[3],"BL",adj=c(0,0),col="darkorchid")
> text(p[2], p[4],"RT",adj=c(1,1),col="darkorchid",
font=2)
> text(mean(p[1:2]),mean(p[3:4]),expression(infinity),
cex=3,adj=0.5)

```

An Example



The next object created is the purple, or darkorchid, solid square. This will be done with the **segments** function, which will draw a line segment from one point to another. The four vectors `x1,y1,x2`, and `y2` define the line segments to be drawn for the square. For example, **segments** draws a line from (2,2)

to $(2, 4)$, which are the first values in the four vectors. The next two options in **segments**, lwd and col, set the width and color of the lines. Notice that lty was omitted and has a default value of 1, which is a solid line.

We now add the arc or quarter of a circle with the **curve** function. The first element is the function to graph, in this case a circle of radius 2 centered at $(2, 2)$. The next two values define the domain of x for plotting, in this case from $x = 2$ to $x = 4$. The **curve** function is a stand alone command; in other words it doesn't need a previous plot to be created to be used. In this case the curve is added to an original plot and the plot domain has been defined. Generally, **curve** is a good choice for mathematics since it graphs functions. Finally, add=TRUE adds the curve or function to the existing graph.

We now add three text objects, BL, RT, and ∞ , to the graph. In order to place the objects in their location we use the **par("usr")** command. A four element vector (v_1, v_2, v_3, v_4) is returned by **par("usr")** and set to p, where (v_1, v_3) is the bottom left corner of the graph and (v_2, v_4) is the top right corner of the graph. The first text command placed BL at $(p[1], p[3])$ which is the left bottom corner of the plot region. The adj=c(0,0) command justifies the letters BL to the left and bottom of the location where it is written. The letters BL, for bottom left, are colored purple, or officially darkorchid. Similarly we place RT at the right top of the plot region with coordinates $(p[2], p[4])$, which is the location of the right top of the graph. The text is justified with the adj=c(1,1) command at right/top of the point $(p[2], p[4])$ and colored darkorchid. The option font=2 boldfaces the text. One last symbol was placed on the graph. The new option here is **expression**, which allows for typesetting mathematical objects, in this case the infinity symbol. Note how **mean** was used twice to get the center of the graph; the infinity symbol was scaled by 300% with cex=3 and center justified with adj=0.5.

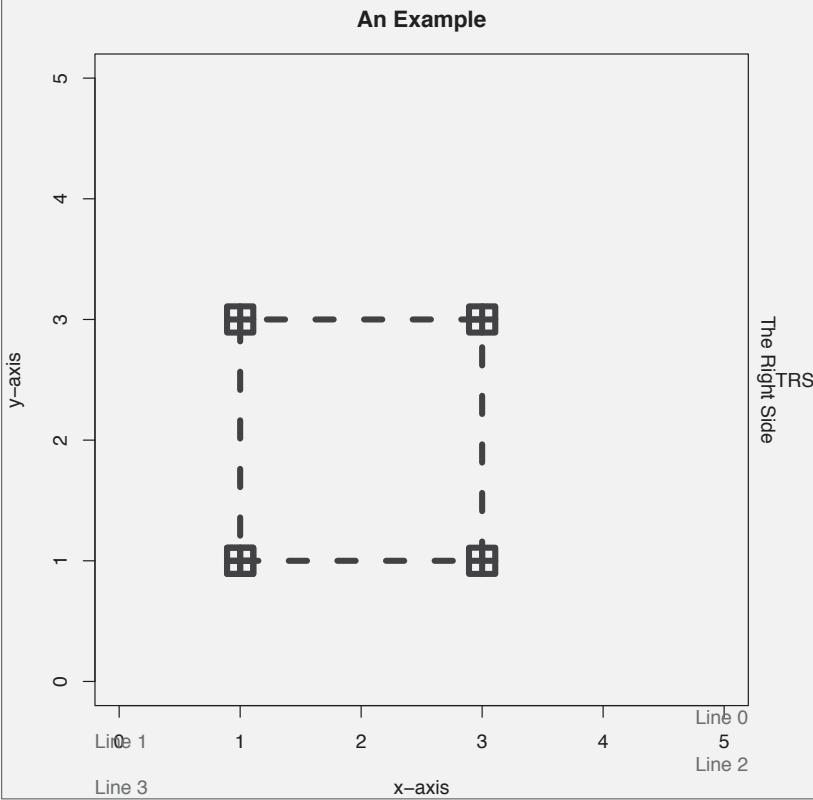
Our next example focuses on the margin. Three of the first four lines define the vectors x0 and y0, and plot the box as in the first example. The second line is a **par** function. We set the four sides of the margin in the order of bottom, left, top, and right with mar=c(4,4,3,3) which provides 4,4,3, and 3, lines for the four sides respectively, around the margin. We already have text in the margin due to the plot command, which added x-axis, y-axis, and An Example. After plot the four **mtext**, margin text, functions place line 0 through line 3 in red in the bottom margin. The desired text is in quotations and side=1 places the text in the bottom margin, whereas 2, 3, and 4 would place the text on the left, top, and right margins respectively. The third option, line=, places the text in the desired line. Recall that **par(mar=c(4,4,3,3))** provided for 4 lines of margin along the bottom. Notice how line 0 aligns with the tick marks, line 1 with the axis numbers, line 2 with a space below the axis numbers, and line 3 with the axis caption. The fourth option, adj, justifies text to its location with 0 left/bottom, and 1 right/top, which places "Line 0" and "Line 2" on the right and the other two on the left. The last option colors the text red.

R Code

```

> x0=c(1,1,3,3,1)
> y0=c(1,3,3,1,1)
> par(mar=c(4,4,3,3))
> plot(x0,y0,type="o",pch=12,cex=3,lwd=5,lty=2,
+ xlim=c(0,5),ylim=c(0,5),xlab="x-axis",ylab="y-axis",
+ col="blue",main="An Example")
> mtext("Line 0",side=1,line=0,adj=1,col="red")
> mtext("Line 1",side=1,line=1,adj=0,col="red")
> mtext("Line 2",side=1,line=2,adj=1,col="red")
> mtext("Line 3",side=1,line=3,adj=0,col="red")
> box("figure", col="red")
> p=par("usr")
> text(p[2]+.15,mean(p[3:4]),"The Right Side",srt=-90,
+ xpd=NA,adj=0.5)
> mtext("TRS",side=4,line=2,adj=0.5,las=2)

```



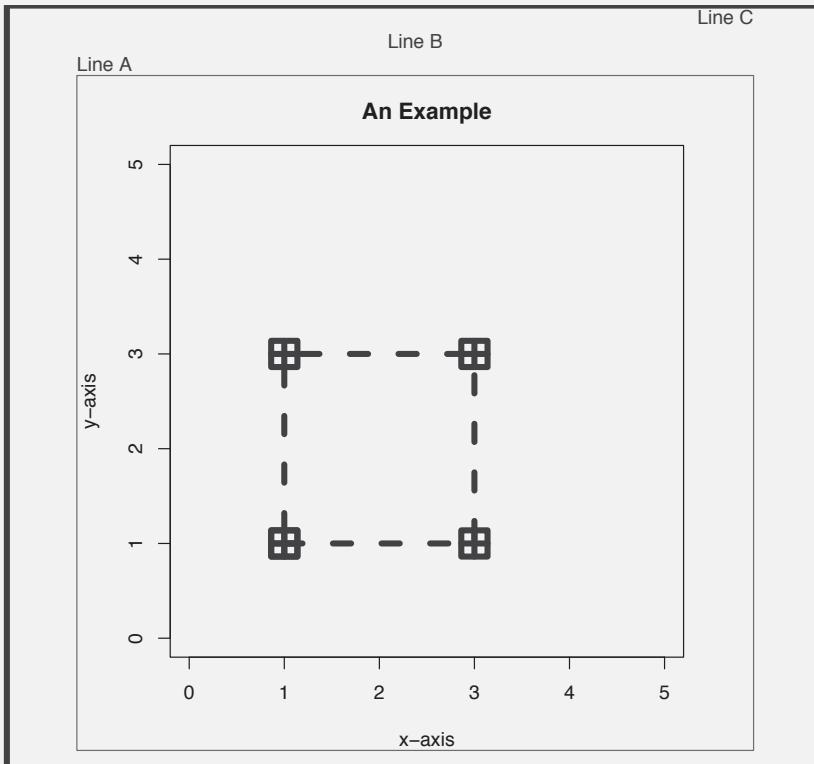
We place a box around the margin with **box("figure",col="red")**. The last three lines place the text on the right margin. Again we define p to be **par("usr")** so that we know the coordinates of the corners to help us place the text using the **text** function. The first two options of **text** define the coordinates for the text. An *x* coordinate of $p[2]+0.15$ places the text outside the graph since $p[2]$ is the right-most value of the graph. The *y* coordinate is the average of $p[3]$ and $p[4]$. We use the **mean** command where $p[3:4]$ are the values of the vector *p* from 3 through 4, which in this case is just the third (bottom) and fourth (top) values. The command *srt*=−90 rotates the text 90 degrees counterclockwise. The command *xpd* can assume three values, FALSE, TRUE, and NA, which clips the text to the plot region, figure region, and device region respectively. This needs to be set to NA so the text appears past the margin instead of being clipped. The last option *adj*=0.5 centers the text and $c(0.5,0.5)$ could also have been used. Another way to place similar text on the graph is to use **mtext**, which is the next command. Here we place the TRS on the fourth side, the right, on the second line with *line*=2, and centered with *adj*=0.5. The last option, *las*=2, places the text perpendicular to the side. Setting *las*=0 would place the text parallel to the side. Note that the **mtext** command limits where text can be placed with *adj*, whereas the **text** command can place text anywhere. Also, with **mtext** the orientation of the text is limited to parallel or perpendicular to the margin; we cannot rotate the text as we did with **text**.

Our final example focuses on the outer margin . Four of the first five lines are the same as the previous example. We define our vectors *x0* and *y0* for plotting with **plot**, and define our margin with **par(mar=c(4,4,3,3))**. The fourth line, **par(oma=c(3,3,3,3))**, defines the outer margin with three lines on all sides. The four values represent the four sides of the outer margin in the order of bottom, left, top, and right, respectively. By default the outer margin is set to 0 for all sides. As before we box the margin in red with **box("figure", col="red")**, but now we also box the outer margin in blue with **box("outer", lwd=5, col="blue")**.

R Code

```
> x0=c(1,1,3,3,1)
> y0=c(1,3,3,1,1)
> par(mar=c(4,4,3,3))
> par(oma=c(3,3,3,3))
> plot(x0,y0,type="o",pch=12,cex=3,lwd=5,lty=2,
+ xlim=c(0,5),ylim=c(0,5),xlab="x-axis",ylab="y-axis",
+ col="blue",main="An Example")
> box("figure",col="red")
> box("outer",lwd=5,col="blue")
> mtext("Line A",side=3,line=0,adj=0,col="blue",
+ outer=TRUE)
```

```
> mtext("Line B", side=3, line=1, adj=0.5, col="blue",
outer=TRUE)
> mtext("Line C", side=3, line=2, adj=1, col="blue",
outer=TRUE)
```



The next three uses of **mtext** are almost exactly the same as those used in the previous example, except `outer=TRUE` places the text in the outer margin. Recall that `par(oma=c(3,3,3,3))` provided for three outer margin lines around the whole graph. With `side=3` the text is placed on the top and we set `adj` to 0, 0.5, and 1 to place the text on the left, center, and right, respectively. The text is colored blue. At this point, it would be a good idea to open R, if you haven't already done so, and use the code here while changing values to help understand the options. Graphs can be saved by going to File -> Save while the graph window is selected.

3.1 Graphing Functions

The most common graphing needs in early mathematics classes is to graph a function. Hence, we begin our examples by illustrating basic graphing by using $\sin(x)$ and its fifth degree Taylor polynomial. We create two examples. The first is done using default options in R, with the goal of getting a quick and easy graph that will be acceptable for basic uses. The next graph will be with the same functions, but we go all out to create a graph with plenty of bells and whistles. The code is daunting at first for the second example, but as we walk through it you should get a sense of how it was pieced together.

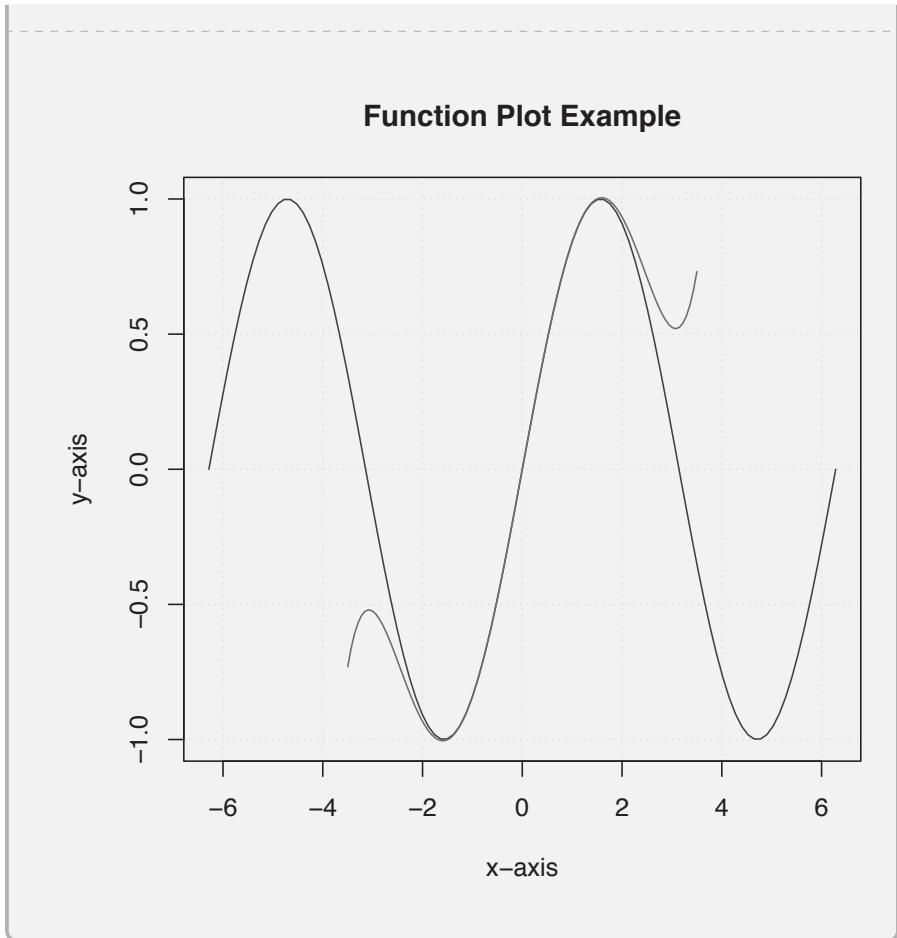
The code for the first graph begins by defining the functions **f** and **g**. In each case we define the function with **function(x)**, declaring x as the variable, and then within braces we explicitly input the desired formula. Note that you don't want to name the function as $f(x)$ since you would then evaluate it as $f(x)(2)$. Once a function is defined it can be used as you might expect. For example, you can evaluate a function by typing **f(pi)**.

The first use of the **plot** functions graphs **f**, sets the color to "blue", and defines the x and y range with **xlim** and **ylim**. Notice that pi is interpreted as π . We add a grid to the graph with **panel.first=grid()**, where **panel.first** is used to draw the grid first so that other objects, such as the functions, are drawn over the grid. Here we use the default setting of **grid** by leaving the options inside the parentheses empty, although this often leaves a grid that is very faint. The last three options create an x and y axis caption, as well as a title for the graph.

The second use of the **plot** function adds the polynomial to the graph, with the first option the function **g**. One of the useful features is that the plot range of the function can be defined differently than the first use of **plot**. Here we set the x range as -3.5 to 3.5. You need to set the x range even if it is the same as the first use of plot; otherwise your function might not plot or only a piece of it will be plotted. We color this function red and set **add=TRUE** so that the function is added to the first use of plot. Note that we chose to color the functions even though it is not necessary and note that the default color is set to black. This could just as easily be done using **curve** instead of **plot**.

R Code

```
> f=function(x){sin(x)}
> g=function(x){x-x^3/factorial(3)+x^5/factorial(5)}
> plot(f,col="blue",xlim=c(-2*pi,2*pi),ylim=c(-1,1),
  panel.first=grid(),xlab="x-axis",ylab="y-axis",
  main="Function Plot Example")
> plot(g,xlim=c(-3.5,3.5),col="red",add=TRUE)
```



We have a useful graph, but there are a number of improvements we might like to make. It is typical for the axis to be in the center of the graph like a crosshair and given that we are graphing $\sin(x)$ we would prefer to label the x -axis from -2π to 2π . The grid could be improved and it would be nice to label the functions in the graph. We do all this with our next example. We start by defining our functions as in the previous example.

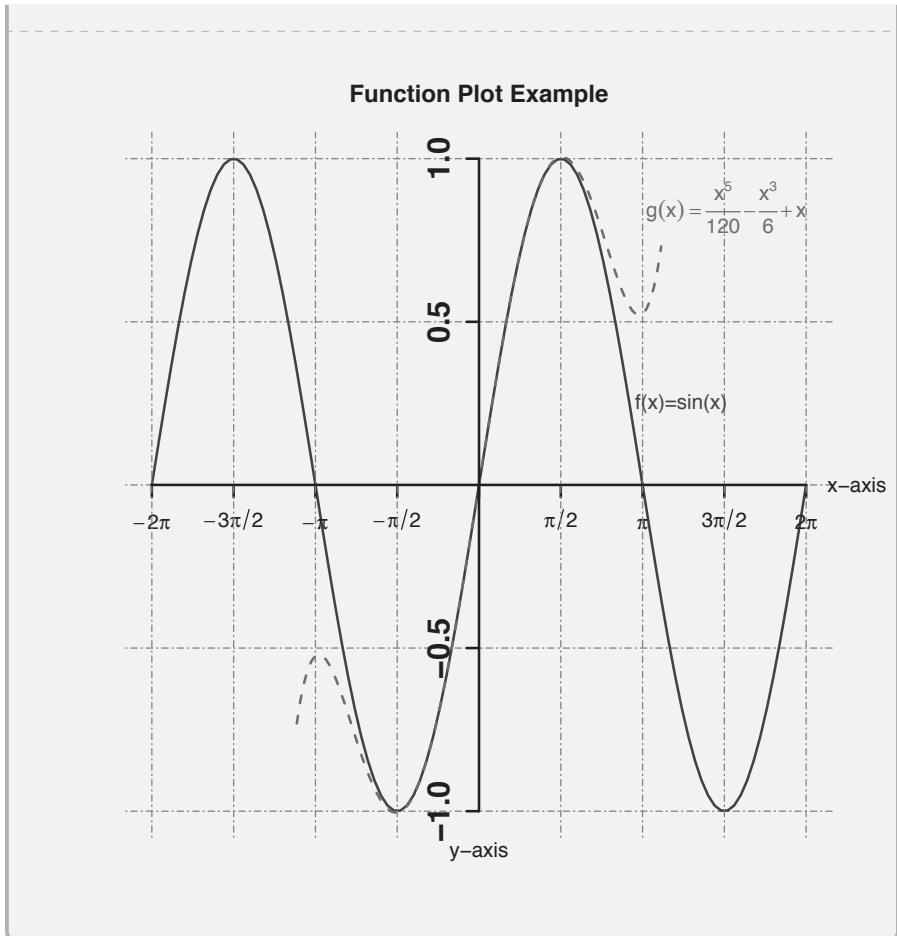
The **par** function sets the margins around the graph to three lines on all sides plus a small buffer of 0.2 lines. The **plot** function plots the function **f** with a line width twice the default set by **lwd=2** and the line type is a solid line set by **lty=1**. The function is colored blue and the x and y ranges are set with **xlim** and **ylim**. The default x and y axes are turned off with **xaxt** and **yaxt** set to "n", since we will define them to be centered later in the code. We could have turned off both with the one option **axes=FALSE**. We also turn off the frame around the plot with **frame.plot=FALSE**. We do not want a caption

along the x or y axis and so `xlab` and `ylab` are set to blank with empty quotes, but a title is added with `main`.

Finally, the `panel.first` command draws the grid first but we do not use the defaults for `grid`. Within `grid` the first two options are the number of boxes in the x and y direction, respectively. Because we will define special tick marks along the x -axis, `grid` has NA which creates no lines along the x -axis. Gridlines for the x -axis will be drawn later in the code. On the other hand, we use default tick marks on the y -axis and `NULL` places gridlines along the default tick marks. For both of these options, a number can be used instead of `NA` or `NULL`. We use `lty=6` to set the lines to a form of dashed lines—there are options 2 through 6 for dashed lines—and color the lines with `gray60`. Note that there are gray options from `gray0`, black, to `gray100`, white, as well as just `gray`.

R Code

```
> f=function(x){sin(x)}
> g=function(x){x-x^3/factorial(3)+x^5/factorial(5)}
> par(mar=c(3,3,3,3)+0.2)
> plot(f,lwd=2,lty=1,col="blue",xlim=c(-2*pi,2*pi),
+ ylim=c(-1,1),xaxt="n",yaxt="n",frame.plot=FALSE,xlab="",
+ ylab="",main="Function Plot Example",
+ panel.first=grid(NA,NULL,lty=6,col="gray40"))
> plot(g,lwd=2,lty=2,col="red",xlim=c(-3.5,3.5),
+ add=TRUE)
> v1=c(-2*pi,-3*pi/2,-pi,-pi/2, 0,pi/2,pi,3*pi/2,2*pi)
> v2=c(expression(-2*pi),expression(-3*pi/2),
+ expression(-pi),expression(-pi/2),"",expression(pi/2),
+ expression(pi),expression(3*pi/2),expression(2*pi))
> axis(side=1,pos=0,lwd=2,labels=v2,at=v1,cex.axis=1)
> axis(side=2,pos=0,lwd=2,c(-1,-0.5,0.5,1),
+ cex.axis=1.5,font=2,las=0)
> v3=c(-2*pi,-3*pi/2,-pi,-pi/2, pi/2,pi,3*pi/2,2*pi)
> abline(v=v3,lty=6,col="gray40")
> mtext("x-axis",side=4,line=1,adj=0.5,las=2)
> mtext("y-axis",side=1,line=0,adj=0.5,las=0)
> text(3,0.25,"f(x)=sin(x)",adj=0,col="blue")
> text(4.75,0.85,expression(g(x)==frac(x^5,120)-
+ frac(x^3,6)+x),adj=0.5,col="red")
```



The second use of **plot** graphs the function **g**. Again we use a `lwd=2`, double the default width, but use a dashed line with `lty=2` which allows us to see both functions when they overlap. The function is colored red and its domain is set with `xlim=c(-3.5,3.5)`, which is smaller than the domain of the graph. The last option, `add=TRUE`, adds the graph to the previously created graph with **plot**, as opposed to drawing a new graph. We now move to creating our axis.

The vectors **v1** and **v2**, define the location of the axis labels and the labels. Vector **v1** is numeric whereas **v2** is a character vector using the **expression** command to create mathematical symbols. The first **axis** command creates the *x*-axis with `side=1` setting it along the bottom or really horizontal since `pos=0` draws the axis at 0 along the *y*-axis. The line width of the axis is set to double the default with `lwd=2`, the tick marks are labeled with the vector **v2** by `labels=v2`, and their locations are defined by the vector **v1** with `at=v1`. The last option, `cex.axis=1`, sets the size of the labels to the default of 1. The

second use of axis creates the y -axis with side=2 for the left side or really vertical since it is located at 0 along the x -axis with pos=0. The axis has a width double the default with lwd=2. If only one vector is included with the axis command, in this case c(-1,-0.5,0.5,1), it is used for both the labels and the locations. The next two options, cex.axis=1.5 and font=2, scales the labels to 1.5 the default size and boldfaces them. Note, the mathematical expressions used on the x -axis cannot be made bold although they can be scaled. We set las=0 so that the labels are parallel to the axis and note that las=2 would orient them to perpendicular.

The next two commands, v3 and **abline**, will create the vertical gridlines. The vector v3 defines the locations for the lines, and notice that we left out 0 since that is the y -axis. The **abline** command draws lines and with v=v3 draws vertical lines at the v3 locations. We use the same line types as we used with grid, lty=6, although it is not necessary, and also color them with gray40. We could have drawn the horizontal gridlines in the same fashion by using **abline** with h=c(-1, -0.5, 0.5, 1), but we wanted to illustrate the use of **grid**, which usually works fine. We now move to adding text to our graph.

The two **mtext** functions label the x and y axes, with x-axis and y-axis. The first **mtext** function places the text on the right with side=4, in the second line of the margin with line=1, centered with adj=0.5, and perpendicular to the side with las=2. The second use of **mtext** places the text on the bottom with side=1, in the first line of the margin with line=0, centered with adj=0.5, and parallel to the side with las=0. We label the $\sin(x)$ graph with the **text** command by placing the text at (0,0.25), left justified with adj=0, and colored blue. We label the polynomial by placing the text at (4.75,0.85). Note the syntax in the **expression** command to create the mathematical symbols. The text is centered with adj=0.5 and colored red. Both of these locations were chosen by a bit of trial and error. Now that is one nice graph.

3.2 Scatter Plots

Possibly the second most common graph is a scatter plot and for this example we will use the Ice data from [Section 1.1](#) and discuss each of the options in the commands. This example assumes that the Arctic Ice Data has been imported as discussed in [Section 1.1](#).

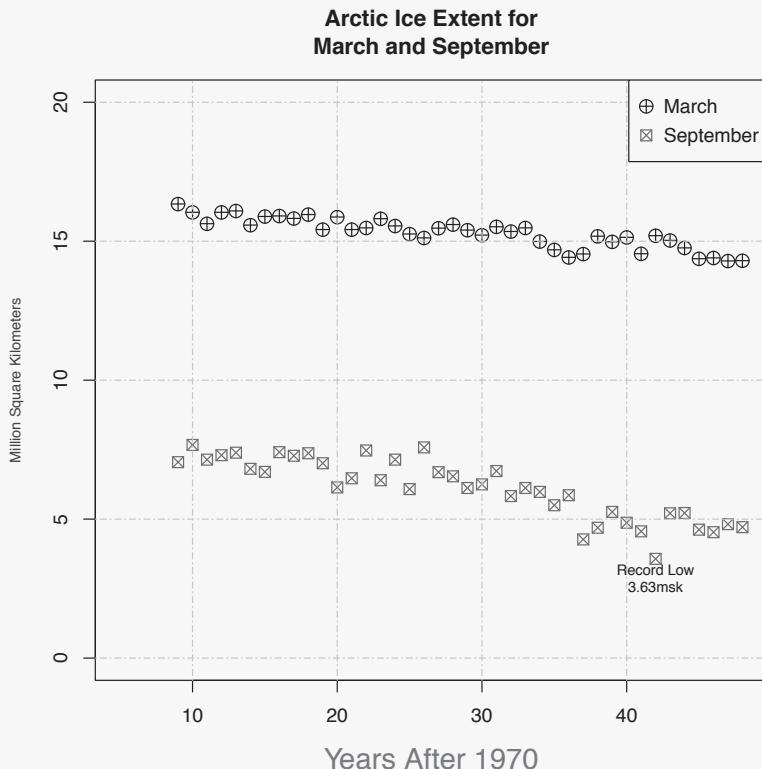
R Code

```
> par(bg="#fdfd7ef")
> plot(Ice$Years.after.1970,Ice$March.Extent.in.MSK,
type="p",cex=1.5,pch=10,xlim=c(5,48),ylim=c(0,20),xlab=
"Years After 1970",ylab="",col.lab="purple",cex.lab=1.5)
```

```

> points(Ice$Years.after.1970,
Ice$September.Extent.in.MSK,type="p",
cex=1.25,pch=7,col="red")
> title(main="Arctic Ice Extent for\nMarch and
September",ylab="Million Square Kilometers",
col.lab="blue",cex.lab=0.75)
> grid (NULL,NULL,lty=6,col="gray")
> lvn=c("March","September")
> legend("topright",lvn,pch=c(10,7),pt.cex=c(1.5, 1.25),
col=c("black","red")),y.intersp=1.25)
> text(42,3.63,"Record Low\n3.63msk",pos=1,cex=0.75)

```



The **par** function is optional and sets the background color of the graph. If it is left out the default is white. The first two variables of the **plot** function are the *x* and *y* data using the names of the columns of the imported data. These must be the first two elements whereas the order of the remaining options can be changed. The type is set to p for points, and we note that two other

common options are l for lines, and o for points and lines. The cex option is a scale factor so that the points are plotted at 1.5 times the default size. The type of points is set by pch and there are numbers 0 through 25 as well as a few select symbols such as + and %. The axes ranges are set by xlim and ylim, and are captioned by xlab and ylab. Note that the text after the axes labeling needs to be in quotes. If they are left out the default is to use the column names as labels. The ylab is left blank here to illustrate the use of label options. The **plot** function finishes up with col.lab for the color of the axes caption, and a scale factor set by cex.lab for the size of the font. Since ylab was blank these commands were applied to the *x* axis caption only.

The **points** function adds another set of points to the main plot. In this case the September data was added to the scatter plot. The options for point are mostly the same as for plot although you do not need to set xlim and ylim since they were set in **plot**. The point type was changed with pch=7 and the col command was used to set the color of the points. In **plot**, the default is black since col was not set.

A title was added to the graph with the **title** function. Note the use of \n to force a line break in the title. If spaces are put around \n then the centering will be off. The *y* axis caption was set here and given a color with col.lab and scaled to be 75% of the default size with cex.lab=0.75. A grid was then added to the graph with the **grid** function. The first two elements are set to NULL which tells R to set them at the default tick marks. The line type is set with lty=6. There are six options from 1 to 6, with 1 being a solid line and 2 through 6 a version of dotted or dashed lines.

The next two commands build the legend. The first, lvn (legend vector names) defines the names of the data for the legend. You can do this directly in the **legend** function but it begins to get lengthy. The first element of **legend** places the legend in the graph. The next is the vector of names. The following three commands define the points we used for each data set so that it is repeated in the legend. The last command is a scale factor for the line spacing in the legend.

Finally, text was added to the graph to point out the record low amount of ice extent. The record low was identified by using **min(September.Extent.in.MSK)**. The first two elements set the coordinates of the text which were chosen by trial and error. The third is the text to be added defined in quotes. The position is set to below with pos=1 and has options 1=below, 2=left, 3=above, and 4=right. There is an offset option as a scale factor, but it wasn't used here. Again, cex is used to scale the font size to be 75% of the default size.

The graph that is created opens in its own window. As you do this you should be working in the editor and so note that if you change a command and rerun the line R will simply add or overwrite to the open window. This can be useful when trying to position text since you can see the changes. To get a clean window to produce your final graph you can simply close out of the window or run **windows**. If you want to work with multiple graphics windows

use **dev.new** to initiate a new window. Note that your graph can be saved in multiple formats and will be saved based on the size of the open window.

3.3 Dot, Pie, and Bar Charts

To create bar, dot, and pie charts we will use the birth month of females, FBM, and males, MBM, as collected by a statistics class. The data will be entered directly into R in two different formats for demonstration purposes. The female data is entered as three letter characters and the males numerically. The data is typed into R and there is nothing returned when it is run.

R Code

```
> FBM=c("Jun", "Nov", "Mar", "Mar", "Aug", "Aug", "Sep", "Sep",
"Sep", "Jul", "Aug", "Nov", "Sep", "Jan", "Mar", "Jan", "Dec",
"Mar", "May", "Mar", "Aug", "Apr", "Jul", "Nov", "May", "Dec",
"Mar", "Oct", "Jan", "Sep", "May", "Feb", "Mar", "Sep", "Dec",
"May", "Aug", "Feb", "Aug", "Sep", "Apr", "Nov", "Mar")
> MBM=c(8,2,2,1,1,1,6,3,8,12,5,11,6,1,10,4,7,3,3,6,9,5,
4,9,11,12,7,4,3,7,1,8,4,7,5,8,6,2,2,4,9,4,4,6,6,2,1,11,
4,4,2,9,7,6,1,4,12,7,4,11,10,6,8,11,6)
```

No Output

We begin with a dotplot for male birth months. The data is entered in the first line as MBM even though we did it above. We follow by defining a vector of months with the vector Month. The two par commands define the spacing around the plot. The default for any plot will leave space in the margins for titles and labels. In particular, the left margin would have space that is not needed for a dotplot. The **par(oma=c(1,1,1,1))** sets the outer margin to 1 line for the bottom, left, top, and right margins. This simply leaves a nice buffer around the graph. The second par command, **par(mar=c(4,1,1,1))**, puts 4 lines along the bottom margin for the axis labels and axis title and then 1 line around the other three margins.

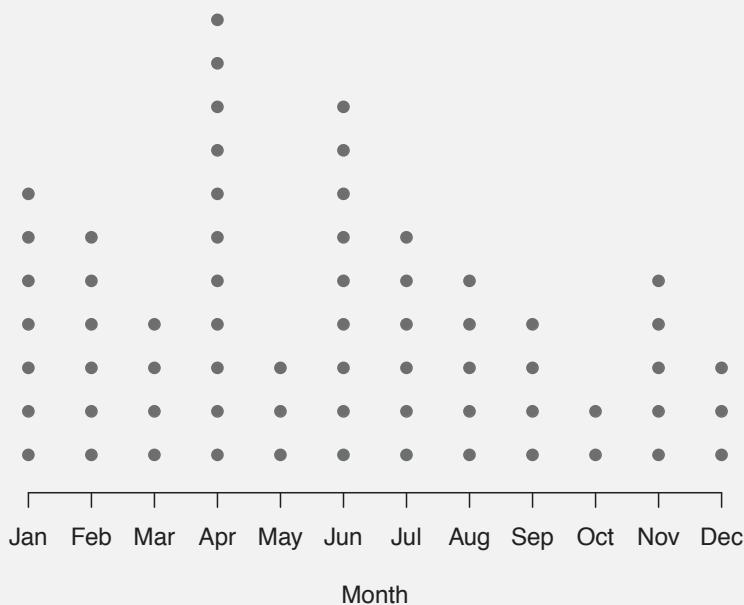
The **stripchart** function creates the dotplot of the MBM data by setting method to stack. The at and offset options start the dots 0.05 above the axis with a 1.5 symbol width spacing between dots. The pch option defines the type of dot and we title the graph with main and caption the *x* axis with xlab. We turn off the axis tickmarks, which would be 1 through 12, by setting axes to FALSE. The following function, **axis**, labels the *x* axis with side set to 1 for bottom and we note that 2 through 4 are for left, top, and right, respectively. The labels will be located at the marks 1 through 12 and they are labelled with the vector Month. If we want a dotplot for females the data needs to be con-

verted to numeric. This can be done by `for(i in c(1:12)){FBM=replace(FBM, FBM==Month[i], i)}` `FBM=as.numeric(FBM)`. We'll work through this code later in the section.

R Code

```
> MBM=c(8,2,2,1,1,1,6,3,8,12,5,11,6,1,10,4,7,3,3,6,9,5,  
4,9,11,12,7,4,3,7,1,8,4,7,5,8,6,2,2,4,9,4,4,6,6,2,1,11,  
4,4,2,9,7,6,1,4,12,7,4,11,10,6,8,11,6)  
> Month=c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",  
"Aug", "Sep", "Oct", "Nov", "Dec")  
> par(oma=c(1,1,1,1))  
> par(mar=c(4,1,1,1))  
> stripchart(MBM, method="stack", at=0.05, offset=1.5,  
pch=19, main="Dotplot of Male Birth Months", xlab="Month",  
ylab="", col="red", axes=FALSE)  
> axis(side=1, at=1:12, labels=Month)
```

Dotplot of Male Birth Months

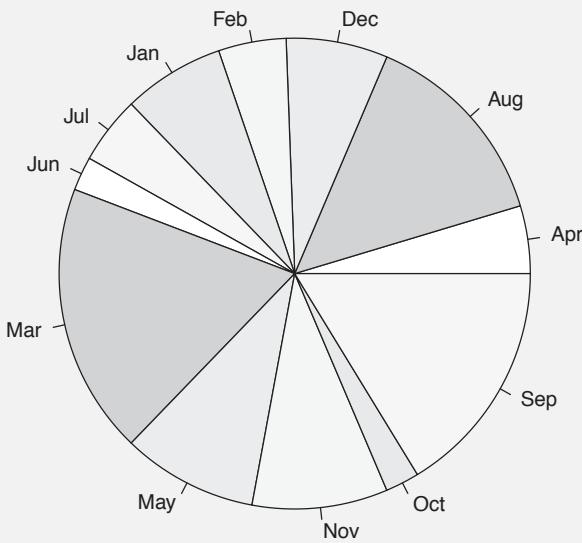


The challenge at this point is that dotplots needs numerical data whereas bar and pie charts need counts. We could have entered counts directly into R, but entering the actual data provides us with an example of converting data to counts. There is more than one way to convert to counts and we will use the table function here but at the end of the section we will demonstrate a method that uses a for loop.

The code begins by entering the female birth month data set, done for completeness even though we entered it above. Next, FBMt set to **table(FBM)** converts the data set to counts and we call it FBMt, with the lower case t standing for table. You should at this point type in FBMt into R to see what was done. In short we now have a table of counts with the first row being the birth months alphabetized. The last command, **pie(FBMt)**, creates the pie chart using the default representation and colors. The same can be done for the male data with the only difference in the pie chart being that the months are labels one through twelve.

R Code

```
> FBM=c("Jun","Nov","Mar","Mar","Aug","Aug","Sep","Sep",
  "Sep","Jul","Aug","Nov","Sep","Jan","Mar","Jan","Dec",
  "Mar","May","Mar","Aug","Apr","Jul","Nov","May","Dec",
  "Mar","Oct","Jan","Sep","May","Feb","Mar","Sep","Dec",
  "May","Aug","Feb","Aug","Sep","Apr","Nov","Mar")
> FBMt=table(FBM)
> pie(FBMt)
```



We have produced a basic pie chart, but we would like something much better. Our goal is to produce side by side pie charts with labels that include the months and percentages, as well as to explore color options. To begin we need to create labels that will include the percent of each category.

R Code

```
> Ftable=as.data.frame(FB Mt)
> FMonth=as.character(Ftable[,1])
> Fcounts=as.numeric(Ftable[,2])
> Fperct=round(Fcounts/sum(Fcounts)*100)
> Flabels=paste(FMonth, Fperct)
> Flabels=paste(Flabels,"%", sep="")
```

No Output.

We begin creating our better pie chart by converting the table, FBMt, to a data frame which will allow us to use the two rows of the table separately. Throughout this example you should run each new variable to see what has changed. For example, type in FBMt and Ftable to see the differences. The next line defines the vector, FMonth, of the female birth months in the order they appear in the table, which in this case is alphabetical. The command Ftable[,1] extracts the data from all rows, by leaving the space before the comma blank, and the first column with 1 after the comma. The function **as.character** informs R to define the data as character data. The next line, Fcounts=**as.numeric(Ftable[,2])**, does basically the same thing except we want data from the 2nd column and it should be numeric.

The next step is to calculate the percentages for each month. The **sum** function sums the values of a vector and so Fcounts/sum(Fcounts)*100 takes each value in the vector Fcounts, divides by the sum and multiplies by 100. We then round each value to the nearest integer with the round function and call it Fperct. The next two lines will **paste** together the vectors to create our labels. We first **paste** (basically combine the vectors element by element) the two vectors with the month labels and the percents, and call the new vector Flabels. We then add the percent sign to the vector by pasting Flables with "%", but add the options of sep with no space between the quotes to tell R not to put a space between each element of the vector Flabels and the appended % sign. If we did this as one step with Flabels=**paste(FMonth, Fperct, "%")** the vector would have spaces between the percentage values for each month and the % sign.

We now do basically the same thing for the male data except we have numbers representing the months instead of the month names. In the code, we create our own character vectors of months, called Months, in the appropriate order for the male data.

R Code

```
> MBMt=table(MBM)
> Mtable=as.data.frame(MBMt)
> Month=c("Jan","Feb","Mar","Apr","May","Jun","Jul",
  "Aug","Sep","Oct","Nov","Dec")
> Mcounts=as.numeric(Mtable[,2])
> Mperct=round(Mcounts/sum(Mcounts)*100)
> Mlabels=paste(Month, Mperct)
> Mlabels=paste(Mlabels, "%", sep="")
```

No Output.

We are now set to put both pie charts on the same output and use the opportunity to demonstrate some options. The code begins with the **layout** function to set up how the graphs appear on the page. We want two side by side pie charts and so we need one row and two columns. Inside the **layout**

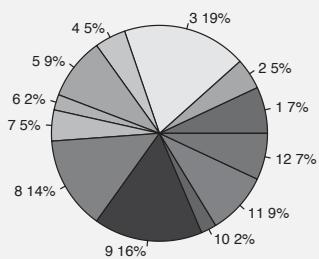
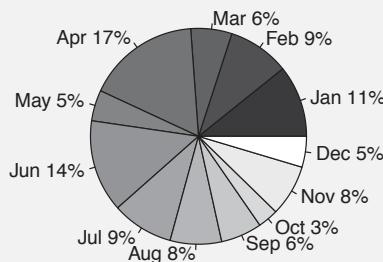
command we set up a 1×2 matrix, where the 1,2 in the middle of the command sets up the matrix. The `c(1,2)` command sets up how the graphs are arranged in the matrix. With the `byrow` command set to TRUE the row is filled out with the first graph and then the second. This seems redundant, and in this case is, but we could set up a 2×2 matrix and use `c(1,1,2,3)` so that the whole first row is used for the first graph and the second and third graph are placed in the second row. In this particular example, `par(mfrow=c(1,2))` will set up a one by two grid for graphing and is easier, but `layout` does much more and we wanted to explain its use.

Moving to the pie charts, we still use the `pie` function but add some options. For the female data we still use the table `FBMt` for the data. We use `main` to provide a title and `labels=Flables` sets the labels to be used as the vector `Flables` we created. We scale the labels to 75% with `cex=0.75` and set the colors using `col=rainbow(12)`. The command `rainbow(12)` is a color palette using 12 colors since we have 12 categories.

For the males pie chart we use the vector of just counts, `Mcounts`, and we could have used `MBMt`, as either is fine. We set the labels to the vector of labels we created, `Mlabels`. We did not add a title at this point as that will be done in the next line. We did not use `cex` and went with the default font size. For colors we went with a set of gray scale colors. In this case `gray` needs twelve values from 0, black, and 1, white. You can use any twelve values, but we went with $1/12, 2/12, 3/12, \dots, 12/12$. This was done by creating the vector of number 1 through 12 with `c(1:12)` and then dividing each by 12. You could let the system choose the 12 shades of gray by using `gray.colors(12)`, which is the black and white analogy to `rainbow`. Finally, we wanted to move the title down two lines and the `title` function is used with line set to -2.

R Code

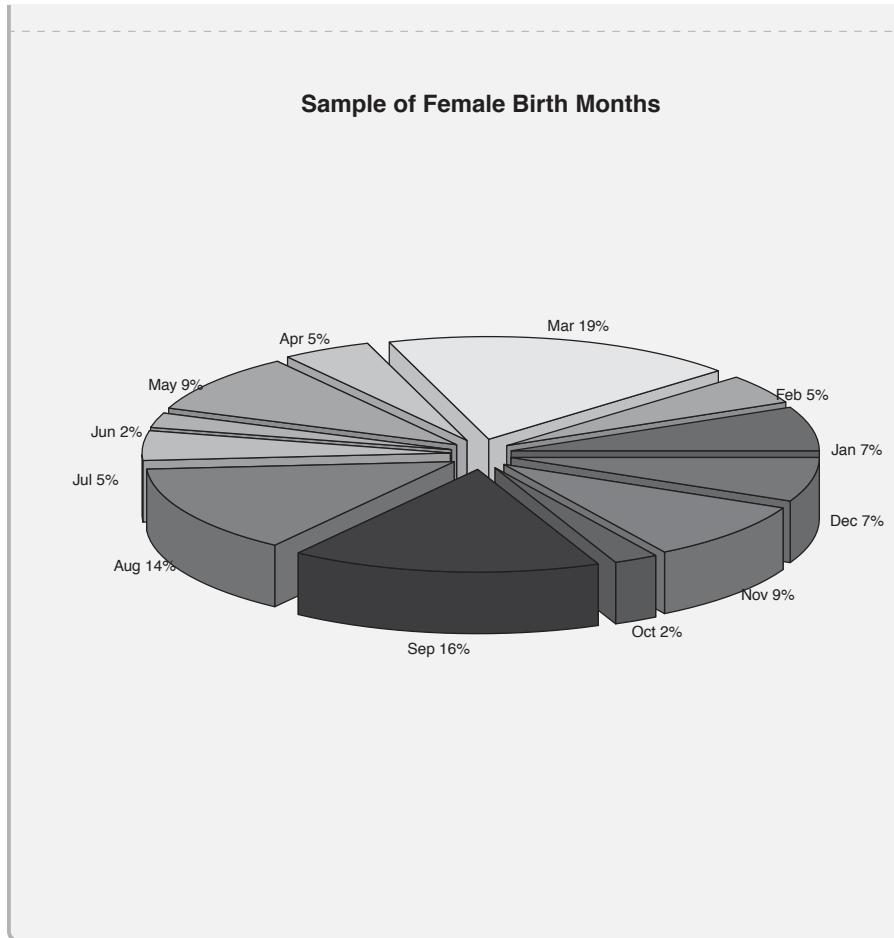
```
> layout(matrix(c(1,2),1,2,byrow=TRUE))
> pie(FB Mt,main="Sample of Female Birth Months",
  labels=Flables,cex=0.75,col=rainbow(12))
> pie(Mcounts,labels=Mlabels,col=gray(c(1:12)/12))
> title("Sample of Male Birth Months",line=-2)
```

Sample of Female Birth Months**Sample of Male Birth Months**

A few details to note. The female title is too far from the pie chart and the title command improves this for the male pie chart. The font from the male pie chart is too large and cex=0.75 improves this in the female pie chart. Both charts are created by starting at about 3:00 and proceeding counter clockwise. Recall that the female data is alphabetized while the male data is in the monthly order. Before moving to bar charts here is a 3D pie chart.

R Code

```
> library(plotrix)
> pie3D(FBMCounts,main="Sample of Female Birth Months",
  labels=Flabels,labelcex=0.75,explode=0.1,
  col=rainbow(12))
```



To create this chart you need to load the **plotrix** package and then call it with the **library(plotrix)** command. The function is **pie3D** with all the options the same except the addition of **explode=0.1**, which scales how far from the center to place each piece of the pie.

If you want a quick and simple bar chart this will do: **barplot(Fcounts, names.arg=FMonth, main="Sample of Female Birth Months", xlab="Month")**. Add **horiz=TRUE** if you want the chart horizontal. We will create a side by side bar chart and then a stacked bar chart. In order to create bar charts with both the female and male data, we will need to combine the data to create a two by two table with names. Once we have done this, the bar charts are simple to create. Our female and male data are ordered differently and so our first step is to change the female data to numeric so that it is ordered the same as the male data.

Before moving to the code, which will contain a for loop, we take a look at a shorter example to replace the months with numeric representations. We

first enter the data for FBM and Month. We then use the **replace** function which takes three options. The first is the vector that needs a replacement, FBM. The second is what to replace. In this case Month[2] = "Feb" and so we look through the vector FBM for instances when an element is "Feb" by using the == equals logical expression. The third option is what is to be used to replace "Feb" and in this case it is 2. The replace command is set equal to FBM to make the change permanent. If we enter FBM now we see that the output now has 2 whenever there was a "Feb", although note that the number 2 is in quotes meaning it is viewed as a character and not a number.

R Code

```
> FBM=c("Jun", "Nov", "Mar", "Mar", "Aug", "Aug", "Sep", "Sep",
       "Sep", "Jul", "Aug", "Nov", "Sep", "Jan", "Mar", "Jan", "Dec",
       "Mar", "May", "Mar", "Aug", "Apr", "Jul", "Nov", "May", "Dec",
       "Mar", "Oct", "Jan", "Sep", "May", "Feb", "Mar", "Sep", "Dec",
       "May", "Aug", "Feb", "Aug", "Sep", "Apr", "Nov", "Mar")
> Month=c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
          "Aug", "Sep", "Oct", "Nov", "Dec")
> FBM=replace(FBM,FBM==Month[2],2)
> FBM

[1] "Jun" "Nov" "Mar" "Mar" "Aug" "Aug" "Sep" "Sep"
[9] "Sep" "Jul" "Aug" "Nov" "Sep" "Jan" "Mar" "Jan"
[17] "Dec" "Mar" "May" "Mar" "Aug" "Apr" "Jul" "Nov"
[25] "May" "Dec" "Mar" "Oct" "Jan" "Sep" "May" "2"
[33] "Mar" "Sep" "Dec" "May" "Aug" "2"   "Aug" "Sep"
[41] "Apr" "Nov" "Mar"
```

At this point we could repeat the **replace** command another 11 times with the remaining number from 1 to 12. This is tedious and a for loop will make this easier. In the next example the first line is a for loop. The loop simply allows *i* to range from 1 to 12 and to execute the **replace** command with each of these values as we showed in the previous example for the case *i* = 2. Since FBM started with characters R replaced them with characters and we really want numbers. We use set FBM to **as.numeric(FBM)** which converts the elements of FBM to numbers. The last two lines create a table from the FBM and the MBM data, but now the tables both have categories from 1 to 12 representing months.

R Code

```
> for(i in c(1:12)){FBM=replace(FBM,FBM==Month[i],i)}
> FBM=as.numeric(FBM)
> FBMt=table(FBM)
> MBMt=table(MBM)
```

No Output.

We are ready to combine the two data sets to create a table. We start by creating FMtable which binds the rows of FBMt and MBMt with the **rbind** command. We define the vector Month and set it to be the column names of our new table with **colnames(FMtable)** set to Month. We have only two row names and so there is no need to create a new vector now and simply use **rownames** to set the names of our rows. By entering FMtable we will get an output to see what our table looks like. As you see we have a table of the counts of the birth months for females and males with two rows and twelve columns. As we move forward we will see that the **barplot** command will use the row and column names in the chart.

R Code

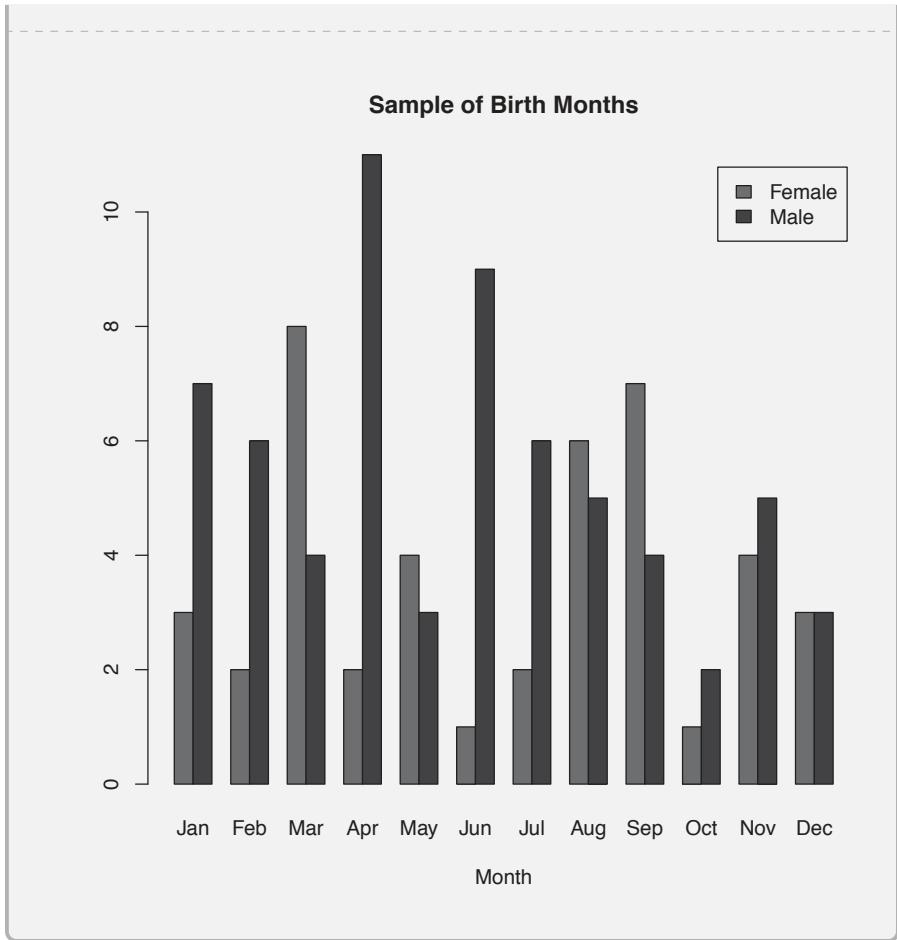
```
> FMtable=rbind(FB Mt, MB Mt)
> Month=c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
  "Aug", "Sep", "Oct", "Nov", "Dec")
> colnames(FMtable)=Month
> rownames(FMtable)=c("Female", "Male")
> FMtable
```

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Female	3	2	8	2	4	1	2	6	7	1	4	3
Male	7	6	4	11	3	9	6	5	4	2	5	3

We now create the side by side plot using **barplot** with input FMtable. We give it a title with the main command and provide a caption for the *x*-axis with *xlab*. The bars are colored red and blue with *col=c("red", "blue")* and we ask for a legend. The last option, *beside=TRUE*, places the bars side by side. Our effort in creating the table pays off here with properly labeled bars and an automatic legend. In our next example the bars are stacked by removing the *beside=TRUE* command and we move the legend to the top left of the graph with *args.legend=list(x="topleft")*.

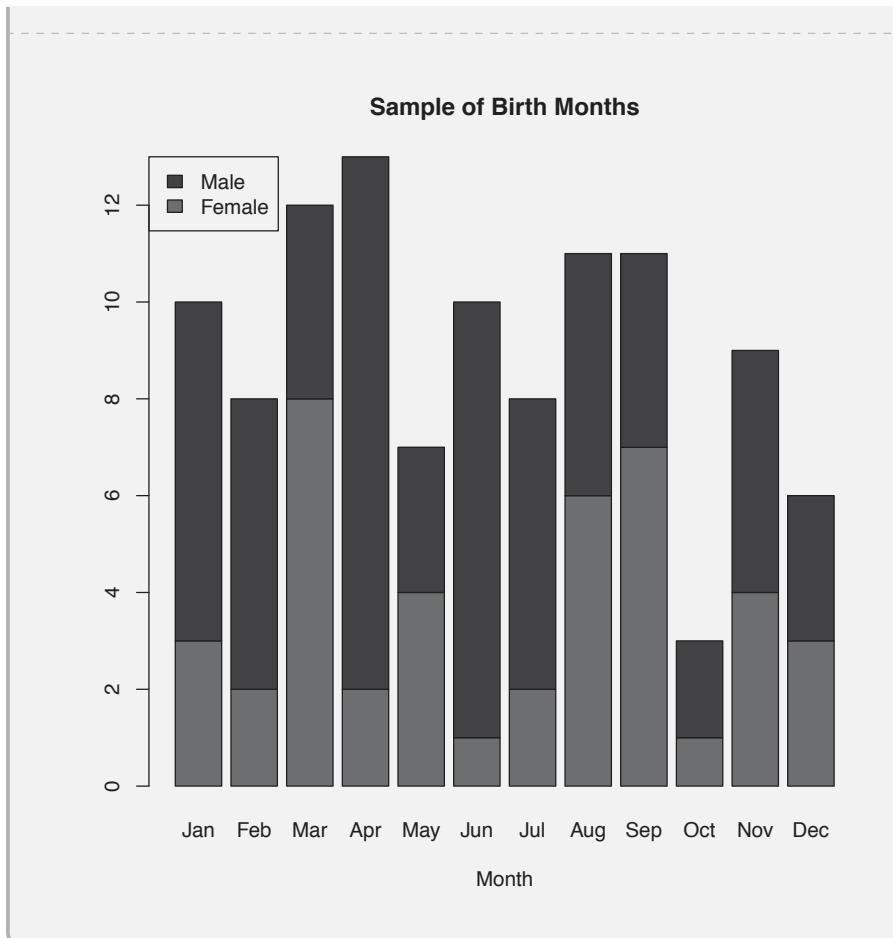
R Code

```
> barplot(FMtable,main="Sample of Birth Months",
  xlab="Month",col=c("red","blue"),legend.text=TRUE,
  beside=TRUE)
```



R Code

```
> barplot(FMtable, main="Sample of Birth Months",
  xlab="Month", col=c("red","blue"), legend.text=TRUE,
  args.legend=list(x="topleft"))
```



3.3.1 A Look at For Loops

As mentioned above there are other methods to get counts from our data. Using a for loop is a bit more complicated than the **table** function, but the for loop is a general tool. We begin by counting the birth months for females by using a for loop.

R Code

```
> Month=c("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
  "Aug", "Sep", "Oct", "Nov", "Dec")
> FBMCounts=vector(mode="numeric", length=12)
> for (i in c(1:12))
{FBMCounts[i]=length(which(FBM==Month[i]))}
```

No Output

The code first defines a vector Month, which will be used to compare to the FBM vector. Next, we define a numeric vector of length 12 called FBMCounts. This vector will be used to store the counts for each month from the FBM vector. The for loop is then started with the variable i, ranging from 1 to 12. Note the parenthesis around the definition of i. A curly bracket, { , starts the code from the loop, which will be ended with a }. Reading from the inside of the loop, when $i=1$ Month[1] is “Jan”, which is the first element of the Month vector. The which command then checks each element of FBM to see if it is equal to “Jan” and the **length** command counts them. Note that ==, two equal signs, are a logical test to see if the elements are the same. Technically, the which function creates a vector with the location of “Jan” in FBM and in this case yields (14, 16, 29). The length function simply returns the length of the vector created by which. We then set FBMCounts[1] = 3 or the number of times “Jan” is listed in FBM. The for loop then does this for each value of i from 1 to 12. It was not shown here, but it is good practice to include a **print** statement within the for loop to check to see if the loop is working as expected, provided the loop isn’t too long. For example before the last } and on a new line include **print(paste("Count for Month", i, "is", FBMCounts[i]))**. This line will print information to the screen for each value of i with the **paste** function creating one line of the text, what is in quotes, and the variables. Give it a try.

We now proceed with almost the same code for the male data. We first define a numeric vector of length 12 to store the results, MBMCounts, and then run a for loop. Since the male data is already integers we don’t need the Month vector. For example, 1 already represents Jan and so the which function simply needs to identify when $MBM==1$.

R Code

```
> MBMCounts = vector(mode="numeric", length=12)
> for (i in c(1:12)){MBMCounts[i]=length(which(MBM==i))}
```

No Output

3.4 Boxplot with a Stripchart

Boxplots are typically used with large continuous data sets and so for this example we will generate data from a normal distributions using **rnorm**. Our

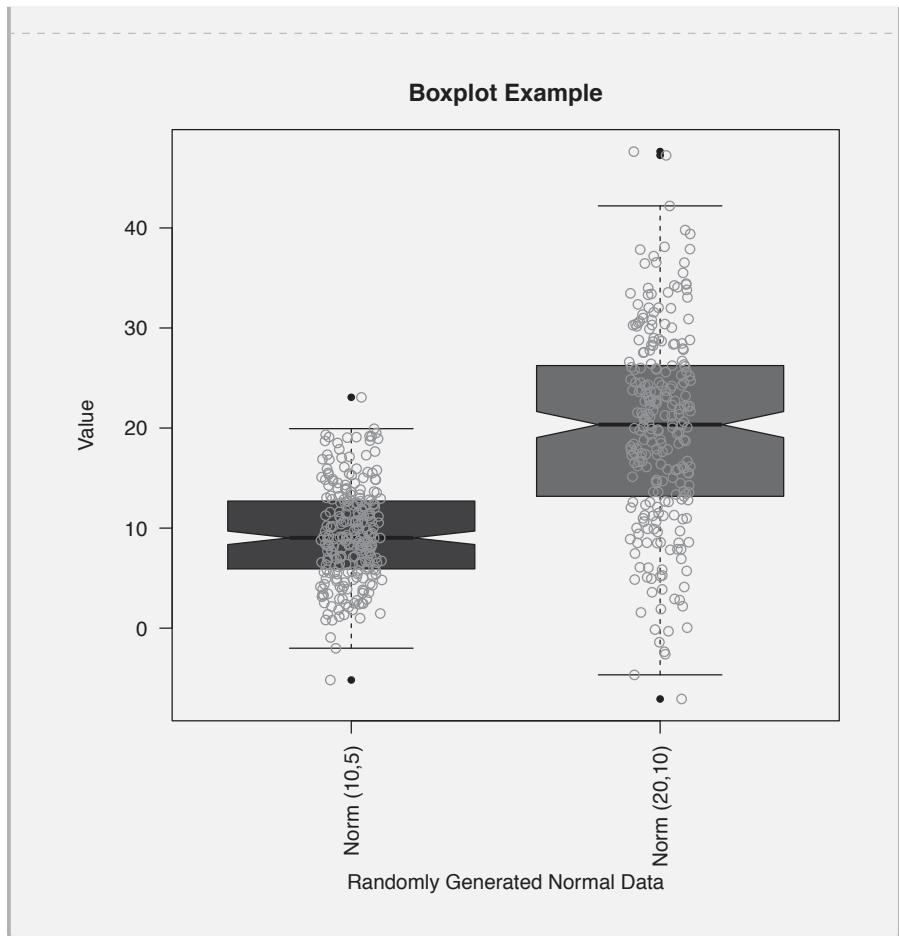
example here will have two boxplots and a stripchart overlay. As we explain the code we will point out how to adjust some features.

The first two lines generates two data sets of size 250 from a normal distribution. The first has mean 10 and standard deviation 5 and the second 20 and 10, respectively. The third line combines the data into one variable called Data. The **par** command sets the number of lines for the margins in the order bottom, left, top, and right. This was done because we needed more space on the bottom to align our labels vertically.

The **boxplot** function creates the boxplot. The first option is the data, which consists of two lists and called Data. You can simply input TenNormFive, TwentyNormTen, but the stripchart used next needed the data in the form of one variable and so we also used it for the boxplot. The pch=20 command sets the style of the outliers to solid circles. This is done to make sure they are distinctive from the actual data graphed by the stripchart. The notches in the boxplot are set with notch=TRUE. If you prefer rectangles without the notch either set notch to FALSE or remove the notch options since the default is without the notch. The next three options create the *x*-axis labels with las=2 placing them perpendicular (0 or deleting las will place them parallel) the at command setting the locations of the labels, and names defining the labels. The boxes are colored blue and red with the col option. If you prefer horizontal boxplots include horizontal=TRUE. A title is created with the main option and the *y*-axis is labelled with ylab. We did not use xlab within the **boxplot** function because it placed the *x*-axis caption within the data labels. This is fixed with the **mtext** command in the last line of the code, with side=1 placing the text at the bottom and line=6 placing the caption on the sixth line below the axis. Note that with the **mtext** command side set to 2, 3, or 4 will place text on the left, top, and right, respectively.

R Code

```
> TenNormFive=rnorm(250,mean=10,sd=5)
> TwentyNormTen=rnorm(250,mean=20,sd=10)
> Data=list(TenNormFive,TwentyNormTen)
> par(mar=c(8,5,3,3))
> boxplot(Data,pch=20,notch=TRUE,las=2,at=c(1,2),
+ names=c("Norm(10,5)","Norm(20,10)"),col=c("blue",
+ "red"),main="Boxplot Example",ylab="Value")
> stripchart(Data,vertical=TRUE,method="jitter",
+ jitter=0.1,add=TRUE,pch=1,col=c("gray50","gray50"))
> mtext("Randomly Generated Normal Data",side=1,line=6)
```



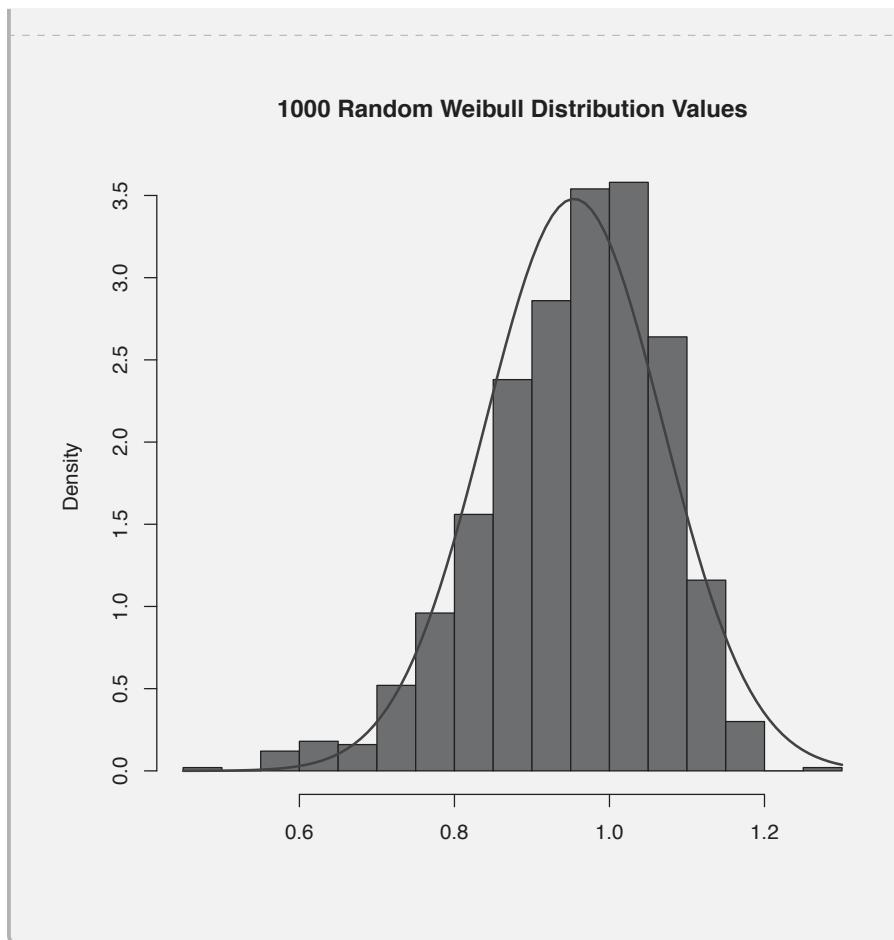
The first option in **stripchart** passes the Data, our two normal data sets, to the command. We set the orientation with vertical=TRUE, which can be removed if you prefer horizontal boxplots above. The next two commands create the plot with the data set spaced out a bit with method = “jitter” and jitter=0.1. A larger jitter value spaces the data out more. The stripchart has to be added to the boxplot chart with add=TRUE. The last two options pch=1 and col = c(“gray50”, “gray50”), sets the dots to open circles; there are over 25 options, and colors both sets with the same gray tone.

3.5 Histogram

Our last plot is a histogram. To begin we need data to graph. In this case we generate 1000 values from a Weibull Distribution and set the variable to WeibullData. We chose a distribution that is not normal since we will overlay the graph with a normal density as part of the example. We can easily change the data to normal by using **rnorm(n,mean,sd)**. The second line creates the histogram with the first object the data variable. The second options sets the number of cells, but note that it is a guideline for R and not strictly enforced. We can set breaks to a vector of cutoffs for more control, and this is left as an exercise. The boxes are colored red, and if col is left out the boxes are not colored. Instead of counts we chose percentages with prob=TRUE. We added a title with main and set the x axis label blank with empty quotes.

R Code

```
> WeibullData=rweibull(1000, shape=10, scale = 1)
> hist(WeibullData,breaks=15,col="red",prob=TRUE,
main="1000 Random Weibull Distribution Values",xlab="")
> m=mean(WeibullData)
> s=sqrt(var(WeibullData))
> curve(dnorm(x,m,s),col="blue",lwd=2,add=TRUE)
```



In order to superimpose a normal density we first calculate the **mean** and standard deviation of the data and set those values to m and s, respectively. We use **curve** and graph the normal density with mean m and standard deviation s with **dnorm**. We color it blue and use a line width of 2. Lastly, add=TRUE adds this to the current plot instead of creating a new plot.

Code Review for the Graphing Chapter

abline(a=,b=,h=,v=) adds lines to a graph. The value of a and b are an intercept and slope. The values set to h are y values for horizontal lines. The values set to v are x values for vertical lines.

axis(i, at=, label=) adds tick marks to the graph at the location given by at with labels given by labels. The values of i are 1, 2, 3, and 4, for the bottom, left, top, and right axis, respectively.

barplot(counts) creates a bar chart of data that is tabulated and stored in counts. If counts is a matrix then a stacked or side-by-side bar chart will be created.

box(option) draws a box in the current graph with option set to “plot”, “figure”, “inner”, and “outer”.

boxplot(data) creates side-by-side box plots where data is a list of vectors of data.

curve(f,a,b) graphs the function f from a to b . When the argument add=TRUE is included the function is added to the current graph. There are numerous options. Execute ?curve for optional graphing parameters.

dev.new() opens a new graphics device.

expression() is a text command that creates mathematical expressions with mostly standard syntax. For example, use x^2 for x^2 and pi for π , but $x[i]$ for x_i .

function(x_1, x_2, \dots, x_n) { expression } defines a function of x_1, x_2, \dots, x_n with expression. The function will return the last line of expression and for clarity use return().

grid(NULL,NULL) adds grid lines at the x and y tick marks of a plot. NULL can be replaced with, n, for the number of grid lines or NA for no grid lines.

hist(data) creates a histogram of the data. Two key options are freq, the default is TRUE, and breaks, to set the breakpoints between cells.

legend() adds a legend to a plot. There are numerous options. Run ?legend to learn more.

mean(data) returns the mean of the data set.

mtext(text, side=, line=) places the text in the margins on side 1 through 4 on the given line.

par(“usr”) returns a vector of four elements that are the values of the left, right, bottom, and top of the graphing frame. In other words the first and third elements represent the bottom left coordinate, while the second and fourth elements represent the top right corner. The function par has many uses, in particular with mflow to set a grid for graphs. Run ?par to learn more.

paste() will concatenate characters, in quotes, and variables, where each term is separated by a comma in paste. The default is to place a space between terms and the option `sep=""`, empty quotes, places no space between terms.

pie(counts) creates a pie chart of data that is tabulated and stored in counts.

plot(x,y) plots the points given by *x* and *y*, which may be vectors.

Plot can interpret and plot a variety of objects beyond a set of *x* and *y* values. There are also numerous graphing options that can be used in plot.

points(x,y) adds the points given by *x* and *y*, which may be vectors, to the current graph.

replace(v,where,what) replaces terms in the vector *v* identified by where with what. The second option where can be a list of indices or a logical statement to identify location in *v* to be replaced.

segments(x_1, y_1, x_2, y_2) draws a line segment from (x_1, y_1) to (x_2, y_2) .

stripchart(data) creates side-by-side strip charts where data is a list of vectors of data. If `add=TRUE` strip chart can be overlayed on box plots.

text(a,b,"text") places text at the coordinate (a, b) .

windows() opens a graphics device, with options for the dimensions of the window.

Selected Graphing Options

adj tells R how to justify text to its location with 0 left/bottom, 0.5 center, and 1 right/top. It can be set as one value or `c(x,y)` if the justification is different in *x* and *y*.

cex scales the size of an object such as the plot points and has additional specifications such as `cex.axis`, `cex.lab`, and `cex.main`.

col sets the color of an object and has additional specifications such as `col.axis`, `col.lab`, and `col.main`.

font sets the type of font with 1=plain, 2=bold, 3=italic, and 4=bold italic. Font has additional specifications such as `font.axis`, `font.lab`, and `font.main`.

las orients text either parallel, =0, to the axis or side or perpendicular, =2.

lty defines the type of line with 1 being solid and 2-6 various dashed lines.

lwd scales the line width.

pch is the plot symbol with options from 1 through 25, plus a few special symbols such as %, #, and +.

side is set to 1, 2, 3, or 4 to place objects in the bottom, left, top, and right margins.

type is the way the points are plotted with common types of p=points, l=lines, and 0=points with lines.

3.6 Exercises

1. Use R to create a jpg or pdf file of your initials. Be creative in your design to demonstrate that you understand different options to control color, thickness, style, etc. You must also use par("usr") to place text or an object in your design.
2. Change the first example in [Section 3.1](#), Graphing Functions, from using `plot()` to using `curve()`.
3. Create a graph to illustrate the limit definition of the derivative at the point $x = 1$ to the function $f(x) = x^2 + 1$. Include three secant lines and the tangent line. Annotate the graph to define each object on the graph, including the values of h used for each secant line.
4. In the bar chart example, add a third category to the data, call it Martians, and add a third bar in the two bar charts.
5. Create a combined boxplot and stripchart graph with three different distributions to demonstrate skewness and to explain the meaning of quartiles. Consider using normal, exponential (`rexp(n, rate = m)`), and Poisson (`rpois(n, lambda=m)`). Choose parameters that create a readable graph and choose smaller data sets so they can be counted on the graph.
6. Recreate the histogram example using your own specified break-points with `break=c(x1, x2, ..., x1)`.
7. Create graph paper. Hint: `pch=NA` in `plot` will not plot the points,

`abline(h=list1, v=list2)` will create horizontal and vertical lines at the list points, `seq(a,b,dx)` will be helpful, and `lty` might be useful.

4

Polynomials

The PolynomF package is designed in such a way that it will do almost anything we might want to do with a polynomial. We highlight this package by demonstrating basic polynomial operations, calculus with polynomials, and working with a list of polynomials. Using the tools of the PolynomF package we explore the roots of degree-three polynomials, create Pascal's triangle and a Taylor polynomial, and test two Legendre polynomial identities. Note that we open the PolynomF in the first example; it is needed for all examples in this chapter.

4.1 Basic Polynomial Operations

We begin this section by defining polynomials by its roots, by its coefficients, and algebraically. The PolynomF package is opened with **library(PolynomF)** and is necessary for all the code that follows. The polynomial **p1** is defined by its roots with **poly.from.zeros**, which returns the monic polynomial with the given roots. The colon command here, **-2:4**, provides the integers **-2, -1, 0, 1, 2, 3** and **4**, which are the roots of p1. The polynomial p1 can be evaluated so that, for example, **p1(1)=0** and **p1(10)=3991680**.

R Code

```
> library(PolynomF)
> p1=poly.from.zeros(-2:4)
> p1
48*x - 28*x^2 - 56*x^3 + 35*x^4 + 7*x^5 - 7*x^6 + x^7
```

Another way to list roots for a polynomial is with **c**, the combine operator. The polynomial **p2** is defined as the monic polynomial with roots **-1** and **1**.

R Code

```
> p2=poly.from.zeros(c(-1,1))
> p2
```

-1 + x²

Polynomials can be defined based on their coefficients with the **polynom** function. The coefficients are listed with **c** in increasing order of the degree. This is demonstrated with polynomial **p3**. Our first three examples used integer roots or integer coefficients, but any real numbers may be used.

R Code

```
> p3=polynom(c(1,2,0,4))
> p3
```

1 + 2*x + 4*x³

Our third method of defining a polynomial is algebraically. We first define **x** as a polynomial object with **x = polynom()**. After that the polynomial can be constructed with standard operations as with polynomial **p4**.

R Code

```
> x=polynom()
> p4=5*(x-4)^3 + 10*x^2 - 5
> p4
```

-325 + 240*x - 50*x² + 5*x³

We can create random polynomials by using either **poly.from.zeros** or **polynom** as illustrated in our next two examples. We first define a vector of five randomly selected integers from 0 through 10 with replacement by using **sample** and setting the result to **rand.int**. In this case **rand.int** is the vector of 1, 10, 3, 9, and 8, and results will vary since it is random. At this point **rand.int** can be inserted into **poly.from.zeros** or **polynom** as seen in defining polynomials **p5** and **p6**.

R Code

```
> rand.int=sample(0:10,5,replace=TRUE)
> p5=poly.from.zeros(rand.int)
> p5
```

-2160 + 3606*x - 1769*x² + 353*x³ - 31*x⁴ + x⁵

R Code

```
> p6=polynom(rand.int)
> p6
9 + 10*x + 3*x^2 + 8*x^3 + x^4
```

The **solve** function is used to find the roots of polynomials. The roots of **p6** are given with **solve(p6)**, whereas solving the equation **p6=9** is achieved with **solve(p6,9)**. The **plot** function recognizes polynomial objects and will choose a window that includes maximum and minimum points of the polynomial as seen with **plot(p6)**.

R Code

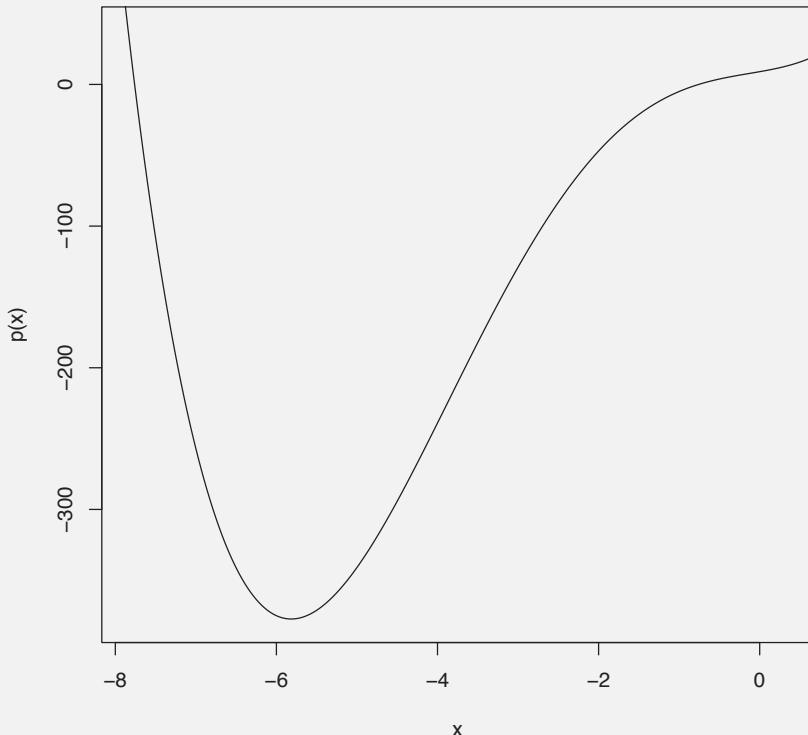
```
> solve(p6)
[1] -7.7602095+0.000000i -0.7574089+0.000000i
0.2588092-1.210059i 0.2588092+1.210059i
```

R Code

```
> solve(p6,9)
[1] -7.779605+0.000000i -0.110198-1.128392i
-0.110198+1.128392i 0.000000+0.000000i
```

R Code

```
> plot(p6)
```



Polynomial arithmetic is executed as expected so that **$2*p1$** , **$p1/3$** , **$p2*p3$** , **$p4^2$** (see example), and **$p5+p6$** return the appropriate polynomials. We note that the functions **round**, **signif**, **floor**, **ceiling**, and **trunc** operate on the coefficients as seen in **round(p1/3, 2)**.

R Code

```
> p4^2
```

```
105625-156000*x+90100*x^2-27250*x^3+4900*x^4-500*x^5+
25*x^6
```

R Code

```
> round(p1/3,2)

16*x-9.33*x^2-18.67*x^3+11.67*x^4+2.33*x^5-2.33*x^6+
0.33*x^7
```

The one operation that requires special attention is division. For comparison we note that $7/3$, $7\%/3$, and $7\%\%3$ return 2.33333 , 2 , and 1 , respectively. In other words, $\%/\%$ returns the integer quotient, while $\%\%$ returns the remainder. With polynomials **p6/p2**, for example, returns the quotient as will **p6\%/\%p2**, while **p6\%\%p2** is the remainder. We can check our work with the logical query **p6\%/\%p2*p2+p6\%\%p2==p6** which does return true.

R Code

```
> p6/p2

4 + 8*x + x^2
```

R Code

```
> p6\%/\%p2

4 + 8*x + x^2
```

R Code

```
> p6\%\%p2

13 + 18*x
```

R Code

```
> p6\%/\%p2*p2+p6\%\%p2==p6

[1] TRUE
```

The **coef** function returns the list of coefficients of the polynomial as in **coef(p5)**. Since **coef** is a vector of values we can ask for a specific coefficient. For example, **coef((x-2)^20)[1]** returns the first coefficient of $(x-2)^{20}$, which is the constant term. On the other hand, **coef((x-2)^20)[21]** returns the last coefficient, which is the coefficient of x^{20} . In general, **coef(p)[i]** returns the coefficient of x^{i-1} .

R Code

```
> coef(p5)
[1] -2160  3606 -1769   353   -31      1
```

R Code

```
> coef((x-2)^20)[1]
[1] 1048576
```

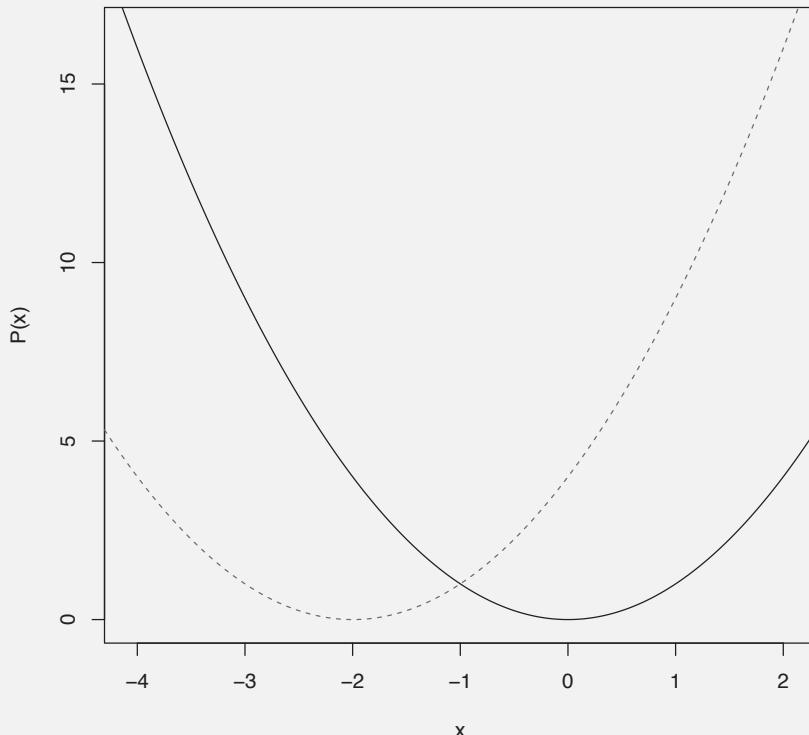
R Code

```
> coef((x-2)^20)[21]
[1] 1
```

The final basic command we will highlight here is **change.origin(p(x), x_0)**, which horizontally shifts the polynomial $p(x)$ to the left by x_0 . For example, we define **p7**= x^2 with **polynom(c(0,0,1))** and **p8=change.origin(p7,2)**. We graph both polynomials with the use of **polylist**, which is interpreted in **plot** as a list of polynomials to plot and selects different colors and line types for each polynomial.

R Code

```
> p7 = polynom(c(0,0,1))
> p8 = change.origin(p7,2)
> plot(polylist(p7,p8),xlim=c(-4,2))
```



4.2 The LCM and GCD of Polynomials

The PolynomF package has functions to calculate the LCM and GCD of polynomials. Our first example defines polynomials **p1** and **p2** with roots $(1,2,3)$ and $(3,4,5)$ respectively. **LCM(p1,p2)** returns the least common multiple of the two polynomials. To check our result, we calculate the roots with **solve(LCM(p1,p2))**. The greatest common divisor of **p1** and **p2** is returned with **GCD(p1,p2)**.

R Code

```
> p1=poly.from.zeros(c(1,2,3))
> p2=poly.from.zeros(c(3,4,5))
> LCM(p1,p2)
```

-120 + 274*x - 225*x^2 + 85*x^3 - 15*x^4 + x^5

R Code

```
> solve(LCM(p1,p2))
```

[1] 1 2 3 4 5

R Code

```
> GCD(p1,p2)
```

-3 + x

Here is a more interesting example. We create three polynomials by selecting roots randomly from -10 to 10. The polynomials **p5**, **p7**, and **p10**, have 5, 7, and 10 roots respectively. We then calculate the GCD and LCM of the three polynomials. Note, we didn't get a nonconstant GCD on the first try and results will vary since these are random. In this case the chance of non-constant GCD is a problem of selecting the same integer in each of rand.int.5, rand.int.7, and rand.int.10. An interesting question, left as an exercise, asks what is the chance of getting a nonconstant GCD if the polynomials are created with random coefficients instead of roots.

R Code

```
> rand.int.5=sample(-10:10,5,replace=TRUE)
> rand.int.7=sample(-10:10,7,replace=TRUE)
> rand.int.10=sample(-10:10,10,replace=TRUE)
> p5=poly.from.zeros(rand.int.5)
> p7=poly.from.zeros(rand.int.7)
> p10=poly.from.zeros(rand.int.10)
> GCD(p5,p7,p10)
```

-70 + 3*x + x^2

R Code

```
> LCM(p5,p7,p10)

-800150400000-1.34311e+11*x+1.0541e+12*x^2+
151815800000*x^3-281656200000*x^4-17934340000*x^5+
29151920000*x^6+404000500*x^7-1485008000*x^8+
27080760*x^9+40676880*x^10-1607378*x^11-607308*x^12+
32826*x^13+4626*x^14-297*x^15-14*x^16+x^17
```

4.3 Illustrating Roots of a Degree-Three Polynomial

Complex roots of polynomials with real coefficients occur in conjugate pairs. This example illustrates the point by plotting the real and imaginary parts of complex roots and the real roots of 50 polynomials with randomly chosen coefficients from -100 to 100 . The red squares represent the imaginary part of a complex root, the black triangles represent the real part of an imaginary root, and the blue circles represent the value of a real root. The symmetry around 0 is evident for the red squares. There is never more than one black triangle and always either one or three blue circles.

The code begins by setting the margins to 3, 3, 2, and 2 lines starting at the bottom and proceeding clockwise. The **plot** function creates an empty graph, type="n", with an x -axis from -5 to 5 and a y -axis from 0 to 50 . Labels for the axis are blank due to ylab=" " and xlab=" ". The first use of **abline** adds a vertical line at $x = 0$ with a line width of 2. The second **abline** creates 50 horizontal lines, one at each of 1 through 50, using seq(1,50,1) to return a sequence of numbers from 1 to 50 with steps of 1. Each line is colored with gray70 and the type of line is dashed with lty=2.

We begin a for loop with i ranging from 1 to 50. We set rand.int to a vector of 4 integers randomly selected from -100 to 100 with replacement. These integers will be the coefficients for the polynomial p in the next line. The roots of p are found with **solve(p)** and set to the variable roots.

Within the loop on i, another for loop is started with j ranging from 1 to 3. Each polynomial has 3 roots and this for loop will graph each of them for the polynomial p. The if statement checks to see if the jth root is real. If it is real then **Im(roots[j]==0** will return TRUE. If it is real then we plot the point using **points** with an x value of **Re(roots[j])** and a y value of i. The point character is 16, a circle, and it is colored blue. Note that the root is real but the output will still include a 0i and so we need to use **Re(roots[j])** instead of simply **roots[j]**. If the root is not real then the second part of the if statement, preceded by else, will be executed. In this case the imaginary part,

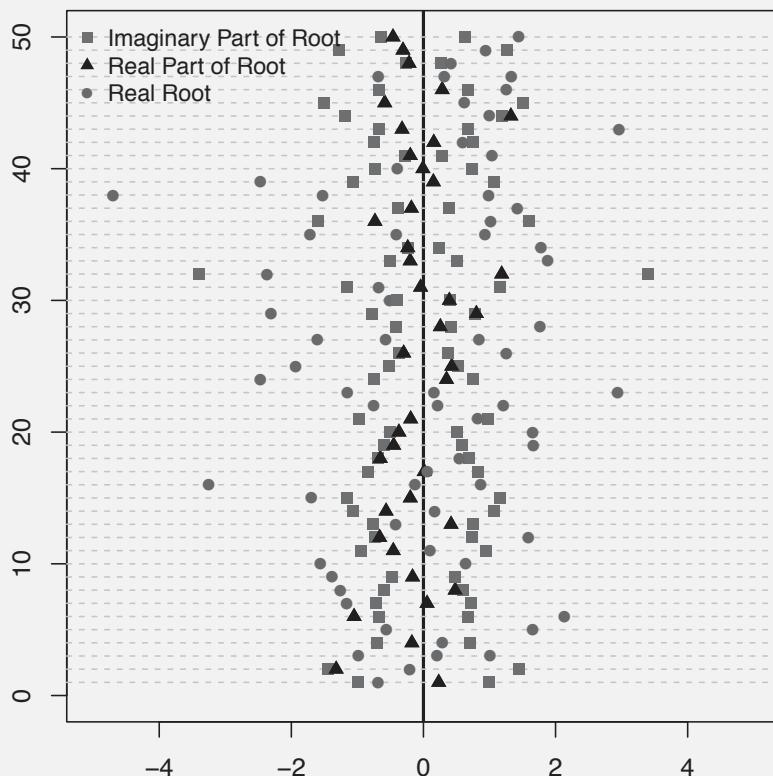
Im(roots[j]), and real part, **Re(roots[j])**, are each plotted with the **points** function with a *y*-value of *i*. The point characters are 15 and 17 for a square and triangle respectively. The points are colored red and black.

The next line has three right curly brackets. The first ends the else part of the if statement. Note that the first part of the if statement was enclosed in curly brackets. The second ends the for loop for *j*, while the third ends the for loop for *i*. What is executed in a for loop is enclosed in curly brackets.

Our last line creates the legend. It is placed in the top left. The next three inputs list the text, the point characters, and the colors. The entries of each are order dependent, meaning the first entries will create the first line of the legend and the same for the second and third. The legend is scaled to 90% with *cex*=0.9 and the box around the legend is removed with *bty*="n".

R Code

```
> par(mar=c(3,3,2,2))
> plot(0,0,type="n",xlim=c(-5,5),ylim=c(0,50),ylab="",xlab="")
> abline(v=0,lwd=2)
> abline(h=seq(1,50,1),col="gray70",lty=2)
> for(i in 1:50){
+ rand.int=sample(-100:100,4,replace=TRUE)
+ p=polynom(rand.int)
+ roots=solve(p)
+ for(j in 1:3){
+ if (Im(roots[j])==0){
+ points(Re(roots[j]),i,pch=16,col="blue")}
+ else {points(Im(roots[j]),i,pch=15,col="red")}
+ points(Re(roots[j]),i,pch=17,col="black")
+ }}}
> legend("topleft",c("Imaginary Part of Root",
"Real Part of Root","Real Root"),pch=c(15,17,16),
col=c("red","black","blue"),cex=0.9,bty="n")
```

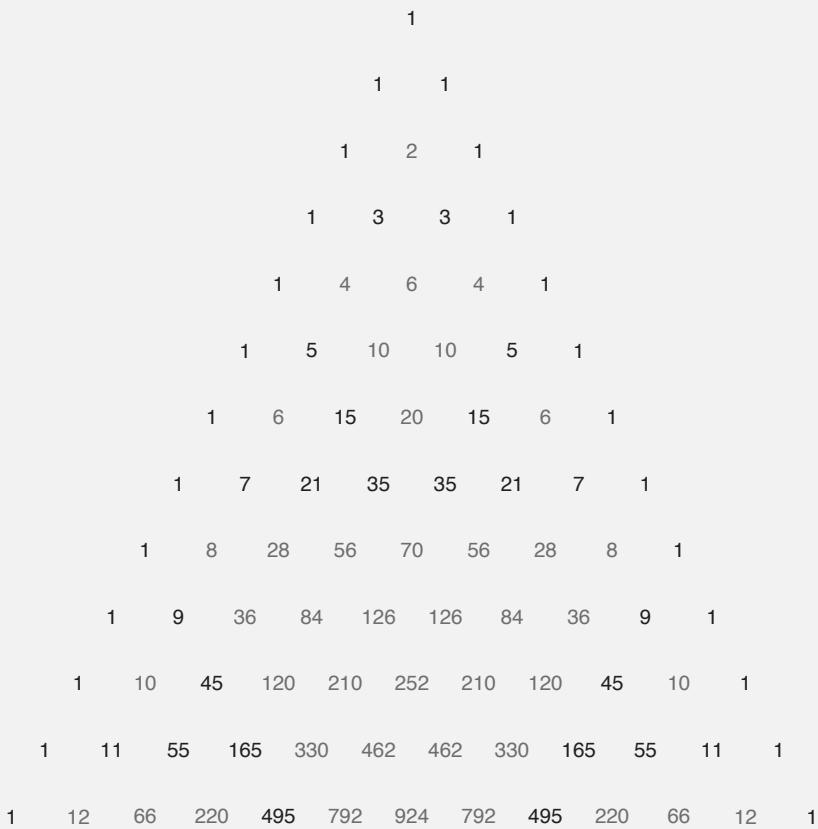


4.4 Creating Pascal's Triangle with Polynomial Coefficients

The binomial coefficient $\binom{n}{k}$ is the coefficient of the x^k term of $(1+x)^n$. We use this fact and the **coef** function to create Pascal's triangle. We begin by defining the variable n, which will be the highest power of $(1+x)^n$ or the number of rows plus 1. We set the margins of the graph frame to 0 on all sides. We let x be a polynomial with **x=polynom()**. We create an empty graph frame with **plot**, type="n", where x ranges from $-n$ to n and y from $-n$ to 0. Axes are suppressed with axes=FALSE and labels are empty with **ylab=""** and **xlab=""**.

R Code

```
> n=12
> par(mar=c(0,0,0,0))
> x=polynom()
> plot(0,0,type="n",xlim=c(-n,n),ylim=c(-n,0),
axes=FALSE,ylab="",xlab="")
> for(i in 0:n) p=(1+x)^i
+ for(j in 0:i){
+ if (coef(p)[j+1]%%2==1){color="black"} else {color=
"red"}+ text(2*j-i,-i,label=coef(p)[j+1],col=color)
+ }}
```



We begin our first for loop with i from 0 to n . This will be the power of $(1 + x)$ and we define $p=(1+x)^i$. Our inner for loop ranges j from 0 to i and will be used to select the coefficients from the polynomial p . We will color our values red if they are even and black if they are odd. This is achieved with the

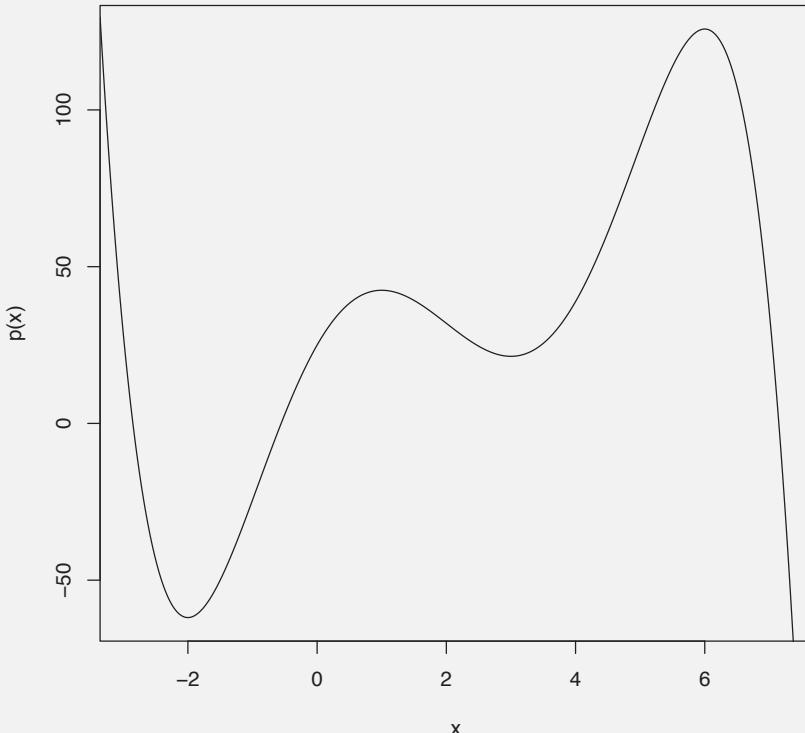
if statement, where **coef(p)[j+1]%%2** is the value of the $j+1$ coefficient of **p** modulo 2. If it is 1, then color is set to “black”; otherwise color is set to “red”. Outside the if then statement, we place the value on the graph using the **text** function. The value is placed at $(2*j-i, -i)$ and given by **label=coef(p)[j+1])** with col set to color. Note how the coefficients are spaced out by using $2*j-1$ as the x -value. Note that both parts of the if then else statement are enclosed in curly brackets as well as the for statements, which account for the two right curly brackets on the last line.

4.5 Calculus with Polynomials

To illustrate the calculus tools available with the PolynomF package we will create a polynomial with local minima at $x = -2$ and $x = 3$, local maxima at $x = 1$ and $x = 6$, and y -intercept of 25. Along the way we will analyze our function and plot it and its first two derivatives. We begin by defining **p** to be a polynomial with roots at $-2, 1, 3$, and 6 , multiplied by -1 . In the next line we let **P** by the integral of **p** plus 25. The **integral** functions sets the constant to 0 and so we add 25 for the desired y intercept. At this point we **plot(P)** to see that we have the desired function. Note that multiplying by -1 could have been done in either of the first two lines of code to achieve the desired results. We also note that **integral** can be used for calculating definite integrals as in the next example, **integral(p,limits=c(-2,6))**.

R Code

```
> p=-1*poly.from.zeros(c(-2,1,3,6))
> P=integral(p)+25
> plot(P)
```



R Code

```
> integral(p,limits=c(-2,6))
```

```
[1] 187.7333
```

What are the inflection points of **P**? We set **dp** to the derivative of **p** and then **solve(dp)**. We get the three inflection points expected based on the graph. On the other hand, **summary(P)** will provide the roots of $P(x)$, $P'(x)$, and $P''(x)$. Further, you can name the output of **summary** and access the specific values. We named the output **poly.info** and **names(poly.info)** lists the information stored as **poly.info**.

R Code

```
> dp=deriv(p)
> solve(dp)

[1] -0.9154759  2.0000000  4.9154759
```

R Code

```
> summary(P)

Summary information for:
25 + 36*x - 18*x^2 - 2.333333*x^3 + 2*x^4 - 0.2*x^5

Zeros:
[1] -2.8513831+0.000000i -0.5564269+0.000000i
     3.1307678-1.105889i
[4]  3.1307678+1.105889i  7.1462743+0.000000i

Stationary points:
[1] -2  1  3  6

Points of inflexion:
[1] -0.9154759  2.0000000  4.9154759
```

R Code

```
> poly.info=summary(P)
> names(poly.info)

[1] "zeros"      "stationaryPoints"    "inflexionPoints"
```

For example `poly.info$zeros` returns the roots of $P(x)$, while **Re(poly.info\$zeros)** returns the real part of the roots. Similarly, if we wanted only the imaginary part of the third root we can use **Im(poly.info=zeros[3])**.

R Code

```
> poly.info=zeros

[1] -2.8513831+0.000000i -0.5564269+0.000000i
     3.1307678-1.105889i
[4]  3.1307678+1.105889i  7.1462743+0.000000i
```

R Code

```
> Re(poly.info$zeros)

[1] -2.8513831 -0.5564269 3.1307678 3.1307678 7.1462743
```

R Code

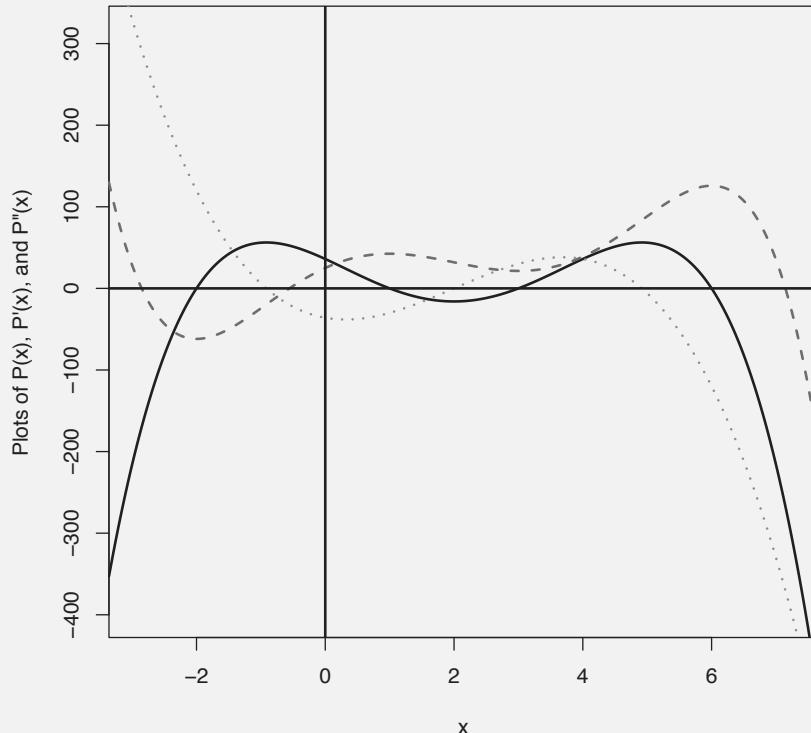
```
> Im(poly.info$zeros[3])

[1] -1.105889
```

Last, we will plot $P(x)$ and its two derivatives. This can be done quickly with **plot** and **polylist**. In this case, the window will be chosen to capture roots, critical points, and inflection points of the polynomials listed. It will also choose the first three colors of the default palette. If we want different colors then we have to redefine the current sessions palette by, for example, **palette(c("black", "blue", "red"))**. The line types are also fixed. If we want more control of the graph we could first plot **P** with **plot(P)**, and then uses **lines(p)** and **lines(dp)**, which will add **p** and **dp** to the graph of **P**. In each case we can use the various graphing options to control the look of the graph. In our example, we used **polylist**, labeled the y -axis and set all lines to width 2. We also added vertical and horizontal lines at 0, with width 2.

R Code

```
> plot(polylist(p,P,dp),ylab="Plots of P(x),P'(x),and
P''(x)",lwd=2)
> abline(v=0,h=0,lwd=2)
```



4.6 Taylor Polynomial of $\sin(x)$

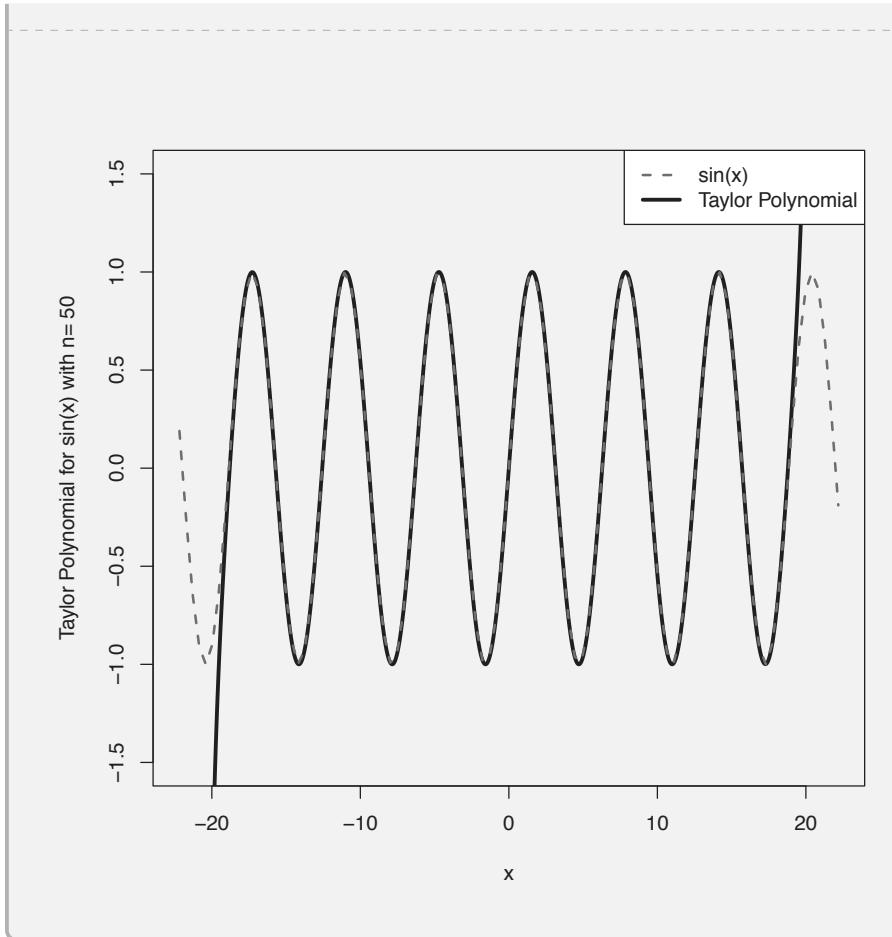
In this example we create the Taylor polynomial for $\sin(x)$ centered at 0 and graph both the Taylor polynomial and $\sin(x)$. To do this, we will use the Deriv package (see Chapter 6 for more on the Deriv package) as it allows us to calculate the i th derivative. We begin by opening this package. We set the variable power=50, which will be the power of the Taylor polynomial, and initiate the vector coef to be numeric of length power. This vector will store the coefficients of the Taylor polynomial that will be generated during the for loop. The for loop allows i to vary from 1 to power and begins with a left curly bracket. We define **d.sin** to be the i th derivative of $\sin(x)$ by using **Deriv**

and noting that `sin` is a built in function. The value in location `i` of the vector `coef` is set to `d.sin(0)/factorial(i)` or, in other words, $\sin^{(i)}(0)/i!$. At this point we end the for loop with a right brace. We now use **`polynom`** to define the Taylor polynomial with its coefficients. The first coefficient is `sin(0)` and so we concatenate `sin(0)` and the vector `coef` to get **`Taylor.poly.sin`**.

We create a graph by plotting **`Taylor.poly.sin`**. We set the range for the y -axis with `ylim`, set the line width to 0 with `lwd`, and label the y -axis with `ylab`. In labeling the axis we use **`paste`** to combine text, in quotes, with the variable power. We add $\sin(x)$ to the graph with **`curve`** by including the option `add=TRUE`. We use a line type of `lty=2`, which is dashed. The line width is 2 and it is colored red. We include a legend in the “topright”. The second option of legend concatenates the text for each line. The next three options are also in pairs setting the color, line type, and line width, respectively, to match with the text. We set the background color to white, `bg="white"`, to “paint over” the polynomial that extends into the box.

R Code

```
> library(Deriv)
> power=50
> coef=numeric(power)
> for (i in 1:power){
+ d.sin=Deriv(sin,nderiv=i)
+ coef[i]=d.sin(0)/factorial(i)}
> Taylor.poly.sin=polynom(c(sin(0),coef))
> plot(Taylor.poly.sin, ylim=c(-1.5,1.5),lwd=3,
ylab=paste("Taylor Polynomial for sin(x) with n=",
power))
> curve(sin,add=TRUE,lty=2,lwd=2,col="red")
> legend("topright",c("sin(x)","Taylor Polynomial"),
col=c("red","black"),lty=c(2,1),lwd=c(2,3),bg="white")
```



4.7 Legendre Polynomials

In this section we illustrate how to define polynomials with a recursive relationship, the Legendre polynomials, and check two known identities. We begin by letting x be a **polynom()** object so we can use it to define polynomials algebraically. We then start a **polylist** with the first two Legendre polynomials and let P represent the list. Once a **polylist()** has been initiated we can add to the list and we do that with a for loop. Our for loop allows n to range from 2 to 14. There is only one line in this for loop, enclosed in braces, and it is the recursive relationship for the Legendre polynomials. The polynomials are referenced in the list by P with double square brackets. We end this code by

exhibiting the 5th polynomial with $P[5]$. Note, if we use $P[[5]]$ it will list only the polynomial by removing List of polynomials. In general, double square brackets will return only the object which is why we used them in the for loop.

R Code

```
> x=polynom()
> P=polylist(x, 1/2*(3*x^2-1))
> for(n in 2:14){
+ P[[n+1]]=((2*n+1)*x*P[[n]]-n*P[[n-1]])/(n+1)}
> P[5]
```

List of polynomials:

```
[1]
1.875*x - 8.75*x^3 + 7.875*x^5
```

We have defined the first 15 Legendre polynomials and now we check the identity $P_n(-x) = (-1)^n P_n(x)$ with them. Again we have a for loop, this time with i from 1 to 15. In the loop we **print** the result of the logical statement, using $==$, of the identity. TRUE is returned 15 times, although we only show the first three lines.

R Code

```
> for (i in 1:15){
+ print(P[[i]](-x)==(-1)^i*P[[i]](x))}
```

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

```
:
```

We now check the identity $P'_n(1) = n(n + 1)/2$, although this time we return both sides of the identity in a question. Our for loop ranges i from 1 to 15. We set **p** to the derivative of the i th polynomial. Within **print** we use **paste** to concatenate text and variables, where **p(1)** is the derivative of $P[i]$ evaluated at 1. Within **paste** we set **sep=""**, empty quotes, which means spaces won't be added at the commas. We put the spaces within the quotes as desired.

R Code

```
> for(i in 1:15){
+ p=deriv(P[[i]])
+ print(paste("Is ", p(1),"=", i*(i+1)/2,"?", sep=""))}
```

```
[1] "Is 1=1?"  
[1] "Is 3=3?"  
[1] "Is 6=6?"  
[1] "Is 10=10?"  
[1] "Is 15=15?"  
[1] "Is 21=21?"  
[1] "Is 28=28?"  
[1] "Is 36=36?"  
[1] "Is 45=45?"  
[1] "Is 55=55?"  
[1] "Is 66=66?"  
[1] "Is 78=78?"  
[1] "Is 91=91?"  
[1] "Is 105=105?"  
[1] "Is 120=120?"
```

Code Review for the Polynomial Chapter

change.origin(p,h) returns the polynomial p horizontally shifted by -h.

coef(p) returns a list of the coefficients of p.

GCD(list) returns the GCD of the list of polynomials.

Im(z) returns the imaginary part of the number z.

integral(p) returns the integral of the polynomial p, with constant of integration set to 0.

LCM(list) returns the LCM of the list of polynomials.

poly.from.zeros(list) returns a polynomial with the zeroes given in the list.

polylist() defines a list of polynomials. It can be used with **plot()** to plot a list of polynomials.

polynom(list) returns a polynomial with the coefficients in the list, from smallest to largest power.

Re(z) returns the real part of the number z.

sample(list,n,replace=TRUE) returns n random numbers from the list with replacement.

solve(p) returns the roots of the polynomial p.

summary(p) returns the roots of the polynomial and its first and second derivative.

4.8 Exercises

1. Use the Pascal's triangle example and create the triangle but color the numbers based on remainder mod3.
2. Create a polynomial with inflection points at $-1, 2$, and 5 . What are the critical points? Graph the polynomial and its first two derivatives using `lines()`.
3. If the coefficients of two polynomials are randomly selected from -10 to 10 , what is the probability that the GCD is at least degree 1?
4. Use the Taylor Polynomial of $\text{Sin}(x)$ example and center the polynomial at π .
5. Use the Taylor Polynomial of $\text{Sin}(x)$ example and create a function with the input the power of the desired Taylor polynomial for $\text{Sin}(x)$ and returns the polynomial. Add an input for the center of the polynomial.
6. Create an example of the Taylor Polynomial of $\ln(x)$.
7. Choose a set of polynomials with a recursive relationship (e.g., Chebyshev, Gegenbauer, or Hermite), define them, and test two identities as we did here.

5

Sequences, Series, and Limits

R has a unique command for creating sequences, which will be used in this chapter. We will begin by considering $1/n$ near 0 and then move on to comparing $1/n$ and $1/n^2$ as $n \rightarrow \infty$. Derivative limits are investigated next and we finish the chapter with the Fibonacci sequence.

We begin by demonstrating the colon operator and how to create a sequence. We will create the sequence $1/(n^3 + 5)$ from $n = 39$ to $n = 42$, where we chose our values of n arbitrarily for the example and kept the range short so there is only one line of output. The code begins by avoiding scientific notation with **options(scipen=9)**, which keeps fixed notation unless scientific notation is needed with a power of 9 or larger, regardless of sign. The key to the code is $n=38:42$ which creates a sequence of numbers from 39 to 42 by steps of 1. The next line cubes each of these integers, adds 5 to all of them, and reassigns the sequence to n. In other words, operations to these sequences are applied to each value of the sequence. We take the reciprocals of each value with $n=1/n$ and then have them printed out with just n. We went through this with more steps than necessary as $n=1/((39:42)^3+5)$ would have completed this in one step.

R Code

```
> options(scipen=9)
> n=39:42
> n=n^3+5
> n=1/n
> n
```

```
[1] 0.00001822257 0.00001685658 0.00001562378
0.00001450831
```

The colon operator is a useful tool. Formally, a:b will create a sequence of number in steps of 1, starting at a and not exceeding b. For example $\pi:5$, will create a sequence of two numbers 3.14 and 4.14, rounded to two decimal places. The colon operator will step down as well as up and negative numbers are acceptable. For example 5:3 returns 5, 4, 3, and -10:-12 returns -12, -11, -10. Basic arithmetic operations applied to the sequence are applied to each value, such as cubed and adding 5 in our example. You can also add, multiply, divide, etc. two sequences, but there is a useful catch. If one sequence is shorter

than the other the shorter sequence will repeat as needed. Review the next example to make sure you understand the output, although we will not use this type of operation in this chapter. If the longer sequence is not a multiple of the shorter sequence you will receive a warning message. Finally, we note that you can find the **min**, **max**, **mean**, **median**, **sum**, **cumsum**, etc. by, for example **min(a)** or **min(1:9)**.

R Code

```
> a=1:9
> b=1:3
> a+b
> a^b

[1]  2  4  6  5  7  9  8 10 12
[1]   1    4   27    4   25  216    7   64  729
```

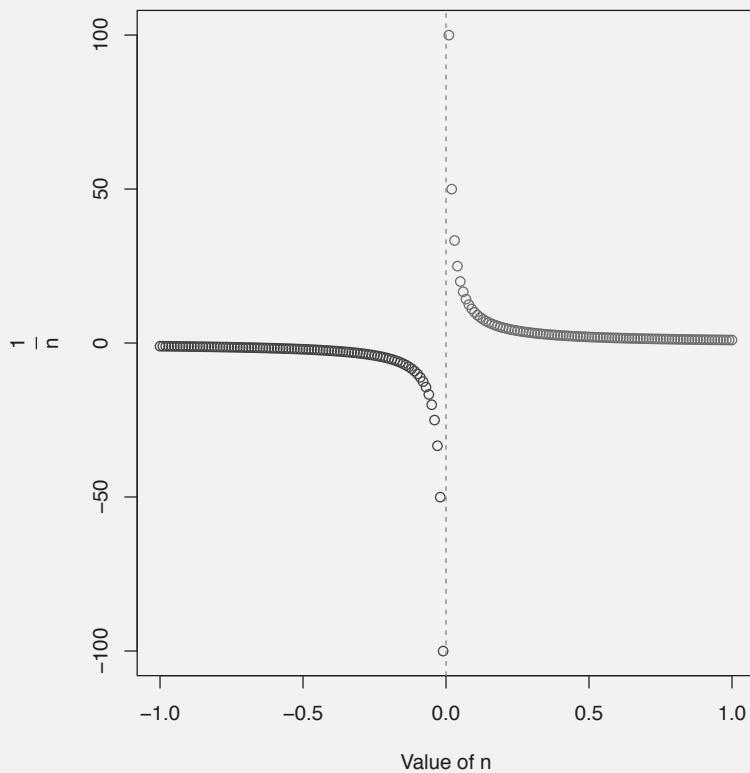
5.1 Sequences and Series

The colon operator allows us to investigate sequences because it is easy to define input values, and, as we will see, we can define our inputs from the left and right so that they may be colored differently in graphs. Our first example considers $1/n$ near 0 and we view $1/n$ as a function where as in the following example we create the sequence as we did in the introduction example. There are always multiple ways to achieve goals in R.

Our example begins by defining input values from the right of 0, nr, and from the left of 0, nl. The values of nr are from 1 down to 1/100 where 100:1 generates the sequence of integers from 100 down to 1 and then each value is divided by 100. Similarly nl is a sequence of values from -1 up to -1/100. The function **f** is defined to be $1/n$ in the next line. The **par** command creates 4,6,4,4 lines around the graphing frame. We set this because the y -axis caption needed extra space. We first plot with the positive input values. In this case a scatter plot is being produced with nr as the x values and **f(nr)** as the y values. Briefly, our **plot** command colors the point red, sets the x and y range with **xlim** and **ylim**, and creates axis captions with **xlab** and **ylab**. The expression command for **ylab** creates the fraction $1/n$. The **points** command adds the negative side of the graph and colors those points blue. For reference, **abline** adds a versicle line a 0, with dashed line type, and colored by gray40.

R Code

```
> nr=100:1/100
> nl=-100:-1/100
> f=function(n){1/n}
> par(mar=c(4,6,4,4))
> plot(nr,f(nr),col="red",xlim=c(-1,1),ylim=c(-100,100),
xlab="Value of n",ylab=expression(frac(1,n)))
> points(nl,f(nl),col="blue")
> abline(v=0,lty=2,col="gray40")
```



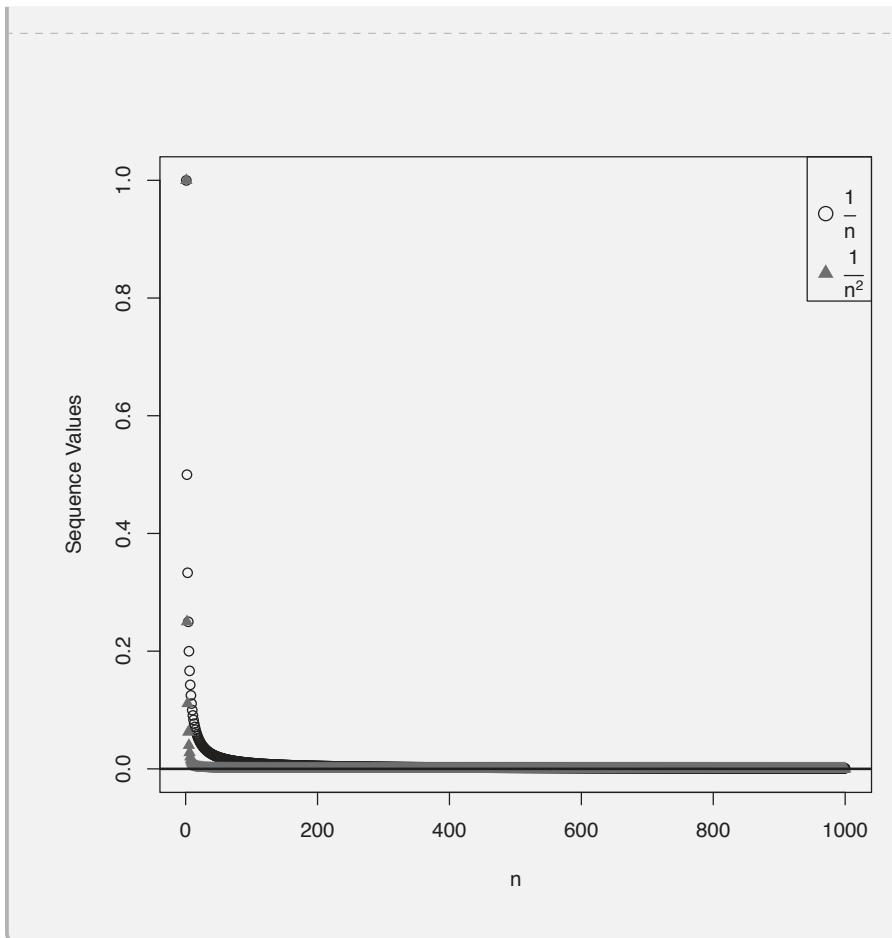
We now move to exploring $1/n$ for “large” values of n while comparing it to $1/n^2$. We begin the code by defining the lower and upper values that will be used by setting them to l and u . If we later decide to use different lower and upper values for our sequence this is the only part of the code that will need to be changed. Next, $n1$ creates a sequence from l to 1000 and then our first sequence, $s1$, is created as $1/n1$. Similarly our second sequence is created

by squaring the values from 1 to 1000 and setting $s2=1/n2$. Our x -axis range is set by l and u , but then our y -axis range is set by $s1$ and $s2$. Our next two lines will be used to set our $ylim$ in `plot`. The command `min` finds the minimum of a sequence and we first find the minimum of $s1$ and $s2$ separately and then set $miny$ to be the smallest of these two values. Similarly, we find $maxy$. We are assuming here that since we are exploring these sequences we don't know which is smaller. Also, this code allows us to easily change $s1$ and $s2$ for other examples.

We now graph $s1$ by using `plot`. Again, we are technically creating a scatter plot so we need to plot $n1$ and $s1$. We set the point type to 1. Note how our $xlim$ and $ylim$ take advantage of the variables l , u , $miny$, and $maxy$, created earlier in the code. We then add the scatter plot for the second sequence with `points` and note that we use a different point type, $pch=17$ which is a solid triangle, and color it red to clearly differentiate it from the first sequence. We also added a legend for clarity and a thicker horizontal line at $y = 0$.

R Code

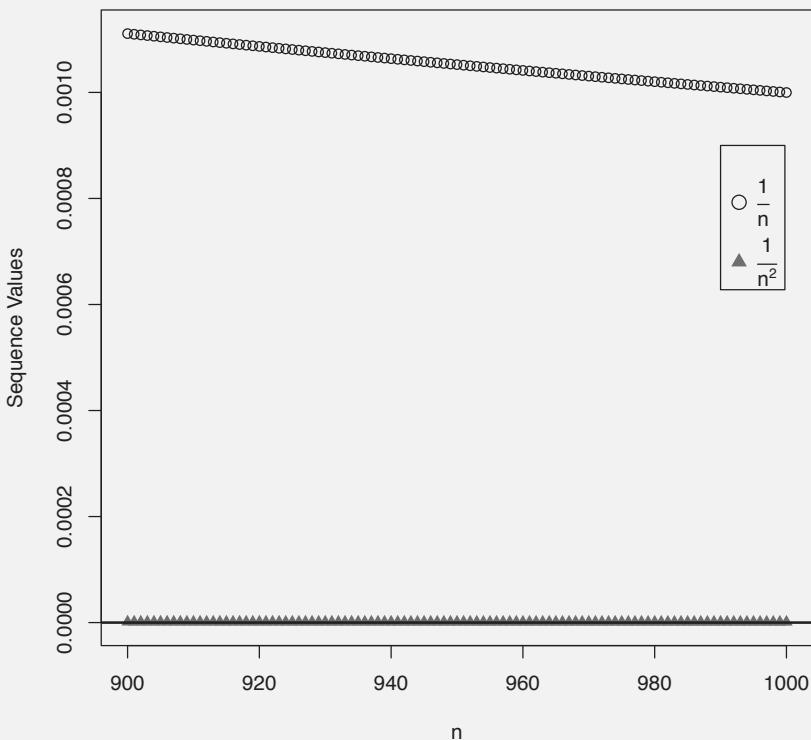
```
> l=1
> u=1000
> n1=l:u
> s1=1/n1
> n2=(l:u)^2
> s2=1/n2
> miny=min(min(s1),min(s2))
> maxy=max(max(s1),max(s2))
> plot(n1,s1,pch=1,xlim=c(l,u),ylim=c(miny,maxy),
  xlab="n",ylab="Sequence Values")
> points(n1,s2,pch=17,col="red")
> legend("topright",c(expression(frac(1,n)),
  expression(frac(1,n^2))),pch=c(1,17),pt.cex=c(1.5,1.25),
  col=c("black","red"),y.intersp=1.25)
> abline(h=0,lwd=2)
```



As we can see this graph really doesn't help us understand the differences, if any, between $1/n$ and $1/n^2$. We will next zoom in on our graph to the section from $x = 900$ to $x = 1000$. Our code will be almost identical to the previous except, for example, instead of considering all of s1 we will use only the values from 900 to 1000 by using $s1[c(l2:u2)]$, where the square brackets set the range of the desired sequence. Our code begins by setting two new lower and upper values with l2 and u2, which are used to select the range of the sequences. We again find min and max y values set to miny2 and maxy2. We restrict scientific notation with **options(scipen=9)**. The **plot** and **points** commands are similar to those used in the previous example. The legend was moved slightly because topright was in the way of the graph so we chose $x = 990$ and $y = 0.0009$ coordinates for the location of the legend. We now have a graph that is a portion of the first graph and one that is more useful in understanding the differences between the two sequences.

R Code

```
> l2=900
> u2=1000
> miny2=min(min(s1[c(12:u2)]),min(s2[c(12:u2)]))
> maxy2=max(max(s1[c(12:u2)]),min(s2[c(12:u2)]))
> options(scipen=9)
> plot(n1[c(12:u2)],s1[c(12:u2)],pch=1,xlim=c(12,u2),
+ ylim=c(miny2,maxy2),xlab="n",ylab="Sequence Values")
> points(n1[c(12:u2)],s2[c(12:u2)],pch=17,col="red")
> legend(990,0.0009,c(expression(frac(1,n)),
+ expression(frac(1,n^2))),pch=c(1,17),pt.cex=c(1.5,1.25),
+ col=c("black","red"),y.intersp=1.25)
> abline(h=0,lwd=2)
```



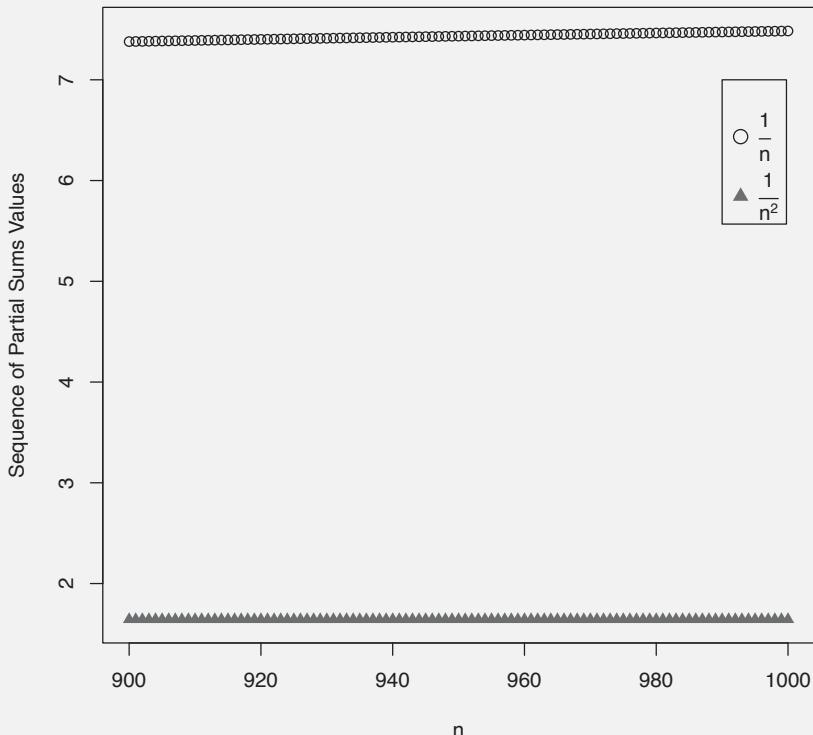
We know that both of these sequences converge to 0 as n gets large, but their convergence is not the same. We now turn our attention to the sum of

these sequences to investigate the questions of whether or not the difference in the convergence to 0 of the sequences impacts their corresponding series, $\sum_0^{\infty} 1/n$ and $\sum_0^{\infty} 1/n^2$. The function **cumsum** allows us to easily generate the sequence of partial sums. Our next graph will have the two sequences of partial sums graphed from 900 to 1000 since that seemed more useful in our previous example (this choice is questioned in an exercise).

We begin by defining l2 and u2. Recall that s1 was the sequence $1/n$, and so ps1 is the sequence of partial sums of $\sum_0^{\infty} 1/n$. For example the first few terms of ps1= $(1/1, 1/1 + 1/2, 1/1 + 1/2 + 1/3, \dots)$. What is the last term of ps1? The sequence ps2 is the sequence of partial sums of $1/n^2$. Everything in the code after ps2 is exactly the same as the previous example except for one change in the **legend** command. What is that change and why was it necessary? We see from the graph that the behaviors of the two sequences of partial sums appear very different. In the end $\sum_0^{\infty} 1/n$ diverges while the $\sum_0^{\infty} 1/n^2$ converges.

R Code

```
> l2=900
> u2=1000
> ps1=cumsum(s1)
> ps2=cumsum(s2)
> miny3=min(min(ps1[c(12:u2)]),min(ps2[c(12:u2)]))
> maxy3=max(max(ps1[c(12:u2)]),min(ps2[c(12:u2)]))
> plot(n1[c(12:u2)],ps1[c(12:u2)],pch=1,xlim=c(l2,u2),
+ ylim=c(miny3,maxy3),xlab="n",
+ ylab="Sequence of Partial Sums Values")
> points(n1[c(12:u2)],ps2[c(12:u2)],pch=17,col="red")
> legend(990,7,c(expression(frac(1,n)),
+ expression(frac(1,n^2))),pch=c(1,17),pt.cex=c(1.5,1.25),
+ col=c("black","red"),y.intersp=1.25)
```



5.2 The Derivative as a Limit

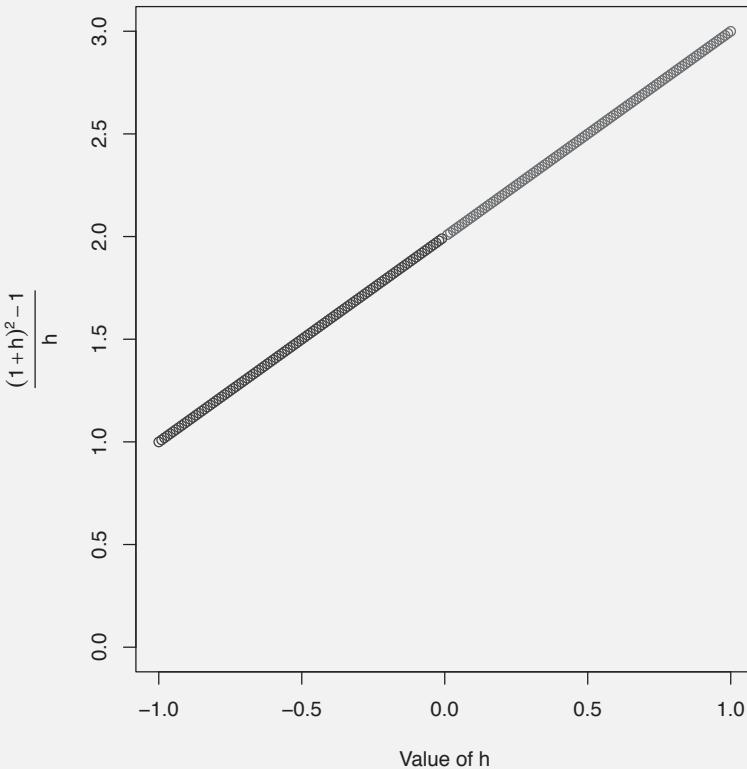
The most important limit in a calculus class is that of the derivative. Here we will see how R may help students understand these limits with three examples dedicated to a derivative limit. The fourth example is a limit associated with a rational function. One of the advantages of using R is that it is somewhat natural to differentiate between the left and right hand limits. Our first three examples begin by defining sequences that get close to 0 using the colon command.

Our first example considers the derivative of x^2 at $x = 1$. The first two lines create a sequence of points tending to 0 from the positive and negative

side. The first hp, for h positive, uses the colon command to create the integers from 100 to 1 and then each point is divided by 100. Hence, hp is a sequence from 100/100 to 1/100. Similarly, hm, for h minus, uses the colon command and division to generate the sequence from $-100/100$ to $-1/100$. The third line defines the function **D**, a function of h , and is the difference quotient or the secant line from $x = 1$ to $x = 1 + h$, for the function x^2 . The **par** command was used to add extra space on the left hand margin. Recall that mar sets the lines for the bottom, left, top, and right margins. Next we plot the positive values of h by using **plot** with hp and **D(hp)**. The points are colored red and the ranges for the x and y axis are set with xlim and ylim. We caption the x and y axis with xlab and ylab. Note the use of **expression** in the ylab option. We needed extra space on the left since our caption is a fraction that takes up extra lines. Lastly, we add the points hm and **D(hm)** and color these blue.

R Code

```
> hp=100:1/100
> hm=-100:-1/100
> D=function(h){((1+h)^2-1)/h}
> par(mar=c(4,6,4,4))
> plot(hp,D(hp),col="red",xlim=c(-1,1),ylim=c(0,3),
xlab="Value of h",ylab=expression(frac((1+h)^2-1,h)))
> points(hm,D(hm),col="blue")
```

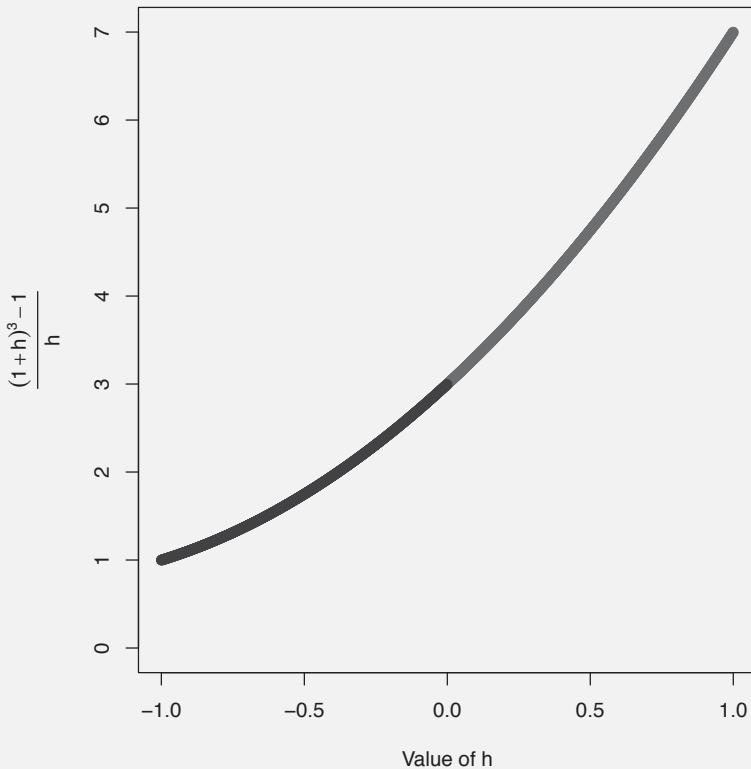


From our graph it appears the both the left and right limits converge to 2. In other words, we can conjecture that $\lim_{h \rightarrow 0} \frac{(1+h)^2 - 1}{h} = 2$. But also notice that the graph is linear. Will the graph of a derivative limit always appear linear? Our next example answers that question by investigating the derivative of x^3 at $x = 1$. The code is almost identical (it pays to save code and use copy and paste). For illustration purposes our sequences hp and hm are from 1 to 1/1000 and -1 to $-1/1000$. We define **D** similarly and add extra space on the left margin with the **par** command. The only changes in the **plot** command are the ylim and ylabel. This graph suggests that the limit is 3, which we know is the derivative of x^3 at $x = 1$, but the convergence is not linear.

R Code

```
> hp=1000:1/1000
> hm=-1000:-1/1000
```

```
> D=function(h){((1+h)^3-1)/h}
> par(mar=c(4,6,4,4))
> plot(hp,D(hp),col="red",xlim=c(-1,1),ylim=c(0,7),
xlab="Value of h", ylab=expression(frac((1+h)^3-1,h)))
> points(hm,D(hm),col="blue")
```



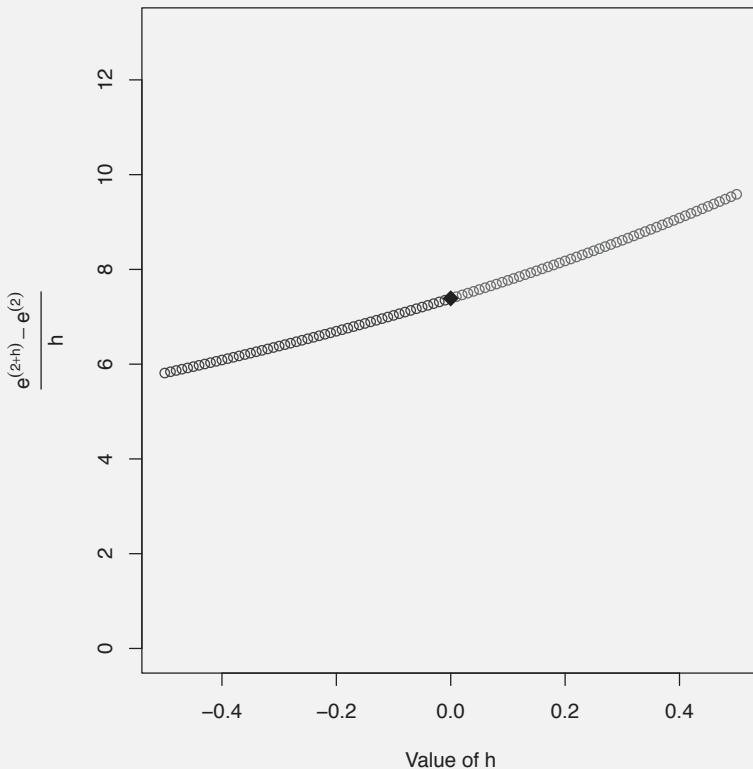
Our final derivative example considers the exponential function and we add some generalization into the code. We define **hp** and **hm** similarly but **D** is now a function of two variables. We again have the **par** command to set the margins. In **plot**, we plot **hp** and **D(2,hp)** so that we are viewing the derivative at 2. We have adjusted **xlim**, **ylim**, and **ylab** accordingly within **plot**. **Points** plots the values from the left again setting $a = 2$ in **D(2,hm)**. Lastly, we added a point, a black diamond with **pch=18** and the default color, at the limit value.

R Code

```

> hp=50:1/100
> hm=-50:-1/100
> D=function(a,h){(exp(a+h)-exp(a))/h}
> par(mar=c(4,6,4,4))
> plot(hp,D(2,hp),col="red",xlim=c(-0.5,0.5),
+ ylim=c(0,13),xlab="Value of h",
+ ylab=expression(frac(e^(2+h)-e^(2),h)))
> points(hm,D(2,hm),col="blue")
> points(0,exp(2),pch=18,cex=1.75)

```

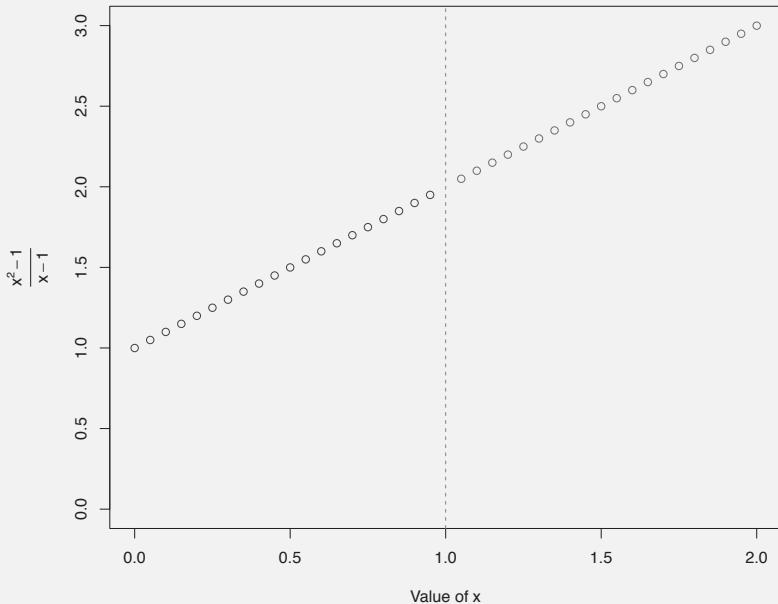


We end this section with a similar example by considering $\lim_{x \rightarrow 1} \frac{x^2 - 1}{x - 1}$. Once you can create a sequence tending to 0, it is easy enough to create a sequence tending to any value. Here xr starts by using the colon command and division to get points from 20/20 to 1/20 but then add 1 to each value. The sequence xl is created in a similar manner. We define the function f with

variable x . The code is now similar to that above except we add a vertical line at $x = 1$.

R Code

```
> xr=1+ 20:1/20
> xl=1- 20:1/20
> f=function(x){(x^2-1)/(x-1)}
> par(mar=c(4,6,4,4))
> plot(xr,f(xr),col="red",xlim=c(0,2),ylim=c(0,3),
  xlab="Value of x",ylab=expression(frac(x^2-1,x-1)))
> points(xl,f(xl),col="blue")
> abline(v=1,lty=2,col="gray40")
```



5.3 Recursive Sequences

In learning how to create recursive sequences we turn to the Fibonacci sequence. Do we really have another choice? We will demonstrate two avenues to work with the sequence. One creates a sequence of n values for a choice of

n , while the other creates a function to return a given value of the sequence. In the first we use a for loop and the other an if then statement.

We begin by generating the first 20 values of the Fibonacci sequence. Our code starts by setting $n = 20$. We don't have to do this as we could just substitute 20 anytime we have n in the code, but this allows us to create different sequence length by changing only one value. Next we define our sequence FibSeq to be a **numeric** sequence of length $n = 20$. The first two values of the sequence are each defined to be 1. We now need our for loop. The first part (i in 3:n) has the value of i range from 3 to n, which we set to 20. The next part sets the ith value of the sequence to the sum of the previous two, $i-1$ and $i-2$. Note that we defined the 1st and 2nd value and started off with $i=3$ so that $i - 1 = 2$ and $i - 2 = 1$ are both defined. Lastly, FibSeq, simply returns the sequence. We now have a list of the first 20 values of the Fibonacci sequence.

R Code

```
> n=20  
> FibSeq=numeric(n)  
> FibSeq[1]=1  
> FibSeq[2]=1  
> for (i in 3:n) {FibSeq[i]=FibSeq[i-1]+FibSeq[i-2]}  
> FibSeq
```

```
[1]   1   1   2   3   5   8  13  21  34  55
[11] 89 144 233 377 610 987 1597 2584 4181 6765
```

Now, the output lacks a numbering and it would be nice to have this in a table with the associated value in the sequence. We do that now. Our first attempt starts by creating a list of numbers from 1 to 20 (or n) with the colon command. The idea here is to combine the two sequences into a matrix. The **rbind** command does that in the second line. The third line prints out the table. This works but is a little awkward since R will add the column number with the [1], etc. outputs. Our second example cheats a bit by simply putting our sequence into a 20 by 1 matrix and takes advantage of the extra output.

R Code

```
> list=1:n  
> FibTable=rbind(list,FibSeq)  
> FibTable
```

```

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
list   1    2    3    4    5    6    7    8    9   10
FibSeq 1    1    2    3    5    8   13   21   34  55
etc...

```

R Code

```
> FibTable2=matrix(FibSeq,n,1)
> FibTable2

[,1]
[1,]    1
[2,]    1
[3,]    2
[4,]    3
[5,]    5
[6,]    8
[7,]   13
[8,]   21
[9,]   34
[10,]  55
etc...
```

What if we would like just one value in the Fibonacci sequence? Our next example does just that. Here we define the function **FibFun** as a function of n . We now use an if statement to work out the details. The first part says that if the value of n is 1 or 2, where `==` is a logical operator for equality and `||` is for or, then return the value of 1. Otherwise, actually else in the code, return the sum of the previous two values. We then ask to return the 30th value of the sequence with **FibFun(30)**. Be careful in using this since all values less than n are kept in memory while it keeps calling the **FibFun** function. It isn't hard to bog down your computer using this function.

R Code

```
> FibFun=function(n) {
+   if (n==1 | n == 2) return (1)
+   else return (FibFun(n-1)+FibFun(n-2))
+ }
> FibFun(30)

[1] 832040
```

We end this chapter by noting the **seq** command. We point out two uses here. First **seq(from, to, by)** will create a sequence from a number to another number counting by a given number. For example, if you want a sequence of multiples of 5 up to 200, then use **seq(0,200,5)**. The second use is to create partitions. If we want to portion the interval from 0 to 3, say to create 4 Riemann sum boxes for a function, we would follow our last example.

R Code

```
seq(0,3, length.out=5)

[1] 0.00 0.75 1.50 2.25 3.00
```

Code Review for the Sequences Series and Limits Chapter

a | b is a logical statement of a or b.

a:b is the colon operator. It creates a vector of number from a to no greater than b, in steps of 1. The values of a and b can be any real number.

a==b is a logical operator that returns true if a==b, and otherwise returns false.

cumsum(v) returns a vector where entry i is the sum of the first i elements of v.

for(i in a:b){expression} evaluates the expression for each value of i from a to b.

function(x₁,x₂,...,x_n){expression} defines a function of x_1, x_2, \dots, x_n with expression. The function will return the last line of expression and for clarity use return().

if (cond) {expression1} else {expression2} is an if-else statement. Expression1 is executed if cond is true; otherwise exprssion2 is executed.

min(v) finds the minimum value of the vector v. Similar commands includes max, mean, median, and sum.

seq(a,b,c) returns a sequence of numbers from a to b in steps of c.

seq(a,b,length.out=c) returns a sequence that partitions the interval from a to b with c values.

5.4 Exercises

1. Explore $\lim_{x \rightarrow \infty} \sin(1/x)$ and $\lim_{x \rightarrow 0} \sin(x)$. What can you say about the two limits?
2. Create a Fibonacci like sequence with two different starting points

and compare it to the Fibonacci sequence by graphing the difference of the two sequences.

3. Find the sum of the first 100 integers in one line of code.
4. Create a sequence from 1 to 100 with integers of the form $4n + 1$ squared, $4n + 3$ cubed, and even numbers replaced by 0.
5. Investigate the “gap” between the $1/n$ and $1/n^2$ sequences as $n \rightarrow \infty$.
6. Plot the sequence of partial sums of $1/n$ and $1/n^2$ from 0 to 1000. Which graph, from 0 to 1000 or 900 to 1000, is more useful in understanding the behavior of the two sequences of partial sums? Why?
7. If you want one particular value of the Fibonacci sequence, which is the better way to find it, creating the sequence or using the function?
8. Create a factorial function.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

6

Calculating Derivatives

The focus of this chapter is to demonstrate symbolic differentiation with R using the `Deriv` package. In general, symbolic differentiation with R doesn't rival Mathematica, but it is certainly sufficient for the needs of calculus and other undergraduate courses. We begin with a quick look at numeric differentiation using the `numDeriv` package.

The following code is straightforward. The `numDeriv` package is opened with `library(numDeriv)` and then we define two functions f and g . The `grad` function takes a function and a value in the domain. As you can see, the output is what we expect.

R Code

```
> library(numDeriv)
> f=function(x){x^2}
> g=function(x){exp(x)}
> grad(f,2)
```

```
[1] 4
```

R Code

```
> grad(g,1)
```

```
[1] 2.718282
```

R does have some built-in functions such as `exp` and `cos`. When using these functions you don't need to define them as we did in the previous example by defining g to be `exp`. Note that in R, `log(x)` is the natural logarithm whereas `log10` and `log2` are built-in for base 10 and 2. More generally, `logb(x, base=b)` is log base b. These functions can be used directly as the next example illustrates.

In the next example, the first line provides the derivative of $\cos(x)$ at $\pi/2$ and the output is -1 . We added two lines to the code here with `options` setting the number of digits for the output, the default being seven, and then running `grad` again. This provides some sense of accuracy or lack of accuracy from `numDeriv`.

R Code

```
> grad(cos, pi/2)
[1] -1
```

R Code

```
> options(digits=20)
> grad(cos, pi/2)
[1] -0.999999999999338152
```

You may find the numDeriv package useful but we really want to focus on symbolic differentiation so we move to those examples now.

6.1 Symbolic Differentiation

Our first example starts by opening the Deriv package, which is necessary for all examples that follow in the section and the rest of the chapter. Once you open a package in an R session you don't need to do it again until after you close R and restart a session. We define the function **f** and set **fp**, for f prime, to be the derivative of **f** with **Deriv(f)**. We output **fp** and then set **fpp** to be the derivative of **fp**, in other words, the second derivative of **f**. We then output **fpp** and get the expected result.

R Code

```
> library(Deriv)
> f=function(x){7*x^2+11*x+42}
> fp=Deriv(f)
> fp
> fpp=Deriv(fp)
> fpp

function (x)
11 + 14 * x
```

R Code

```
> fpp=Deriv(fp)
> fpp
```

```
function (x)
14
```

Our next example illustrates three points. First the function defined includes the constant a , so when we use **Deriv** we need to specify the variable. Ok, we really don't, since **g** is defined as a function of x only, but it is a bad habit not to clarify the variable when using **Deriv**. We did not set the derivative to a variable. It isn't necessary to do so unless you want to use the derivative in a further calculation, and we will do so in subsequent examples. Third, we use the `nderiv` option in **Deriv**, which sets the desired derivative. In this example, we calculated the ninth derivative.

R Code

```
> g=function(x){sin(a*x)}
> Deriv(g,"x")

function (x)
a * cos(a * x)
```

R Code

```
> gp9=Deriv(g,"x", nderiv=9)
> gp9

function (x)
a^9 * cos(a * x)
```

The output isn't always nice especially when the derivative becomes complicated with a chain rule. Our next example is a product rule with chain rules as you can see by the definition of **h**. The output has $.e1=5*x$ as a variable that must be substituted into the line below. But **hp** can still be evaluated as we see with the last line even though the algebraic output of **hp** isn't as clean as we might like.

R Code

```
> h=function(x){exp(x^2)*cos(5*x)}
> hp=Deriv(h)
> hp

> hp
function (x)
{
```

```
.e1 <- 5 * x
(2 * (x * cos(.e1)) - 5 * sin(.e1)) * exp(x^2)
}
```

R Code

```
> options(digits=20)
> hp(5)

[1] 761364459126.21741
```

Our final single-variable example illustrates use of the quotient rule. Note in the definition of **k** that **log** is the natural log and for reference **log10** is log base 10. We were sloppy here in not specifying that we are requesting the derivative with respect to *x*. The output has three parts with .e1 and .e2 needing to be substituted into the last line. As in the previous example, we can evaluate **kp** at any number and get a numeric answer if d has a value.

R Code

```
> k=function(x){x*log(x)/(x^2+d)}
> kp=Deriv(k)
> kp

function (x)
{
  .e1 <- x^2
  .e2 <- d + .e1
  ((1 - 2 * (.e1/.e2)) * log(x) + 1)/.e2
}
```

We have two final examples for this section. The first is a multivariate function and the second uses **Deriv** as part of a function. For the first, the multivariate function **m** is defined as a function of *x* and *y*. We take the derivative with respect to *x* and call it **m.xp** and then take the derivative with respect to *y*. In this example you must specify the variable since **m** is defined as a function of two variables.

R Code

```
> m=function(x,y){x*exp(y^2)+ y*log(x) + 3*x}
> m.xp=Deriv(m,"x")
> m.xp
```

```
function (x, y)
3 + exp(y^2) + y/x
```

R Code

```
> Deriv(m.xp,"y")

function (x, y)
1/x + 2 * (y * exp(y^2))
```

Finally, we embed **Deriv** inside a function. We begin by defining $\sin(x)$ as the function **f**. We don't need to do this since $\sin(x)$ is a defined function, but sometimes it makes it easier to follow the code when we do. The function **f.np** is defined as the nth derivative of **f**. We see that **f.np(5)** is $\cos(x)$ and that **f.np(5)(pi)** is -1. Be warned that the output of **f.np(n)** can be messy as no simplification happens along the way. Note that the definition of **f.np** uses the **function** command and returns a function. We'll see other examples of defining function that returns functions further in the chapter.

R Code

```
> f=function(x){sin(x)}
> f.np=function(n){Deriv(f,nderiv=n)}
> f.np(5)

function (x)
cos(x)
```

R Code

```
> f.np(5)(pi)
```

```
[1] -1
```

As you can see R is capable of symbolic differentiation. We now move to three examples that are a bit more interesting. Next we include the use of the rootSolve package to find max, min, and inflection points and create a graph with those points identified. After that we place a number of tangent lines on a curve with a for loop while illustrating the use of a color palette. Lastly we identify the inflection points of a normal density curve and demonstrate shading the tails of the curve.

6.2 Finding Maximum, Minimum, and Inflection Points

Our example here takes advantage of the rootSolve package. The rootSolve package is opened with **library(rootSolve)** after which we define the function **f** for this example. We use **Deriv** to define **fp**, the derivative of **f**, and **fpp**, the second derivative of **f**. For **fpp** we could have used **Deriv(f,nderiv=2)**. The function **uniroot.all**, from the rootSolve package, will find all roots of the given function in the defined interval. We set First.D.roots and Sec.D.roots to be the roots of the first and second derivative respectively. The output provides us with two roots each for both the first and second derivative. We now move to creating our graph.

R Code

```
> library(Deriv)
> library(rootSolve)
> f=function(x){(x^2-1)*exp(-x)}
> fp=Deriv(f)
> fpp=Deriv(fp)
> First.D.roots=uniroot.all(fp, c(-10,10))
> First.D.roots

> First.D.roots
[1] -0.4142331  2.4142303
```

R Code

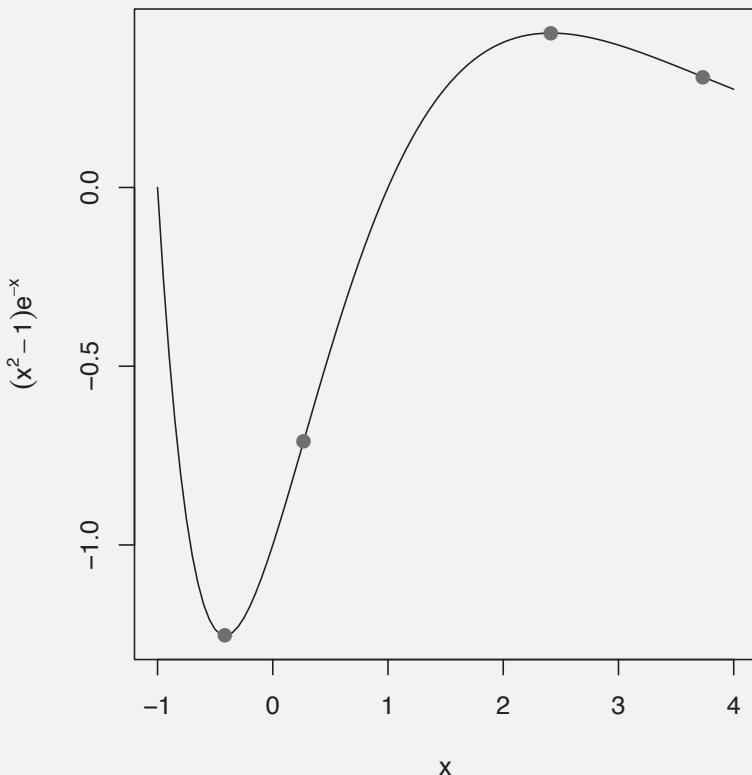
```
> Sec.D.roots=uniroot.all(fpp, c(-10,10))
> Sec.D.roots

[1] 0.2679472 3.7320688
```

We set the margins with **par** so that the bottom, left, top, and right margins have 4, 5, 3, and 3 lines respectively since the expression for the y -axis required more lines than the default. We plot **f** for values of x from -1 to 4 and label the y -axis with a mathematical **expression**. The vectors, **xValues**, of roots are defined and used in the following line with the **points** function. The four points are added to the graph with their associated y -values. The points are colored red, the **cex=1.25** option scales the point by a factor of 1.25, and **pch=16** makes the point character a solid.

R Code

```
> par(mar=c(4,5,3,3))
> plot(f,xlim=c(-1,4),ylab=expression((x^2-1)*e^-x))
> xValues=c(First.D.roots, Sec.D.roots)
> points(xValues,f(xValues),col="red",cex=1.25,pch=16)
```



6.3 Graphing a Function and Its Derivative

We create a graph with a function and its derivative with different scales for the left and right y -axis. In the first three lines we load the `Deriv` package, define the function `f` to be $x + x \sin(4x) - 1$, and set `fp` to the derivative of `f`.

We use **par** so that the bottom, left, top, and right margins have 4, 5, 3, and 3 lines respectively.

We **plot** the function **f** with an x -axis from 0 to 4.5, a y -axis from -2 to 8, with a line width of 2, and colored blue. The x -axis label is suppressed and left blank with **xlab** set to empty quotes. Both **yaxs** and **xaxs** are set to "i", which suppresses the 4% extra space which is the default. The y -axis is labeled using **expression** for formatting mathematical expressions. Note that two equal signs typesets an equal sign and * is used to denote multiplying. The label is scaled to 1.25 with **cex.lab** and colored blue with **col.lab**. The function **axis** generates a left axis with 2 and colors the line blue. Note that the axis in this case was already created with **plot** and so the use here is just to color the left y -axis. A horizontal line at $y = 0$ is added with **abline**, **h=0**, and colored blue.

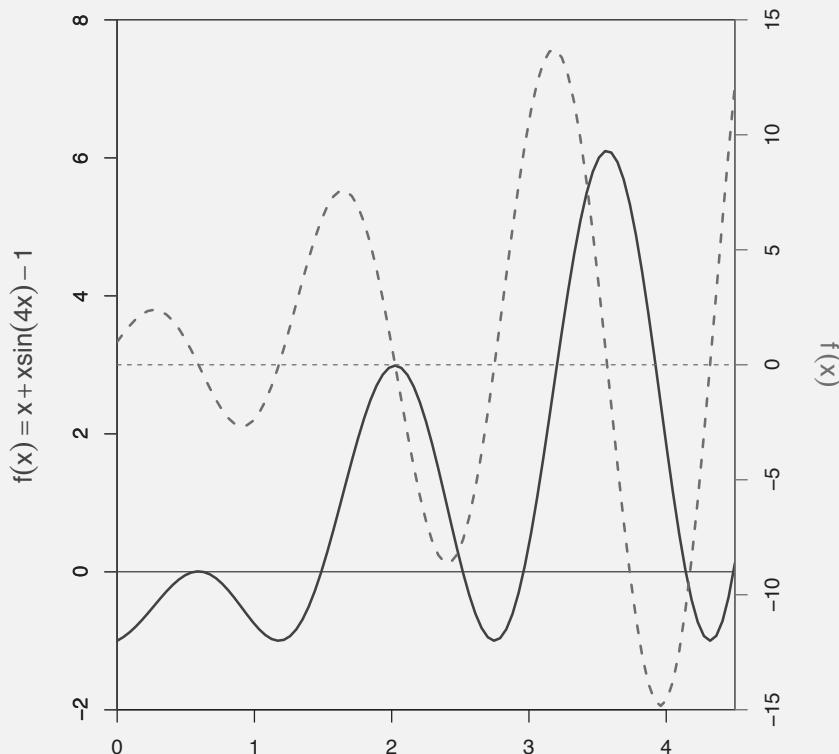
To overlay another plot on top of the one created we use **par(new=T)**. We use **plot** again the plot **fp** with the same x -axis range but a y -axis from -15 to 15. The graph is colored red and the line type is set to 2 for a dashed line. Both axis labels are blank with empty quotes. Again, both **yaxs** and **xaxs** are set to "i" to suppress the 4% extra space. Axes aren't displayed with **axes=F**, since axes are already drawn. We use **axis** to create the right y -axis, the first argument is 4, and color it red. A horizontal line at $y = 0$ is added with **abline**, **h=0**, and colored red.

The last two lines are to label the right y -axis. A four element vector (v_1, v_2, v_3, v_4) is returned by **par("usr")** and set to **p**, where (v_1, v_3) is the bottom left corner of the graph and (v_2, v_4) is the top right corner of the graph. The first two arguments of **text** are the coordinates where text is to be placed. The value **p[2]+.7**, where **p[2]** is the second element of **p**, is 0.7 to the right of the right y -axis and **mean(p[3:4])**, the average of the third and fourth value of **p**, is the center of the right y -axis. The third argument of **text** is the text to be added to the graph, which is defined bye **expression** to get $f'(x)$. The text is rotated 90° clockwise with **srt**. By default text is clipped beyond the plot region. This is changed to clipping beyond the device region with **xpd=NA**. We set **adj=0.5** so that the text is centered. The text is scaled with **cex** and colored red.

R Code

```
> library(Deriv)
> f=function(x){x+x*sin(4*x)-1}
> fp=Deriv(f)
> par(mar=c(4,5,3,5))
> plot(f,xlim=c(0,4.5),ylim=c(-2,8),lwd=2,col="blue",
+ xlab="",yaxs="i",xaxs="i",ylab=expression(f(x)==
+ x+x*sin(4*x)-1),cex.lab=1.25,col.lab="blue")
> axis(2,col="blue")
> abline(h=0,col="blue")
```

```
> par(new=T)
> plot(fp,xlim=c(0,4.5),ylim=c(-15,15),col="red",lty=2,
xlab="",ylab="",lwd=2,yaxs="i",xaxs="i",axes=F)
> axis(4,col="red")
> abline(h=0,lty=2,col="red")
> p=par("usr")
> text(p[2]+.7,mean(p[3:4]),expression(f^{'''}*(x)),
srt=-90,xpd=NA,adj=0.5,cex=1.25,col="red")
```

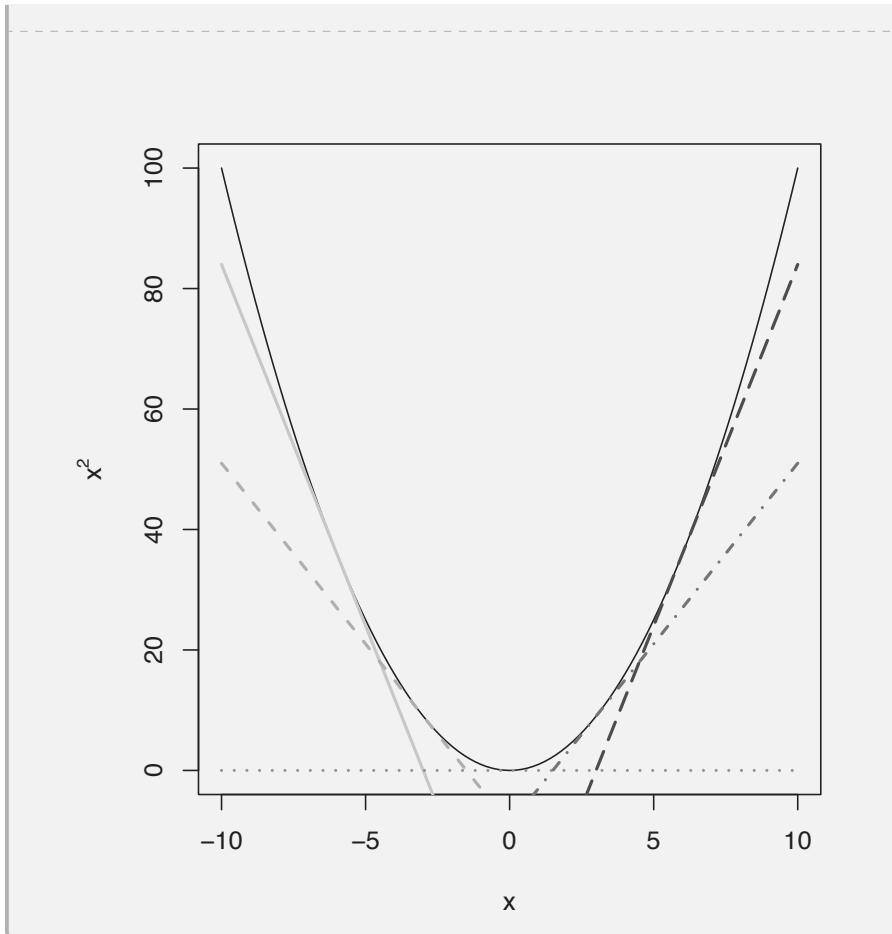


6.4 Graphing a Function with Tangent Lines

We now graph x^2 and add five tangent lines to the graph. As we do this, we will illustrate a few bells and whistles. We begin by defining **f** to be x^2 and **fp** to be its derivative. The function **f.tl**, f tangent line, is defined as a function of two variables, which allows us to generate the equation of the tangent line at any point a . The function within **f.tl** is the equation of the tangent line using **f** and **fp**.

R Code

```
> library(Deriv)
> f=function(x){x^2}
> fp=Deriv(f)
> f.tl=function(x,a){fp(a)*(x-a)+f(a)}
> palette=colorRampPalette(c("steelblue1","royalblue4"))
> colors=palette(5)
> plot(f,xlim=c(-10,10),ylab=expression(x^2))
> for(i in -2:2){curve(f.tl(x,3*i),lwd=2,
+ col=colors[i+3],lty=i+3,add=TRUE)}
```



We plan on graphing a number of tangent lines and wish to color them in a systematic fashion. We define palette to be a palette of colors ranging from steelblue1 to royalblue4. The function **colorRampPalette** creates this range of colors. You should enter palette to see exactly what was created. We define colors to be five values from the palette, which will be chosen spaced equally within palette.

We now plot the function **f** from -10 to 10 and label the y -axis with x^2 . Our for loop that follows is set up to add tangent lines at -6 , -3 , 0 , 3 , and 6 . Within the for loop we use **curve** to add the lines to the graph. The function **curve** is similar to **plot** but will only take a function, not points or other objects. All options that are available to **plot** are also available to **curve**.

As i ranges from -2 to 2 in our for loop **curve** plots the tangent line at $a = 3 * i$. The line width is set to 2 times the default with **lwd=2**. The colors of each line will be taken in order from the vector of colors, but since the elements of colors are from 1 to 5 , we must use $i + 3$. Similarly, we define the

line type, lty=i+3, so that the line types chosen are from 1 to 5. Lastly, we need the option add=TRUE so that each line is added to the previous graph, without which a new graph would be created with each call to **curve**.

In this example, we defined the tangent line as **f.tl=function(x,a){fp(a)*(x-a)+f(a)}** and it does work, but only for the specific function **f**. We can do better than this, by defining **f.tl.new=function(f,a){return(function(x){f(a)+Deriv(f)(pt)*(x-a)})}**. Now, **f.tl.new** has a function and a point as inputs and returns a function. So, for example **f.tl.new(exp(x),0)** is the equation of the tangent line to e^x at $x = 0$ and **f.tl.new(exp(x),0)(0.1)** is the value at $x = 0.1$ on the tangent line.

6.5 Shading the Normal Density Curve Outside the Inflection Points

Our final example defines the normal density function, **Norm.u.s**, as a function of three variables. We use **Deriv** to take the second derivative of **Norm.u.s** with respect to “x” by setting nderiv=2. The function **uniroot.all** will need a function of one variable and so we set **NormInflFix.u.s**, normal inflection fix u and s, to be the function **Norm.u.s.pp** with $u = 0$ and $s = 2$. This is followed by using **uniroot.all** to find the roots of **NormInflFix.u.s**, the second derivative of **Norm.u.s** with $u = 0$ and $s = 2$, over the interval from -5 to 5 . As expected **uniroot.all** returns -2 and 2 . Note that we defined **Norm.u.s** here as an example, when we could have used the normal density function that is already defined as **dnorm(x,mean,sd)**.

R Code

```
> library(Deriv)
> library(rootSolve)
> Norm.u.s=function(x,u,s){ 1/sqrt(2*pi*s^2)*
exp(-(x-u)^2/(2*s^2))}
> Norm.u.s.pp=Deriv(Norm.u.s,"x",nderiv=2)
> NormInflFix.u.s=function(x){Norm.u.s.pp(x,0,2)}
> roots=uniroot.all(NormInflFix.u.s,c(-5,5))
> roots
```

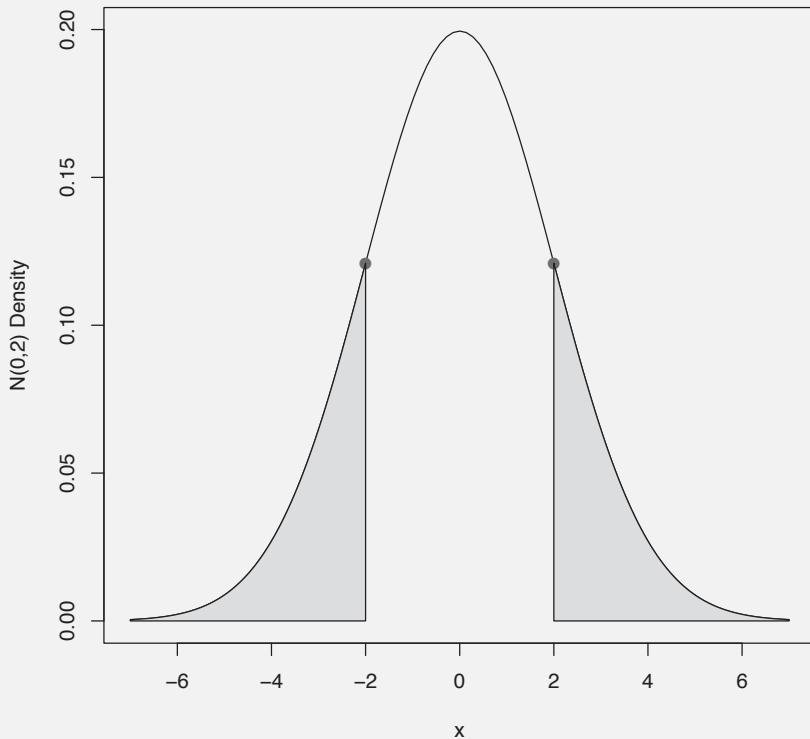
[1] -2 2

We now turn to creating a graph of our results. We set xValues to our two points of interest, -2 and 2 , and use **curve** to plot our function with $u = 0$ and $s = 2$ over the interval from -7 to 7 . The **points** function adds the two inflection points to the graph colored red, scaled to 1.25 times the default size (cex=1.25), with the point character selected to solid (pch=16). Note that

with **points** you can enter a list of x -values and the associated y -values can be evaluated by a function.

R Code

```
> curve(Norm.u.s(x,0,2),-7,7,ylab="N(0,2) Density")
> points(roots, Norm.u.s(roots,0,2), col="red", cex=1.25,
  pch=16)
> xlist=seq(-7,-2,0.01)
> polygon(c(xlist,rev(xlist)),c(Norm.u.s(xlist,0,2),
  0*rev(xlist)),col="gray85",border="black")
> polygon(c(-xlist,rev(-xlist)),c(Norm.u.s(xlist,0,2),
  0*rev(xlist)),col="gray85",border="black")
```



Our last goal is to shade the tails of the density curve starting at the inflection points. The function **polygon**, which shades a polygon region, will be used. We need to define a region that follows the density curve and then along

the x -axis with points close enough so that the triangular regions appear curved. The function `seq(-7, -2, 0.01)` creates a sequence of points from -7 to -2 in steps of 0.01 , which is defined to be `xlist`. The **`polygon`** function needs a set of x and y values to draw a polygon by connecting the points. Here **`c(xlist, rev(xlist))`** creates a string of x values by appending **`rev(xlist)`**, the sequence of x values we created but in reverse order, to the sequence `xlist`. In other words, we have a list of x values that range from -7 to -2 in steps of 0.01 and then back from -2 to -7 . Our y -values are similarly created in two parts. The first part, **`Norm.u.s(xlist, 0, 2)`** evaluates the values from `xlist` to create a sequence of points on the curve. We follow that with a list of zeroes to follow along the x -axis which is done by multiplying **`0*rev(xlist)`**. We could have multiplied **`0*xlist`**, but since we used **`rev(xlist)`** in the second part of the x -values we used it to insure that we have the correct number of y -values, since the number of x -values needs to equal the number of y -values. The polygon created follows along the density curve from -7 to -2 , then down to $(-2, 0)$, and then follows the x -axis back to -7 . The polygon is shaded with `gray85` and given a black border. The right half is done similarly and all we need to do is use `-xlist`, which multiplies each element of `xlist` by -1 .

Code Review from Calculating Derivatives Chapter

`colorRampPalette(c("color 1", "color 2"))` creates a color palette from color 1 to color 2.

`Deriv(f, "x", nderiv=n)` calculates the nth derivative of the function f with respect to x . If f has only one variable " x " can be omitted. For the first derivative, $n=1$, `nderiv` can be omitted. Uses the `Deriv` package.

`function(x){body}` defines a function of x .

`grad(f,a)` finds the derivative of f at a . Uses `numDeriv` package.

`options(digits=n)` sets the number of digits for the output to n . The default is 7.

`polygon(xlist,ylist,col="color 1",border="color2")` creates a polygon connecting the points given by `xlist` and `ylist` that is shaded color 1 and has color 2 for its border.

`uniroot.all(f,c(a,b))` finds all the roots of f in the interval a to b .
Uses the `rootSolve` package.

6.6 Exercises

1. Use a for loop to find the inflection points of the normal density for values of s from 1 to 100. Imbed it in another for loop to also vary u .
2. Choose your favorite functions and create a set of graphs, one with the function and one with its derivative. For each pair, add a tangent line to the graph of the function and a corresponding dot on the derivative graph. Tip: `par(mfrow=c(a,b))` will allow for $a \times b$ plots in a single window with a rows and b columns. The next $a \times b$ calls for a graph will be placed in the same frame.
3. Using only `Deriv()`, create a function that takes a function and a point and returns the n th derivative of the function at the given point.
4. Redo the example in [Section 6.4](#) using `t.fl.new` as defined at the end of the section.
5. Redo the example in [Section 6.5](#) using `dnorm(x,mean,sd)` instead of defining the normal density function.
6. Create a graph of $f(x) = \sqrt{x}$, with the tangent line at $x = 4$. Calculate and label the tangent line approximation to $\sqrt{4.1}$. Generalize this by creating a function that accepts a function, a value for the location of the tangent line, and a value to approximate with the tangent line. The function should return a graph and the approximation value.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Riemann Sums and Integration

The computation and graphical abilities of R can be applied to integration, and although there is little support for symbolic integrations (one package supports symbolic integration of polynomials; see [Chapter 4](#)) we can effectively compute and illustrate Riemann sums, perform numeric integration, and graph antiderivatives.

R Code

```
> f=function(x){(5+exp(x)+(2.5)^x*sin(2*pi*x))/3000}
> dx=(11-5)/19
> mid=seq(5+dx/2,11,dx)
> sum(f(mid)*dx)

[1] 18.37172
```

We begin this chapter by calculating the sum of 19 midpoint boxes from $x = 5$ to $x = 11$ for the function $f(x) = (5 + e^x + 2.5^x \sin(2\pi x))/3000$; purposely choosing and “ugly” function to illustrate the point that with R we aren’t constrained to using “nice” functions. Textbook examples and problems are typically nice and simple, but with R the number of subintervals, the interval, and the function, can be anything without extra effort. The code begins by defining the function and setting the width of the boxes to `dx`. The `seq` function does all the work by generating a sequence of points starting from $5+dx/2$ (the first midpoint), ending by 11, in steps of `dx`. The list of values will not exceed 11, so we do not have to give the exact last value of $11-dx/2$. The variable `mid` is set to this list of values. In our last line, `f(mid)` is the list of values created by evaluating each element of `mid` by `f`. Then `f(mid)*dx` multiplies each element of `f(mid)` by `dx`, creating a list of the areas of each box. Finally `sum` adds the values of `f(mid)*dx`.

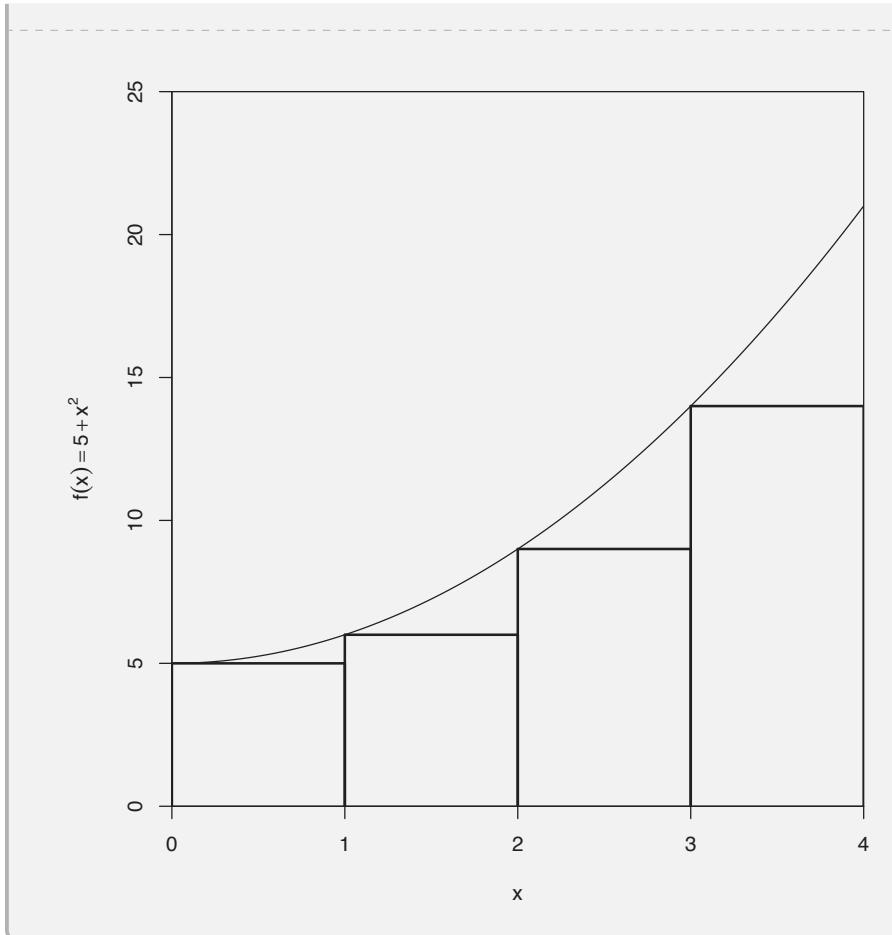
It is simple enough to change the function, interval, or number of boxes from this code. In the exercises, we ask the reader to change this example to calculate left and right boxes, and then to turn the procedure into a function. Later in the chapter we use the function here and create a graph with the midpoint boxes. As we move through this chapter, we will demonstrate how to create graphs with boxes, perform numerical single and double integration, calculate the area between two curves, and graph an antiderivative.

7.1 Riemann Boxes

In this section we will draw Riemann boxes on graphs in three ways. We first do this by placing line segments on the graph one at a time to create the boxes. This is tedious but illustrates how the boxes are created. We then improve on this by nesting the drawing of segments into a loop and we shade the boxes. Lastly, all of this will be placed within a function.

R Code

```
> f=function(x){5+x^2}
> par(mar=c(4,5,2,2))
> curve(f,0,4,ylim=c(0,25),yaxs="i",xaxs="i",
ylab=expression(f(x)==5+x^2))
> segments(0,0,0,f(0),lwd=2)
> segments(0,f(0),1,f(0),lwd=2)
> segments(1,f(0),1,0,lwd=2)
> segments(1,0,1,f(1),lwd=2)
> segments(1,f(1),2,f(1),lwd=2)
> segments(2,f(1),2,0,lwd=2)
> segments(2,0,2,f(2),lwd=2)
> segments(2,f(2),3,f(2),lwd=2)
> segments(3,f(2),3,0,lwd=2)
> segments(3,0,3,f(3),lwd=2)
> segments(3,f(3),4,f(3),lwd=2)
> segments(4,f(3),4,0,lwd=2)
```



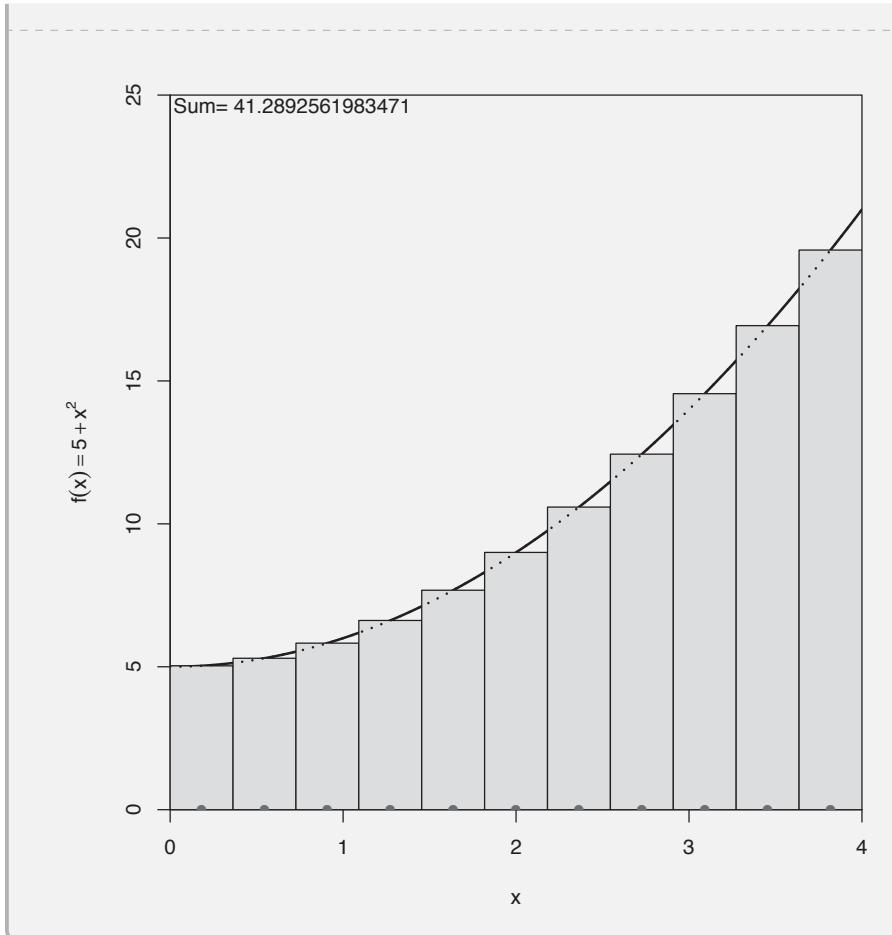
For our first example will use the function $f(x) = 5 + x^2$ and four left boxes from 0 to 4. The first line of code defines the function. The **par** command sets the margins to 4, 5, 2, and 2 lines for the bottom, left, top, and right, respectively. (The default doesn't leave enough space on the left to label it with $f(x) = 5 + x^2$.) We create a graph of the function from $x = 0$ to $x = 4$ with **curve**. We also set the y -axis range from $y = 0$, to $y = 25$ with `ylim=c(0,25)`. By default Graphs in R leaves some space between the axis and beginning of the graph. Visually, this is particularly unsightly with Riemann boxes and so the space is removed with `yaxs="i"` and `xaxs="i"`. The x -axis is labelled with x by default, but the y -axis is labelled with $f(x) = 5 + x^2$ by using **expression**. The **expression** function allows for mathematical typesetting with a few quirks, one of which is that == typesets an = sign. The next 12 lines create the boxes. The **segments** function adds a line segment to the graph from the first point to the second point. In each case, `lwd=2` draws a line that is twice the default thickness. All of the usual graphing options will work in

segments such as col, if you want to color the segments, or lty if you want a dashed line type.

The next example will use the same function, but we will begin to generalize the code as well as use 11 mid boxes that are shaded. We first define the function, and then the values for the left endpoint a=0, right endpoint b=4, and number of boxes n=11. By defining these variables first we can use a, b, and n in the rest of the code, and if we want to change a value then it only needs to be changed once. In fact, if you want to change the function, the interval, the number of subsets, or all of these, nothing beyond the first four lines of code needs to be modified. As in the intro example, we set dx to be the width of a box and define mid to be the list of midpoint values for the boxes. As in the previous example, **par** sets the number of lines for the margins and **curve** graphs the function. Note that we use a and b in **curve** instead of 0 and 4. We changed the line width to 2 for this example.

R Code

```
> f=function(x){5+x^2}
> a=0
> b=4
> n=11
> dx= (b-a)/n
> mid=seq(a+dx/2, b, dx)
> par(mar=c(4,5,2,2))
> curve(f,a,b,ylim=c(0,25),lwd=2,yaxs="i",xaxs="i",
ylab=expression(f(x)==5+x^2))
> for(i in 1:n){
+ mid.box.x=c(mid[i]-dx/2,mid[i]-dx/2,mid[i]+dx/2,
mid[i]+dx/2)
+ mid.box.y=c(0, f(mid[i]), f(mid[i]), 0)
+ polygon(mid.box.x,mid.box.y,col="gray85",
border="black") }
> curve(f,a,b,lty=3,lwd=2,add=TRUE)
> points(mid,0*mid,col="red",pch=16)
> p=par("usr")
> text(p[1], p[4],paste("Sum=",sum(f(mid)*dx))),adj=c(0,1))
```



To create the shaded boxes we use a for loop, (i in 1:n), with the code to be executed for each i beginning with a { and ending with a }. For each value of i from 1 to n, we will create a shaded box using the **polygon** function. The **polygon** function needs a set of x and y values to define the corners of the box, which we define to be mid.box.x and mid.box.y. Each of these variables has four points to define the four x and four y values for the corners of each box. Recall that mid is the set of midpoint values for each box, so mid[i] is the x -value of the middle of box i. Hence, subtracting and adding $dx/2$ yields the left and right x -values of the box. For mid.box.y two of the y -values are 0 and the other two are obtained by evaluating the function at mid[i]. In the end, our four points for the ith polygon are ($mid[i]-dx/2, 0$), ($mid[i]-dx/2, f(mid[i])$), ($mid[i]+dx/2, f(mid[i])$), and ($mid[i]+dx/2, 0$). The **polygon** function has the list of x -values as mid.box.x and y -values as mid.box.y. The boxes are shaded with gray85 and given a black border.

After the for loop is completed we redraw the curve using lty=3, which

is a dotted line. R creates the graph by drawing a new layer on top of the previous layer. The result here is the boxes cover up parts of the curve. By redrawing the curve with dotted lines we have dotted lines for the curve when the original drawing is covered by a box. We next add points to represent the midpoint of each box with the **points** function, which needs a list of x -values and y -values. We already have our x -values of midpoints defined by mid and the corresponding y -values are 0, which is accomplished by 0^*mid . Note that we can't just use one 0 for our y -values since the number of y -values must be the same as the number of x -values. The syntax 0^*mid , multiplies each value of mid by 0.

Our last two lines of code places the value of the sum of the boxes in the top left of the graph. We use **par("usr")** for the values of the bottom left and top right corners of the graph frame. The first and third values define the point of the lower left corner, $(\text{p}[1],\text{p}[3])=(0,0)$, and the second and fourth values define the point for the top right corner, $(\text{p}[2],\text{p}[4])=(4,25)$. In this case, we know these values but if we use a different function and/or interval then we do not have to change this. We use **text** to place text on the graph. The first two values are the location of the text where $\text{p}[1]$ and $\text{p}[4]$ give the location for the top left of the frame. The **paste** function concatenates the text $\text{Sum}=$, as it is in quotes, with the value of the sum which is calculated by **sum(f(mid)*dx)**. Lastly $\text{adj}=\text{c}(0,1)$ sets the alignment of the text where 0 is left or bottom and 1 is right or top alignment. In this case, we aligned the text to the left and top of the location.

We now generalize the process of graphing a function with n midpoint boxes from a to b , by creating a function. We begin by naming our function MidBox, a function of four variables, a the left endpoint, b the right endpoint, n the number of boxes, and f the function. The code for the MidBox function begins with { and ends with }. As before, we define dx , the width of the boxes, and mid , the list of midpoints using **seq**. The **par** function sets our margins, although the default margins would have worked. There are two changes to **curve**. We leave out ylim and go with the default range for the y -axis. We also label the y -axis as simply "function," since it is an unknown function.

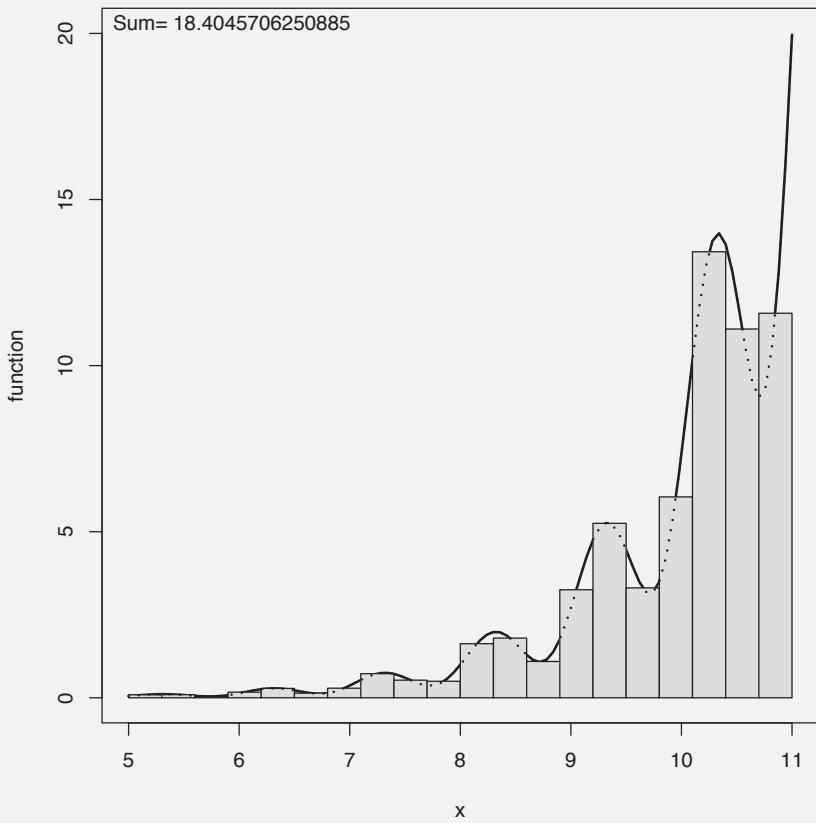
R Code

```
> MidBox=function(a,b,n,f){
+ dx=(b-a)/n
+ mid=seq(a+dx/2, b, dx)
+ par(mar=c(4,4,2,2))
+ curve(f,a,b,lwd=2,ylab="function")
+ for(i in 1:n){
+ polygon(c(mid[i]-dx/2,mid[i]-dx/2,mid[i]+dx/2,
+ mid[i]+dx/2),c(0,f(mid[i]),f(mid[i]),0),col="gray85",
+ border="black") }
+ curve(f,a,b,lty=3,lwd=2,add=TRUE)
```

```

+ p=par("usr")
+ text(p[1]+.1, p[4]-.2,paste("Sum=",sum(f(mid)*dx))
,adj=c(0,1) )
> h=function(x){(5+exp(x)+(2.5)^x*sin(2*pi*x))/3000}
> MidBox(5,11,20,h)

```



We simplify the for loop by not defining `mid.box.x` and `mid.box.y` and simply imbedding those terms inside the **polygon** function. We redraw the curve with `lty=3`, dotted line, so we see the function in places where the boxes covered the original drawing. We use **par** and **text** as before to place the sum of the boxes on the graph, but adjust the location a bit so that there is some space between the text and the frame. Note that when you type multiple lines in the editor and then send them to the console, you get the + signs. The line within `MidBox` without a + signs was wrapped to fit the page but is one line in the editor. Finally, we let `h` be our function from the introduction and run `MidBox`, which produces the graphical output.

7.2 Numerical Integration

We now turn our attention to numerical integration. Our first example defines the function $f(x) = x^2$. We use **integrate(f,0,2)** to compute the definite integral of f from 0 to 2, or $\int_0^2 f(x)dx$.

R Code

```
> f=function(x){x^2}
> integrate(f,0,2)
```

2.666667 with absolute error < 3e-14

We can use **integrate** for a built-in function such as sin, noting that pi is the value of π . For this example we expect the answer to be 0, which we basically have when we consider the reported absolute error, which emphasizes the point that this is numerical, not exact, integration.

R Code

```
> integrate(sin,0, 2*pi)
```

2.221482e-16 with absolute error < 4.4e-14

R can handle improper integrals; the notation for infinity is Inf. Our next example defines the function $g(x) = e^{-x}$ followed by three integrals. In each case we get the expected result.

R Code

```
> g=function(x){exp(-x)}
> integrate(g,0,1)
```

0.6321206 with absolute error < 7e-15

R Code

```
> integrate(g,0,Inf)
1 with absolute error < 5.7e-05
```

R Code

```
> integrate(g,-Inf,Inf)
```

```
Error in integrate(g,-Inf,Inf):non-finite function value
```

There is one caveat with improper integrals. You should not put in a “large” value for an endpoint when you really want Inf. The next example shows what can happen and the reader might ponder why.

R Code

```
> g=function(x){exp(-x)}
> integrate(g,0,20000)
```

```
2.170732e-08 with absolute error < 4.3e-08
```

7.2.1 Numerical Integration of Iterated Integrals

Calculating an iterated integral with R necessitates the use of **sapply**, so we begin with two examples illustrating **sapply**. The first example defines *y* to be the sequence of integers from 0 to 10. For each value of *y*, **sapply** will calculate *y*² and return the list of results.

R Code

```
> y=0:10
> sapply(y,function(y){y^2})
```

```
[1] 0 1 4 9 16 25 36 49 64 81 100
```

In the next example, *y* is the sequence of integers from 0 to 10, each divided by 10. In other words, $\{0, 1/10, 2/10, \dots, 1\}$. For each value of *y*, **sapply** will integrate $x * y$ from 0 to 2 and return the result. Note that we needed to append \$value after the **integrate** function to capture the value without the error term. At this point we have calculated values of the inside integral of $\int_0^1 \int_0^2 xy \, dx \, dy$, for the given values of *y*. In this particular case, we have values of the function $2y$ from the *dx* integral.

R Code

```
> y=0:10/10
> sapply(y,function(y){integrate(function(x){x*y},0,2)
$value})
```

```
[1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
```

To complete the calculation, we simply wrap an integral around the function of *y* created from the **sapply** example, which is the next example. Ba-

sically, to calculate the outside integral a list of values will be sent to the function with **sapply** to obtain the list of numerical values needed to numerically calculate the integral.

R Code

```
> integrate(function(z){sapply(z, function(y) {
+ integrate(function(x){x*y},0,2)$value
+ })},0,1)

1 with absolute error < 1.1e-14
```

We finish this section with two more examples. The first calculates $\int_1^3 \int_1^2 (4 + x - y) dx dy$ and the second $\int_{-1}^1 \int_0^{\pi/2} xe^y \sin(x) dx dy$.

R Code

```
> integrate(function(z){sapply(z,function(y){
+ integrate(function(x){4+x-y},1,2)$value
+ })},1,3)

7 with absolute error < 7.8e-14
```

R Code

```
> integrate(function(z){sapply(z,function(y){
+ integrate(function(x){x*exp(y)*sin(x)},0,pi/2)$value
+ })},-1, 1)

2.350402 with absolute error < 2.6e-14
```

7.3 Area Between Two Curves

A classic calculus problem is finding the area bounded by two curves. In this example, we find the area bounded by the curves $f(x) = (10x^2 - 1)e^{-x}$ and $g(x) = x^3$ and create a graph with the bounded area shaded and text displaying the area.

We begin by loading the **rootSolve** package, since we will need to find the intersection of the functions, and defining the functions **f** and **g**. We then define the function **h** as the difference of **f** and **g**. We use **uniroot.all**, from the **rootSolve** package, to find the roots of **h**, the intersection of the functions **f** and **g**. The second argument in **uniroot.all**, **c(-1,2)**, sets the bounds so that

all roots will be found between -1 and 2 . This is required as bounds must be entered. In this case, previous exploration of the functions allowed us to set fairly tight bounds on the search for roots. The variable `roots` is our list of three intersection points.

R Code

```
> library(rootSolve)
> f=function(x){(10*x^2-1)*exp(-x)}
> g=function(x){x^3}
> h=function(x){f(x)-g(x)}
> roots=uniroot.all(h,c(-1,2))
> roots
```

[1] -0.3126725 0.3235736 1.7238132

We calculate our areas by integrating `h` from the first root, `roots[1]`, to the second root, `roots[2]`. Note that you cannot integrate `f-g` as `integrate` must have one function of one variable, not a combination of functions, although it could be done in one line as `integrate(function(x)f(x)-g(x),1,2)`. In order to calculate the total area we use `$value` for the value from `integrate`, omitting the absolute error part, and subtracting the first area since the integral was negative. We could have also used `abs(integrate(h,roots[1],roots[2])$value)` to get the absolute value of the resulting integral or `integrate(function(x)abs(h(x)), roots[1],roots[3])` for the total area in one line.

R Code

```
> integrate(h,roots[1],roots[2])
> integrate(h,roots[2],roots[3])
> integrate(h,roots[2],roots[3])$value-
integrate(h,roots[1],roots[2])$value

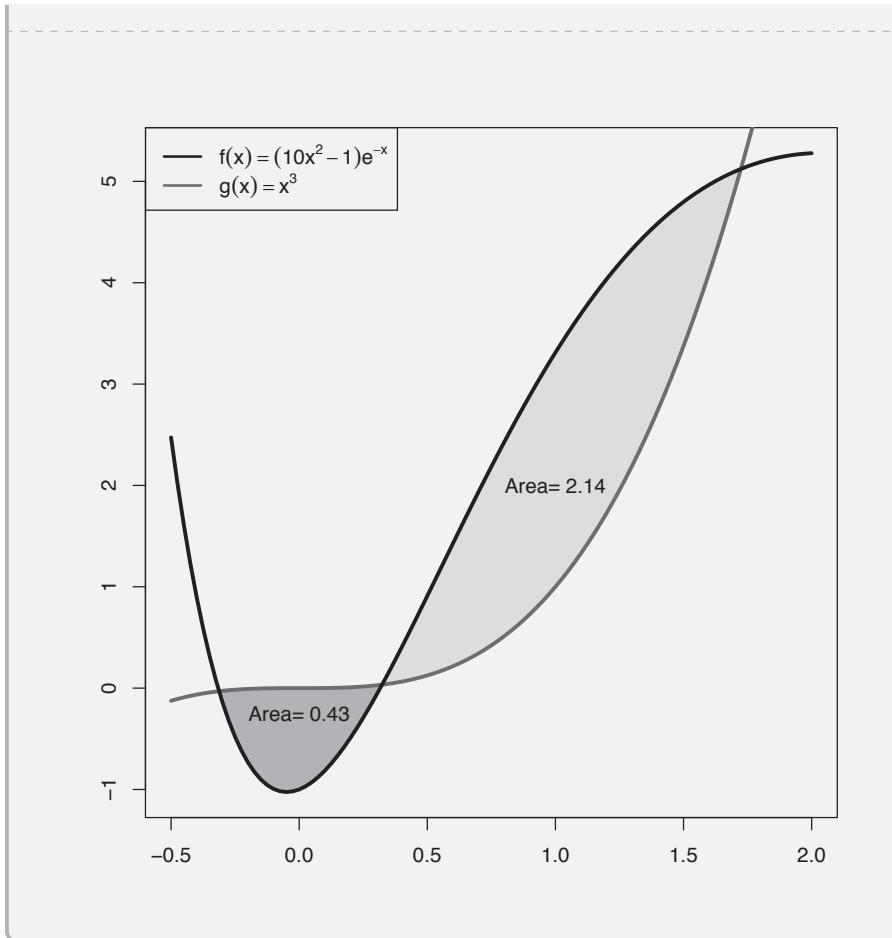
> integrate(h,roots[1],roots[2])
-0.4260403 with absolute error < 4.7e-15
> integrate(h,roots[2],roots[3])
2.14281 with absolute error < 2.4e-14
> integrate(h,roots[2],roots[3])$value-
integrate(h,roots[1],roots[2])$value
[1] 2.56885
```

We now turn to creating a graph to illustrate our results. Our first line sets our margins and then we plot the function `f`. The x -axis is set with `xlim=c(-0.5,2)` and we let the y -axis be set by default. The line width, `lwd`, is set to 2 and the x and y labels are set blank by using empty quotes. The function `g` is added to the graph with `curve` using the same x -axis and `lwd`,

but coloring the curve red. We have `add=TRUE` to add this to the previous plot; otherwise it will create a new plot. A legend is added to the top left of the graph with four parts defining the elements of **legend**. The first part provides the two function names using **expression** for mathematical expressions. The last three are the line types with 1 being a solid line, the line width used, and the color. In each case **c(,)** is used because there are two elements in the legend, one for each function.

R Code

```
> par(mar=c(4,4,3,3))
> plot(f,xlim=c(-.5,2),lwd=2,xlab="",ylab="")
> curve(g,-.5,3,lwd=2,col="red",add=TRUE)
> legend("topleft",c(expression(f(x)==(10*x^2-1)*e^-x),
expression(g(x)==x^3)),lty=c(1,1),lwd=c(2,2),
col=c("black","red"))
> xlist1=seq(roots[1],roots[2],0.01)
> xlist2=seq(roots[2],roots[3],0.01)
> polygon(c(xlist1,rev(xlist1)),c(g(xlist1),
f(rev(xlist1))),col="gray65")
> polygon(c(xlist2,rev(xlist2)),c(f(xlist2),
g(rev(xlist2))),col="gray85")
> curve(g,-.5,2,lwd=3,col="red",add=TRUE)
> curve(f,-.5,2,lwd=3,col="black",add=TRUE)
> text(0,-0.25,paste("Area=",
-round(integrate(h,roots[1],roots[2])$value,2)))
> text(1,2,paste("Area=",
round(integrate(h,roots[2],roots[3])$value,2)))
```



We will use **polygon** to shade the enclosed area. To do this we need to define the shape of each area with points. We define `xlist1` to be a sequence of points from the first root, `roots[1]`, to the second root, `roots[2]`, in steps of 0.01. Similarly, `xlist2` defines a list of points from the second to the third root. The input to **polygon** is the x -values and then the y -values of the points that will start at the first intersection point of the function, follow along the function **g** to the second intersection point, and then back to the first intersection point by way of **f**. We do this by combining `xlist1` with its reverse, **rev(xlist1)**, for our x -values. Our y -values combine the points of **g(xlist1)**, evaluating each value in `xlist1` with **g**, with **f(rev(xlist1))**, evaluating each value of `xlist1` with **f** but in reverse order. We color the polygon with gray65. We could have defined a border color with `border="some color"`, but it is not necessary here. The second use of **polygon** shades the second intersected area in the same way, but we shade with a lighter gray, gray85.

Next, we redraw both the **g** and **f** curves. The uses of **polygon** leaves some

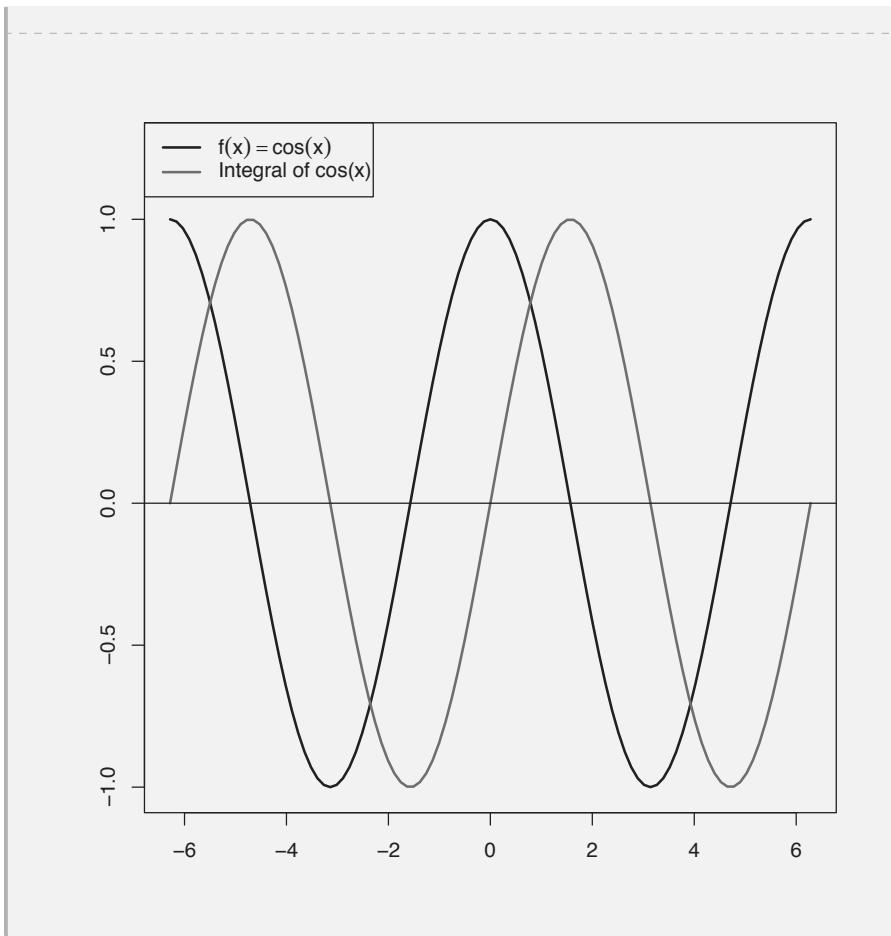
jagged edges along the curves. The commands are executed and layered in the order they are listed. The redrawing in of the two curves creates a much better looking graph. Our last two commands, **text**, places text on the graph. The first two values are the location of the text. The **paste** function is used to combine text, **Area=**, with values where **round** rounds the value of **integrate** to two decimal places. Note that **\$value** was appended to **integrate** to get the value without the error term.

7.4 Graphing an Antiderivative

Our last example graphs the antiderivative of a function, in this case $\cos(x)$. We start by defining the function **f** to be $\cos(x)$. The function **g** is then defined as the integral of **f**, or $\int_0^x \cos(t)dt$. Note that we append **\$value** to the **integrate** so that it returns the value without the error term. At this point, we can evaluate **g**; for example, **g(2)** returns 0.9092974, but for technical reasons **g** is not an object that can be graphed. The next line of the code deals with this technicality by letting **h** be the **Vectorize** of **g**.

R Code

```
> f=function(x){cos(x)}
> g=function(x){integrate(f,0,x)$value}
> h=Vectorize(g)
> par(mar=c(4,4,3,3))
> plot(f ,xlim=c(-2*pi,2*pi),ylim=c(-1,1.25),lwd=2,
+ xlab="",ylab="")
> curve(h(x),-2*pi ,2*pi,lwd=2,col="red",add=TRUE)
> abline(h=0,lwd=1)
> legend("topleft",lty=c(1,1),lwd=c(2,2),
+ c(expression(f(x)==cos(x))),paste("Integral of",
+ expression(cos(x)))),col=c("black","red"))
```



We begin our graph by setting the margins and then we plot **f**. We set the x -axis from -2π to 2π and the y -axis from -1 to 1.25 . In this case, we set the y -axis to 1.25 instead of 1 to create space for the legend. The line width is set to 2 and both axis labels are set to blank. We add the graph of **h** to the plot with **curve**, setting `add=TRUE` and using the same x -axis limits. The line width is again 2 and we color the graph red. For reference we added a horizontal line at $y = 0$ with **abline**. Lastly, we added the **legend** to the “topleft” of the graph. The line types are both 1 , `lty=c(1,1)`, and the line widths are both 2 , `lwd=c(2,2)`. We use **expression** to create the math text for the function and then **paste** to append “Integral of” with the expression for $\cos(x)$. The last argument sets the color for each graph in the legend.

Code Review from Riemann Sums and Integration Chapter

integrate(f,a,b) returns the $\int_a^b f(x)dx$ and an absolute error.

integrate(f,a,b)\$value returns the $\int_a^b f(x)dx$, without the absolute error term. This is necessary for using the integral in further calculations.

polygon(xlist,ylist,col="color1",border="color2") shades the polygon defined by the list of (x, y) values with color1 and colors the border with color2.

sapply(xlist,function(x){expression}) evaluates the function for each value of xlist and returns the list of values.

segments(x_1, y_1, x_2, y_2) draws a line segment from (x_1, y_1) to (x_2, y_2) .

seq(a,b,dx) returns a list of numbers starting at a, increasing by steps of dx, and stopping so that the list does not exceed b.

sum(list) returns the sum of a list of values.

7.5 Exercises

1. Calculate the right and left box sums for the first example.
2. Create a function that accepts a function, left endpoint, right endpoint, and number of boxes, that will return the Riemann sum using mid point boxes. Check this with the first example.
3. Create functions LeftBox and RightBox, similar to MidBox.
4. Find the area bounded by the function e^x and $x^5(1 + \cos(x))$ for $x > 3$. Create a graph with the bounded areas shaded and labelled.
5. Calculate the triple integral $\int_0^1 \int_0^1 \int_0^1 xyz \, dx \, dy \, dz$.
6. Graph the antiderivative of the function in the introduction.

8

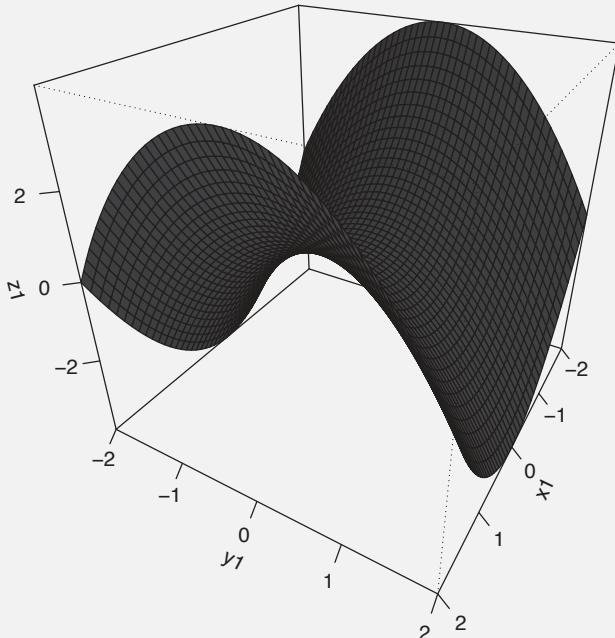
Planes, Surfaces, Rotations, and Solids

Base R has limited ability in graphing three-dimensional objects. If all we need to do is to plot one surface or create a contour plot, base R will do. If, on the other hand, we want to plot a surface and a tangent plane or illustrate estimating volume with disks, we will need a package. In using the rgl package we will get the added feature of interactive graphs that can be rotated. and we also add the ability to zoom in and out.

We begin with a surface plot by defining the saddle function $x^2 - y^2$. In order to create a graph we need a set of x , y , and z points. We use **seq** to generate a set of 50 points from -2 to 2 , setting the result to both x and y at the same time in one line. The values of z must be a matrix, and there is an easy way to create this matrix, **outer**. The function **outer** will evaluate **saddle** at all pairs x and y , and return a matrix where the (i, j) value is the result of **saddle(x_i, y_i)**. The function **persp** creates the surface plot given the vectors x and y and the matrix z . The values for theta and phi provide the viewing angle, where theta rotates in the xy plane and phi in the z direction. The graph is colored blue and shade is set to 0.2 , where 0 is no shading and 1 will shade black in the shade of the light source. Ticktype set to “detailed” draws ticks similar to two-dimensional graphs. The other option for ticktype is “simple”. Options for **persp** include **xlab**, **ylab**, and **zlab**, for labeling the axis, and **main** for a title. Axis ranges can be set with **xlim**, **ylim**, and **zlim**. There is an option **r** to set the distance of the eye point to the center of the box and expand, 0 to 1 , scales down the plotting in the z direction.

R Code

```
> saddle=function(x,y){x^2-y^2}
> x=y=seq(-2,2,length=50)
> z=outer(x,y,saddle)
> persp(x,y,z,theta=120,phi=30,col="blue",shade=0.2,
+ ticktype="detailed")
```

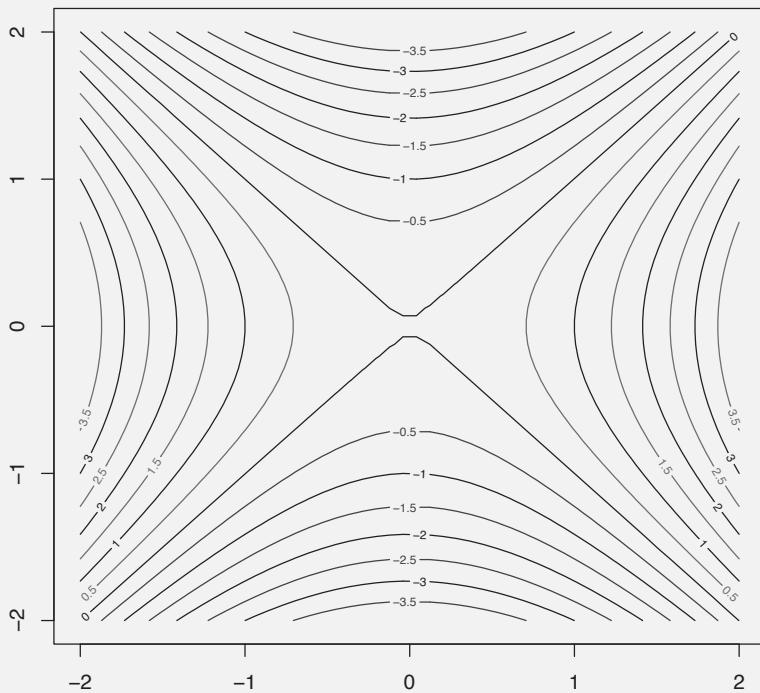


Our next example of base R graphics is a contour plot. We will use `x`, `y`, and `z`, as defined in the previous example. Our first use of `contour` creates the black contour lines by default and we create a title with `main`. In the next two uses, we demonstrate how contour lines can be defined with the `level` option. In the first case we add the blue contour lines and in the second we add the red lines. Note that the option `add=TRUE` adds to the graph; otherwise a new graph is created. There is also the `nlevels` option, which is set to the number of levels desired and cannot be used with `levels`. In deciding on the contour lines `min(z)` and `max(z)` may be helpful. The `labels` option can be used to label the contour lines and is set to a vector equal to the number of levels. The line type and line width for the contour lines can be set with `lty` and `lwd`, respectively.

R Code

```
> contour(x,y,z,main="Contour Plot for the Saddle")
> contour(x,y,z,levels=c(0.5,1.5,2.5,3.5),col="red",
add=TRUE)
> contour(x,y,z,levels=c(-0.5,-1.5,-2.5,-3.5),
col="blue",add=TRUE)
```

Contour Plot for the Saddle



The graphs for the remainder of this chapter use the `rgl` package. The package produces interactive graphs, meaning that they can be rotated in any direction and we can zoom in and out.

8.1 Interactive: Surface Plots

Our first example of a surface plot with rgl is the same saddle as above but we add a tangent plane at $(2, 1, 3)$. We begin by loading the package with **library** and opening a graph window with **open3d**. Note the graph windows are different than the windows from using **persp** or **contour** above. We don't need to use **open3d** unless we want more than one graph viewable at a time as **persp3d** will open the window without using **open3d**. The function **saddle** is defined using **function**. As with **persp** we need two vectors and a matrix of values to create the plot. The **seq** function returns a vector of 100 points from -4 to 4 and is set to x and y . The function **outer** will evaluate **saddle** at all pairs x and y , and returns a matrix where the (i, j) value is the result of **saddle(x_i, y_i)**.

The first two arguments of **persp3d** are the x and y vector of points, while the third is the matrix of values returned by the function **outer**; in this case z . The surface is colored blue and the z range is defined by **zlim**. We use the default ranges for x and y as they are naturally set when x and y are defined, but we still have the options **xlim** and **ylim**. The option **front** set to "lines" has the front side of the saddle drawn in as a wire frame and allows us to see through the frame. The default is "filled", which creates a solid surface, and we allow that for the back. This creates a surface that is solid on one side and a wire frame on the other. We label the axis with **xlab**, **ylab**, and **zlab**. The graph that is created can be rotated and zoomed and has a "light source" that can be "turned off" with **lit=FALSE**.

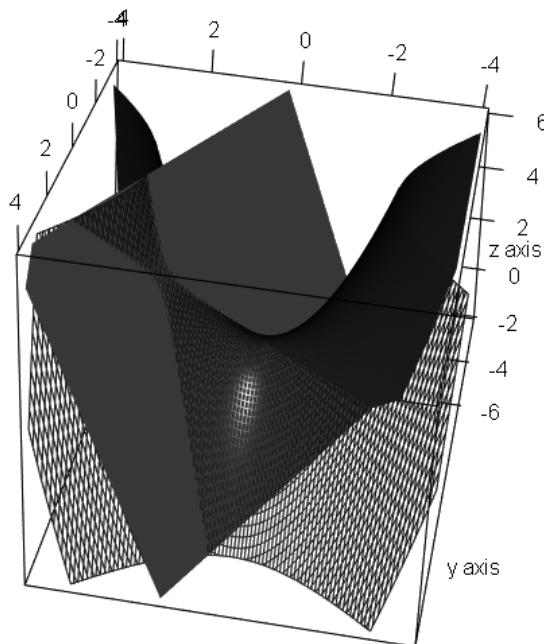
The function **planes3d** graphs a plane with the parametrization $ax + by + cz + d = 0$ with the first four arguments the values of a , b , c , and d . The values $4, -2, -1$ and -3 create the tangent plane to the saddle at $(2, 1, 3)$. The plane is colored red. When rotating the graph the axis tick marks and labels will automatically switch sides. In the example here the x -axis label is still in the back and hidden.

We saved the graph using the code **rgl.snapshot("fileName.png", fmt="png", top=TRUE)**. The option **top=TRUE** ensures that the snapshot contains only the graphing window. There is also the function **rgl.postscript("fileName.pdf", fmt="pdf", drawText=FALSE)** that saves the image as pdf file and also supports the formats ps, eps, tex, svg, and pgf. Note that the files are saved to the directory that R is set to.

R Code

```
> library(rgl)
> open3d()
> saddle=function(x,y){x^2-y^2}
> x=y=seq(-4,4,length=100)
```

```
> z=outer(x,y,saddle)
> persp3d(x,y,z,col="blue",zlim=c(-6,6),front="lines",
xlab="x axis",ylab="y axis",zlab="z axis")
> planes3d(4,-2,-1,-3,col="red")
```

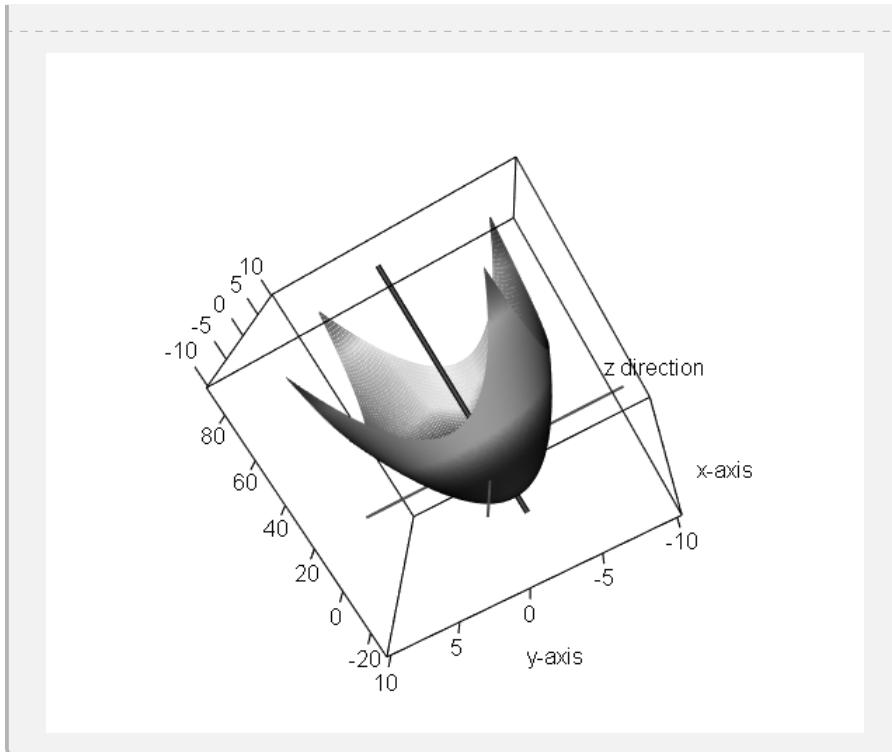


The next example graphs a paraboloid with two added features, coloring depending on heights and axis centered on the origin. We define the two variable function `parab` to be $x^2 + y^2 - 20$. We create a set of points to plot by setting `x` and `y` to be a sequence of values from -7 to 7 of length 100 and then use `outer` to create the matrix `z` as in the previous example. The next three lines set up the coloring scheme. We will use 100 colors set to the variable `z.col`. The function `rainbow` generates colors along the spectrum of a rainbow. The first argument is the number of colors. The second and third arguments are options and are used to set the range. We start at 0, red, and stop at $5/6$, which is magenta. If we stop at 1 we get back to red. If we stop at say $4/6$, we don't get enough in the blue spectrum. The function `rev` reverses a list of values. In this case we want red to be at the end of the list because it will be associated with larger `z` values. This list is set to `colors`. (See [Section 6.4](#) for creating user defined color palettes.) The last step for coloring uses the `cut` function to partition the `z` values into 100, `n.col`, levels, with the result set to `z.col`. The paraboloid is graphed with `persp3d` using the `x`, `y`, and `z`

values with col set to colors[z.col] which will associate the z.col cuts to the list of z.col colors. The ranges for the axis are set with xlim, ylim, and zlim and are larger than the function values for better perspective. The option front set to lines creates a grid on the front part of the graph. We add axes by setting axes=TRUE and label the axis with xlab, ylab, and zlab. We add axis lines with the **abclines3d** function. The first three values are a point and the fourth vectors defining the direction of the lines. Here **diag(3)** is the identity matrix, which provides directions for each axis. For example, **abclines3d(0, 0, 0, c(0,0,1))** would add lines from the origin in the z direction only. The last two arguments define the color and line width. Note that there is an **axis3d()** function for more control of tick mark and labels.

R Code

```
> parab=function(x,y){x^2+y^2-20}
> x=y=seq(-7,7,length=100)
> z=outer(x,y,parab)
> n.col=100
> colors=rev(rainbow(n.col,start=0,end=5/6))
> z.col=cut(z,n.col)
> persp3d(x,y,z,col=colors[z.col],xlim=c(-10,10),
ylim=c(-10,10),zlim=c(-30,90),front="lines",axes=TRUE,
xlab="x-axis",ylab="y-axis",zlab="z direction")
> abclines3d(0,0,0,diag(3),col="gray40",lwd=2)
```



8.2 Interactive: Rotations around the x -axis

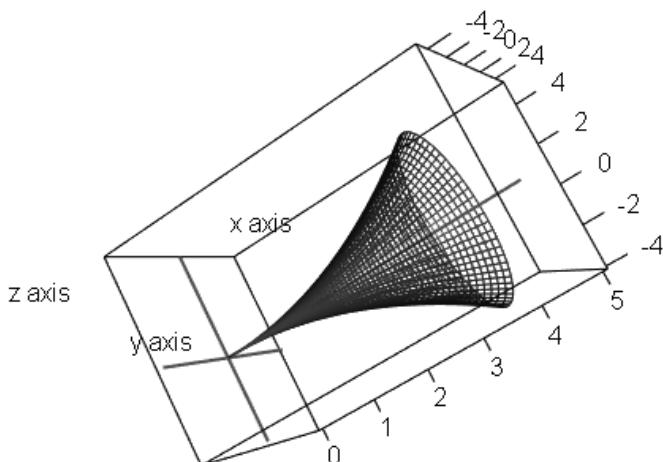
The two examples in this section will illustrate the **turn3d** function in the **rgl** package. Note that objects can only be turned, or rotated, around the x -axis. The first example will rotate a curve around the x -axis and add four discs to estimate the volume. The second will rotate the area bounded by two curves around the x -axis.

First, we will rotate $y = x^2$ from 0 to 4 around the x -axis. We will then add four discs of width 1 to the graph. The **persp3d** function creates a nice set of axes and the first line takes advantage of this by graphing an object outside the desired ranges so that the axes are created. The values for x and y must be at least two points, and so we use two, which then means that z must be a 2×2 matrix. For both x and y we use the two points 98 and 99, which are outside our range of interest for this example. Any matrix will do for z and so we use the identity matrix created by **diag(2)**. We set the axes ranges with **xlim**, **ylim**, and **zlim**. We set an aspect of $(3, 1, 1)$ for (x, y, z) to stretch out the function along the x -axis for better visualization. The axes

are labelled with `xlab`, `ylab`, and `zlab`. We add axis lines with the **`abclines3d`** function. The first three values are a point and the fourth is a set of vectors defining the direction to add lines. Here **`diag(3)`** is the identity matrix, which provides directions for each axis. To generate the curve to be rotated we need a set of x and y points. The values for x are returned from `seq`: fifty points from 0 to 4. We define y by squaring and dividing by 4 every value in x . The **`turn3d`** function rotates the values around the x -axis, where n is the number of polygons used to approximate the rotation and `smooth` set is to TRUE. To display the image we use **`wire3d`** and color the object blue. We can also use **`shade3d`**, which will create solid walls.

R Code

```
> persp3d(c(98,99),c(98,99),diag(2),xlim=c(0,5),
  ylim=c(-5,5),zlim=c(-5,5),aspect=c(3,1,1),
  xlab="x axis",ylab="y axis",zlab="z axis")
> abclines3d(0,0,0,diag(3),col="gray40",lwd=2)
> x=seq(0,4,length=50)
> y=x^2/4
> wire3d(turn3d(x,y,n=60,smooth=TRUE),col="blue")
```

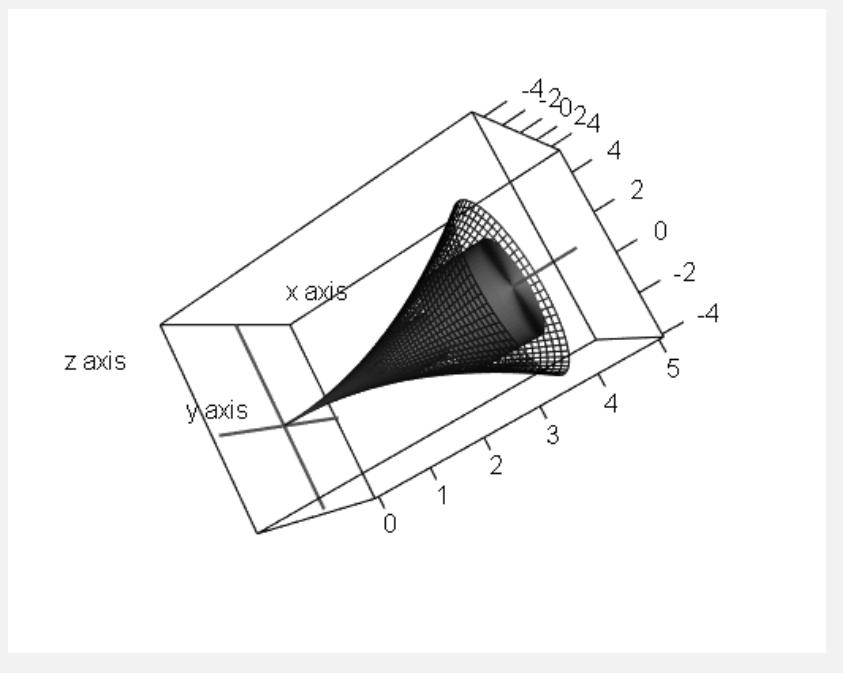


We now add four cylinders based on left-hand endpoints of unit width to the graph. To do this, each cylinder will need four points to define the region to be rotated. We begin by defining f to be $x^2/4$. This was not done in the

first part as it was unnecessary, but here it will help clarify the code. Since these are “left cylinders” we can start with the one from 1 to 2. We define the four points 1,1,2, and 2, and y_1 to be 0, $f(1)$, $f(1)$, and 0, which define the corners of the box to be rotated. We use **turnd3d** for the rotation and display it with **shade3d**, coloring the cylinder red. The other two cylinders are created in the same manner. Note that **points3d**, **lines3d**, and **segments3d** exist to add points, lines, and line segments to plots.

R Code

```
> f=function(x){x^2/4}
> x1=c(1,1,2,2)
> y1=c(0,f(1),f(1),0)
> shade3d(turn3d(x1,y1,n=60),col="red")
> x2=c(2,2,3,3)
> y2=c(0,f(2),f(2),0)
> shade3d(turn3d(x2,y2,n=60),col="red")
> x3=c(3,3,4,4)
> y3=c(0,f(3),f(3),0)
> shade3d(turn3d(x3,y3,n=60),col="red")
```

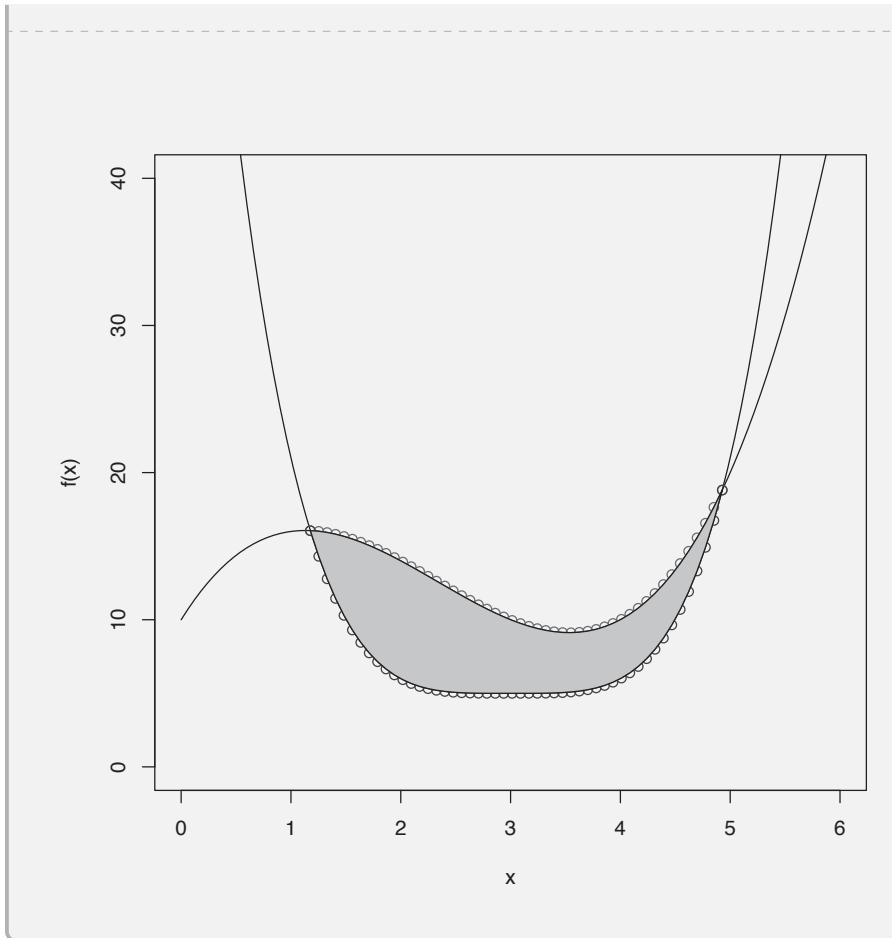


We now move to rotating the area bounded by two curves around the x -axis. We consider the two functions $f(x) = x^3 - 7x^2 + 12x + 10$ and $g(x) = (x - 3)^4 + 5$, defined in the first two lines. The third and fourth

lines graph **f** from 0 to 6 with y ranging from 0 to 40 and adds **g** to the plot with `add=TRUE`. We need to find the intersection points of the two function, so we load the `rootSolve` package. The function **h** is defined as the difference of **f** and **g**, and roots is set to the roots of **h** on the interval from 0 to 10 with the function **uniroot.all**. We now create a set of points to define the boundary of the intersected region. The variable **x.int** is a sequences of 50 points of x -values from the first root to the second root. The variables **y.f** and **y.g** are the corresponding y -values. The points along **f** and **g** are plotted with **points** and colored red and blue, respectively. In order to shade a region we need a set of points that follow the region clockwise or counterclockwise. We define **x.rot** to be the set of points of **x.int** and the same set in reverse. Similarly **y.rot** is the y -values of **y.f** followed by the values along **g** in reverse, `rev(y.g)`. The function **polygon** shades the region defined by the values of x and y given in the first two arguments and colored gray. This provides a nice visual of the region that will be rotated.

R Code

```
> f=function(x){x^3-7*x^2+12*x+10}
> g=function(x){(x-3)^4+5}
> curve(f,0,6,ylim=c(0,40))
> curve(g,0,6,add=TRUE)
> library(rootSolve)
> h=function(x){f(x)-g(x)}
> roots=uniroot.all(h,c(0,10))
> x.int=seq(roots[1],roots[2],length=50)
> y.f=f(x.int)
> y.g=g(x.int)
> points(x.int,y.f,col="red")
> points(x.int,y.g,col="blue")
> x.rot=c(x.int,rev(x.int))
> y.rot=c(y.f,rev(y.g))
> polygon(x.rot,y.rot,col="gray")
```



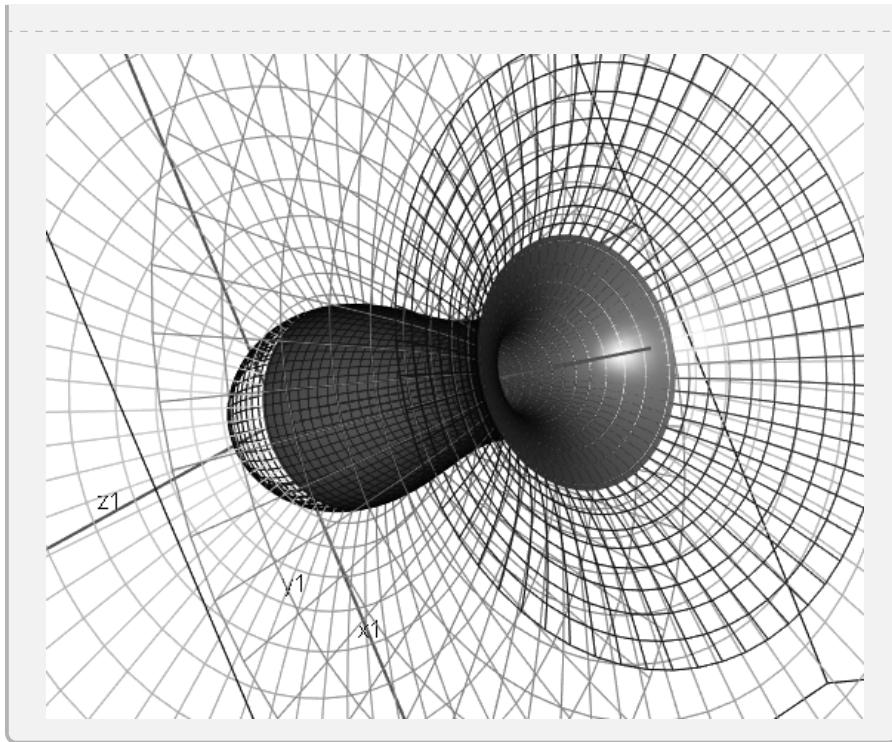
Our goal is to rotate the shaded region around the x -axis. This visual will proceed in four parts. We will graph a “dummy” function because **persp3d** creates a nice set of axes; then we rotate **f** and **g** separately. The last part will rotate the shaded region as defined by $x.\text{rot}$ and $y.\text{rot}$. We start by setting the aspect to 15, 1, and 1 for x , y , and z , respectively with **aspect3d**. The next four lines create the “dummy” function. We set $x1$ and $y1$ both to be the two points 0 and 1. We define the constant function as $0x + 0y - 99$ and use **outer** to create the matrix $z1$, which will be a two-by-two matrix with the values -99. We plot $x1, y1, z1$ with **persp3d**, set the ranges for the axes, include the axes with $\text{axes}=\text{TRUE}$, and set $\text{aspect}=\text{FALSE}$ since it has already been set. We add axis lines with the **abclines3d** function. The first three values are the coordinates of a point and the fourth is a set of vectors defining the directions to add lines. At this point, we have a box with labelled axes and lines representing the center of the axes.

To add **f** to the graph we let $x2$ be a sequence of 50 points from 0 to 6.

Set y_2 to the function values of x_2 , or $f(x_2)$. The function **turn3d** rotates the points and **wire3d** graphs the object as a wire mesh and colored blue. Similarly, we define y_3 to be $g(x_2)$ and graph the rotated object as a wire mesh, but colored green. Our last line rotates the points defined by $x.\text{rot}$ and $y.\text{rot}$ and this time we display the object with **shade3d** colored red. The graph here was rotated and zoomed before saving with **rgl.snapshot("filename.png", fmt="png", top=TRUE)**.

R Code

```
> aspect3d(15,1,1)
> x1=y1=c(0,1)
> constant=function(x,y){0*x+0*y-99}
> z1=outer(x1,y1,constant)
> persp3d(x1,y1,z1,xlim=c(0,6),ylim=c(-20,20),
zlim=c(-20,20),axes=TRUE,aspect=FALSE)
> abclines3d(0,0,0,a=diag(3),col="gray40",lwd=2)
> x2=seq(0,6,length=50)
> y2=f(x2)
> wire3d(turn3d(x2,y2,n=60,smooth=TRUE),col="blue")
> y3=g(x2)
> wire3d(turn3d(x2,y3,n=60,smooth=TRUE),col="green")
> shade3d(turn3d(x.rot,y.rot,n=60,smooth=TRUE),
col="red")
```



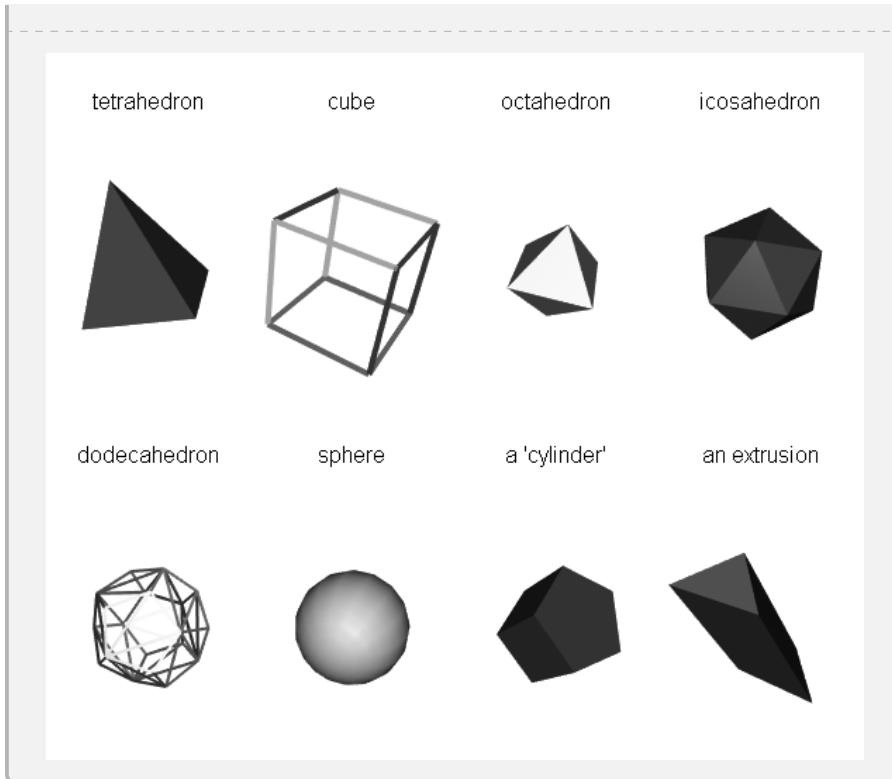
8.3 Interactive: Geometric Solids

Our last example exhibits some of the predefined shapes with the `rgl` package and this example is similar to the one given by Duncan Murdoch [21]. We begin by setting `cols` to eight colors from `rainbow`, which will be used to color our solids, and then set our layout to be a four-by-four matrix with the heights of the rows set to 1, 3, 1, and 3. This creates a 4-by-4 layout with the first and third row with height 1 and the second and fourth row with height 3. The code first places text and then a shape, with `next3d` advancing to the next location in the matrix. The first three values are coordinates for the text in the context of the cell in the matrix. In two cases we use `wire3d` instead of `shade3d` for a wire frame instead of a solid object, in which case we set the line width with `lwd`. Note that the objects are added by columns to the graph frame. Two shapes need some explanation. The ‘cylinder’ has four arguments. The first is a matrix to define the local coordinates of the space curve. Here we simply use the identity matrix given by `diag(3)`. The second and third are the radius and number of sides. For the last we note that if closed is 0

then the ends are open, while -1 and -2 will cap one or both sides. The last object, the “extrusion,” takes the shape defined by the points x and y and creates a solid cylindrical object. Note that we can use **plot(x,y,type="p")** and **polygon(x,y)** to see the shape in the xy plane. Note also that in the layout all eight objects can be manipulated.

R Code

```
> cols=rainbow(8)
> layout3d(matrix(1:16, 4,4),heights=c(1,3,1,3))
> text3d(0,0,0,"tetrahedron"); next3d()
> shade3d(tetrahedron3d(col=cols[1])); next3d()
> text3d(0,0,0,"dodecahedron"); next3d()
> wire3d(dodecahedron3d(col=cols[2]),lwd=2); next3d()
> text3d(0,0,0,"cube"); next3d()
> wire3d(cube3d(col=cols[3]),lwd=4); next3d()
> text3d(0,0,0,"sphere"); next3d()
> spheres3d(0,y=NULL,z=NULL, radius=3,col=cols[4]);
next3d()
> text3d(0,0,0,"octahedron"); next3d()
> shade3d(octahedron3d(col=cols[5])); next3d()
> text3d(0,0,0,"a 'cylinder'"); next3d()
> shade3d(cylinder3d(diag(3),radius=1,sides=5,
closed=-1),col=cols[6]); next3d()
> text3d(0,0,0,"icosahedron"); next3d()
> shade3d(icosahedron3d(col=cols[7])); next3d()
> text3d(0,0,0,"an extrusion"); next3d()
> x=c(0,1,2,0)
> y=c(0,1,0,0)
> shade3d(extrude3d(x,y,thickness=3),col=cols[8])
```



Code Review for the Planes Surfaces Rotations and Solids Chapter

abclines3d(x, y, z, a, b, c) adds vectors to a three-dimensional plot where x , y , and z are coordinates of the points in the direction of vectors given by a , b , c . The vectors may be given as one matrix. Requires rgl package.

contour(v_x, v_y, A) returns a contour plot given by the vector of points v_x and v_y and the matrix A where the (i, j) component is the z value corresponding to x_i and y_i of the vectors v_x and v_y .

outer($v_x, v_y, f(x, y)$) returns a matrix where the (i, j) component is the value of $f(x_i, y_i)$ where x_i and y_i are the i th and j th elements of the vectors v_x and v_y .

persp(v_x, v_y, A) returns a surface plot given by the vector of points v_x and v_y and the matrix A where the (i, j) component is the z value corresponding to x_i and y_i of the vectors v_x and v_y .

persp3d(v_x, v_y, A) returns an interactive surface plot given by the vec-

tor of points v_x and v_y and the matrix A where the (i, j) component is the z value corresponding to x_i and y_j of the vectors v_x and v_y .

planes3d(a,b,c,d) graphs a plane parameterized as $ax + by + cz + d = 0$. Requires rgl package.

shade3d(shape**)** creates a solid graph of a three-dimensional shape, such as that created by turnd3d(). Requires rgl package.

turn3d(v_x,v_y) generates a three-dimensional object by rotating the figure given by the points of the vectors v_x and v_y . Requires rgl package.

wire3d(shape**)** creates a wire mesh graph of a three-dimensional shape, such as that created by turnd3d(). Requires rgl package.

8.4 Exercises

1. Create a basic (use persp()) surface plot of $f(x, y) = \sin(x) - x \cos(y)$ for x and y between 0 and π .
2. Create a contour plot of $f(x, y) = \sin(x) - x \cos(y)$ for x and y between 0 and π . Use at least two colors for the contour lines and at least two line types.
3. Create an interactive surface plot of $f(x, y) = \sin(x) - x \cos(y)$ for x and y between 0 and π along with a tangent plane at $(\pi/2, \pi/2)$.
4. Modify the example of rotating $y = x^2/4$ around the x -axis with four cylinders by using 6 cylinders. Alternate the colors of the cylinders.
5. Create an extruded star.

9

Curve Fitting

In this chapter we demonstrate fitting curves to data, in particular, exponential, polynomial, log, logistic, and power curves. The two main functions we use are **lm**, linear model, and **nls**, non-linear least squares. In three of the examples, exponential, polynomial, and logistic, we use the real-world data U.S. solar capacity, Arctic sea ice extent, and Germany solar capacity. In the other two, we illustrate curve fitting by first generate random data. We provide more detail in the first section, Exponential Fit, with the understanding that functions such as **summary** or **names** return similar results in the other sections, but demonstrating this in all sections would be repetitive. Rossiter's *Technical Note: Curve Fitting with the R Environment for Statistical Computing* is a good primer for curve fitting in R [24].

9.1 Exponential Fit

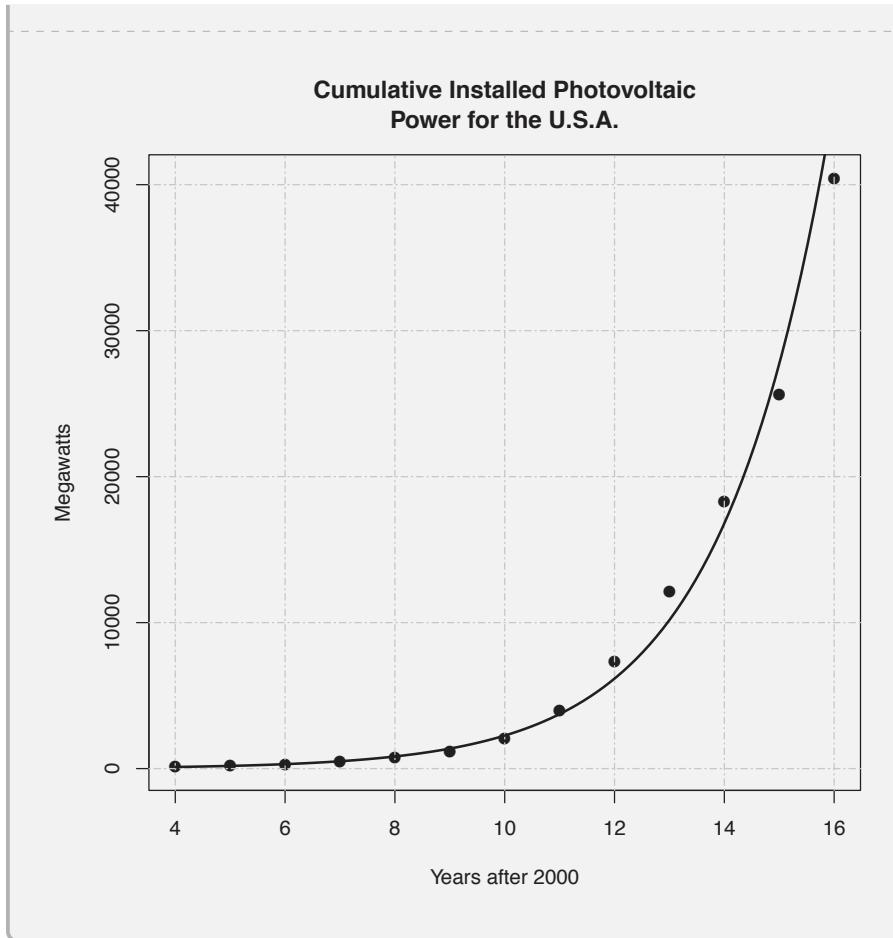
We begin by defining the variables for the data. In many cases this is likely to be achieved by importing the data, see for example [Section 1.1](#), but for simplicity we enter the data here. The independent data is years after 2000 from 2004 through 2016. The dependent data is cumulative installed photovoltaic power in megawatts for the United States of America [13]. In the third line, we plot the data with **plot**. The first two arguments are the independent and dependent data. The point character is set to 16, pch=16, for a solid dot, and scaled to 1.25 times the default size, cex=1.25. The x range is set with xlim, while the y range is set by default. The axes are labeled with xlab and ylab. A title is added with **title**. We can include main as an argument within **plot** for a title, but this can make the inputs to **plot** lengthy and difficult to follow. Note that within **title** \n is used for a line break. A grid is added with **grid**. With NULL as the first two arguments, grid lines are placed at the tick marks set by **plot**. We can also use the number of grid lines desired or NA for none in either of the first two arguments. The line type is set by lty and the lines are colored gray. For greater control of grid lines we can use **abline**.

We use **lm**, linear model, and fit $\ln(y) = ax + b$ with the syntax $\log(\text{Solar.usa}) \sim \text{Years.after.2000}$. The final model is thus given by $y = e^b e^{ax}$. The result of **lm** is stored as Solar.usa.fit. We define the function **Exp.Solar.usa**

noting that `coef(Solar.usa.fit)[[1]]` and `coef(Solar.usa.fit)[[2]]` return the intercept and slope from the linear model. The function `Exp.Solar.usa` is added to the graph from 4 to 16 using `curve`. The line width is set to 2 and `add=TRUE` is necessary so that the function is added to the current graph.

R Code

```
> Years.after.2000=4:16
> Solar.usa=c(119,198,303,463,761,1190,2040,3961,7330,
12106,18317,25600,40436)
> plot(Years.after.2000,Solar.usa,pch=16,cex=1.25,
xlim=c(4,16),xlab="Years after 2000", ylab="Megawatts")
> title(main="Cumulative Installed Photovoltaic\n
Power for the U.S.")
> grid(NULL,NULL,lty=6,col="gray")
> Solar.usa.fit=lm(log(Solar.usa)~ Years.after.2000)
> Exp.Solar.usa=function(x)
{exp(coef(Solar.usa.fit)[[1]])*
exp(coef(Solar.usa.fit)[[2]]*x)}
> curve(Exp.Solar.usa,4,16,lwd=2,add=TRUE)
```



For standard results from a linear regression model we use **summary(Solar.usa.fit)**. This provides us with estimates, standard errors, p-values, R-square, etc.

R Code

```
> summary(Solar.usa.fit)

Call:
lm(formula = log(Solar.usa) ~ Years.after.2000)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.141339 -0.087250 -0.005822  0.069876  0.173753
```

```

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.712571 0.091418 29.67 7.51e-12 ***
Years.after.2000 0.501164 0.008562 58.53 4.47e-15 ***
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 0.1 1

Residual standard error: 0.1155 on 11 degrees of freedom
Multiple R-squared: 0.9968, Adjusted R-squared: 0.9965
F-statistic: 3426 on 1 and 11 DF, p-value: 4.473e-15

```

To see a list of values that can be returned from the model we use **names(Solar.usa.fit)**. For example Solar.usa.fit\$coefficients will return the coefficients of the model, whereas Solar.usa.fit\$coefficients[[1]] will return only the first value in the list of coefficients. This is equivalent to **coef(Solar.usa.fit)[[1]]**, the syntax we used above. Note that, for example, Solar.usa.fit\$coefficients[1] will return a value with a header whereas Solar.usa.fit\$coefficients[[1]] returns just the value. In other words, there is a difference between single and double brackets.

R Code

```

> names(Solar.usa.fit)

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

```

R Code

```

> Solar.usa.fit$coefficients

(Intercept) Years.after.2000
2.712571      0.501164

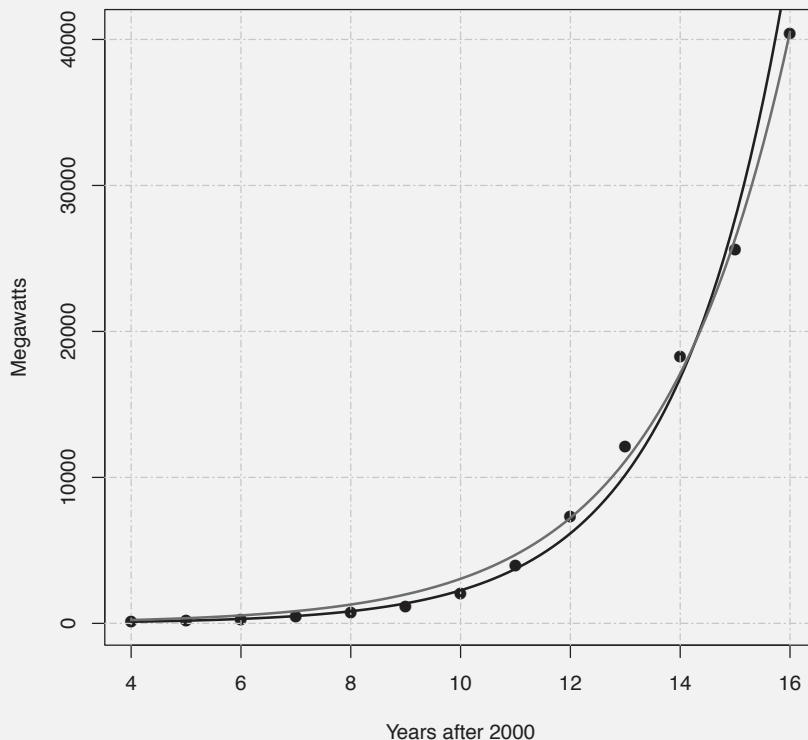
```

We can also use **nls** for the model with the syntax $\text{Solar.usa} \sim P * \exp(r * \text{Years.after.2000})$. In using **nls** we need reasonable starting points for the algorithm and they are defined by $\text{start} = \text{list}(P=1, r=1)$. We leave as an exercise to check that adding a constant parameter to the model is not significant. As above, we define the function **ExpSolar.usa.nls** and add it to the graph using **curve**, with the only difference being that the curve is colored red. We can use **summary(Solar.usa.nls)** and **names(Solar.usa.nls)** to return information similar to that above. Note that the results from **lm** and **nls** are not the same as can be seen in the graph.

R Code

```
> Solar.usa.nls=nls(Solar.usa~  
P*exp(r*Years.after.2000),start=list(P=1,r=1))  
> Exp.Solar.usa.nls=function(x)  
{coef(Solar.usa.nls)[[1]]*  
exp(coef(Solar.usa.nls)[[2]]*x)}  
> curve(Exp.Solar.usa.nls,4,16,add=TRUE,col="red",lwd=2)
```

Cumulative Installed Photo voltaic Power for the U.S.A.



9.2 Polynomial Fit

We demonstrate a polynomial fit using the average monthly Arctic Sea Ice extent for 2017 [10]. Month is defined as the integers from 1 to 12, using the

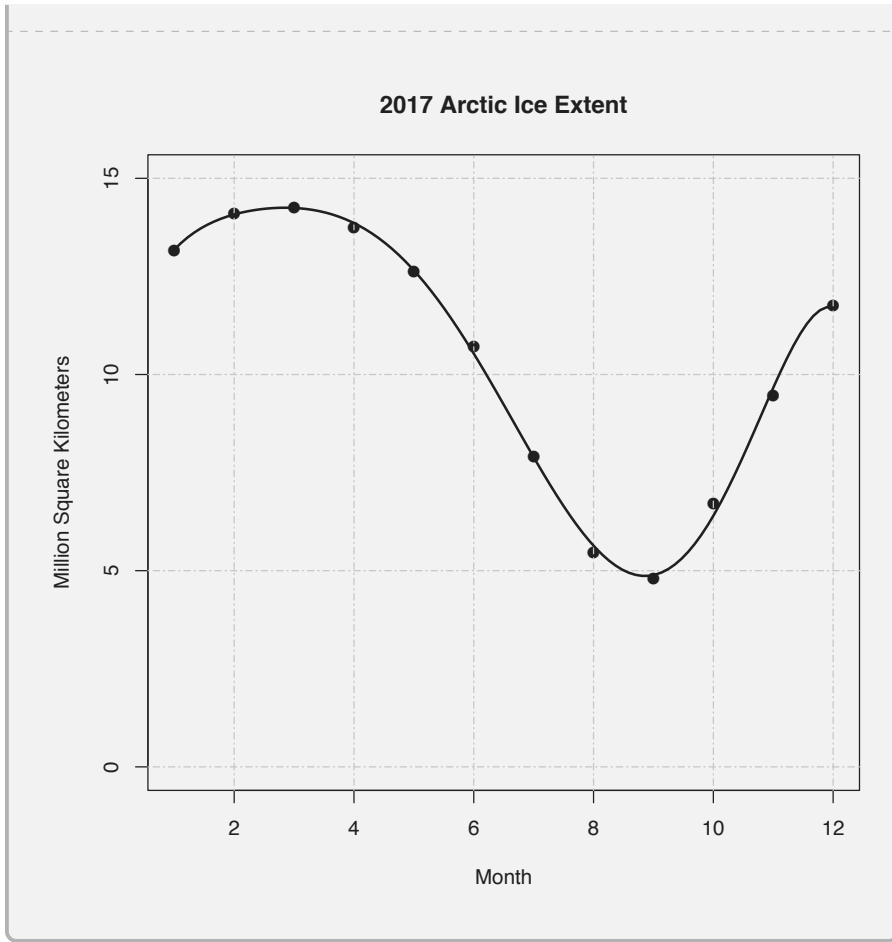
colon command, and Arctic.Ice.2017 is the average monthly sea ice extent measured in million square kilometers. We plot the two variables with point character set to 16, solid dots, and scaled to 1.25 times the default size with cex. We allow for the default range for the x -axis, but set the y axis to range from 0 to 15; otherwise the default setting will not start at 0. The axes are labelled with xlab and ylab. We add a title to the graph with **title**, and note that main could have been an argument within **plot**. A grid is added with **grid**. With NULL as the first two argument grid lines are placed at the tick marks set by **plot**.

The data is modeled by a sixth degree polynomial using **lm**, with the format $y \sim \text{polynomial}$. The polynomial is defined by the **poly** function where the first argument is the x or independent variable, the second is the degree of the polynomial, and the third sets raw=TRUE. If raw is FALSE, the default, then each monomial of the polynomial is viewed as orthogonal. The model can also be expressed as $\text{Arctic.Ice.2017} \sim I(x) + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6)$. The **I** operator ensures that an object is treated “as is”.

To save some typing we define Coef to be the coefficients of the model. We then define the function **Poly.Ice** to be the sixth degree polynomial returned from the model. Note that this can be done faster if the polynomF package is used (see Chapter 4) or with the more sophisticated code, presented without explanation, `function(x){return(sum(x^(0:(length(Coef)-1))*as.numeric(Coef)))}`. The code is finished by using **curve** to add the polynomial to the graph from month 1 to 12, with line width 2. Note that **summary(Ice.fit)** and **names(Ice.fit)** will return similar information as in the Exponential Fit section.

R Code

```
> Month=1:12
> Arctic.Ice.2017=c(13.17,14.11,14.27,13.76,12.62,10.72,
7.9,5.47,4.8,6.71,9.46,11.75)
> plot(Month,Arctic.Ice.2017,pch=16,cex=1.25,
ylim=c(0,15),xlab="Month",
ylab="Million Square Kilometers")
> title(main="2017 Arctic Ice Extent")
> grid (NULL,NULL,lty=6,col="gray")
> Ice.fit=lm(Arctic.Ice.2017~poly(Month,6,raw=TRUE))
> Coef=Ice.fit$coefficients
> Poly.Ice=function(x){Coef[[1]]+Coef[[2]]*x+
Coef[[3]]*x^2+Coef[[4]]*x^3+Coef[[5]]*x^4+
Coef[[6]]*x^5+Coef[[7]]*x^6}
> curve(Poly.Ice,1,12,lwd=2,add=TRUE)
```



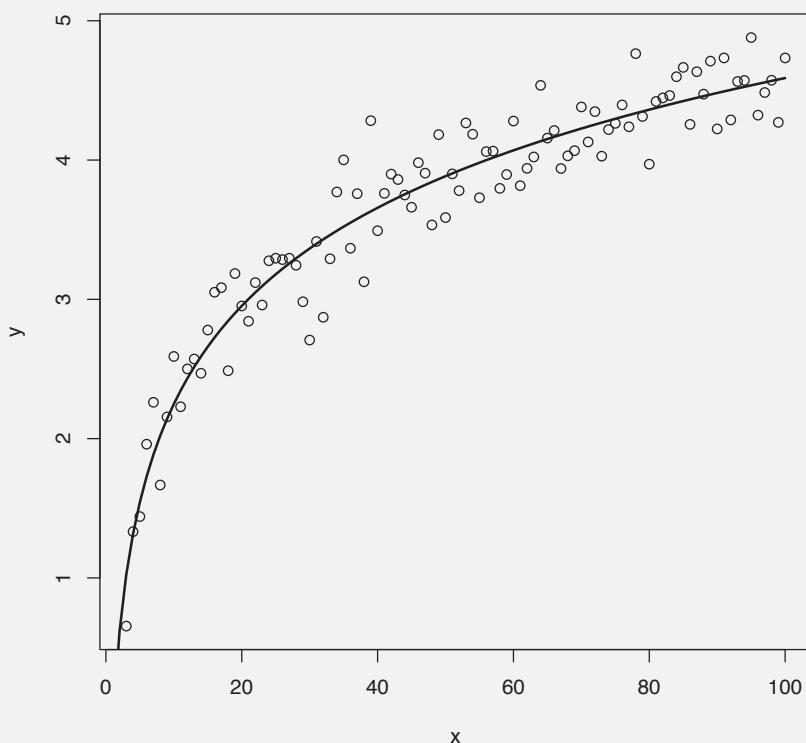
9.3 Log Fit

We use **rnorm** to generate random data for our example of fitting a log function to data. We have not set a seed so results will differ. We let x be the integers from 3 to 100 using the colon command. We then define y to be $\log(x)$ (in R \log is the natural log function) plus some “noise” added with **rnorm**. Here our call to **rnorm** returns a vector of $\text{length}(x)$ values from a random normal distribution with mean zero and standard deviation 0.25. Note that y is a vector with the same length as x since the addition of the two vectors, **log(x)** and **rnorm**, is element-wise. A simple plot is created with **plot(x,y)**. The **lm** functions returns the results from fitting the data with the model $y = \log(x) +$

b , which is set to the variable Log.fit. The function **Log.fit.function** is defined using the coefficients returned from **coef(Log.fit)** and added to the graph with **curve**. The functions **summary(Log.fit)**, **names(Log.fit)**, and **coef(Log.fit)** will return information similar to that demonstrated in the Exponential Fit Section.

R Code

```
> x=3:100
> y=log(x) + rnorm(length(x),0,0.25)
> plot(x,y)
> Log.fit=lm(y~log(x))
> Log.fit.function=function(x)
{coef(Log.fit)[2]*log(x)+coef(Log.fit)[1]}
> curve(Log.fit.function,0,100,lwd=2,add=TRUE)
```

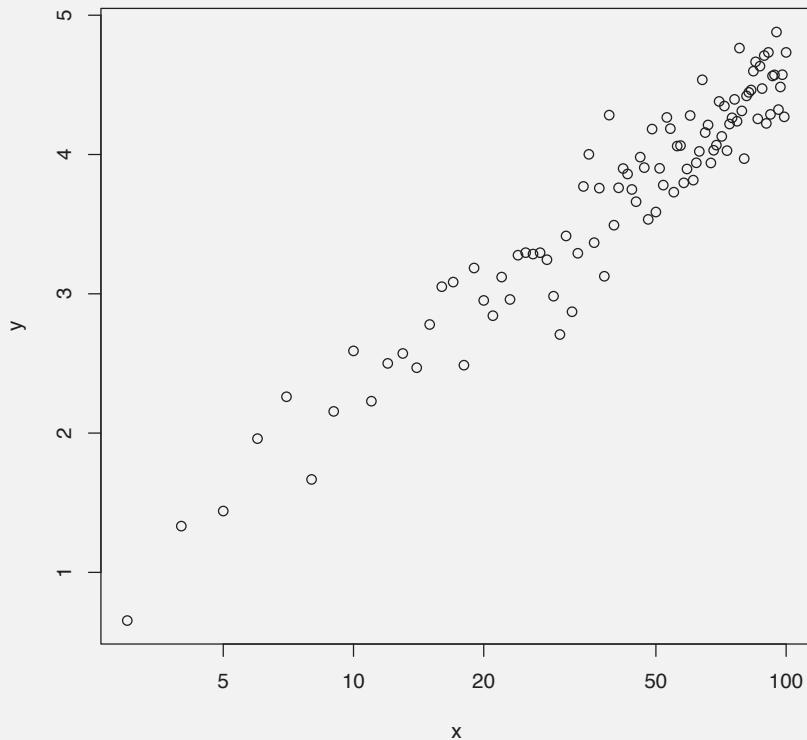


This is an opportune moment to note that the **plot** function will return

log scale graphs. In this case we add the argument `log="x"` so that the x -axis is a log scale. Similarly, `log="y"` is an optional argument for `plot`.

R Code

```
> plot(x,y,log="x")
```



9.4 Logistic Fit

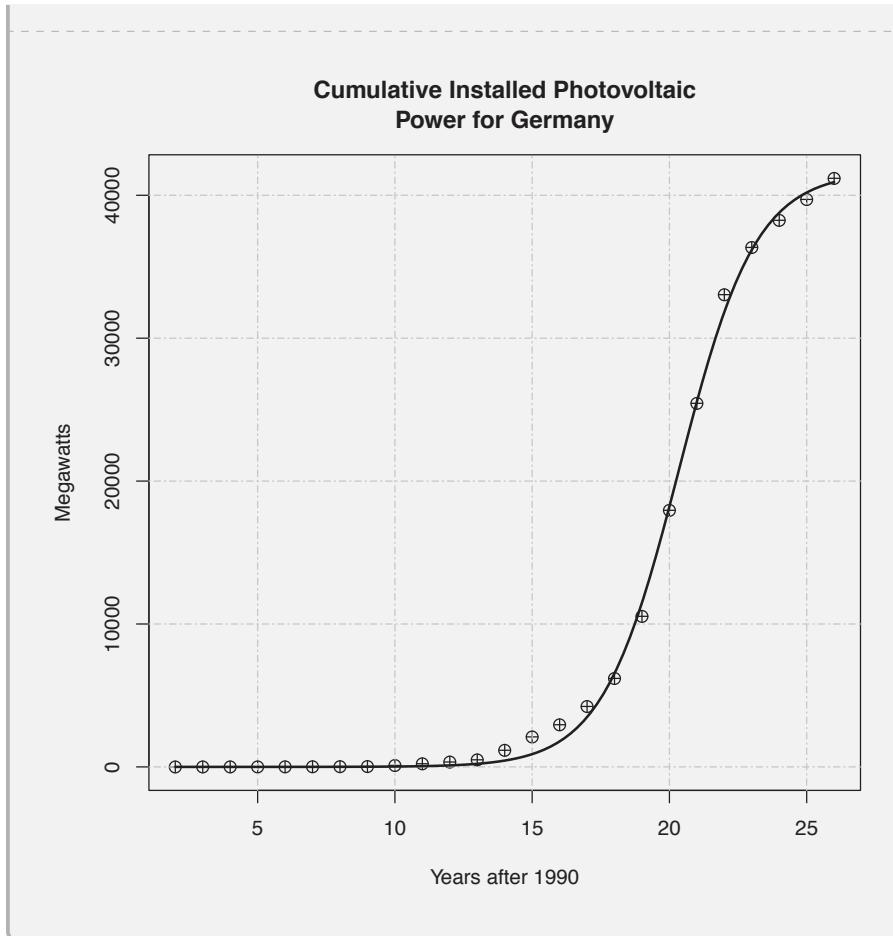
In this example, we model Germany's cumulative installed photovoltaic power with a logistic curve [13]. The data are from 1992 through 2016 and so `Year.After.1990` is defined to be 2 through 26, using the colon command. `Solar.Ger` is the cumulative installed power in megawatts. The data is graphed

with **plot** using point character 10 and scaled to 1.25 times the default with cex. The axes ranges are set by default and the axes are labelled with xlab and ylab. A title is added with **title** and note that \n is used for a line break. A grid is added with **grid**. With NULL as the first two argument grid lines are placed at the tick marks set by **plot**.

The logistic model is returned using **nls** with the argument Solar.Ger ~ SSlogis(). The function **SSlogis**, self starting logistic, identifies starting values for the parameters. The logistic model is of the form $\frac{\text{Asym}}{1+e^{(x\text{mid}-x)/\text{scal}}}$. The first argument of **SSlogis** is the independent variable and the other three correspond to the parameters in the model. We define the logistic function, Ger.logistic.function, using the **coef(Ger.logistic)** vector for the estimates of the parameters. We add the function to the graph from 2 to 26, with **curve** setting the line width to two and using add=TRUE to add it to the current plot.

R Code

```
> Years.After.1990=2:26
> Solar.Ger=c(2.9,4.3,5.6,6.7,10.3,16.5,21.9,30.2,
  103.4,222.5,343.6,496.0,1165.4,2100.6,2950.4,4230.1,
  6193.1,10538.1,17956.4,25441.6,33045.6,36349.9,38249.9,
  39710.0,41186.0)
> plot(Years.After.1990,Solar.Ger,pch=10,cex=1.25,
  xlab="Years after 1990",ylab="Megawatts")
> title(main="Cumulative Installed Photovoltaic\nPower
  for Germany")
> grid(NULL,NULL,lty=6,col="gray")
> Ger.logistic=
  nls(Solar.Ger~SSlogis(Years.After.1990,Asym,xmid,scal))
> Ger.logistic.function=function(x)
  {coef(Ger.logistic)[[1]]/(1+exp((coef(Ger.logistic)
  [[2]]-x)/coef(Ger.logistic)[[3]]))} 
> curve(Ger.logistic.function,2, 26,lwd=2,add=TRUE)
```



Another way to return a logistic function with **nls** is **nls(Solar.Ger ~ L/(1+A*exp(-r*Years.After.1990)))**, start=list(L=41600,A=2056895,r=.7)), but by not using **SSlogis** we need reasonable starting points for the parameters. An error ‘singular gradient matrix at initial parameter estimates’ will occur if starting points are not reasonable.

9.5 Power Fit

The final example of this chapter fits the model $y = x^r + c$, for four different values of r , using data with randomly generated error. We provide four examples using a for loop with r equal to -0.2 , 2.5 , 0.5 , and -1.5 . The code begins by setting x to the integers from 1 to 50 using the colon command. The

variable error, set to 0.05, will be used for the standard deviation when adding “noise” to the data. We use **par(mfrow=c(2,2))** to create a two-by-two grid for four graphs.

The for loop begins letting r range over the four values $-0.2, 2.5, 0.5$, and -1.5 . We set c to a randomly selected integer between 5 and 15 using the function **sample**. The last argument in **sample** is the number of values to return. The dependent data y is x^r plus a value selected from a random normal distribution with mean 0 and standard deviation error plus c. Note that in the definition of y the first two variables are vectors of length the length of x while the last term is a constant. The terms of the vector are added together and the constant is added to each term.

We use **nls** to fit the model. There are two issues to note. First the model is written in the form $y \sim I(x^{\text{power}}) + c$ and not $y \sim x^{\text{power}} + c$. The **I** function is used to clarify the model. For instance, if we used $y \sim x^{\text{power}}$ we could mean to use a linear model with the two vectors given by y and x^{power} . The use of **I** states that we want the formula ‘as is’ and not to create a vector of data out of x^{power} .

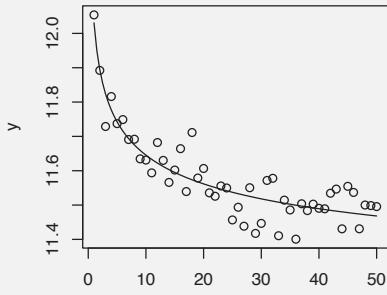
Our second issue is the need for starting points for the algorithm. In this example the starting points, power=1 and c=10, are sufficient for all four examples even though r and c change. The function **coef** returns the vector of coefficients from the model and is used to define the function **f**. Our next two lines set up the title and label for the x -axis. In both cases we use **paste** to concatenate characters, in quotes, and variables. For xlabel, we set sep to empty quotes so that no space is placed between arguments separated by commas. In this case the spaces are added within the quotes. We plot the data with all the defaults except we add a title with main and label the x -axis with xlabel. We add **f** to the graph from 1 to 50 with **curve**. The final right brace closes the for loop which was started with a left brace.

R Code

```
> x=1:50
> error=0.05
> par(mfrow=c(2,2))
> for (r in c(-.2, 2.5, 0.5, -1.5)){
+ c=sample(5:15,1)
+ y=x^r+rnorm(length(x),0,error)+c
+ Power.fit=nls(y ~ I(x^power)+c,
start=list(power=1,c=10))
+ f=function(x){x^coef(Power.fit)[[1]]+
coef(Power.fit)[[2]]}
+ title=paste("Data: x^(",n,"+N(0,", error,")+",c)
+ xlabel=paste("nls() estimates: r= ",
round(coef(Power.fit)[[1]],2),", constant=",
round(coef(Power.fit)[[2]],2),sep="")}
```

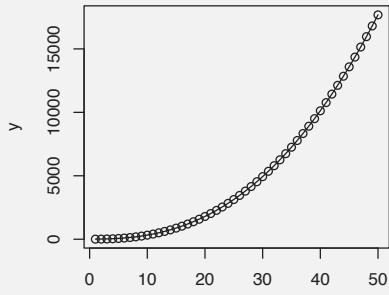
```
+ plot(x,y,main=title,xlab=xlabel)
+ curve(f,1,50,add=TRUE)}
```

Data: $x^{-0.2} + N(0, 0.05) + 11$



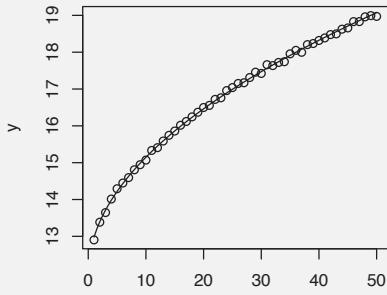
nls() estimates: n= -0.21, constant=11.03

Data: $x^{2.5} + N(0, 0.05) + 9$



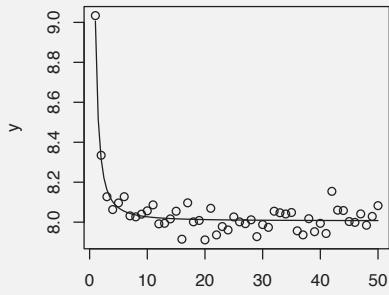
nls() estimates: n= 2.5, constant=9

Data: $x^{0.5} + N(0, 0.05) + 12$



nls() estimates: n= 0.5, constant=11.98

Data: $x^{-1.5} + N(0, 0.05) + 8$



nls() estimates: n= -1.69, constant=8.01

Code Review for the Curve Fitting Chapter

coef(model name) returns the vector of coefficients of a model.

lm(model) returns the results of a linear model. In this chapter, the models used are $y \sim x$, $\log(y) \sim x$, $y \sim \text{poly}(x, 6, \text{raw}=TRUE)$, and $y \sim \log(x)$. Note that lm() is also used for multiple linear regression.

names(model name) is used in this chapter to return the names of the variables available from a model output.

nls(model,start=list()) returns the results of a non-linear least-

squares procedure. Unless using a self starting model (see `SSlogis`) initial estimates of the parameters are listed in `list()`.

`SSlogis(x,Asym,xmid,scal)` is a self starting model of the form $\frac{\text{Asym}}{1+e^{(x\text{mid}-x)/\text{scal}}}$ and used in this chapter with `nls()`.

`summary(model name)` was used in this chapter to return the summary information of a model.

9.6 Exercises

1. Using the data in [Section 9.1](#), is C a significant parameter in the model `Solar.usa ~ P*exp(r*Years.after.2000) + C`?
2. Would a seventh degree polynomial be a better model for the data in [Section 9.2](#)? Hint: Consider the p-value.
3. How do the results of the model in [Section 9.3](#) change if the 0.25 in `rnorm()` is changed to 0.5?
4. Remove the last value in the data set of [Section 9.4](#). How does this change the results of the model?
5. Create an example fitting data with a logistic model by generating 50 data points similar to [Section 9.3](#).
6. Is a power model a potentially better model for the data in [Section 9.1](#)?

10

Simulation

We provide examples of six simulation in this chapter that illustrate a number of functions that are available in R. For example, **replicate**, **sapply**, and **apply**, along with functions that generate random values. While this is not the only place where you will find a simulation in this book, the focus of the simulations in this chapter are to estimate a probability. The examples are ordered from basic, simulating coin flips, to the complicated simulation in the deadly board game section, while including some classic probability problems such as the Buffon needle problem.

10.1 A Coin Flip Simulation

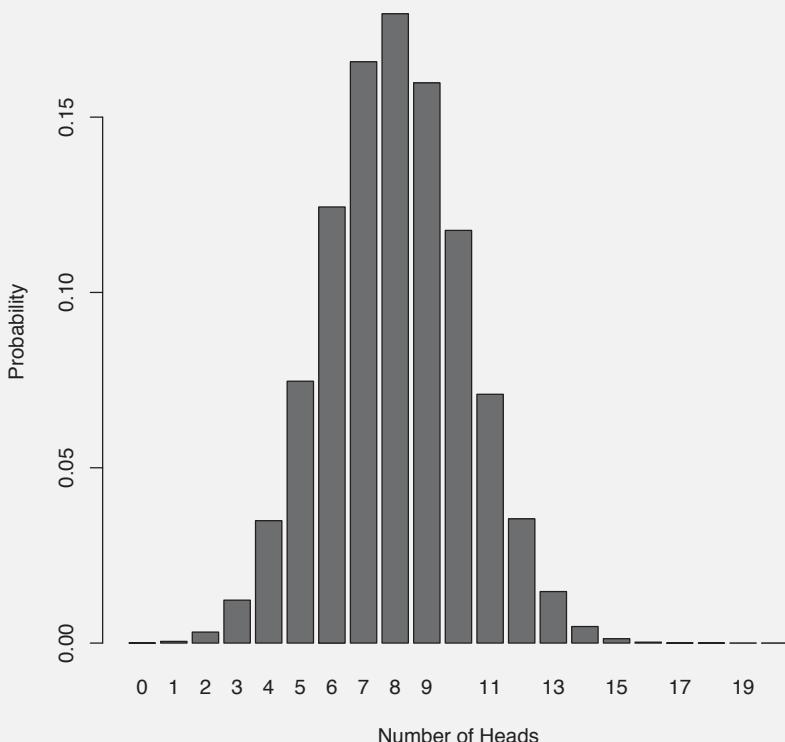
What is the probability of a coin landing on heads 10 or more times in 20 coin flips if the coin has a 40% chance of landing on heads? Our first simulations will estimate this probability, but first we will create a bar chart of our simulation and then calculate an estimate of the probability. In addition, we will calculate an estimated cumulative probability distribution.

We begin by defining the variable `flips` for the number of coin flips, `trials` for the number of simulations, and `p.succ` for the probability of a success. Doing this first allows us to make changes to the code easily. Generating our random sample is done in the third line with **rbinom**, which will count the number of successes in `flips` tosses of our coin with probability of success `p.succ`. This will be repeated `trials` times. As an aside we note the existence of the functions **dbinom**, **pbinom**, and **qbinom** for the density, distribution, and quantile function for the binomial distribution. The results of **rbinom** are assigned to `simulation`, which is a vector of length `trials` where each value is a number from 0 to `flips`. The function **tabulate** counts the number of times a 1 occurs, a 2 occurs, etc., but it will not count zeroes. To get around this, `simulation+1` is the simulation vector but with 1 added to each entry. Thus `simulation+1` is a vector of values from 1 to `flips+1`. With this shift our data will be tallied and the largest value tallied will be `flips+1`. The first value in the vector totals is the number of times 0 occurred, the second is the number of times 1 occurred, and the last is the number of times the number `flips` occurred.

R Code

```
> flips=20
> trials=100000
> p.succ=0.40
> simulation=rbinom(trials,flips,p.succ)
> totals=tabulate(simulation+1,flips+1)
> prob=totals/trials
> barplot(prob,col="red",names.arg=0:flips,ylab=
"Probability",xlab="Number of Heads", main=
paste("A coin flipped", flips,"times with",
p.succ*100,"% chance of heads\nsimulated",
format(trials,scientific=FALSE, big.mark=",","times.")))
```

A coin flipped 20 times with 40% chance of heads simulated 1,000,000 times.



Each value in the vector totals is divided by the value of trials. We can also accomplish this with **prop.table(totals)**. We now plot our results with **barplot**, which will create bars for each value in prob, color the bars red,

and label each bar with the values from 0 through 20 with `names.arg=0:flips`. Recall that the first value in totals in the number of times 0 heads occurred and so the first value in prob is the percentage of times 0 heads occurred. The **barplot** function creates bars with the heights the values in the vector and does not associate those heights to any particular value. In using `names.arg` we need to be aware of the association and we note that `names.arg` can be set to a character vector and it is not limited to numeric. We label the *x*- and *y*-axis with `xlab` and `ylab`. We set the title of the graph with `main` and use **paste** to concatenate characters in quotes and variables unquoted, while separating each term with commas. Within **paste** we use **format** so that the number of trials is not in scientific notation and we separate every three digits with a comma, `big.mark=", "`.

The bar chart is illustrative but doesn't really answer our question. The values in the 11th through 21st, `flips+1`, location of our prob vector are the probabilities associated with 10 through 20 heads. This list of values is obtained with `prob[c(11:(flips+1))]`, where the square brackets refer to values in the location given with `11:(flips+1)`. Here the colon command, `a:b`, list the values from `a` to `b` in unit increments.

R Code

```
> prob[11:(flips+1)]
```

```
[1] 0.11770 0.07096 0.03544 0.01469 0.00472 0.00125
[7] 0.00026 0.00004 0.00001 0.00000 0.00000
```

What we really want is the sum of these values and so we input `prob[11:(flips+1)]` into the **sum** function. While not necessary for our question, it is worth pointing out here that **cumsum** will provide the cumulative sum of a vector or in this case the cumulative distribution. We note that there is also a **cumprod** function.

R Code

```
> sum(prob[c(11:(flips+1))])
```

```
[1] 0.245062
```

R Code

```
> cumsum(prob)
```

```
[1] 0.00004 0.00053 0.00366 0.01591 0.05081 0.12549
[7] 0.24988 0.41567 0.59516 0.75494 0.87264 0.94360
[13] 0.97904 0.99373 0.99845 0.99969 0.99995 0.99999
[19] 1.00000 1.00000 1.00000
```

10.2 An Elevator Problem

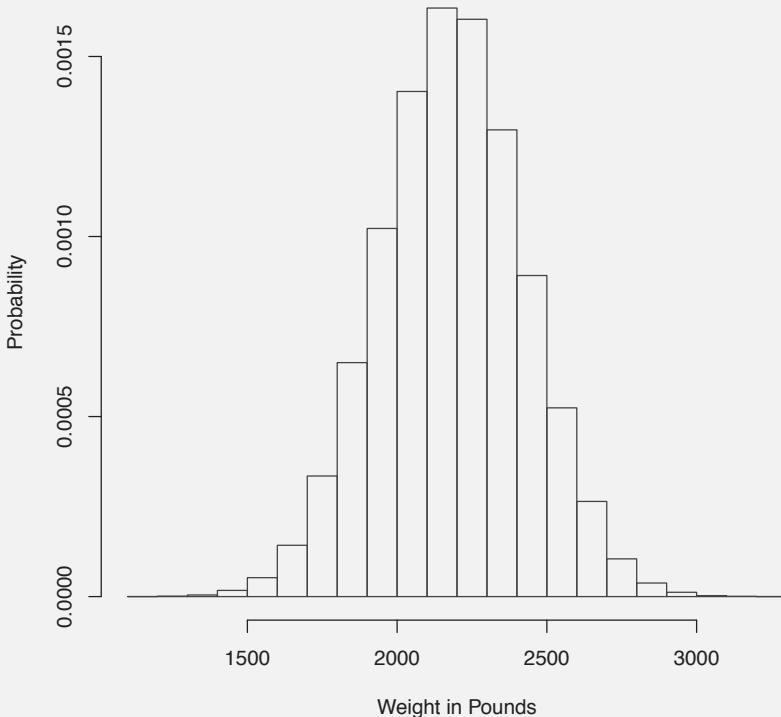
Suppose an elevator has a capacity of 2400 pounds and states that the maximum occupancy is 12 people. If 12 people get on the elevator what is the probability that capacity is exceeded? We will assume that the weight of women is normally distributed with mean of 168.5 lbs and standard deviation of 67.76 lbs and that the weight of men is normally distributed with mean of 195.7 lbs and standard deviation of 68.02 pounds [5]. We note that weights are actually skewed, but assuming normal is sufficient for this example. This simulation is a three step process as we have to first selected 12 people assuming males and females are equally likely, then for each person randomly select a weight, and finally sum the weights. We will create a function that will execute one simulation and then use **replicate** to repeat the process.

We begin by defining trials for the size of the sample and people for the number of people on the elevator. We define the function Selection as a function of n. In step one we assign Adults to be a random sample of size n from 0 to 1 with replacement, where we use 0 to represent a female and 1 a male. Note that we could use **rbinom** here. The for loop, i ranging from 1 to n, consists of an if then else statement. If Adults[i]==0, in other words the ith person is female, then replace the 0 with a random value chosen from a normal distribution with mean 168.5 and standard deviation 67.76, Adults[i]=**rnorm(1,168.5,67.76)**. Otherwise, else, replace the ith value of Adults with a random value chosen from a normal distribution with mean 195.7 and standard deviation 68.02. Note that there are two right curly brackets at the end of the else line. The first closes the else statement and the second ends the for loop. Now, outside the for loop, but still within the function definition, we have **return(sum(Adults))**, which says that the function Selection will return the sum of the values in the Adults vector. The line ends with a right curly bracket which ends the function definition. Our simulation is created with **replicate**, which will return a vector of length trials where each value is the output of an evaluation of **Selection** with input the value People. In summary, **replicate** repeats the second options the first option times.

We create a histogram of our results with **hist**. It is a relative frequency histogram due to freq=FALSE and the borders of the boxes are blue. We label the x- and y-axis with xlab and ylab. We set the title of the graph with main and use **paste** to concatenate characters in quotes and variables unquoted while separating each with commas. Within **paste** we use **format** so that the number of trials is not in scientific notation and we separate every three digits with a comma, big.mark=“,”. The bins are set by default, but they can be defined with breaks=(a vector of values). We still need to calculate the percentage of time we exceed 2400 lbs.

R Code

```
> Trials=100000
> People=12
> Selection=function(n){
+ Adults=sample(0:1,n,replace=TRUE)
+ for (i in 1:n){
+ if (Adults[i]==0){Adults[i]=rnorm(1,168.5,67.76)}
+ else {Adults[i]=rnorm(1,195.7,68.02)}}
+ return(sum(Adults)) }
> simulation=replicate(Trials, Selection(People))
> hist(simulation,freq=FALSE,border="blue",ylab=
"Probability",xlab="Weight in Pounds",main=
paste(format(Trials,scientific=FALSE,big.mark=","),
"Simulations of the Sum of Weights of",People,"Adults"))
```

100,000 Simulations of the Sum of Weights of 12 Adults

Square brackets refer to terms within a vector. So, for instance, simula-

tion[4], would return the value of the fourth simulation. On the other hand, if a logical statement is placed within the square brackets we get a vector of the values that satisfy the statement. The vector over is a list of values from simulation that exceed 2400. Finally, **length(over)**, the number of times the elevator is over capacity, divided by the value of Trials, is the estimated probability we exceed capacity with 12 adults on the elevator.

R Code

```
> over=simulation[simulation>2400]
> length(over)/Trials
```

```
[1] 0.182
```

10.3 A Monty Hall Problem

A version of the Monty Hall problem starts with three cards: one has both sides colored red, one has both sides colored blue, and one has one side red and the other blue. Suppose we pick a card at random and put it down on a table. If the side showing is red, then what is the probability the other side is blue? For this simulation we will create a 2×3 matrix to represent our three cards. We will randomly select a card and randomly select a side to view. From there we can answer the question.

The **matrix** function will return a matrix. The first inputs are the values for our matrix, where 0 represents red and 1 represents blue. The second input, **byrow=TRUE**, sets the matrix to be filled by rows, while the last sets the number of rows. This matrix is called cards. We use **rownames(cards)** and **colnames(cards)** to add names to the rows and columns, respectively. This wasn't absolutely necessary, but this way the output has clear meaning. The matrix is returned by running cards.

R Code

```
> cards=matrix(c(1,1,0,0,1,0),byrow=TRUE,nrow=3)
> rownames(cards)=c("Card 1", "Card 2", "Card 3")
> colnames(cards)=c("Side 1", "Side 2")
> cards
```

	Side 1	Side 2
Card 1	1	1
Card 2	0	0
Card 3	1	0

With our matrix constructed, we begin the simulation by setting Trials to 100000. The vector simulation is Trials long with each element a number randomly chosen from 1 to 3, with 1:3. In other words, a vector of card selections. Our for loop will randomly select a side of each card, check if it is red and if so “flip” it, and if not assign a value of 9. Our for loop has k ranging from 1 to Trials. Our first line sets i to the value in the kth location of simulation. We let j be a random value selected from 1 and 2. Now, if cards[i,j]==0 (in other words, check to see if side j of card i is red) then set the value of simulation[k] to the other side of the card. Here j%2 is j mod2 and adding 1 will switch a 1 to 2 and vice versa; we “flipped” the card. If the value of card[i,j] is not 0 (in other words it is blue) the set simulation[k] to 9. Our probability question focused on cards where red is showing so we will need to ignore 9’s further down. Simulation[1:12] shows the first 12 values of the simulation vector.

R Code

```
> Trials=100000
> simulation=replicate(Trials,sample(1:3,1))
> for(k in 1:Trials){
+ i=simulation[k]
+ j=sample(c(1:2),1)
+ if (cards[i,j]==0){simulation[k]=cards[i,j%%2+1]}
else{simulation[k]=9}}
> simulation[1:12]
```

[1] 0 0 0 9 9 9 0 9 9 1 1 1

We set flip to be the vector of 1s from simulation. Here simulation[simulation==1] returns the elements of the simulation vector that satisfy the logical statement, in this case simulation==1. The length of vector flip is the number of times a card changed from red to blue when it was flipped over. The stay vector is similar but the length here is the number of times the card remained red when it was flipped over. The last line, **length(flip)/(length(flip)+length(stay))**, is our desired probability.

R Code

```
> flip=simulation[simulation==1]
> stay=simulation[simulation==0]
> length(flip)/(length(flip)+length(stay))
```

[1] 0.3346325

10.4 Chuck-A-Luck

Chuck-A-Luck is a casino game played with three dice with six different ways to bet. We will focus on the number bet, where you bet on a number from 1 to 6, and for our example the payout will be \$1 if your number comes up once, \$2 if it comes up twice, and \$5 if it comes up six times. How much would you be willing to pay to play this game?

We begin by simulating rolls of three dice. We set the variables `n.trials`, `n.sides`, and `n.rolls` for the number of simulations, sides on the dice, and how many dice to roll, respectively. The function `sample`, inside the `replicate` function, will select three (rolls) random integers from 1 up to die with replacement. This will be repeated trials times and replicate returns a $3 \times$ trials matrix assigned simulation. Note if we aren't sure if an object is a data frame or a matrix we can use `is.data.frame` and `is.matrix` to check. Within the square brackets of simulation the first location is blank, which is read as return all rows, while the second value is `1:10`, which is read as return columns 1 through 10.

R Code

```
> n.trials=100000
> n.side=6
> n.rolls=3
> simulation=replicate(trials,
sample(n.side,n.rolls,replace=TRUE))
> simulation[,1:10]
```

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

Since the values 1 through 6 are equally likely, without loss of generality, we can count the number of times the die showed 1 in each column of simulation. Within the `colSums` function, which sums the columns of a matrix, we have the logical statement `simulation==1`. Hence, `colSums` will only sum the cells that satisfy the statement. We set the output to chuck and list out the first 10 values with `chuck[1:10]`. To check our work we can compare the output with that of `simulation[,1:10]`.

R Code

```
> chuck=colSums(simulation==1)
> chuck[1:10]
```

```
[1] 1 0 0 2 2 0 0 0 0 1
```

To calculate expected winnings we set the variables m1, m2, m3 to the payouts for rolling a one 1 to 3 times. We **tabulate** the vector chuck for values up to 3 so that wins is a vector of length three for the number of times one was rolled once, twice, and three times. The value of return is the total won in playing the game trials times. Our expected winnings is return/trials. Paying more than \$0.51 to play is unwise.

R Code

```
> m1=1
> m2=2
> m3=5
> wins=tabulate(chuck,3)
> return=wins[1]*m1+wins[2]*m2+wins[3]*m3
> return/trials
```

```
[1] 0.5121
```

Out of curiosity we ask the question, how will our winnings change if the number of sides of the die increases? We answer this question with a graph. We will take the previous work and create a function for repeated use with different sizes of dice.

We set game to be a function of the number of trials n.trials, the number of sides on the dice n.sides, the number of dice to roll n.dice, and the payout for 1, 2 or 3 ones on the dice in pay.1, pay.2, pay.3. The code within the curly brackets for function is essentially the same as above with some changes in variable names. The one key is that we need to add **return(expected.money)** before ending the function. We then set max.die=20 for the maximum number of sides on the dice. The **sapply** function applies what is in the first input into the function of the second input and returns a vector of the same size. Note that **game** is already a function and it would seem that all we would need is **game(1000,x,3,1,2,5)** in the second input. But any variable can be used in the second input so it isn't clear that we wish to apply the numbers 1:max.die to x. Hence the use of **function(x)**. In **game** we have 100000 trials, 3 dice and 1, 2, and 5 as our payouts. The result of **sapply** is set to dist. We list the first five elements of dist. We should note that **sapply** and for loops are very similar and we could have used **dist=numeric(max.die); for(i in 1:max.die)dist[i]=game(100000,i,3,m1,m2,m3)** instead. In using a for loop here we have to create the vector dist first whereas we don't need to do this when using **sapply**.

R Code

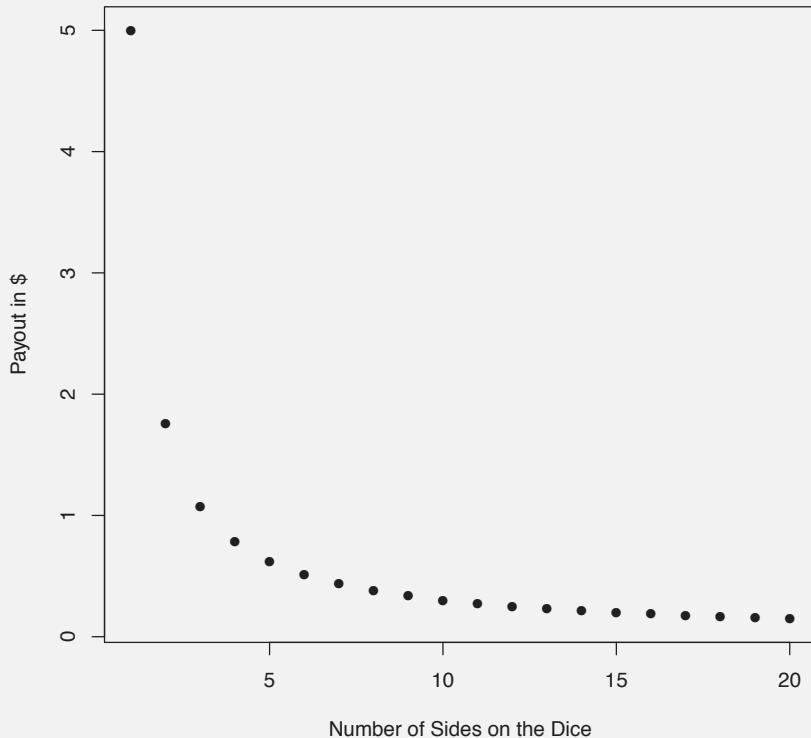
```
> game=function(n.trials,n.sides,n.dice,pay.1,pay.2,
+ pay.3){
+ simulation=replicate(n.trials,sample(n.sides,n.dice,
+ replace=TRUE))
+ chuck=colSums(simulation==1)
+ wins=tabulate(chuck,3)
+ money=wins[1]*pay.1+wins[2]*pay.2+wins[3]*pay.3
+ expected.money=money/n.trials
+ return(expected.money)}
> max.die=20
> dist=sapply(1:max.die,function(x)
{game(100000,x,3,1,2,5)})
> dist[1:5]
```

```
[1] 5.00000 1.74779 1.07258 0.78285 0.61214
```

We plot our results with **plot**. The x values are 1:max.die, and the y values are dist. The point character is set with pch=16, a solid dot, and the axes are labeled with xlab and ylab. Notice that if there is only 1 side on each die we always get three ones and so our payout is always \$5. Moving to dice with two numbers cuts our expected value to under \$2.

R Code

```
> plot(1:max.die,dist,pch=16,xlab=
"Number of Sides on Dice",ylab="Payout in $")
```



10.5 The Buffon Needle Problem

If we drop a needle of length d inches onto a floor with parallel lines spaced l inches apart, with $d < l$, what is the probability the needle crosses a line? This is known as the Buffon Needle Problem. For the simulation here we will use a needle of length $1/2$ units on a floor with parallel lines spaced 1 unit apart. We are particularly interested in the inverse of the probability that the needle crosses a line. We will estimate that probability first and then create a graph of randomly placed needles on our floor. In our simulation we will select a location for the center of the needle and an angle. Due to symmetry, we can limit the center of the needle from 0 to $1/2$, instead of 1. Similarly, the angle

of the needle can be limited from 0 to $\pi/2$. If the center of the needle is d units from the line with angle θ , then the needle crosses if $\frac{1}{4} \sin \theta \geq d$ or $\frac{\sin \theta}{d} \geq 4$. Note that half the length of the needle is $1/4$, which is the hypotenuse of our triangle.

With the math completed the simulation is only a few lines. We set Trials=100000. The function **runif(1,0,pi/2)**, for example, uniformly selects one random value from the interval $[0, \pi/2]$. We take the sin of that value and divide by a uniform random variable from $[0, 1/2]$. This is repeated Trials time so that **replicate** returns a vector of length Trials of these values. We need to count the values that are 4 or greater. We set cross to simulation[simulation>=4], which selects the values of the simulation vector that are 4 or greater. The probability is then **length(cross)/Trials**, but we wanted the inverse. Note that the probability the needle crosses the line is $2d/\pi l$.

R Code

```
> Trials=100000
> simulation=replicate(Trials,sin(runif(1,0,pi/2))/runif(1,0,1/2))
> cross=simulation[simulation>=4]
> Trials/length(cross)

[1] 3.146831
```

We now create a graph to illustrate the Buffon Needle Problem. We randomly place the center of the needle in a $[0, 1] \times [0, 1]$ box with an angle from $[0, \pi]$. We start by creating a graph frame with **plot**. We use the points $(0, -0.25)$ and $(1, 1.25)$ and an aspect ratio of 1/1 with **asp=1**. Since the y -values range from -0.25 to 1.25, the aspect ratio forces the same for x . We use **pch=NA** so that the points are not drawn as we used them simply to set the graph range. The labels on the axes are also left blank with empty quotes. We have set a window with space for our needles to cross outside the box where they will be centered. Two red horizontal lines with line width 2 are drawn at $y = 0$ and $y = 1$ using **abline**. The function **abline** will draw horizontal lines at all the values set equal to h. It will also draw vertical lines at all values set to v. With our graph frame set we turn our attention to randomly drawing needles.

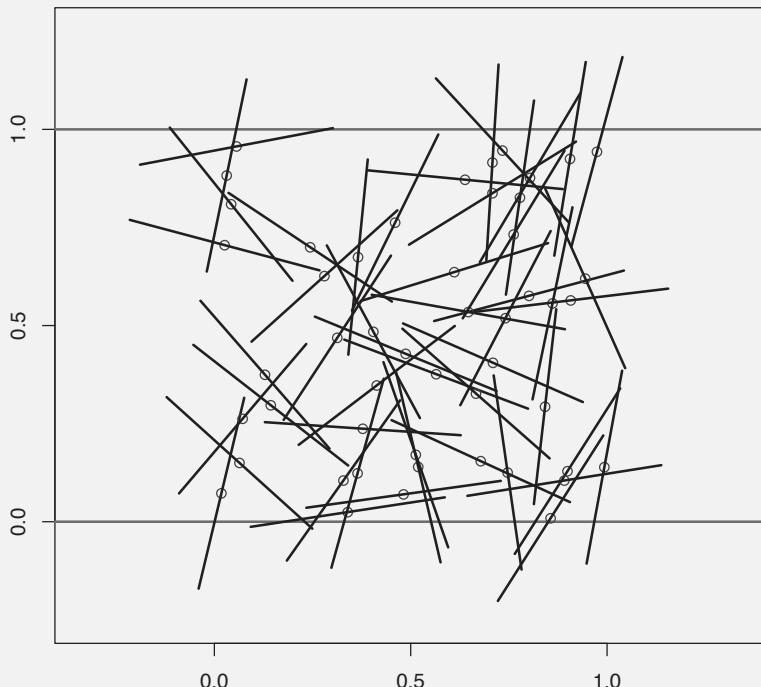
R Code

```
> plot(c(0,1),c(-0.25,1.25),asp=1,pch=NA,xlab="",ylab="")
> abline(h=c(0,1),lwd=2,col="red")
> Pins=50
> for(i in 1:Pins){
```

```

+ x=runif(1,min=0,max=1)
+ y=runif(1,min=0,max=1)
+ theta=runif(1,0,pi)
+ points(x,y,col="blue")
+ if(theta <= pi/2)
+ {segments(x-cos(theta)/4,y-sin(theta)/4,
x+cos(theta)/4,y+sin(theta)/4,lwd=2)}
+ else{theta.new=pi-theta
+ segments(x-cos(theta.new)/4,y+sin(theta.new)/4,
x+cos(theta.new)/4,y-sin(theta.new)/4,lwd=2)}}

```



We set the number of Pins to drop to 50 and then begin a for loop where i ranges from 1 to Pins. Two uniform $[0, 1]$ random values are selected and set to x and y. A uniform $[0, \pi]$ random value is set to theta. The center of the needle is plotted with **points** and colored blue. We use the default point character, pch, which is an open circle. We will graph our needles using **segments()** and

to do that we will need to know the endpoints of our needle, which depends on the value of theta. If theta is less than or equal to $\pi/2$, then the two points are $(x - \cos(\theta)/4, y - \sin(\theta)/4)$ and $(x + \cos(\theta)/4, y + \sin(\theta)/4)$. The first part of the if statement graph this line segment with segments() and a line width of 2. Now, if theta is not less than or equal to $\pi/2$ then we set theta.new = pi - theta, call it $\hat{\theta}$, as the first line after else. The endpoints of our needle are $(x - \cos(\hat{\theta})/4, y + \sin(\hat{\theta})/4)$ and $(x + \cos(\hat{\theta})/4, y - \sin(\hat{\theta})/4)$, which are again graphed with **segments**. Two right curly brackets end the code, the first ending the else statement and the second ending the for statement.

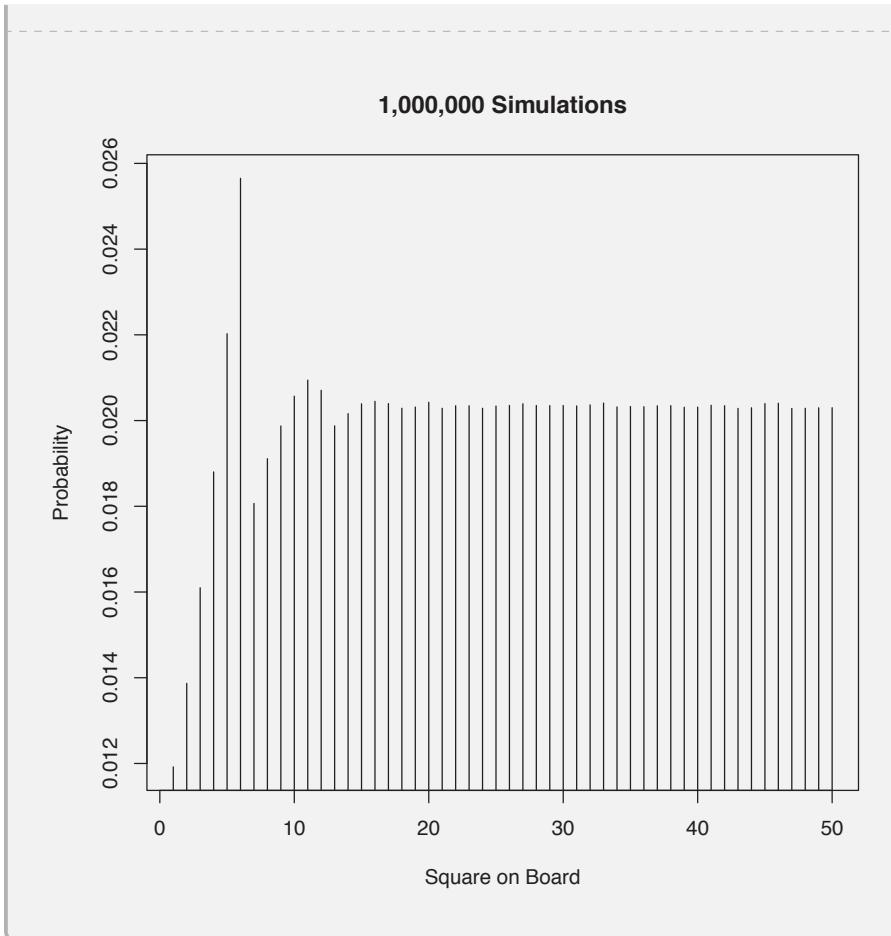
10.6 The Deadly Board Game

Our last simulation is a modification of the one produced by Robinson [23]. He provides a nice story to go with the puzzle; in short we have a board numbered from 0 to 1000. You may place three coins on the board. Starting on square 0, you roll a single six-sided die repeatedly until you pass your coins. If you land on one of your coins you win. Otherwise you lose. Where should you place your three coins? In the story for the game, the game is used to decide guilt or innocence. If you lose you are deemed guilty and put to death. Hence, the name of the game.

We start by first simplifying the game by placing only one coin and using only 50 squares on the board. We set trials=1000000, max.square=50, and rolls=50. Trials is the size of the simulation and max.square is the length of the board. We will roll the die 50 times, which it is really more than necessary, but does ensure that we always get to the end of the board. If computing power were an issue around 20 is enough since the average roll is 3.5 and max square divided by 3 would be reasonable.

R Code

```
> trials=1000000
> max.square=50
> rolls=50
> simulation=replicate(trials,cumsum(
sample(6,rolls,replace=TRUE)))
> totals=tabulate(simulation, max.square)
> probabilities=totals/sum(totals)
> plot(1:max.square,probabilities,type="h",xlab=
"Square on Board",ylab="Probability",main=paste
(format(trials,scientific=FALSE,big.mark=","),
"Simulations"))
```



To simulate rolling the die and advancing along the board, we use **sample** within **cumsum**. With **sample(6,rolls,replace=TRUE)** we get a vector of length rolls of random values from 1 through 6 with replacement, which is then converted to cumulative sums with **cumsum**. For example, **cumsum(1,4,2,5,3)=(1,5,7,12,15)**, which represents the squares we occupy during that sequence of rolls. This process is repeated with **replicate** trials times so that simulation is a rolls-by-trials matrix.

The function **tabulate** will tally all values in the matrix simulation less than or equal to max.square. Totals is a vector of length max.square where the first number represents the times square 1 was occupied, the second number is the times square 2 was occupied, and so on. In the next line, the **sum(totals)** is the total number of rolls so that totals/**sum(totals)** are probabilities. We can also obtain these values with **prop.table(totals)**. We plot the probabilities with **plot**. The *x*-values are 1:max.square and the probabilities are the *y*-values. We use type="h" for a histogram-like plot. We label the axes with xlab and

`ylab`. The title of the plot is created with `main`. We use `paste` to concatenate values and characters within quotes. The `format` function formats the number of trials so that it is not scientific notation and so that every three digits are separated by a comma with `big.mark=“,”`. It is clear that for the one coin game, we should place the coin on square 6. What are the exact percentages associated with each square?

To answer the question we want to order the vector probabilities, and we also need the associated squares with each probability. We assign `square` to be the numbers 1 through `max.square`. We convert probabilities to percentages. We define the data frame `Percent.Position` with the columns `square` and `percent`. We then sort the `Percent.Position` data frame by ordering the rows of the data frame by the `percent` column, `Percent.Position$percent`. The “`-`” sign in `-Percent.Position$percent` orders from largest to smallest; otherwise it would be smallest to largest. Leaving the column location after the comma blank returns all columns. In reading the last line, we have all the columns of `Percent.Position` with the rows ordered from largest to smallest based on the `percent` column.

R Code

```
> square=1:max.square
> percent=probabilities*100
> Percent.Position = data.frame(square, percent)
> Percent.Position[order(-Percent.Position$percent),]

  square      prob
6       6 2.565157
5       5 2.202578
11      11 2.094269
12      12 2.070616
10      10 2.056853
16      16 2.044828

:
4       4 1.880249
7       7 1.806643
3       3 1.609853
2       2 1.386912
1       1 1.192080
```

We now move to solving the problem when we are allowed to place three coins on the board. We should note that landing on squares is not independent and so we shouldn't necessarily use square 6, 5, and 11 from the one-coin analysis. From the one-coin analysis it is clear that the variability of landing on squares diminishes the farther along the board we go. Hence, we expect that

the optimal locations for placing the three coins occur in the early squares. Consequently we set max.square to 20 after setting trials to 100000 and rolls to 8. In this case, the average roll of the die is 3.5 and so on average we should cover 28 squares. This is good enough and it will make it easier to follow the output with the examples below. As in the one-coin example, we use **replicate** to generate our simulation of landing on squares with **sample** returning rolls random numbers from 1 to 6 with replacement to which we apply **cumsum** for the cumulative sum of the numbers. Executing simulation[,1:5] returns all rows by leaving no argument in the space before the comma for “all” and columns 1 through 5 of the matrix simulation.

R Code

```
> trials=100000
> rolls=8
> max.square=20
> simulation=replicate(trials,
cumsum(sample(6,rolls,replace=TRUE)))
> simulation[,1:5]
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	6	4	6	1	3
[2,]	12	9	7	5	8
[3,]	14	12	10	8	9
[4,]	19	17	14	14	14
[5,]	23	23	20	17	15
[6,]	27	25	21	23	17
[7,]	32	28	27	28	23
[8,]	33	34	33	33	27

We are going to construct a new two-column matrix from our simulation matrix. The first column will be the columns of simulation stacked on one another; the second column will denote the column number from the simulation matrix. We will call this new matrix landing.squares. The **cbind** command, column bind, combines vectors as columns in a matrix. Here **c(simulation)** returns the values of simulation as a vector by stacking the columns. Within the **rep**, replicate, function we have two elements. The first is **seq_len(trials)**, a sequence from 1 to trials, and the second sets the number of times each element of **seq_len(trials)** is repeated. For example, **rep(seq_len(5),each=2)** returns 1 1 2 2 3 3 4 4 5 5. In summary, **cbind** is combining two vectors into a matrix where the first column is the columns of simulation stacked and the second column is the sequence from 1 to trials with each value repeated rolls times. We return the first 17 rows and all columns of landing.squares with landing.squares[1:17,], which can be compared to simulation[,1:5] above.

R Code

```
> landing.squares=cbind(c(simulation),
  rep(seq_len(trials),each=rolls))
> landing.squares[1:17,]
```

	[,1]	[,2]
[1,]	6	1
[2,]	12	1
[3,]	14	1
[4,]	19	1
[5,]	23	1
[6,]	27	1
[7,]	32	1
[8,]	33	1
[9,]	4	2
[10,]	9	2
[11,]	12	2
[12,]	17	2
[13,]	23	2
[14,]	25	2
[15,]	28	2
[16,]	34	2
[17,]	6	3

Our next step is to remove rows of `landing.squares` whenever the value exceeds `max.square`. We call this new matrix `landing.squares.cut` and we will use a logical statement within the square brackets of `landing.squares`. With `landing.squares[, 1] <= max.square` we apply a logical test to see if each value in the first column is less than or equal to `max.square` and returned is either TRUE or FALSE, creating a logical vector of TRUE and FALSE. With this vector as the first index of `landing.squares`, the TRUE rows will be kept. The empty second index means all columns are returned. We see the first 17 rows of `landing.squares.cut` and both columns with `landing.squares.cut[1:17,]`. We can compare this to `landing.squares[1:17,]` above.

R Code

```
> landing.squares.cut=
  landing.squares[landing.squares[, 1] <= max.square,]
> landing.squares.cut[1:17,]
```

	[,1]	[,2]
[1,]	6	1
[2,]	12	1

[3,]	14	1
[4,]	19	1
[5,]	4	2
[6,]	9	2
[7,]	12	2
[8,]	17	2
[9,]	6	3
[10,]	7	3
[11,]	10	3
[12,]	14	3
[13,]	20	3
[14,]	1	4
[15,]	5	4
[16,]	8	4
[17,]	14	4

We will now create a matrix of 0's and 1's where each column will be one of our simulations, each row a square on the board, and a 1 will indicate that the associated square was occupied during the simulation. We do this by first creating a matrix of zeroes, games, with max.square rows and trials columns. The next statement assigns 1 based on the value in landing.squares.cut. In this context, rows of landing.squares.cut are viewed as a pair representing a row and a column of games. These locations in the matrix games will be assigned a 1. For example the first row of landing.squares.cut is 6 and 1 and games[6,1] has a value of 1. We return the first 17 rows and five columns of games to be compared with landing.squares.cut above.

R Code

```
> games=matrix(0,nrow=max.square,ncol=trials)
> games[landing.squares.cut]=1
> games[1:17,1:5]
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0	0	0	1	0
[2,]	0	0	0	0	0
[3,]	0	0	0	0	1
[4,]	0	1	0	0	0
[5,]	0	0	0	1	0
[6,]	1	0	1	0	0
[7,]	0	0	1	0	0
[8,]	0	0	0	1	1
[9,]	0	1	0	0	1
[10,]	0	0	1	0	0

[11,]	0	0	0	0	0
[12,]	1	1	0	0	0
[13,]	0	0	0	0	0
[14,]	1	0	1	1	1
[15,]	0	0	0	0	1
[16,]	0	0	0	0	0
[17,]	0	1	0	1	1

Our goal now is to check all combinations of the three squares we could place coins on to see which combination yields the highest probability of winning. The $3 \times \text{max}(\text{square})$ matrix `coin.placements` is all combinations of coin placements returned by `combn`. We return all rows and the first 5 columns with `coin.placements[,1:5]`.

R Code

```
> coin.placements=combn(max.square,3)
> coin.placements[,1:5]

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

We next create a function that will check a given combination of coin placements, which we will use within the `apply` function. We call the variable of the `calculate.success` function choice and it will be a three-element vector representing an option for placing the three coins.

To illustrate this function, we will consider a particular coin placement; namely 6,9,12. Based on the first 5 columns of the games matrix above, consider what `colSums(games[c(6,9,12),])` returns. In this case we get the vector 2, 2, 1, 0, 1 since, for example, in the first column of games we occupied squares 6,12,14. Since `colSums(games[c(6,9,12),]) > 0` is a logical statement it returns TRUE, TRUE, TRUE, FALSE, TRUE, in other words, a TRUE whenever the column sum is positive. We now take the mean of a logical vector, which will assign a 1 to TRUE and 0 to FALSE and compute the mean. In our example, `mean(colSums(games[c(6,9,12),]) > 0)` is 0.80; we won in 80% of the games. As our function only computes one value it will return that value without a `return` statement.

The arguments of the `apply` function are a matrix (`sapply` takes a vector only), and a second variable, 2, which specifies to use the columns as inputs to the function, and a function. Every three-coin placement (i.e., every column of `coin.placement`) will be evaluated by `calculate.success`, and the probability of winning is returned. The first 5 values of `three.coin.probabilities` is displayed

by `three.coin.probabilities[1:5]`. The first value, 0.50200, is the probability of winning if we place the coins on squares 1, 2, and 3.

R Code

```
> calculate.success=function(choice)
{mean(colSums(games[choice, ]) > 0)}
> three.coin.probabilities=apply(coin.placements,2,
calculate.success)
> three.coin.probabilities[1:5]
```

```
[1] 0.50200 0.53001 0.56142 0.60023 0.47628
```

The largest value in `three.coin.probabilities` represents our highest probability of winning, found by `max(three.coin.probabilities)`, but this doesn't tell us where to place the coins. The function `which.max` returns the location that produces the max (or specifically the first location of the max if there are ties). The 461st column of `coin.placements` is our best placement and found by `coin.placements[,which.max(three.coin.probabilities)]`. We have a 79% of winning if we place our three coins on 4, 5, and 6.

R Code

```
> max(three.coin.probabilities)
[1] 0.79346
```

R Code

```
> which.max(three.coin.probabilities)
[1] 461
```

R Code

```
> coin.placements[,which.max(three.coin.probabilities)]
[1] 4 5 6
```

Code Review for the Simulation Chapter

`apply(A,i,function)` (for the example in this chapter) returns a vector from applying the function to either the columns ($i = 2$) or rows ($i = 1$) of the matrix A .

barplot(v) creates a bar plot with the elements of tallies in vector v . Typical graphing options are available to use with `barplot()`.

cumsum(v) returns a vector where the i element is the sum of the values up to and including the i th element of the vector v .

format(r,scientific=,big.mark="") formats the number r for output as either scientific (TRUE or FALSE) with every three digits separated by what is placed within the quotes (typically a comma).

matrix(v,byrow=,nrow=) creates a matrix from the vector v by-row (TRUE or FALSE) with the given number of rows.

max(v) returns the maximum element of the vector v .

rbinom(s,n,p) returns a vector of length s where each element is the number of successes in n trials of a binomial random variable with probability of success p .

replicate(n,function()) returns a matrix where the columns are formed by output to repeated applications of the function.

rnorm(n,μ,σ) returns a vector of n values from a normal distribution with mean μ and standard deviation σ .

sapply(v,f(x){expression}) (for the example in this chapter) returns a vector from applying the function described by expression to each element of v , where f is a function of x .

tabulate(v,m) returns a vector of length m where the i th element is the number of times i occurs in vector v .

which.max((v)) returns the location of the first occurrence of the maximum element of the vector v .

10.7 Exercises

1. Estimate the probability of 10 or more tails in 20 coin flips if the probability of heads is 40%. Find at least two ways to modify the code in [Section 10.1](#) to estimate this probability.
2. From [Section 10.2](#), calculate the probability the elevator is over capacity if there is a 60% chance of selecting a female and 13 people are on the elevator. Hint use `rbinom()`.

3. In [Section 10.3](#) add a fourth card that is red/blue and answer the same question.
4. Add a fourth die to the example in [Section 10.4](#) and a payout of \$8 for winning on all four dice. How much would you be willing to play this game? How much would the payout have to be for winning on all four so that it is worth paying \$1 to play?
5. From [Section 10.5](#) generalize the simulation by creating a function with inputs simulation size, needle length, and line spacing, and returns the inverse probability of a needle crossing a line.
6. In the one-coin game from [Section 10.6](#), instead of rolling one die, roll two four-sided dice and advance based on the sum of the dice. Where should you put your one coin to maximize the chance of landing on the coin?
7. In the three-coin game from [Section 10.6](#), what is the best placement of coins if we may not place the coins on adjacent squares (variant mentioned by Robinson [23])?



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

11

The Central Limit Theorem and Z-test

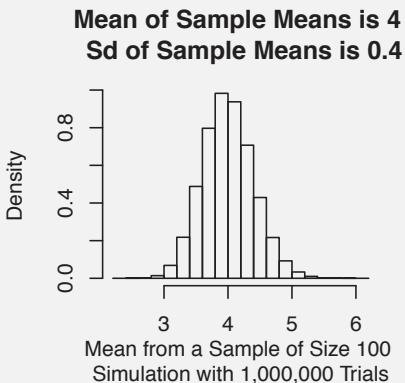
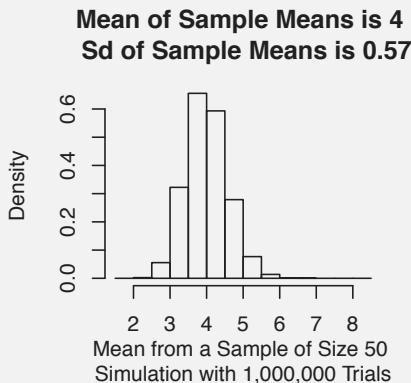
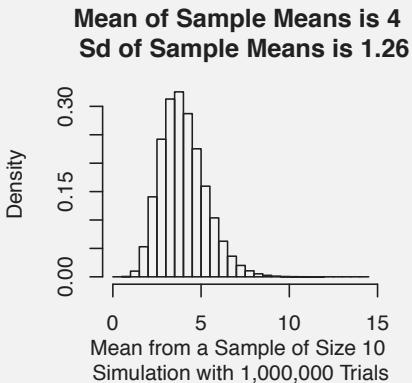
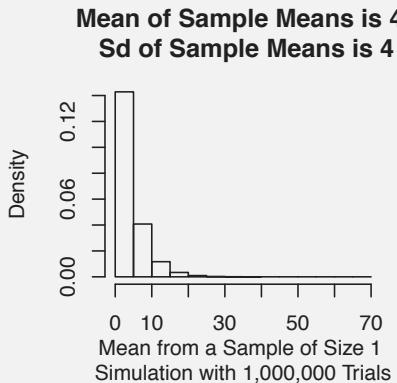
In this chapter we provide two simulations to illustrate the central limit theorem. We follow that up by demonstrating how to perform a z-test and calculate associated confidence intervals. For both, we show how to perform tests and compute confidence intervals when working from a data set and from summary statistics. The data we use in the chapter for the z-test and interval will be generated randomly from a normal distribution, while the simulations will use data generated from exponential and uniform distributions. The `z.test` and `zsum.test` in this chapter use the `BSDA` package. Please note that you should only load a package once in an R session, although the code below will have **library(BSDA)** whenever it is needed as a reminder that it is being used.

11.1 A Central Limit Theorem Simulation

We demonstrate two central limit theorem simulations, with the differences being the distribution we sample from and how the breaks are set in the histograms. Our first example begins by setting the graph frame, `par(mfrow=c(2,2))`, to accept four graphs in a 2-by-2 grid. We set `trials=1000000`, which is the size of the simulation. This isn't necessary to do here, but it makes it easier to change the simulation size in one place in the code. We begin a for loop for values of $i = 1, 10, 50$, and 100 , which will be the sizes of our samples for the distribution given two lines later. Note that the for loop begins with { and ends with }. Depending on your computer, this simulation size make take a few minutes to complete. The real work in the code is in the next line where `simulations` stores the output of `replicate(trials,mean(rexp(i,0.25)))`. Unraveling this from the inside out, `rexp(i,0.25)` selects i random values from the exponential distribution with $\lambda = 0.25$, and `mean(rexp(i,0.25))` calculates the mean of these values. The `replicate(n,f)` function repeats an operation `f`, `n` times. In our case we repeat `mean(rexp(i,0.25))` `1000000` (`trials`) times. The output is a vector of values, in this case of size `1000000`, which is set to the variable `simulation`. The next two lines calculate the mean and standard deviation of the vector of values in `simulation` and sets them to `x.bar` and `s`.

R Code

```
> par(mfrow=c(2,2))
> trials=1000000
> for(i in c(1,10,50,100)){
+ simulation=replicate(trials,mean(rexp(i,0.25)))
+ x.bar=mean(simulation)
+ s=sd(simulation)
+ xlabel=paste("Mean from a Sample of Size", i,
+ "\nSimulation with",format(trials, big.mark=",",
+ scientific=FALSE) , "Trials")
+ title=paste("Mean of Sample Means is",round(x.bar,2),
+ "\n Sd of Sample Means is",round(s,2))
+ hist(simulation,freq=FALSE,xlab=xlabel,main=title)
+ }
```



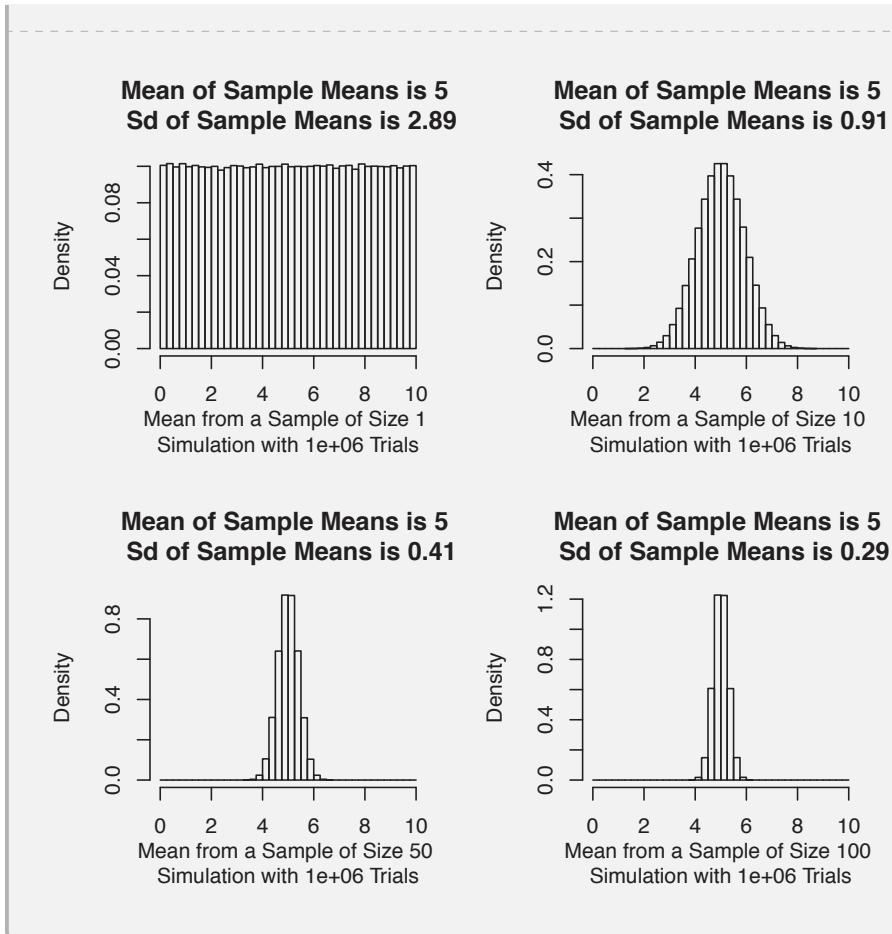
The rest of the code is focused on creating the histogram. We set xlabel to

desired text printed below the x -axis. The **paste** function combines characters, in quotes, and variables, not in quotes, where `\n` is the new line character, which forces a line break. We use the **format** function to add commas to the number associated to trials, `big.mark=","`, and to make sure we do not use scientific notation. The next line creates the title. Again we use **paste** and this time `u` and `s` are rounded to two decimal places, although **round** will drop zeroes for the output. Compare these lines of code to what appears in the histograms. Our last line creates the histograms with **hist** of the simulation data. The output is percentages instead of frequencies because `freq=FALSE`. The `xlab` and `main` are set to our code from `xlabel` and `title`. We could have put `xlabel` and `title` code directly into the **hist** function, but it would become difficult to read and follow. In this example, the histogram bins are set by default; in the next example we will set the bins. More details on histograms can be found in [Section 3.5](#).

The next example is almost identical to the first and so we only highlight the differences. In this case we sample from a uniform distribution over the interval $[0, 10]$, **runif(i,0,10)** (this is `r(random)` `unif(orm)` not `run if`). As an aside there is also **dunif**, **punif**, and **qunif** for the uniform density, distribution, and quantile functions respectively. This is consistent with other distributions in that `r(distribution name)`, `d(distribution name)`, `p(distribution name)`, and `q(distribution name)` is the naming convention for these functions. Because we know that the values of the simulation will always fall between 0 and 10 we set the breaks for the histogram with `breaks` set to **seq(0,10,by=0.25)**. In general, `breaks` can be set by providing a vector of values for the breaks. In this case we created the vector with **seq**, which generates a sequence of numbers from 0 to 10 by increments of 0.25.

R Code

```
> par(mfrow=c(2,2))
> trials=1000000
> for(i in c(1,10,50,100)){
+   simulation=replicate(trials,mean(runif(i,0,10)))
+   x.bar=mean(simulation)
+   s=sd(simulation)
+   xlabel=paste("Mean from a Sample of Size", i,
+ "\nSimulation with",trials, "Trials")
+   title=paste("Mean of Sample Means is",round(x.bar,2),
+ "\n Sd of Sample Means is",round(s,2))
+   hist(simulation,freq=FALSE,breaks=seq(0,10,by=0.25),
+ xlab=xlabel,main=title)
+ }
```



11.2 Z Test and Interval for One Mean

The Z test is not a base function for R and so we will use the **BSDA** package. The code begins by opening this package. In this example, we will generate data from a standard normal and then test to see if it has mean 0. Our second line of code, **set.seed(42)**, sets the seed for the random number generator to 42, which is arbitrary. Sometimes for testing purposes it is convenient to run a program repeatedly with the same “random” data. Specifying a seed ensures that you will get the same “random” numbers every time. If you want the same results here you will need to do this; otherwise leave out the line of code. We use **rnorm(100,0,1)** to sample 100 values from a normal

distribution with mean 0 and standard deviation 1. The vector of 100 values is set to sample.1. The test is performed with the **z.test** function. The first value must be the data; in this case sample.1. The null hypothesis is set by mu=0. The alternative hypothesis is set by alternative with the options “two.sided”, “less”, or “greater”, where “two.sided” is the default so we didn’t need to include it here. Similarly, the default values for mu and sigma.x are 0 and 1 respectively. The default confidence interval is 95%, and can be set to say 90% with conf.level=0.90. The last line is result, which outputs the summary information of the z-test. By setting result=z.test we are able to extract specific values from the output as we will see next. If we don’t need to access specific values we can simply run z.test without saving the information to result.

R Code

```
> library(BSDA)
> set.seed(42)
> sample.1=rnorm(100,0,1)
> result=z.test(sample.1,alternative="two.sided",mu=0,
sigma.x=1)
> result
```

One-sample z-Test

```
data: sample.1
z = 0.32515, p-value = 0.7451
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
-0.1634816 0.2285112
sample estimates:
mean of x
0.03251482
```

To find out what we can access by setting result=z.test, we use the **names** function. The output lists eight variables that are produced by the test. So, for example, result\$p.value returns the p-value from the **z.test** output.

R Code

```
> names(result)

[1] "statistic"  "p.value"    "conf.int"   "estimate"
[5] "null.value" "alternative" "method"    "data.name"
```

R Code

```
> result$p.value
[1] 0.7450689
```

Similarly, `result$conf.int` returns the confidence interval. If we want to access individual endpoints of the interval then `result$conf.int[1]` and `result$conf.int[2]`, will provide them. As a rule [i] accesses values from a vector.

R Code

```
> result$conf.int
[1] -0.1634816  0.2285112
attr(,"conf.level")
[1] 0.95
```

R Code

```
> result$conf.int[1]
[1] -0.1634816
```

R Code

```
> result$conf.int[2]
[1] 0.2285112
```

There are situations where we have summary statistics instead of the data for our test. Our next example demonstrates this situation. We will use the same data set as above and our first two lines calculate the sample mean and sample size, set to `mu.1` and `n.1`, respectively. The **`zsum.test`** is part of the **BSDA** package also and so we use **`library(BSDA)`**. Note, you only have to load a package once during an R session. You should not keep invoking **`library(BSDA)`** if it has already been done. The **`zsum.test`** function is very similar to **`z.test`**. The first three entries provide our sample statistics and a value for sigma. We set the value of the null hypothesis with `mu=0`, which is actually the default. The alternative hypothesis choices are “two.sided”, “less”, or “greater”, with “two.sided” as the default and so we didn’t need to include it here. The default confidence interval is 95% and we used `conf.level=0.90` for a 90% confidence interval in this example. As above we can use `result.sum.1` to access the same values as with `z.test`. Use **`names(result.sum.1)`** to see. Note that the summary information here is the same as for **`z.test`**, as it should be in this case.

R Code

```
> mu.1=mean(sample.1)
> n.1=length(sample.1)
> library(BSDA)
> result.sum.1=zsum.test(mean.x=mu.1,sigma.x=1,n.x=n.1,
mu=0,alternative="two.sided",conf.level=0.90)
> result.sum.1
```

One-sample z-Test

```
data: Summarized x
z = 0.32515, p-value = 0.7451
alternative hypothesis: true mean is not equal to 0
90 percent confidence interval:
-0.1319705 0.1970002
sample estimates:
mean of x
0.03251482
```

11.3 Z Test and Interval for Two Means

Performing a z-test and calculating a confidence interval for two means is almost exactly the same as for one mean. We will use sample.1 from above for one data set. Run **set.seed(42)** and **sample.1=rnorm(100,0,1)** if necessary. We generate another data set of size $n = 125$ from **rnorm** with mean 0 and standard deviation 1. Again we need the BSDA package. We use **z.test** handing it the data sets, sample.1 and sample.2, as the first two arguments. After that we set the null hypothesis, the difference of population means, with $\mu_1=0$ and the alternative hypothesis (choices “two.sided”, “less”, or “greater”). We provide the population standard deviation for both data sets and set the confidence level. As above result.2 contains the output of the test and can be used to access specific values.

R Code

```
> sample.2=rnorm(125,0,1)
> library(BSDA\)
> result.2=z.test(sample.1,sample.2,mu=0,alternative=
"greater",sigma.x=1,sigma.y=1,conf.level=0.90)
> result.2
```

Two-sample z-Test

```

data: sample.1 and sample.2
z = 0.96948, p-value = 0.1662
alternative hypothesis: true difference in means is
greater than 0
90 percent confidence interval:
-0.04186927      NA
sample estimates:
mean of x  mean of y
0.03251482 -0.09755410

```

If we have summary statistics instead of the data we can use **zsum.test**, which is also used for a 2-sample test by including information about the second sample. Everything works in the same way as the examples above.

R Code

```

> mu.1=mean(sample.1)
> mu.2=mean(sample.2)
> length.1=length(sample.1)
> length.2=length(sample.2)
> library(BSDA)
> result.sum.2=zsum.test(mean.x=mu.1,sigma.x=1,
n.x=length.1,mean.y=mu.2,sigma.y=1,n.y=length.2,
alternative="two.sided",mu=0,conf.level=0.95)
> result.sum.2

```

Two-sample z-Test

```

data: Summarized x and y
z = 0.96948, p-value = 0.3323
alternative hypothesis: true difference in means is not
equal to 0
95 percent confidence interval:
-0.1328878  0.3930257
sample estimates:
mean of x  mean of y
0.03251482 -0.09755410

```

Code Review from The Central Limit Theorem and Z-test Chapter

hist(data) creates a histogram of the data. Two key options are freq, the default is TRUE, and breaks, to set the breakpoints between cells.

replicate(n, f) returns a matrix of n columns where the columns are the output of f.

rexp(i,λ) returns a vector of i random variables from an exponential distribution with parameter λ .

rnormr(i,mu,sig) returns a vector of i random variables from a normal distribution with mean mu and standard deviation sig.

runif(i,a,b) returns a vector of i random variables from a uniform distribution on the interval $[a, b]$.

zsum.test(mean.x=,sigma.x=,n.x=,mean.y=,sigma.y=,n.y=, alternative=,mu=,conf.level=) is used for a one or two sample z-test with summary statistics instead of data. The .y options are removed for a one sample test. The alternative options are “two.sided”, “less”, or “greater”. This uses the BSDA package.

z.test(data.x,data.y,sigam.x=,sigma.y=,alternative=,mu=, conf.level=) is used for a one or two sample z-test. The .y options are removed for a one sample test. The alternative options are “two.sided”, “less”, or “greater”. This uses the BSDA package.

11.4 Exercises

1. Create a CLT simulation with a different distribution. Run ?Distributions to see what distributions are available. Create six graphs instead of the four used in the examples.
2. Create a CLT function that takes the sample size and the simulation size, and returns a histogram. Add the breaks for the function as an input.
3. Calculate the margin of error for the z.test example.
4. Create a graph of margin of error vs. confidence level for a z interval.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

12

The T-Test

The main goal of this chapter is to illustrate t-tests: one-sample, two-sample, and paired. Along the way, we will test data for normality and create a qqplot. As a secondary goal, this chapter provides some basics on working with data frames. Our examples here will use one of the data sets available with R, and we note that when we use **read.table** to import data (see [Section 1.1](#)) the form of a the data is a data frame. In short, a data frame is an array of data where the data can be numeric, logical, character, etc., whereas entries in a matrix are numeric.

12.1 T Test and Intervals for One and Two Means

Our examples will use the mtcars data set available in R. The mtcars data set is from the 1974 Motor Trend US magazine. We use **data(mtcars)** to make the data set available in an R session and then **str** to see the structure of the data. The output informs us that there are 32 observations with 11 variables that are all numeric. To get a five-number summary and the mean for each data set use **summary(mtcars)**.

R Code

```
data(mtcars)
str(mtcars)

'data.frame': 32 obs. of 11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 ...
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num 160 160 108 258 360 ...
 $ hp : num 110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 ...
 $ wt : num 2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num 16.5 17 18.6 19.4 17 ...
 $ vs : num 0 0 1 1 0 1 0 1 1 1 ...
 $ am : num 1 1 1 0 0 0 0 0 0 0 ...
```

```
$ gear: num  4 4 4 3 3 3 3 4 4 4 ...
$ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

To illustrate the t-test we will test to see if this set of cars exceeds the mpg ratings in 1974 of 13.6 miles per gallon and then compare the 4-cylinder cars to the 6- and 8- cylinder cars. We begin by checking to see if the data set is normal. We use the Shapiro-Wilk normality test, **shapiro.test**, with the mpg data, mtcars\$mpg. The null hypothesis states that the data is normal and with a p-value of 0.1229 we fail to reject.

R Code

```
> shapiro.test(mtcars$mpg)

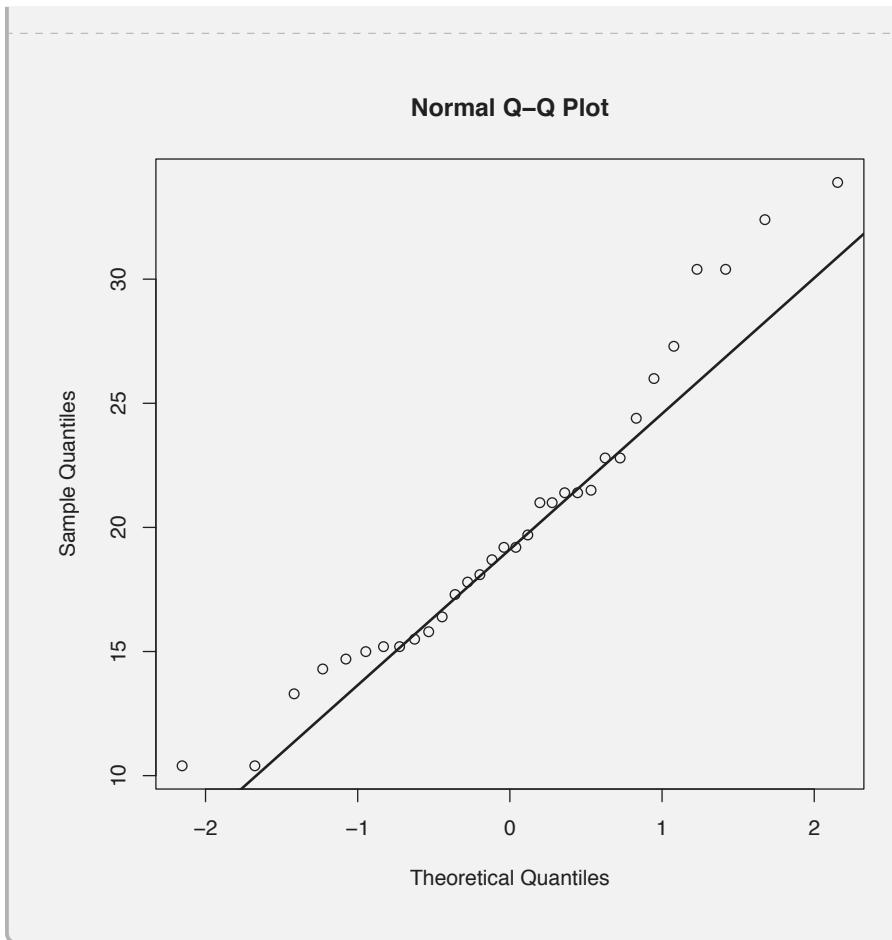
Shapiro-Wilk normality test

data: mtcars$mpg
W = 0.94756, p-value = 0.1229
```

A standard plot is the quantile-quantile plot, **qqnorm**, and we add the line with **qqline**. The **qqline** function will accept options such as line width, here set to 2, and color, col.

R Code

```
> qqnorm(mtcars$mpg)
> qqline(mtcars$mpg,lwd=2)
```



We perform a t-test with the **t.test** function. The first input is our data set and the second sets $\mu_0 = 13.6$. As we are testing to see if the cars exceed 13.6 we have alternative="greater", with the "less" and "two.sided" as the other two options. The default confidence level is 95% and can be changed with conf.level=0.xx. We set the output to result so that we can access specific values. This is not necessary if all we wanted was the results. In this case we reject the null hypothesis and conclude cars exceed the 13.6 mpg standard. Of course, there is no reason to believe that we have a representative sample of cars from 1974.

R Code

```
> result=t.test(mtcars$mpg,mu=13.6,  
+alternative="greater")  
> result
```

One Sample t-test

```

data: mtcars$mpg
t = 6.0921, df = 31, p-value = 4.723e-07
alternative hypothesis: true mean is greater than 13.6
95 percent confidence interval:
 18.28418      Inf
sample estimates:
mean of x
20.09062

```

To see what values we can obtain from the **t.test** we use **names(result)**, which gives us a list of variables associated with the test by using **test\$xxx**. For example **result\$estimate** outputs the sample mean.

R Code

```

> names(result)

[1] "statistic"   "parameter"   "p.value"     "conf.int"
[5] "estimate"    "null.value"   "alternative" "method"
[9] "data.name"

```

R Code

```

> result$estimate

mean of x
20.09062

```

Similarly, **result\$conf.int** outputs the confidence interval. If we want the specific values of the endpoints we use **result\$conf.in[1]** and **result\$conf.in[2]**.

R Code

```

> result$conf.int

[1] 18.28418      Inf
attr(,"conf.level")
[1] 0.95

```

R Code

```
> result$conf.int[1]
[1] 18.28418
```

R Code

```
> result$conf.int[2]
[1] Inf
```

We now compare the 4-cylinder cars to the 6- and 8-cylinder cars. To do this we will create new data frames for the two categories of cars. The key to this is in our first line, `cars.4cyl=mtcars[mtcars$cyl==4,]`. Mtcars is a data frame and if we typed `mtcars[3,4]` we would get the value in the third row and fourth column, 93. In this case, we included a logical statement in the first location, `mtcars$cyl==4`, to obtain all rows where cyl is 4. The use of double equal signs, `==`, is a logical statement. We left the second location, columns, empty so that all columns are included. We really only need the mpg column and so could have used `mtcars[mtcars$cyl==4, "mpg"]` or `mtcars[mtcars$cyl==4, 1]` since mpg is the first column. This data frame is set to `cars.4cyl` and we check the first three rows with **head**. Note: The last column carb has been cut from the output.

R Code

```
> cars.4cyl=mtcars[mtcars$cyl==4,]
> head(cars.4cyl, n=3)

      mpg cyl disp hp drat wt qsec vs am gear
Datsun 710 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4
Merc 240D 24.4   4 146.7 62 3.69 3.19 20.00  1  0    4
Merc 230 22.8   4 140.8 95 3.92 3.15 22.90  1  0    4
```

Similarly, we create the data frame of 6- and 8-cylinder cars with `mtcars[mtcars$cyl!=4,]` where `!=` is not equal. Other logical operators are `<`, `<=`, `>`, `>=`, `|` (or), and `&` (and). Again, we use **head** to check the first three rows. Note: The last two columns gear and carb have been cut from the output.

R Code

```
> cars.not.4cyl=mtcars[mtcars$cyl!=4,]
> head(cars.not.4cyl,n=3)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0

We still use **t.test** for a two-sample test by including the independent data sets as the first two entries to the function. All other options remain available. In the example here we perform a two-sided test and create a 90% confidence interval. The output of the test is set to result.2. In this case, we reject the null hypothesis. As in the previous example, **names(result.2)** will list the variables of the test that can be accessed, which are the same as in the previous example. The one difference is that when appropriate the output will have two values instead of one. For example result.2\$estimate provides two values, whereas result\$estimate has only one. In this case, result.2\$estimate[1] and result.2\$estimate[2] are the means of the first and second data set, respectively.

R Code

```
> result.2=t.test(cars.4cyl$mpg,cars.not.4cyl$mpg,
alternative="two.sided",conf.level=0.90)
> result.2

Welch Two Sample t-test

data: cars.4cyl$mpg and cars.not.4cyl$mpg
t = 6.5737, df = 15.266, p-value = 8.09e-06
alternative hypothesis: true difference in means is
not equal to 0
90 percent confidence interval:
 7.348035 12.683999
sample estimates:
mean of x mean of y
26.66364 16.64762
```

If we have summary statistics instead of data, we will need a different function. For an example, we calculate the statistics for the two-sample test above. We then use the BSDA package. It is mostly a package of data sets, which can be useful, but also has some statistical tests.

R Code

```
x.bar.4=mean(cars.4cyl$mpg)
sd.4=sd(cars.4cyl$mpg)
n.4=length(cars.4cyl$mpg)
x.bar.not.4=mean(cars.not.4cyl$mpg)
```

```
sd.not.4=sd(cars.not.4cyl$mpg)
n.not.4=length(cars.not.4cyl$mpg)
```

No output

We begin the code by opening the BSDA package. The **tsum.test** function takes the mean, standard deviation, and sample size of the data. In the example here we perform the same two-sample test as above, and we get the same output. We can use **tsum.test** for one sample, but then we will need to set $\mu_0 = \mu_0$ for some value μ_0 . The default is 0, so it was left out of the two-sample test above. The **tsum.test** function has the same options as **t.test**. The alternative can be set to “two.sided”, “greater”, or “less”, with “two.sided” as the default. The confidence level is defined with conf.level=0.xx, with 0.95 the default.

As above, **names(result.sum.2)** provides variable names associated with result.sum.2 and result.sum.2\$estimate, result.sum.2\$conf.int, result.sum.2\$conf.int[1], and result.sum.2\$conf.int[2] provide values as above.

R Code

```
> library(BSDA)
> result.sum.2=tsum.test(x.bar.4, sd.4, n.4, x.bar.not.4,
sd.not.4, n.not.4)
> result.sum.2
```

Welch Modified Two-Sample t-Test

```
data: Summarized x and y
t = 6.5737, df = 15.266, p-value = 8.09e-06
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
 6.773357 13.258678
sample estimates:
mean of x mean of y
26.66364 16.64762
```

12.1.1 Paired T-Test

We really don't need a separate section to demonstrate the paired t-test (but by creating a separate section the paired t-test shows up in the Table of Contents). For a paired t-test, use **t.test** with the same values as with a two-sample test, but include paired=TRUE because the default is false.

12.2 Illustrating the Meaning of a Confidence Interval Simulation

A 90% confidence interval for a population mean has a 90% chance of containing the population mean. The next two examples illustrate this fact. The first provides numeric output while the second has a graphical component. In both cases, we will generate data from a normal distribution and then use the **t.test** function to create confidence intervals.

Our first example begins by setting the variable `misses` to 0, which will be used as a counter in the for loop that begins a few lines later. We set our `sample.size` to 200, `sim` (simulations) to 10000, and `c.level` to 0.90 in the second, third, and fourth lines so that it is easy to change them. For example, `sim` is used twice and if we wanted to change the size of the simulation we would have to do it in both places instead of once at the beginning. The idea behind the for loop is to generate a data set of size `sample.size` (in this case 200), calculate a confidence interval based on the data, and test to see if the interval contains the mean. We repeat this `sim` times. So, `for(i in 1:sim)` controls the loop where `i` varies from 1 to `sim`. The for loop begins with a `{` and ends with `}`. We use **rnorm** to generate a data set of size `sample.size` from a normal distribution with mean 10 and standard deviation 2. We create a 90% confidence interval with **t.test** and set it to `result`. Now, if the left endpoint of the interval, `result$conf.int[1]`, is greater than 10 or, `|`, the right endpoint, `result$conf.int[2]`, is less than 10, then the population mean of 10 is outside the interval. In this case, increase the counter by 1, `misses=misses+1`. This process is repeated `sim` times.

The last line, outside the for loop, produces the output by pasting the value of `misses/sim*100`, and the characters in quotes into one line. We use `sep=""` with no space between the quotes so that no space is placed between the value and the character string. If left out, the default is to place a space at the comma. The result is that close to 10% of the confidence intervals did not contain the population mean, which is what we expect. Note that your results will not necessarily be the same as we didn't set a seed here.

R Code

```
> misses=0
> sample.size=200
> sim=10000
> c.level=0.90
> for(i in 1:sim){
+ data=rnorm(sample.size,10,2)
+ result=t.test(data,mu=10,conf.level=c.level)
+ if(10 < result$conf.int[1] || 10 > result$conf.int[2])
+ misses=misses+1
}
> misses/sim*100
```

```
{misses=misses+1} }
> paste(misses/sim*100,"%","of the Intervals Missed
the True Mean")

[1] "9.73% of the Intervals Missed the True Mean"
```

Our next example creates a visual of the relationship between confidence intervals and the population mean. We begin by setting two counters, misses and samp.mean.est, to 0 and the number of simulations to 100. In this case, each confidence interval will be represented graphically and more than 100 becomes difficult to visualize. We next initiate a plot. The plot is of type="n" or empty. In other words, the point (0,0) won't appear. We set the *x*-axis to extend from 9.5 to 10.5 and the *y*-axis to extend from 0 to 7% more than the number of simulations. (A bit of trial and error determined that this created enough extra space at the top of the graph for a legend.) Axes are suppressed with axes=FALSE. The labels for the *x*-axis and *y*-axis were set to blank with empty quotes. We define our own *x*-axis with the **axis** function. The first element 1, is for the *x*-axis, with 2,3, and 4 moving clockwise around the graph. The at option says where to put the labels. We want numbers at 9.5, 10, and 10.5, so we create a sequence, **seq**, from 9.5 to 10.5 in steps of 0.5. We also want those as the labels and so label is also **seq(9.5,10.5, 0.5)**. We add a vertical red line at *x* = 10 of width 2 with **abline(v=10,lwd=2,col="red")**, where v=10 creates a vertical line at *x* = 10.

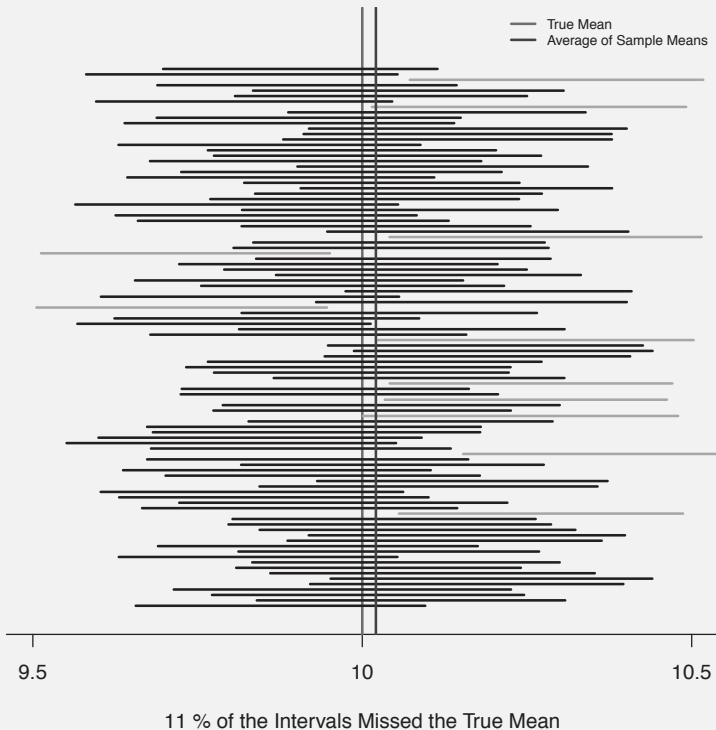
R Code

```
> misses=0
> samp.mean.est=0
> sim=100
> c.level=0.90
> plot(0,0,type="n",xlim=c(9.5,10.5),
+       ylim=c(0,1.07*sim),axes=FALSE,ylab="",xlab="")
> axis(1,at=seq(9,11,0.5),label=seq(9,11,0.5))
> abline(v=10,lwd=2,col="red")
> for(i in 1:sim){data=rnorm(200,10,2)
+   result=t.test(data, mu=10, conf.level=c.level)
+   if( 10<result$conf.int[1] | 10>result$conf.int[2]){
+     misses=misses+1
+     segments(result$conf.int[1],i,result$conf.int[2],i,
+               lwd=2,col="green")}
+   else {segments(result$conf.int[1],i,
+                 result$conf.int[2],i,lwd=2)}
+   samp.mean.est=samp.mean.est+result$estimate}
> mtext(paste( misses/sim*100,"%","of the Intervals
```

```

Missed the True Mean"), side=1, line=3)
> abline(v=samp.mean.est/sim, lwd=2, col="blue")
> legend("topright", c("True Mean", "Average of Sample
Means"), lwd=c(2,2), col=c("red", "blue"), bty="n",
cex=0.65)

```



The for loop begins in the same way as the previous example, with data 200 random values from a normal distribution with mean 10 and standard deviation 2. The output of **t.test** is set to result. Again, if the left endpoint of the interval, test\$conf.int[1], is greater than 10 or (|) the right endpoint, test\$conf.int[2], is less than 10, then the population mean of 10 is outside the interval. In this case, increase the counter by 1, misses=misses+1. But also, we have the first new line with **segments** which graphs a horizontal line at $y = i$ from the left of the confidence interval, test\$conf.int[1], to the right of the interval, test\$conf.int[2], with a line width of 2 and colored green. The line that begins else executes drawing a line segment when the population mean is

contained in the interval, but this time the default color of black is used. We use samp.mean.est=samp.mean.est+result\$estimate to add the sample means each time we go through the for loop.

The last three lines are outside the for loop and add elements to the graph. The first, **mtext**, places text in the margin. The text is created with **paste** so that the value of misses/sim*100, and the characters in quotes are put into one line. We use sep="" with no space between the quotes so that no space is placed between the value and the character string. If left out, the default is to place a space at the comma. Side=1 represents the x -axis and it is placed at the third line below the graph. We place a blue vertical line at the average of the sample means from each interval by setting v=samp.mean.est/sim. The line has width 2. Last, we place a **legend** in the top right. The legend has two elements in this case and the second, third, and fourth options use the concatenate function **c**. In order, we define the text, the line width, and colors of each element. We remove the box around the legend with bty="n" and scale the text to 65% with cex=0.65.

Code Review from the T-Test Chapter

logic operators are <, <=, >, >=, | (or), and & (and).

qqnorm(x) creates a qqplot with the data x.

qqline(x) creates a qqline with the data x.

rnorm(n,u,s) returns n random numbers from a normal distribution with mean u and standard deviation s.

shapiro.test(x) performs the Shapiro-Wilk normality test on the data x.

str(df) provides basic information, or structure, about the data frame df.

tsum.test(mean.x=,sigma.x=,n.x=,mean.y=,sigma.y=,n.y=, alternative=,mu=,conf.level=) is used for a one or two sample t-test with summary statistics instead of data. The .y options are removed for a one sample test. The alternative options are “two.sided”, “less”, or “greater”. This uses the BSDA package.

t.text(data.x,data.y,alternative=,mu=,conf.level=) is used for a one or two sample t-test. The .y options are removed for a one sample test. The alternative options are “two.sided”, “less”, or “greater”. For a paired t-test use the option paired=TRUE.

12.3 Exercises

1. Create a data frame of 4-cylinder cars and just the mpg, hp, and wt columns. Hint use `c()`.
2. Create a dataframe where the cars weigh at least 3 tons and have at least 80 horsepower.
3. Test to see if the not 4-cylinder cars exceeds 13.6 mpg. Test to see if just 8-cylinder cars exceeds 13.6 mpg.
4. Test to see if the 4-cylinder cars are heavier or lighter than the 8-cylinder cars.
5. Use the built-in data set iris to test if there is any difference in petal length and sepal width between the versicolor and virginica iris species.
6. Create a function from the first confidence interval theory example. It should have four inputs: a simulation size, a sample size, a mean, and a standard deviation. The output is the number of times the confidence interval missed the true mean.
7. Create a graph to illustrate the relationship between margin of error and confidence level.

13

Testing Proportions

In testing a single proportion we present two options. The first is the traditional z-test, **prop.test**, and the second is an exact test, **binom.test**. The **prop.test** generalizes to two or more populations and we provide an example of testing two proportions. Our examples here will use data generated from a binomial distribution using **rbinom**. We end the chapter with an example of the error associated with the choice of α .

13.1 Tests and Intervals for One and Two Proportions

We begin this section by generating data to illustrate testing proportions. We let heads be the number of successes of a binomial random variable with $n=123$ and a probability of successes of 0.5. In this case, we have 66 successes and results will vary. We will test to see if this outcome is distinguishable from a probability of success of 0.55 in two ways. We will use a z-test and an exact test.

R Code

```
> heads=rbinom(1,size=123,prob=0.5)
> heads
[1] 66
```

The function **prop.test** performs a one-proportion z-test. We provide the number of successes, stored in `heads`, the sample size, 123, and a value for p_0 , 0.55. The alternative hypothesis is set by `alternative` with the options `two.sided`, `less`, or `greater` placed in quotes, where `two.sided` is the default. The default confidence interval is 95%, and can be set to, say, 90% with `conf.level=0.90`. By default `correct=TRUE` so that the continuity correction is used. It can be set to `FALSE`. Our p-value is 0.8349, so we fail to reject the null hypothesis.

R Code

```
> result.z=prop.test(heads,123,p=0.55,
alternative="two.sided")
> result.z

 1-sample proportions test with continuity correction

data: heads out of 123, null probability 0.55
X-squared = 0.043443, df = 1, p-value = 0.8349
alternative hypothesis: true p is not equal to 0.55
95 percent confidence interval:
 0.4447004 0.6261536
sample estimates:
      p
0.5365854
```

Alternatively, we can perform an exact test by using **binom.test**. The inputs are the same as that for **prop.test**, although there is no continuity correction option since it is an exact test.

R Code

```
> result.exact=binom.test(heads,123,p=0.55,
alternative="two.sided",conf.level=0.95)
> result.exact

Exact binomial test

data: heads and 123
number of successes=66, number of trials=123, p-value=
0.7862
alternative hypothesis:
  true probability of success is not equal to 0.55
95 percent confidence interval:
 0.4444475 0.6269206
sample estimates:
probability of success
 0.5365854
```

In both cases, setting the output to a variable, `result.z` or `result.exact`, allows us to access specific values of the test output. To obtain a list of values we use **names(result.z)**. The output of **names(result.exact)** is the same.

R Code

```
> names(result.z)

[1] "statistic"   "parameter"    "p.value"     "estimate"
[5] "null.value"  "conf.int"     "alternative" "method"
[9] "data.name"
```

R Code

```
> result.z$p.value

[1] 0.8348942
```

For example, `result.z$p.value` returns the p-value of the test and `result.z$conf.int` returns the confidence interval. For the endpoints of the intervals use `result.z$conf.int[1]` and `result.z$conf.int[2]`.

R Code

```
> result.z$conf.int

[1] 0.4447004 0.6261536
attr(,"conf.level")
[1] 0.95
```

R Code

```
> result.z$conf.int[1]

[1] 0.4447004
```

R Code

```
> result.z$conf.int[2]

[1] 0.6261536
```

We can also use **prop.test** when testing two proportions. Our example begins by generating another data set by letting `heads.2` be the number of successes from a binomial random variable with `n=45` and a probability of success of 0.5. We will test our two data sets to see if they both have the same probability of success. In **prop.test** the first entry is the number of successes and when there is more than one group we use **c**, combine, to create a vector of successes. Similarly, the second entry is the list of sample sizes and again we

use **c**. After the first two entries, the options are the same and can be in any order. We note that **prop.test** can be used for testing multiple proportions. We set the output of the test to **test.z.2**. An exact test for two proportions can be done using **fisher.test**.

R Code

```
> heads.2= rbinom(1,size=45,prob=0.5)
> result.z.2=prop.test(c(heads,heads.2),c(123,45),
+ alternative="two.sided",conf.level=0.90)
> result.z.2

2-sample test for equality of proportions with
continuity correction

data: c(heads, heads.2) out of c(123, 45)
X-squared = 0.0019103, df = 1, p-value = 0.9651
alternative hypothesis: two.sided
90 percent confidence interval:
-0.1766766 0.1387362
sample estimates:
prop 1     prop 2
0.5365854 0.5555556
```

The output of **names(result.z.2)** is the same as above. For example, if we want to recover the number of successes from the second group we use **result.z.2\$estimate[[2]]** and multiply by the sample size.

R Code

```
> test.z.2$estimate[2]*45
```

13

13.2 Illustrating the Meaning of α Simulation

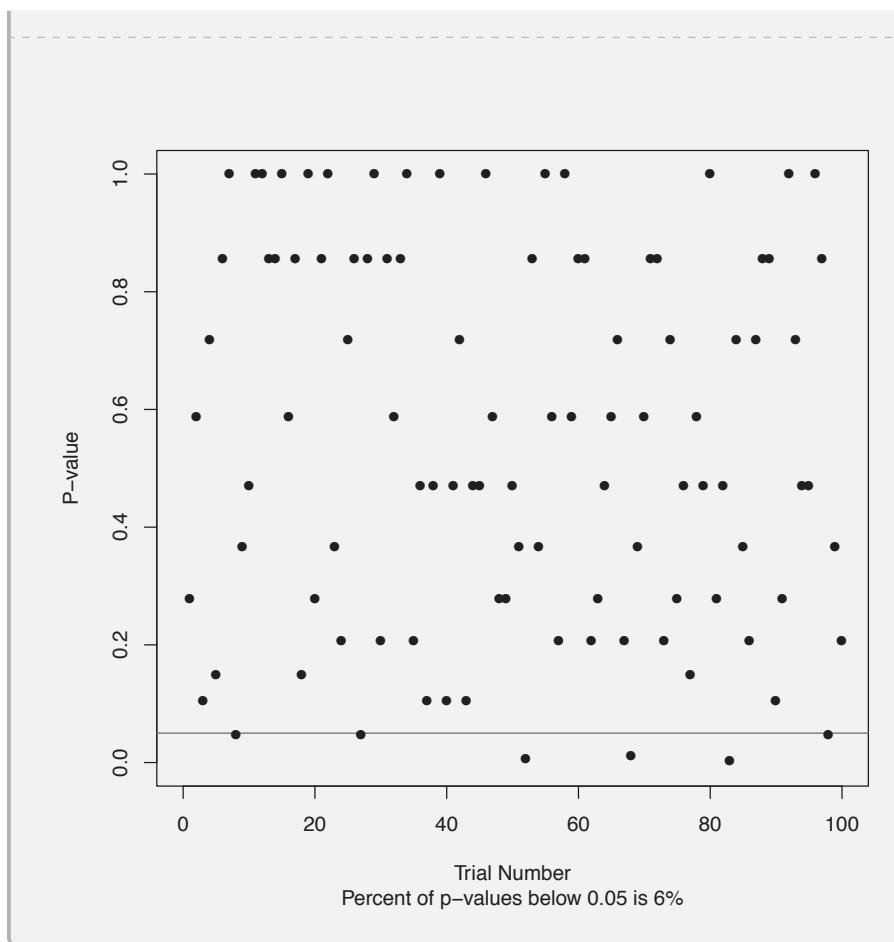
With $\alpha = 0.05$, about 5% of the time we would reject the null hypothesis when we shouldn't. Our example here illustrates this point. We will repeatedly sample from a binomial distribution with proportion **pop.p=0.5** and test the data to see if it is different from that probability of success. Our first five lines of code set values used throughout. We do this so that if we want to change any of these values it only has to be done once at the beginning of the code. We will perform 100 simulations (**sim=100**), our sample size will be 123

(`pop.size=123`), our population proportion is 0.5 (`pop.p=0.5`), and our value of $p_0 = 0.5$ also (`p.0=0.5`). We set `reject=0`, which will be used as a counter in the for loop below. We create a **plot** where the *x*-axis is for each trial and the *y*-axis the p-value from that trial. We use `type="n"` for an empty plot, `xlim` sets the *x*-axis from 0 to the number of simulations, and `ylim` is from 0 to 1 to plot our p-values. The axes are labelled with `xlab` and `ylab`. We add a red horizontal line at 0.05 with **abline**.

Our for loop begins with `i` ranging from 1 to the number of simulations, in this example 100. The variable `samp` captures the output from **rbinom**, which is the number of successes with a sample size of `pop.size`, 123, and probability of success 0.5, `pop.p`. The output of **prop.test** is set to `result`. We plot the p-value of the test with **points**, where the *x*-value is *i* and the *y*-value is `test$p.value`. The plot character is set to 16, `pch=16`, for a solid dot. The if statement checks to see if the p-value is less than $\alpha = 0.5$, and if it is, then increase our number of rejects by one with `reject=reject+1`. The for loop is closed with `}`. Our last line adds text at the bottom of our graph with **mtext**. Within **mtext** we use **paste** to combine characters with variables. At the end of `paste()` we have `sep=""` so that there are no spaces between the parts separated by commas. This was done so that there isn't a space between the variable `reject/sim*100` and the percent sign. But, in doing this we had to add spaces within the character strings.

R Code

```
> sim=100
> pop.size=123
> pop.p=0.5
> p.0=0.5
> alpha=0.05
> reject=0
> plot(0,0,type="n",xlim=c(0,sim),ylim=c(0,1),
xlab="Trial Number",ylab="P-value",)
> abline(h=alpha,col="red")
> for(i in 1:sim){
+   sample=rbinom(1,size=pop.size,prob=pop.p)
+   result=prop.test(sample,pop.size,p=p.0,
alternative="two.sided",conf.level=(1-alpha))
+   points(i,result$p.value,pch=16)
+   if( result$p.value < alpha){reject=reject+1}
+ }
> mtext(paste("Percent of p-values below ", alpha,
" is ",reject/sim*100,"%",sep=""),side=1,line=4)
```



Code Review from the Testing Proportions Chapter

`binom.test(succ,n,alternative=,p=,conf.level=)` is used for an exact test for a single proportion. The alternative options are “two.sided”, “less”, or “greater”.

`prop.test(c(succ.x, succ.y), c(n.x,n.y),alternative=,p=,conf.level=)` is used for a one or two sample test of proportions. The `.y` options are removed for a one sample test in which case `c()` can also be removed. The alternative options are “two.sided”, “less”, or “greater”.

`rbinom(x,n,p)` returns a vector of length x where each entry is the number of success from a binomial random variable with n trials and probability of success p .

13.3 Exercises

1. Physically flip a coin 25 times and record the number of heads. Test to see if the coin is fair.
2. Physically flip a nickel and a quarter 25 times each and record the number of heads. Test to see if the probability of a head is different between the two coins.
3. On average, how much do the `prop.test()` and `binom.test` differ? Create a simulation that estimates this difference.
4. Use the proportion test to create confidence interval theory examples as in [Section 12.2](#).



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

14

Linear Regression

We will begin with an example using the Ice data from [Section 1.1](#). The example assumes that the Arctic ice data has been imported. Recall that `read.table("Arctic-Ice-Data-R.csv", header=TRUE, sep=",")` is the code that imports the data with the data set being called `Ice`. We could use `names(Ice)` to get a list of the column names and `Ice$March.Extent.in.MSK` is the syntax to reference the March ice extent data.

R Code

```
> linear.model.M=lm(Ice$March.Extent.in.MSK ~  
Ice$Years.after.1970)  
> summary(linear.model.M)

Call:  
lm(formula=Ice$March.Extent.in.MSK~Ice$Years.after.1970)

Residuals:  
      Min       1Q   Median       3Q      Max  
-0.56691 -0.20553  0.03769  0.21075  0.46576

Coefficients:  
              Estimate Std. Error t value Pr(>|t|)  
(Intercept) 16.502932   0.108415 152.22 < 2e-16 ***  
Ice$Years. after.1970 -0.042112   0.003526 -11.94 1.96e-14 ***  
---  
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1  1

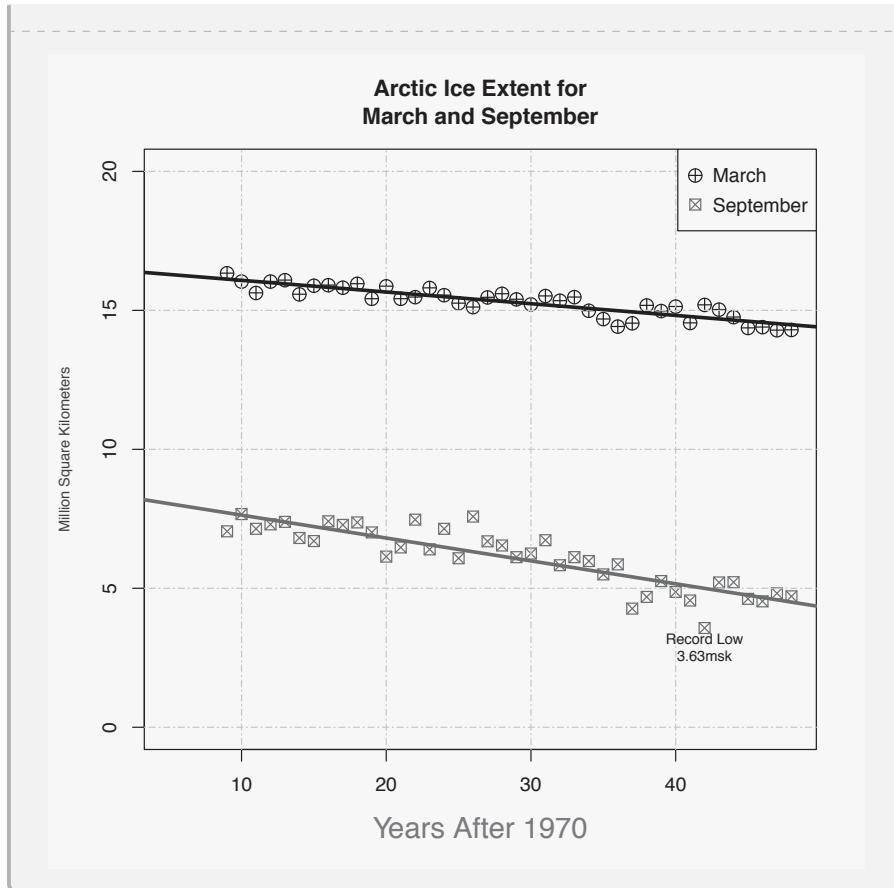
Residual standard error: 0.2574 on 38 degrees of freedom  
Multiple R-squared:  0.7897, Adjusted R-squared:  0.7841  
F-statistic: 142.7 on 1 and 38 DF,  p-value: 1.965e-14
```

The syntax here is **lm(dependent-variable ~ independent-variable)** with the real key being that you give the result a name. Here we call it `linear.model.M`, which is an object that contains information about the model. In the example, `summary(linear.model.M)` produces the information one should expect, including the coefficient, p-values, and R-squared. We cre-

ate the analogous object for the September data setting linear.model.S to $\text{lm}(\text{Ice\$September.Extent.in.MSK} \sim \text{Ice\$Years.after.1970})$, and add the lines to the scatter plot from [Section 3.2](#). The last two lines of code, `abline`, add the lines. The code before that is the same as in [Section 3.2](#).

R Code

```
> par(bg = "#fdf7ef")
> plot(Ice$Years.after.1970,Ice$March.Extent.in.MSK,
+       type="p",cex=1.5,pch=10,xlim=c(5,48),ylim=c(0,20),
+       xlab="Years After 1970",ylab="",col.lab="purple",
+       cex.lab=1.5)
> points(Ice$Years.after.1970,Ice$September.Extent.in.
+         MSK,type="p",cex=1.25,pch=7,col="red")
> title(main="Arctic Ice Extent for\nMarch and
+       September",ylab="Million Square Kilometers",
+       col.lab="blue",cex.lab=0.75)
> grid(NULL,NULL,lty=6,col="gray")
> v3=c("March","September")
> legend("topright",v3,pch=c(10,7),pt.cex=c(1.5,1.25),
+       col=c("black","red"),y.intersp=1.25)
> text(42,3.63,"Record Low\n3.63msk",pos=1,cex=0.75)
> abline(linear.model.M, lwd=3)
> abline(linear.model.S, lwd=3, col="red")
```



One of the many nice functions of R is its ability to interpret information. The function **abline** interprets `linear.model.M` and `linear.model.S` as lines and adds them to the graph. The scalar object `lwd` is used to increase the thickness of the lines to 3 times the default and the September line is colored red. Another option for graphing the line, as well as evaluating values on the line, is to create a function. If you invoke `names(linear.model.M)` you will see the objects associated with `linear.model.M`; one of which is `coefficients`. The first two lines of the next code returns the intercept and slope of `linear.model.M`. Note the use of `$` to call one of the objects associated with `linear.model.M`, as well as the use of `[[]]`. As a rule one set of brackets calls an item in a list. In this case, `lineM$coefficients[1]` returns the intercept but with the column header (try it). The use of `[[]]` will remove the column header and simply return the number. We use those two values to define the line `March.Line`. At this point you can evaluate `March.Line` or add it to the graph with **curve(March.Line, a, b, add=TRUE)**, where the line will be graphed from $x = a$ to $x = b$, which can be useful if you only want to graph part of the line. All the typical graphing options such as `lty`, `lwd`, and `col`, are available with **curve**.

R Code

```
> linear.model.M$coefficients[[1]]
[1] 16.50293
```

R Code

```
> March.Line=function(x){linear.model.M
$coefficients[[2]]*x+linear.model.M$coefficients[[1]]}
> March.Line(30)
[1] 15.23958
[1] -0.04168837
```

A 95% confidence interval for the slope of a regression line is typically desired, which is our next code. We use the **confint** function with the first element the object `lineM`. The second is the related parameters whose coefficient we are constructing the confidence interval. The last element is the confidence level.

R Code

```
> confint(linear.model.M,"Ice$Years.after.1970",
level=0.95)

           2.5 %      97.5 %
Ice$Years.after.1970 -0.04924924 -0.03497402
```

If the second option “`Ice$Years.after.1970`” is left out then **confint** provides intervals for all variables. In this case you would also obtain the interval for the intercept. If you want to use the interval values then name your confidence interval, say `March.Slope.ConfInt=confint((linear.model.M,` “`ce$Years.after.1970`”, `level=0.95)`), and use `March.Slope.ConfInt[1]` to obtain the lower value. Running **confint** for September yields the 95% confidence interval of $(-0.09712047, -0.06757747)$ and notice that the two intervals do not overlap. In other words, ice extent is declining at a significantly faster rate in September as compared to March.

The next two intervals we create are a confidence interval for the mean extent and a prediction interval for 2011 and 2018, which are represented as 41 and 48 years after 1970. (Note: As this is time series data the interpretation here doesn’t make sense.) For technical reasons we start by defining x to be the years after 1970 data. Note that we could have done this at the beginning for all the data columns we are using to cut down on the typing associated with the data names. The second line reruns the regression for March. The third line sets the variable new to the x , or years after 1970, values where we will

create our intervals, in other words, 41 and 48 for 2011 and 2018 respectively. The values must be put into the data.frame format and you can do this for as many locations as desired. The **predict** function creates the intervals for the desired line. The first argument is the model, the second is the new points, the third defines the type as confidence, and the last argument is the confidence level of 90%. The default confidence level is 95%. The output is the fitted value and the lower and upper values of the confidence interval, but you have to keep track of the associated values from new. We solve this in the next line, which outputs the associated values for 95% prediction intervals. We use **cbind**, column bind, to concatenate the value from new with the output of **predict**. In **predict** we change interval to prediction and use the same level of 90%. The output now includes a table with the associated value from new. Note that the prediction intervals are wider as they should be.

R Code

```
> x=Ice$Years.after.1970
> linear.model.M2=lm(Ice$March.Extent.in.MSK ~ x)
> new=data.frame(x=c(41,48))
> predict(linear.model.M2,new,interval="confidence",
level=0.90)

      fit      lwr      upr
1 14.77635 14.67521 14.87750
2 14.48157 14.34687 14.61628
```

R Code

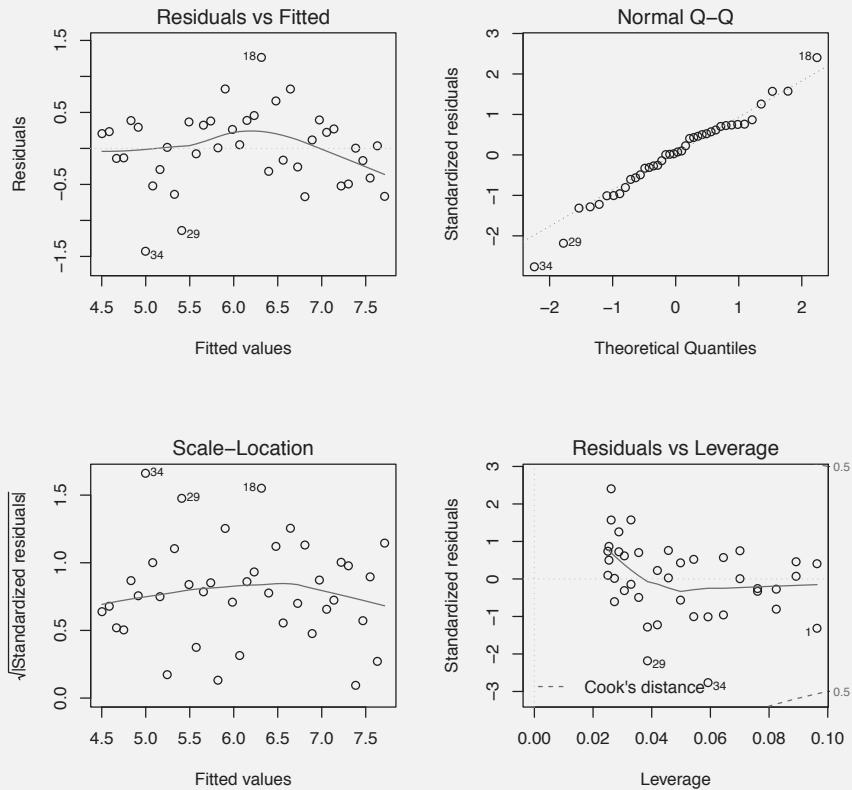
```
> cbind(new,predict(linear.model.M2,new,interval=
"prediction",level=0.90))

      x      fit      lwr      upr
1 41 14.77635 14.33075 15.22196
2 48 14.48157 14.02717 14.93598
```

We now move to some diagnostics of our lines. The next two lines of code quickly output the four standard diagnostic plots. The **par** function sets our graphing frame to be a 2-by-2 grid. This is necessary because the **plot** function in the next line outputs four graphs. The function **plot** is sophisticated and interprets the input of lineS as a regression line and is set to output the four graphs.

R Code

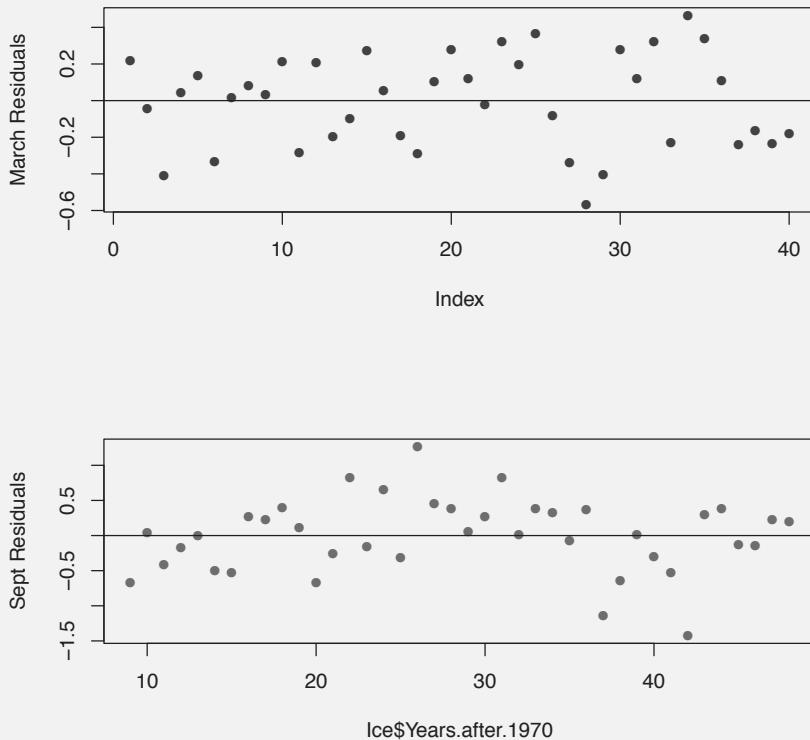
```
> par(mfrow=c(2,2))
> plot(linear.model.S)
```



Our next set of graphs produces the residual plots for both March and September. We begin using **par** to set a 2 by 1 grid for graphing. The first use of **plot** inputs just the residual data from the March line with a *x*-axis simply labelled by the number of values. The point character is set to 16 for solid dots, the dots are colored blue and we label the *y*-axis. We then use **abline** to add a horizontal line at 0. Our next use of **plot** graphs the September residuals, but this time we provide *x* and *y* data so that the *x*-axis corresponds to the year. We also add a horizontal line at 0. In both graphs, we left the *x* label to be the default. In looking at the residual plots, the September data seems to have a bend while the March doesn't.

R Code

```
> par(mfrow=c(2,1))
> plot(linear.model.M$residuals,pch=16,col="blue",
ylab="March Residuals")
> abline(h=0)
> plot(Ice$Years.after.1970,linear.model.S$residuals,
pch=16,col="red",ylab="Sept Residuals")
> abline(h=0)
```

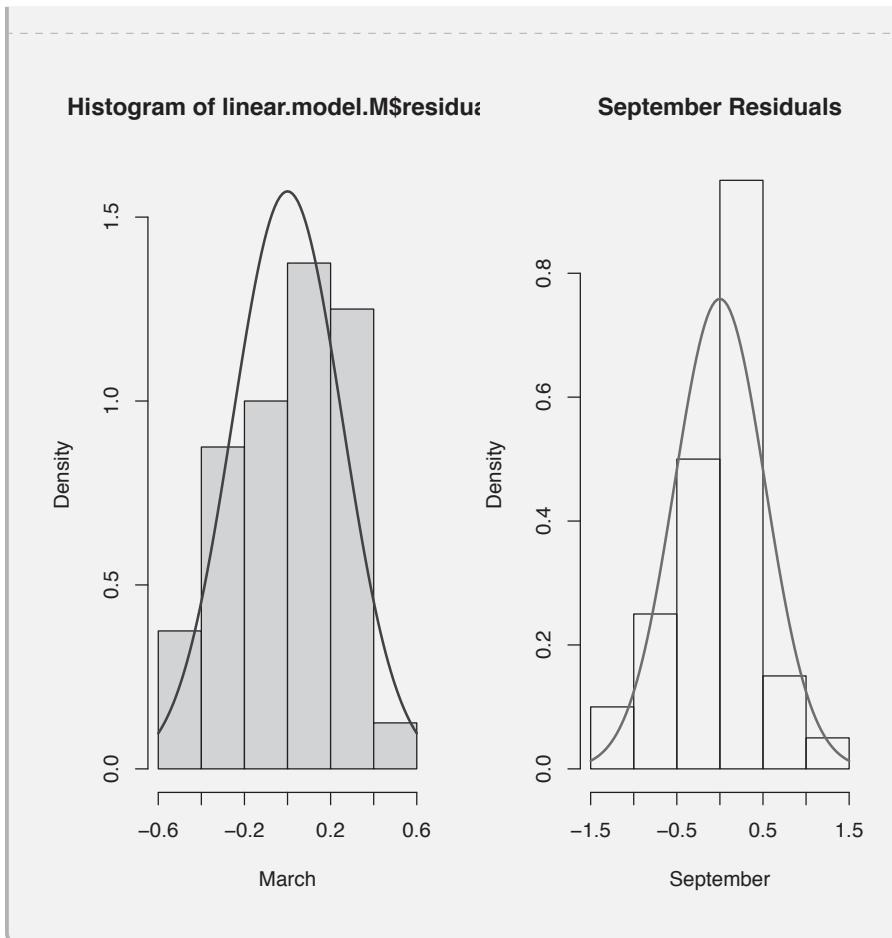


Our last set of plots creates histograms of the residual with a normal density overlay. We begin by setting a 1 by 2 graphing frame. Next, **hist** plots a histogram for the March residuals. We added a y -axis range with `ylim` because the without it the default doesn't fit the density added later. We set `prob=TRUE` to get relative frequencies, with the default set to frequency. We color the boxes lightface and label the x -axis. The next two lines calculate the mean and standard deviation of the March residuals. This is followed by

curve which plots the normal density, **dnorm**, with the mean and standard deviation of the March residuals. Note that add=TRUE is necessary to add the curve to the plot as opposed to generating a new plot. The remaining four lines repeat the process for September, the only difference being we don't color the histogram boxes but add a title with main. Note that for the first use of **hist** we did not provide a title and by default the name of the data is used in the title. In this case, the default title is too long and is cut off.

R Code

```
> par(mfrow=c(1,2))
> hist(linear.model.M$residuals,ylim=c(0,1.6),prob=TRUE,
+       col="lightblue", xlab="March")
> mM=mean(linear.model.M$residuals)
> sM=sqrt(var(linear.model.M$residuals))
> curve(dnorm(x,mM,sM),col="blue",lwd=2,add=TRUE)
> hist(linear.model.S$residuals,prob=TRUE,xlab=
+       "September",main="September Residuals")
> mS=mean(linear.model.S$residuals)
> sS=sqrt(var(linear.model.S$residuals))
> curve(dnorm(x,mS,sS),col="red",lwd=2,add=TRUE)
```



14.1 Multiple Linear Regression

Continuing with the Ice data, it would seem that knowing the ice extent for March and June would help to predict the September extent. Multiple linear regression works in essentially the same way as simple linear regression. We use **lm** but now add additional explanatory variables. In this case we add March and June ice and **summary** returns the standard regression output. Notice that March isn't significant which makes sense because June is closer to September. Note also there is a correlation between March and June, which we could check. In short, March doesn't add information to the model.

R Code

```
> mult.linear.model=lm(Ice$September.Extent.in.MSK ~
  Ice$Years.after.1970+Ice$March.Extent.in.MSK+
  Ice$June.Extent.in.MSK)
> summary(mult.linear.model)
```

Call:

```
lm(formula = Ice$September.Extent.in.MSK ~
  Ice$Years.after.1970 + Ice$March.Extent.in.MSK
  + Ice$June.Extent.in.MSK)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.21335	-0.23500	0.03537	0.29484	0.90034

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-4.03545	6.51598	-0.619	0.53961
Ice\$Years.	-0.03403	0.02133	-1.596	0.11928
after.1970				
Ice\$March.	-0.13219	0.30525	-0.433	0.66755
Extent.in.MSK				
Ice\$June.	1.13191	0.35314	3.205	0.00283 **
Extent.in.MSK				

Signif. codes:	0 ***	0.001 **	0.01 *	0.05 . 0.1 1

Residual standard error: 0.4826 on 36 degrees of freedom
 Multiple R-squared: 0.8213, Adjusted R-squared: 0.8064
 F-statistic: 55.16 on 3 and 36 DF, p-value: 1.534e-13

We can still use **plot(mult.linear.model)** for the four diagnostic graphs, again preceded by **par(mfrow=c(2,2))**. We can also obtain residuals with **mult.linear.model\$residuals**. For confidence intervals for each corresponding slope we use **confint(mult.linear.model, level=0.95)**. Confidence and prediction intervals for given values are similar and demonstrated in the next code. As with simple regression we redefine the input values as x1, x2, and x3, and rerun the multiple regression. Our given data, newMult, is a **data.frame** with inputs for all of x1, x2, and x3. Our last line uses **cbind** with **predict** to create the output of 90% confidence intervals for the mean September ice (Note: As this is time series data the interpretation here doesn't make sense.)

R Code

```

> x1=Ice$Years.after.1970
> x2=Ice$March.Extent.in.MSK
> x3=Ice$June.Extent.in.MSK
> mult.linear.model2=lm(Ice$September.Extent.in.MSK
~ x1+x2+x3)
> newMult = data.frame(x1=c(41,46),x2=c(14.55,14.40),
x3=c(10.75,10.41))
> cbind(newMult,predict(mult.linear.model2,newMult,
interval="confidence", level = 0.90))

      x1     x2     x3      fit      lwr      upr
1 41 14.55 10.75 4.813892 4.547744 5.080041
2 46 14.40 10.41 4.278705 3.953452 4.603958

```

Code Review for the Regression Chapter

abline(line) will add the regression line to a plot. All the typical graphing options work with abline().

confint(line, "variable", level=C) creates a confidence interval for the coefficient associated with the variable of the line. C is the confidence level as a decimal.

line\$coefficients[[1]] is the value for the intercept of the line. Changing 1 to 2 returns the slope.

line=lm(y~x) performs simple linear regression and creates the output object line (you can call it what you want). You can just run lm(y x) but naming it allows one to call output of the regression.

line\$residuals are the residuals associated with line.

Mline=lm(y~x₁ + x₂ + ⋯ + x_n) performs a multiple linear regression. Most options that use line above also work for Mline.

predict(line,new,interval="confidence",level=C) will calculate a confidence interval for the mean y value associated with all x values listed in new. Changing "confidence" to "prediction" will return an associated prediction interval.

summary(line) provides standard regression output of the regression object line.

14.2 Exercises

1. Use $x=1:100$ and $y=x+rnorm(100,0, 10)$ to perform a regression analysis using all the tools presented here.
2. Use $x=-50:50$ and $y=x+rnorm(100,0, 10)+x^2/75$ to perform a regression analysis using all the tools presented here.
3. Complete a multiple regression analysis for September ice predicted by years after 1970 and June ice. You should create a histogram of the residuals with a normal density.
4. Add a 95% confidence interval band around the March ice predicted by year. This is a bit challenging. Bonus if you can shade the band.

15

Nonparametric Statistical Tests

We quickly run through common nonparametric statistical tests, with the expectation that before reading this chapter other statistics chapters have been reviewed. Thus, this chapter will be briefer than the other statistics-related chapters.

We begin by creating data to be used to illustrate the tests. The **set.seed(41)** insures that the random uniform data is the same for all users. The function **runif(n,a,b)** returns n values from a uniform random distribution from a to b. The vectors x1 and x2 are selected from 0 to 10, while x3 is from 0 to 5. The values of x1 are returned.

R Code

```
> set.seed(41)
> x1=runif(20,0,10)
> x2=runif(20,0,10)
> x3=runif(20,0,5)
> x1

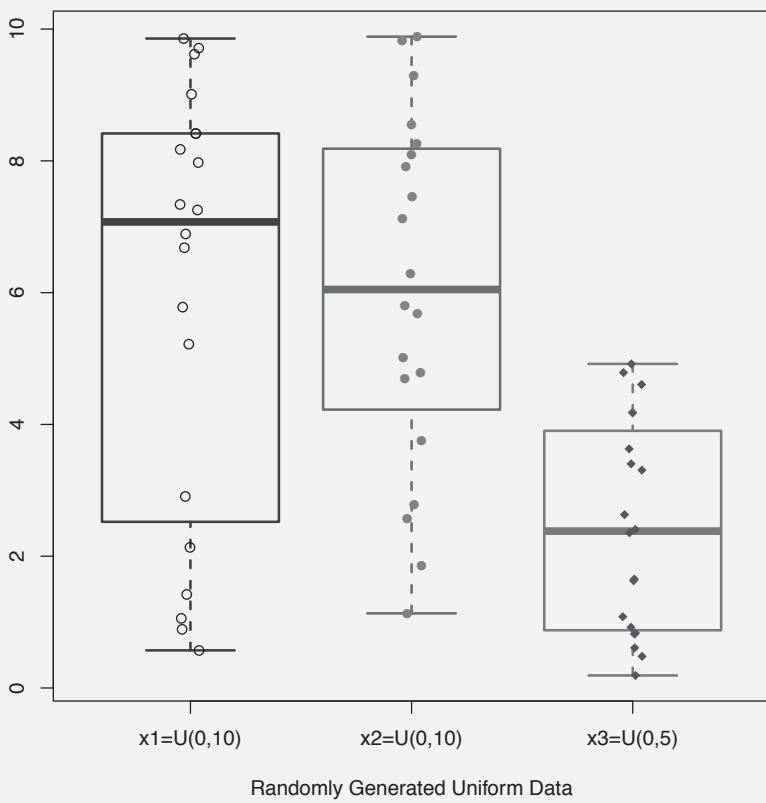
[1] 2.1349051 9.7122835 5.7818700 1.4207998
[5] 8.4175679 9.6203301 9.0127060 5.2192307
[9] 8.1746681 6.6829136 6.8922948 7.3391263
[13] 7.2550884 2.9084979 0.5709845 7.9755688
[17] 8.4149488 0.8912729 9.8566548 1.0578927
```

We create a boxplot to visually display the data. The values of x1, x2, and x3 are put into the variable Data as three lists with **list**. The **par** function here sets the margins around the graph frame to be 4, 3, 3, and 3, for the bottom, left, top, and right margins respectively. The **boxplot** function creates boxplots of each of the vectors in Data. The plots are located at 1, 2, and 3 along the *x*-axis. The names are given for each of the plots, the line width of the borders are set to 2, and the border colors are set. The boxes can also be shaded with the col option. The **stripchart** function overlays the boxplots with the data. The method set to jitter and the jitter value moves the points off a straight line to avoid overlap of data points. The options add needs to be set to TRUE to add the graph to the current graph. The point characters for each data set are given by pch and the dots are colored with col. Note how pch and col

have three values combined with **c**, each being associated with the data in the same order.

R Code

```
> Data=list(x1,x2,x3)
> par(mar=c(4,3,3,3))
> boxplot(Data,at=c(1,2,3),names=c("U(0,10)","U(0,10)",
"U(0,5)"),lwd=2,border=c("blue","red","purple"))
> stripchart(Data,vertical=TRUE,method="jitter",
jitter=0.05,add=TRUE,pch=c(1,16,18),col=c("black",
"gray40","gray20"))
> mtext("Randomly Generated Uniform Data",side=1,line=3)
```



15.1 Wilcoxon Signed Rank Test for a Single Population

We test to see if the `x1` data is from a population with median 5. The function `wilcox.test` takes `x1` and tests for symmetry about `mu=5`, with an alternative hypothesis of greater. For alternative the default is two.sided and there is the option of less. To also return a confidence interval, `conf.int` is set to TRUE and the confidence level is set to 0.90, with a default of 0.95. The output of the test is set to `result` and then displayed.

R Code

```
> result=wilcox.test(x1,mu=5,alternative="greater",
conf.int=TRUE,conf.level=0.90)
> result

Wilcoxon signed rank test

data: x1
V = 137, p-value = 0.1227
alternative hypothesis: true location is greater than 5
90 percent confidence interval:
 4.900396      Inf
sample estimates:
(pseudo)median
 5.978191
```

The virtue of setting the output to the variable `result` is so that values can be accessed. The list of values set to `result` is given by `names`. For example `result$p.value` would return the p-value of the test. Throughout the rest of the chapter we will return test results directly without setting them to a variable, but what was done in this example can be done with the other tests also.

R Code

```
> names(result)

[1] "statistic" "parameter" "p.value"    "null.value"
[5] "alternative" "method"     "data.name"   "conf.int"
[9] "estimate"
```

15.2 Wilcoxon Rank Sum Test for Independent Groups

The Wilcoxon rank sum test, also known as Mann-Whitney, is the nonparametric equivalent of the two-sample t-test. Specifically, the null hypothesis is that the distributions of the two groups are the same, $\mu_1 = \mu_2$, or differ by a shift of $\mu_1 - \mu_2$. The **wilcox.test** function requires the first two arguments to be the data sets. The default value of $\mu_1 - \mu_2$ is 0. Options for alternative and conf.level are the same as above.

R Code

```
> wilcox.test(x1,x2,conf.int=TRUE)

Wilcoxon rank sum test

data: x1 and x2
W = 203, p-value = 0.9467
alternative hypothesis: true location shift is
not equal to 0
95 percent confidence interval:
-1.783510 2.109774
sample estimates:
difference in location
0.08668388
```

The data can also be input in a stacked format. As an example we stack the two data sets x_1 and x_2 by creating a data frame. The first column is called data and is the value of x_1 followed by the values of x_2 . The second column is named factor and consists of “a” repeated **length(x1)** times, and then a string of “b” again using the **rep** function. The first few and last few rows of the data frame are shown. Note that **wilcox.test(Data.Stacked.1\$data ~ Data.Stacked.1\$factor, conf.int=TRUE)** will produce the same output as above.

R Code

```
> Data.Stacked.1=data.frame("data"=c(x1,x2),
 "factor"=c(rep("a",times=length(x1)),
 rep("b",times=length(x2))))
> Data.Stacked.1

   data factor
1  2.1349051      a
2  9.7122835      a
3  5.7818700      a
```

```
:  
38 1.8558491      b  
39 9.8852910      b  
40 9.2952160      b
```

15.3 Wilcoxon Signed Rank Test for Dependent Data

For paired or dependent data, we test to see if the difference is symmetric about mu. Again the **wilcox.test** function is used with the difference from the rank sum test being that paired is set to TRUE. Options of mu with a default of 0, alternative, and conf.level are the same as above.

R Code

```
> wilcox.test(x1,x3,paired=TRUE,conf.int=TRUE)  
  
Wilcoxon signed rank test  
  
data: x1 and x3  
V = 193, p-value = 0.0003948  
alternative hypothesis: true location shift is  
not equal to 0  
95 percent confidence interval:  
 1.800270 5.271995  
sample estimates:  
(pseudo)median  
 3.506802
```

15.4 Spearman's Rank Correlation Coefficient

Spearman's rank correlation test is performed using **cor.test**. The first two arguments are the two data sets. We set method to "spearman" for this test and note that method can also be set to "pearson" or "kendall". As with the tests above, the default alternative is "two.sided" and alternative can be set to "less" or "greater". We demonstrate this test using the x1 and x2 data sets.

R Code

```
> cor.test(x1,x2,method="spearman")

Spearman's rank correlation rho

data: x1 and x2
S = 1054, p-value = 0.3783
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.2075188
```

15.5 Kruskal-Wallis One-Way Analysis of Variance

A nonparametric one-way ANOVA is performed with **kruskal.test**, but first we need to stack our data. we create a data framed with two columns using **data.frame**. The first column is called **data** and consists of the three vectors **x1**, **x2**, and **x3**. The second column, labeled **factor**, will associate “a”, “b”, and “c” with the data sets **x1**, **x2**, and **x3** respectively. To do this we create three vectors of data using **rep()**, where we repeat, for example, the letter “a” **length(x1)** times. The three vectors are combined using **c**. Part of the data frame **Data.Stacked.2** is displayed.

R Code

```
> Data.Stacked.2=data.frame("data"=c(x1,x2,x3),"factor"=
c(rep("a",times=length(x1)),rep("b",times=length(x2)),
rep("c",times=length(x3))))
> Data.Stacked.2

   data factor
1  2.1349051     a
2  9.7122835     a
3  5.7818700     a
:
58  0.8316096     c
59  0.1901612     c
60  1.6524501     c
```

The data is represented by Data.Stacked.2\$data and the factors by Data.Stacked.2\$factor. In using **kruskal.test** the input Data.Stacked.2\$data ~ Data.Stacked.2\$factor can be read as testing the data by the given factors.

R Code

```
> kruskal.test(Data.Stacked.2$data~  
Data.Stacked.2$factor)  
  
Kruskal-Wallis rank sum test  
  
data: Data.Stacked.2$data by Data.Stacked.2$factor  
Kruskal-Wallis chi-squared = 18.103, df = 2,  
p-value = 0.0001172
```

Code Review for the Nonparametric Statistical Tests Chapter

boxplot(data) creates side-by-side boxplots where data is a list of vectors of data.

cor.test(x1,x2,method=“spearman”) returns the results of Spearman’s rank correlation test.

data.frame() creates a data frame with the input options of the form “name”=data for each column of data.

kruskal.test(data~factor) returns the results of the nonparametric one-way ANOVA, where data is a vector of data and factor is an associated vector denoting the groups.

runif(n,a,b) returns a vector of n values from a uniform distribution from a to b .

stripchart(data) creates side-by-side strip charts where data is a list of vectors of data. If add=TRUE stripchart can be overlayed on boxplots.

wilcox.test() returns the results of the non-parametric test for a single population, independent groups, and dependent groups (add paired=TRUE).

15.6 Exercises

1. Perform the signed rank test with the x3 data and mu=5.

2. Perform the rank sum test with x2 and x3.
3. Perform the signed rank test (dependent) with x2 and x3.
4. Perform Spearman's rank correlation test with x2 and x3.
5. Create three new sets of data with $\text{rexp}(30,0.25)$, $\text{rexp}(30,0.25)$, and $\text{rexp}(30,0.5)$. Create side-by-side boxplots and perform a Kruskal-Wallis test.

16

Miscellaneous Statistical Tests

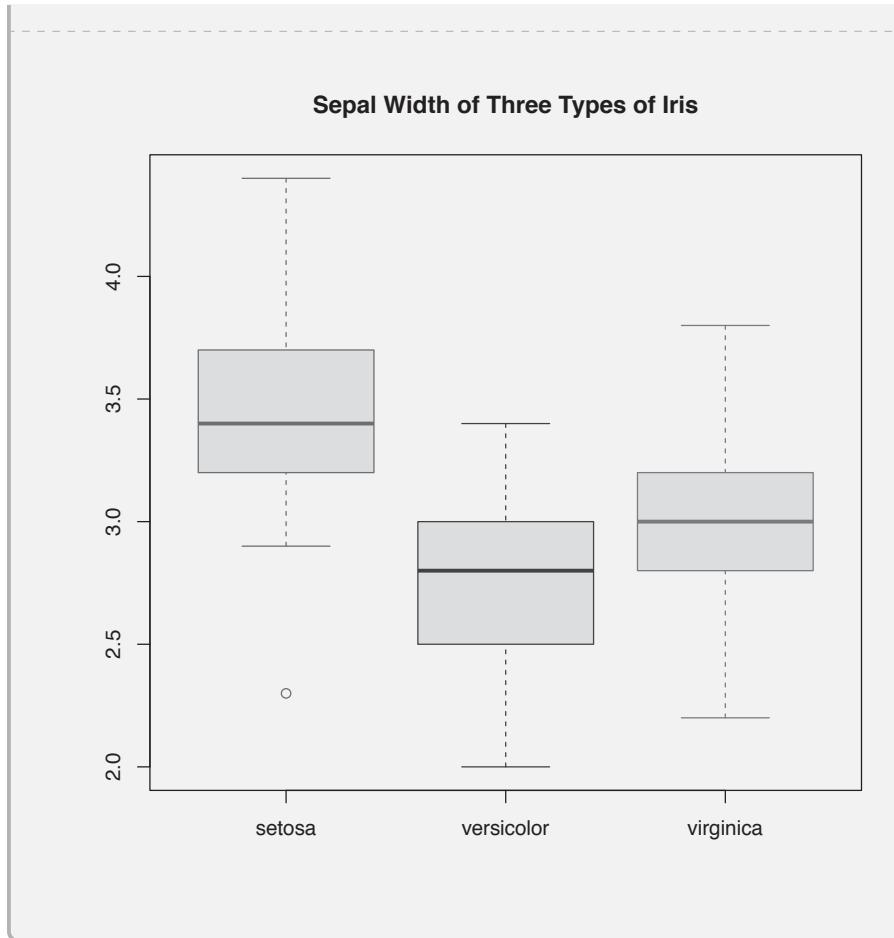
We cover three statistical tests that are common in introductory statistics classes; one-way ANOVA, chi-square tests, and testing standard deviations. In the ANOVA sections we include test for assumptions and a post-hoc test. The chi-square section covers goodness of fit as well as test for homogeneity and independence. The last section includes tests for standard deviations of one and two populations.

16.1 One-way ANOVA

We use the iris data, which is available in R and loaded with the command **data(iris)**, to illustrate a one-way ANOVA by testing the mean sepal width of the three species. There are 50 samples of each species for a balanced design, which is found with **summary(iris\$Species)**. A boxplot of sepal width for each species is created with the **boxplot** function. The command `iris$Sepal.Width ~ iris$Species` informs the **boxplot** function to create boxplots of sepal width by species. The data is the iris data, the boxes are colored with gray85, and a vector of colors is set to border to color the borders. A title is added with `main`. Note: If data is in the format of separate columns see stacking data at the end of this section.

R Code

```
> data(iris)
> boxplot(iris$Sepal.Width~iris$Species,data=iris,
  col="gray85",border=c("red","blue","purple"),
  main="Sepal Width of Three Types of Iris")
```



The output of **aov**, which performs ANOVA, is set to `result.aov`. As with **boxplot** the data is `iris$Sepal.Width ~ iris$Species` which is sepal width sorted by species. The data is from the `iris` data set. The results of the test are displayed with **summary(result.aov)**.

R Code

```
> result.aov=aov(iris$Sepal.Width ~ iris$Species,
  data=iris)
> summary(result.aov)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
iris\$Species	2	11.35	5.672	49.16	<2e-16 ***
Residuals	147	16.96	0.115		

```
Signif. codes: 0 *** 0.001 ** 0.01* 0.05 . 0.1 1
```

The results of ANOVA are significant, meaning at least one sepal width differs from the others. For pairwise comparisons we use **TukeyHSD**, which shows that the setosa variety has larger sepal widths than the other two species, and that, in fact, all three differences are significant.

R Code

```
> TukeyHSD(result.aov)

Tukey multiple comparisons of means
95% family-wise confidence level

Fit: aov(formula=iris$Sepal.Width ~ iris$Species,data=iris)

$`iris$Species'
      diff      lwr      upr
versicolor-setosa -0.658 -0.81885528 -0.4971447
virginica-setosa -0.454 -0.61485528 -0.2931447
virginica-versicolor 0.204  0.04314472  0.3648553

      p adj
versicolor-setosa 0.0000000
virginica-setosa 0.0000000
virginica-versicolor 0.0087802
```

The information stored in result.aov can be seen with **names(result.aov)**. In order to return or access specific variables use result.aov\$name such as result.aov\$residuals for a list of the residuals. For the residual standard error (the standard deviation of the residuals) use **sigma(result.aov)**.

R Code

```
> names(result.aov)
```

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

R Code

```
> sigma(result.aov)
```

```
[1] 0.3396877
```

To check the assumption of equal population variance we use **bartlett.test** again with the input of `iris$Sepal.Width ~ iris$Species`.

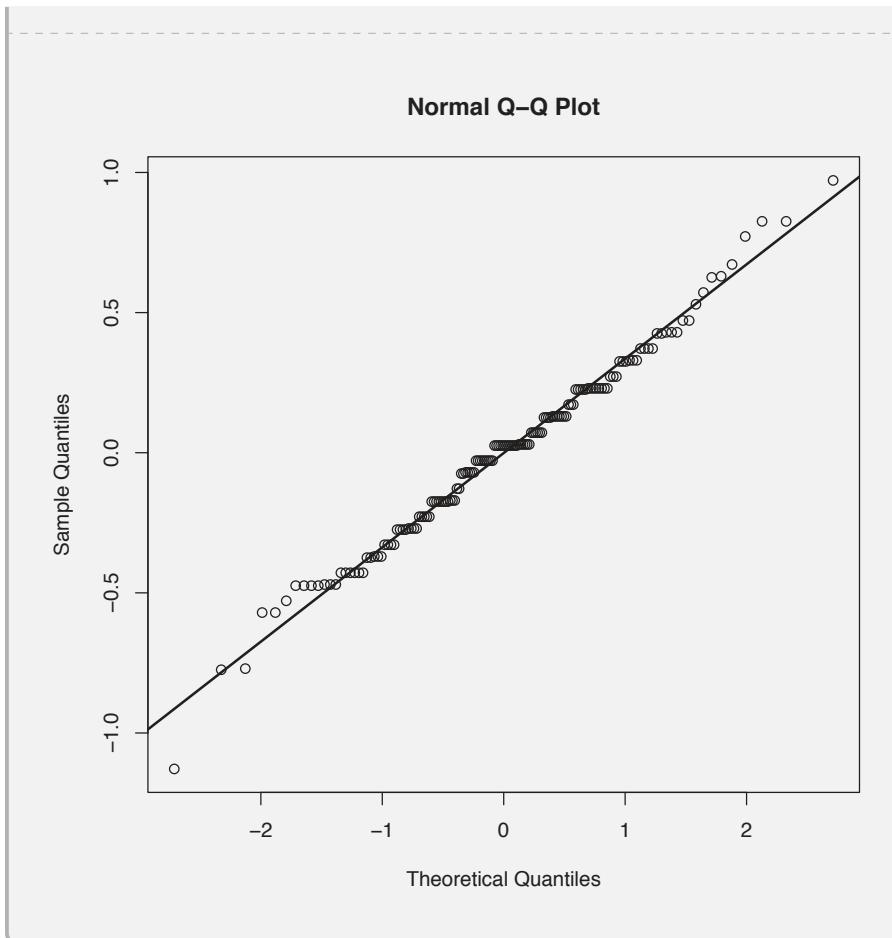
R Code

```
> bartlett.test(iris$Sepal.Width ~ iris$Species,  
  data=iris)  
  
Bartlett test of homogeneity of variances  
  
data: iris$Sepal.Width by iris$Species  
Bartlett's K-squared = 2.0911, df = 2, p-value = 0.3515
```

To check the normality assumption we create a normal q-q plot with **qqnorm** with the input data the residuals. A qq line is added with **qqline** with a line width of 2. To perform the Shapiro-Wilk test of normality use **shapiro.test(result.aov\$residuals)**.

R Code

```
> qqnorm(result.aov$residuals)  
> qqline(result.aov$residuals, lwd=2)
```



16.1.1 Stacking Data

If data is in a data frame where each level is in a separate column there is a command to convert it to a stacked data set. For this example we create a small data frame, although in many instances a data frame will be created by importing data from a spreadsheet. We define `df.columns` to be a data frame with three columns labeled C1, C2, and C3. Column C1 contains the integers from 1 to 3 generated by the colon command. Column C2 is defined by a list of numbers. Column C3 is created from `seq`, which produces a sequence from 100 to 300 in steps of 100.

R Code

```
> df.columns=data.frame("C1"=1:3,"C2'=c(5,10,15),
" C3'=seq(100,300,by=100))
> df.columns

  C1 C2  C3
1  1  5 100
2  2 10 200
3  3 15 300
```

To convert this data set to a stacked data set we use the **stack** function and set it to df.stack. The new data frame is given column names with the **colnames** function. The output is not stacked and we can used df.stack\$data ~ df.stack\$type in the commands in the ANOVA section. Note that when creating a data frame the columns must be the same length.

R Code

```
> df.stack=stack(df.columns)
> colnames(df.stack)=c("data","type")
> df.stack

  data type
1     1   C1
2     2   C1
3     3   C1
4     5   C2
5    10   C2
6    15   C2
7   100   C3
8   200   C3
9   300   C3
```

16.2 Chi-Square Tests

In this section we use **chisq.test** for a goodness of fit test, with equal and unequal proportions, and a homogeneity test, which is performed the same way as an independence test. We test to see if a proposed loaded die is fair and then test to see if it fits a particular distribution. We then test to see if another die has the same distribution as the first.

We are given a die and told it is not fair. To test this we roll the die

200 times and find that 1, 2, 3, 4, 5, and 6 occur 55, 45, 52, 20, 16, and 12 times respectively. The data is set to sample.1. The default distribution of **chisq.test** is uniform and the results are set to chi.results.1. Requesting chi.results.1 returns the results of the test. We reject the null hypothesis and conclude that the die is not fair.

R Code

```
> sample.1=c(55,45,52,20,16,12)
> chi.results.1=chisq.test(sample.1)
> chi.results.1

Chi-squared test for given probabilities

data: sample.1
X-squared = 56.62, df = 5, p-value = 6.056e-11
```

The information stored in chi.results.1 is given by **names(chi.results.1)**. The output from chi.results.1\$expected allows us to check the assumption that the expected values are greater than 5. The residuals and standard residuals are given by chi.results.1\$residuals and chi.results.1\$stdres.

R Code

```
> names(chi.results.1)

[1] "statistic" "parameter" "p.value"
[4] "method"     "data.name"  "observed"
[7] "expected"   "residuals" "stdres"
```

R Code

```
> chi.results.1$expected

[1] 33.3333 33.3333 33.3333 33.3333 33.3333 33.3333
```

We are told that the die is loaded so that 1, 2, 3, 4, 5, and 6 occur with proportions of 0.3, 0.2, 0.2, 0.1, 0.1, and 0.1 respectively. The proportions are set to test.p. To test this we use **chisq.test** and include p=test.p. Note the p= must be included. From the output from chi.results.2 we fail to reject the null hypothesis that the distribution of the values on the die is given by test.p.

R Code

```
> test.p=c(0.3,0.2,0.2,0.1,0.1,0.1)
> chi.results.2=chisq.test(sample.1,p=test.p)
> chi.results.2
```

Chi-squared test for given probabilities

```
data: sample.1
X-squared = 8.6417, df = 5, p-value = 0.1242
```

We are given a second die and told that it is loaded in the same way as the first die. Again, we roll the die 200 times and find that 1, 2, 3, 4, 5, and 6 occur 45, 40, 30, 25, 38, and 22 times respectively. We set these values to sample.2. Before using **chisq.test** the data will be put into a matrix. We set samp.matrix to the output of **matrix** which takes the two data sets, **c(sample.1,sample.2)**, and initializes a matrix by row with two rows. The functions **colnames** and **rownames** are used to name to the columns and rows. The results are seen by invoking samp.matrix.

R Code

```
> sample.2=c(45,40,30,25,38,22)
> samp.matrix=matrix(c(sample.1,sample.2),byrow=TRUE,
nrow=2)
> colnames(samp.matrix)=c("s1","s2","s3","s4","s5","s6")
> rownames(samp.matrix)=c("Die 1","Die 2")
> samp.matrix

      s1 s2 s3 s4 s5 s6
Die 1 55 45 52 20 16 12
Die 2 45 40 30 25 38 22
```

We now perform a chi-square test for independence to see if the two dice have the same distribution with **chisq.test**, using the matrix. The results are set to chi.results.3 and then outputted. We reject the null hypothesis that the distributions are the same.

R Code

```
> chi.results.3=chisq.test(samp.matrix)
> chi.results.3
```

Pearson's Chi-squared test

```
data: samp.matrix
X-squared = 19.656, df = 5, p-value = 0.00145
```

Note that chi.results.3\$expected may be used in the same way as above. In particular, chi.results.3\$expected is the matrix of expected values.

R Code

```
> chi.results.3$expected

      s1   s2   s3   s4   s5   s6
Die 1 50 42.5 41 22.5 27 17
Die 2 50 42.5 41 22.5 27 17
```

If the data isn't already tabulated then this can be achieved with the **table** command. For example, here we set data.col to a list of characters, in quotes, and **table(data.col)** tallies the results. This also works with number and longer character strings. At this point **chisq.test(table(data.col))** is interpreted properly, although the "sample size" is small, which generates a warning. We can also **table** within **matrix** so that, for example, **matrix(c(table(data.1),table(data.2)),byrow=TRUE,nrow=2)** is interpreted properly.

R Code

```
> data.col=c("r","g","b","y","g","r","0","b","g",
"y","y")
> table(data.col)

data.col
b g 0 r y
2 3 1 2 3
```

16.3 Testing Standard Deviations

We use the iris data again for examples of testing standard deviations or variances. The data is loaded with **data** and the structure of the data is displayed by **str**. A useful function to see the number of species is **unique**.

R Code

```
data(iris)
str(iris)

'data.frame': 150 obs. of 5 variables:
 $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 ...
 $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3 ...
 $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 ...
```

```
$ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.2 0.4 0.3 ...
$ Species      : Factor w/ 3 levels "setosa",
  "versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

R Code

```
> unique(iris$Species)

[1] setosa      versicolor virginica
Levels: setosa versicolor virginica
```

We will test if the variance of Versicolor petal length is more than 0.25 cm. To do this we create a data frame of just Versicolor species. The logical statement within `iris[iris$Species=="versicolor",]` refers to all the rows with species type Versicolor, `iris$Species=="versicolor"` and includes all columns, which is indicated by the blank after the comma. We display the first five rows of our new data frame, `iris.versi`, with **head**.

R Code

```
> iris.versi=iris[iris$Species=="versicolor",]
> head(iris.versi, n=5)

  Sepal.Length Sepal.Width Petal.Length
51          7.0        3.2       4.7
52          6.4        3.2       4.5
53          6.9        3.1       4.9
54          5.5        2.3       4.0
55          6.5        2.8       4.6

  Petal.Width   Species
51          1.4 versicolor
52          1.5 versicolor
53          1.5 versicolor
54          1.3 versicolor
55          1.5 versicolor
```

In order to test a single variance, we use the EnvStats package and load it with **library**. The results of **varTest** using the Petal.Length column of the `iris.versi` data frame is set to `result.versi.var`. Within **varTest** we set the alternative to greater, the confidence level to 0.95, and test against a variance of 0.25. The results of the test are displayed by executing `result.versi.var`. The information stored in `result.versi.var` is displayed with **names**.

R Code

```
> library(EnvStats)
> result.versi.var=varTest(iris.versi$Petal.Length,
  alternative="greater",conf.level=0.95,
  sigma.squared=0.25)
> result.versi.var
```

Results of Hypothesis Test

```
-----
Null Hypothesis:      variance = 0.25
Alternative Hypothesis:True variance is greater than 0.25
Test Name:           Chi-Squared Test on Variance
Estimated Parameter(s):variance = 0.2208163
Data:                 iris.versi$Petal.Length
Test Statistic:       Chi-Squared = 43.28
Test Statistic Parameter: df = 49
P-value:              0.7032852
95% Confidence Interval: LCL = 0.1631025
                                UCL =      Inf
```

R Code

```
> names(result.versi.var)
```

```
[1] "statistic" "parameters" "p.value"   "estimate"
[5] "null.value" "alternative" "method"    "data.name"
[9] "conf.int"
```

We move to testing two standard deviations by comparing the petal width of the Versicolor with that of the Setosa. As with Versicolor, we create a data frame of the Setosa species and call it `iris.setosa`. Note: The last column of **head**, Species with setosa for the output, is removed for space reasons.

R Code

```
> iris.setosa=iris[iris$Species=="setosa",]
> head(iris.setosa,n=5)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	5.1	3.5	1.4	0.2
2	4.9	3.0	1.4	0.2
3	4.7	3.2	1.3	0.2
4	4.6	3.1	1.5	0.2
5	5.0	3.6	1.4	0.2

To test to see if the variances of the width of Setosa and Versicolor are the same we use **var.test**, which is available in base R. We do not need the EnvStats package for this test. The results of **var.test** are set to `result.iris.width.var`. Within **var.test** the first two variables are the data sets, the alternative is set to two-sided and the confidence interval to 0.95. The output of the results is displayed by executing `result.iris.width.var`.

R Code

```
> result.iris.width.var=var.test(iris.versi$Petal.Width,
iris.setosa$Petal.Width,alternative="two.sided",
conf.level=0.95)
> result.iris.width.var

F test to compare two variances

data:iris.versi$Petal.Width and iris.setosa$Petal.Width
F = 3.5211, num df = 49, denom df = 49,
p-value = 2.129e-05
alternative hypothesis: true ratio of variances is
not equal to 1
95 percent confidence interval:
1.998158 6.204898
sample estimates:
ratio of variances
3.521132
```

Again **names(result.iris.width.var)** will display the information tied to `result.iris.width.var`. For example, `result.iris.width.var$p.value` returns the p-value, while `test.iris.width.var$conf.int[1]` and `test.iris.width.var$conf.int[2]` returns the left and right value of the confidence interval of the difference of the two variances.

R Code

```
> result.iris.width.var$p.value

[1] 2.128794e-05
```

R Code

```
> test.iris.width.var$conf.int[1]

[1] 1.998158
```

R Code

```
> test.iris.width.var$conf.int[2]  
[1] 6.204898
```

Code Review for the Miscellaneous Statistics Chapter

bartlett.test(v ~ f,data=name) returns the results of the Bartlett test of homogeneity of the data in v by factor f, from data frame name.

boxplot(v ~ f,data=name) creates box plots of the data in v by factor f, from data frame name. The options col and border color the boxes and the borders, respectively. The option main is for a title.

chisq.test(A) returns the results of the chi-square test of independence or homogeneity with the data given by the matrix A.

chisq.test(sample,p=test.p) returns the results of a chi-square test, testing if the data in the sample fits the distribution given by test.p.

colnames(A) is used to define the column names for A.

data(name) loads the available data set name. To see available data sets run data().

matrix(v,nrow=i) creates a matrix from vector v with i rows filling by column. If byrow=TRUE then the matrix is filled by rows.

qqline(v) adds a q-q line to a normal q-q plot.

qqnorm(v) creates a normal q-q plot of the data in vector v.

rownames(A) is used to define the row names for A.

sigma(result.aov) returns the residual standard error from the output of ANOVA stored as result.aov. More generally will produce the residual standard error from the result of appropriate statistical methods.

stack(df) stacks the columns of a data frame df; the first column will contain the data and the second the associated column name from df.

table(v) returns a table of counts of the objects in v.

TukeyHSD(result.aov) returns a Tukey pairwise comparison from the results of ANOVA stored as result.aov.

var.test(data1,data2) returns the results of testing the equality of the variances of the data sets data1 and data2. The option alternative ("two.sided", "less", "greater") sets the alternative hypothesis while conf.level defines the confidence level.

varTest(data,sigma.squared=s2) returns the results of testing whether the variance of the data is equal to s2. The option alternative ("two.sided", "less", "greater") sets the alternative hypothesis while conf.level defines the confidence level. Requires the package EnvStats.

16.4 Exercises

1. Use ANOVA to test if the sepal lengths of the three Iris species are equal. Include a boxplot and post hoc test in the analysis.
2. Using the mtcars data set, test to see if there is a differences in 1/4 mile time, qsec, by number of cylinders.
3. A third die is said not to be fair. Die 3 is rolled 300 times with 1, 2, 3, 4, 5, and 6 occurring 66, 63, 61, 35, 49, and 26 times respectively. Is die 3 different from die 1 or 2?
4. Use the mtcars data set test to see if the variation in mpg is different between four cylinder cars and six and eight cylinder cars. Hint: See [Chapter 12](#).

17

Matrices

The goal of this chapter is to provide the basics for working with matrices. We include how to create a matrix, basic operations, and calculating eigenvalues and eigenvectors. We begin by creating a matrix. Our first line of code defines the matrix A with the **matrix** command, where we first enter the values of the matrix within **c**. By default the values are entered by column. The next two entries of **matrix** define the number of rows and columns. You do not need to type nrow and ncol as **matrix(c(1,2,3,3,5,6),3,2)** will work, but it does make the code easier to follow. As you can see it looks like we wanted the values of the matrix to be 1 through 6, but we mistyped and repeated 3. We'll fix that with the next example. Note we enter A into the code so that the matrix is returned as output.

R Code

```
> A=matrix(c(1,2,3,3,5,6),nrow=3,ncol=2)
> A

 [,1] [,2]
[1,]    1    3
[2,]    2    5
[3,]    3    6
```

Informally, since A is a matrix $A[1, 2]$ represents the entry in the first row and second column. Simply setting $A[1, 2] = 4$ assigns 4 to that location in the matrix. Similarly, if you simply want to know a value of the matrix, entering $A[i, j]$ with fixed values of i and j will return that value in the matrix. We'll see further down that, for example, $A[, 1]$ and $A[1 : 2, 1 : 2]$ will return the entire first column and the 2×2 submatrix leaving out the last row, respectively.

R Code

```
> A[1,2]=4
> A

 [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```
[3,]    3    6
```

We can add another column to the matrix with the **cbind**, column bind, command. We are not looking to create a new matrix so we set **A=cbind(A, c(7,8,9))**, which lets our new *A* matrix equal the old *A* matrix but with an extra column appended. The **cbind** command will always add the new column at the end. There is also a **rbind** command, which we'll see in a moment.

R Code

```
> A=cbind(A,c(7,8,9))
```

```
> A
```

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

Here we create a new matrix *B*. It looks like we did exactly the same thing when we created *A*, but this time we added the option `byrow=TRUE`, which fills the matrix by rows instead of the default by columns.

R Code

```
> B=matrix(c(1,2,3,4,5,6),nrow=3,ncol=2,byrow=TRUE)
```

```
> B
```

```
[,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Our next example exhibits the use of the **rbind**, row bind, command. Analogous to **cbind**, the new row is added at the bottom of the matrix.

R Code

```
> B=rbind(B,c(7,8))
```

```
> B
```

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

The **rbind** command can be used to combine a number of rows. In our

next example, we are going to insert a row of zeroes between the second and third rows of the matrix A . We will call this new matrix C . The first element of **rbind** is $A[1:2,]$, which represents the first two rows of A . Check this by entering it to see the output. Using $1:2$, represents the rows 1 through 2, and leaving the column entry blank returns all columns. The second entry is our new row of three zeroes. The third entry, similar to the first, represents the third row of A . These three pieces are combined to make the matrix C .

R Code

```
> C=rbind(A[1:2,],c(0,0,0),A[3,])
> C
```

	[,1]	[,2]	[,3]
[1,]	1	4	7
[2,]	2	5	8
[3,]	0	0	0
[4,]	3	6	9

We next multiply the matrices C and A and call the product D . You can make sure the matrices can be multiplied with **dim(C)** and **dim(A)**, which will return the number of rows and columns of each matrix. Matrix multiplication is performed with `%*%`. If you use `C*A` you will get element-wise multiplication, or at least that is what you would get if the matrices were the same size. In this case you will get an error.

R Code

```
> D=C%*%A
> D
```

	[,1]	[,2]	[,3]
[1,]	30	66	102
[2,]	36	81	126
[3,]	0	0	0
[4,]	42	96	150

We end our introduction example to matrices by noting that **t** returns the transpose of a matrix.

R Code

```
> E=t(D)
> E
```

```
[,1] [,2] [,3] [,4]
[1,] 30   36   0   42
[2,] 66   81   0   96
[3,] 102  126  0   150
```

Here are a few other helpful commands. We can create a diagonal matrix with **diag**. For example **diag(c(1,1,1,1))** will create a 4×4 identity matrix. If the argument of **diag** is a matrix, then **diag(A)** will create a vector of the diagonal elements of A . You can calculate all the row and column means and sums with **rowMeans**, **rowSums**, **colMeans**, and **colSums**.

17.1 Eigenvalues, Eigenvectors and Other Operations

In order to illustrate finding eigenvalues and eigenvectors we will use an example with some ornamentation to illustrate some useful commands. We will use a 2×2 matrix to make it easier to follow and to keep the space of the output minimal. We begin by creating our 2×2 matrix but this time the elements will be randomly chosen integers from 1 to 100. The **replicate** command will output columns for your matrix for each repetition. In this case, we repeat the **sample(1:100,2)** function twice because the first entry of replicate is 2. Here **sample(1:100,2)** returns two random integers from the range 1:100. There are other functions that can be used in **replicate** and basically anything that returns a string of numbers will work. Two functions that might be useful are **rnorm(n,mean=a,sd=b)**, which will return n values from a random normal distribution with mean a and standard deviation b , and **seq(a,b,c)**, which returns a sequence of numbers starting at a by steps of c and stops when it exceeds b . Keep in mind that these functions can be nested. For instance, what will **D=replicate(10, rnorm(20,mean=sample(1:10,1),sd=1))** produce? Note that **colMeans(D)** will return the means of each column, and **colSds(D)** will return the column standard deviations, but this requires the matrixStats package.

We next output our matrix to see its elements and find the determinant of the matrix with **det(A)**. The determinant is 2488 and so A has an inverse. Our next line sets B to be the result of multiplying A with its inverse, produced by **solve(A)**. Since a rounding error is likely, check this by outputting B with it digits rounded to one place. As we see, B is the identity matrix as it should be. Note that in this example we are creating a matrix with random elements and so each time this code is executed A will almost certainly be different.

R Code

```
> A=replicate(2, sample(1:100, 2))
> A
```

```
[,1] [,2]
[1,] 71 84
[2,] 38 80
```

R Code

```
> det(A)
```

```
[1] 2488
```

R Code

```
> B=A%*%solve(A)
> round(B,digits=1)
```

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

Our next example illustrates solving matrix equations. We create the vector b with the two elements 1 and 3. There is nothing special about this choice. We check b and then set x to be the solution of $Ax = b$ by using **solve(A,b)**. We see that the solution is $(-0.069, 0.070)$, which you can check by performing $A\%*%x$.

R Code

```
> b=c(1,3)
> b
```

```
[1] 1 3
```

R Code

```
> x=solve(A,b)
> x
```

```
[1] -0.06913183280 0.07033762058
```

We now turn our attention to the eigenvalues and eigenvectors of A . The command **eigen(A)** calculates these values and we set them to y . Note that

y contains a fair amount of information. The next two lines separate out the vectors and values of y by setting `Evals` to the eigenvalues, `y$val`, and `Evecs` to the eigenvectors, `y$vec`. We enter `El` and `Ec` to see those values. Our last line checks the first of our eigenvectors and eigenvalues by multiplying the matrix A by the first column of `Ec` and then subtract the result of multiplying the first eigenvalue times the first eigenvector. Notice the use of `%*%` and `*` in this line of code. Our output should be a zero vector and it is.

R Code

```
> y=eigen(A)
> Evals=y$val
> Evals
[1] 132.1767148 18.8232852
```

R Code

```
> Evecs=y$vec
> Evecs
[,1] [,2]
[1,] -0.8083431292 -0.8494643832
[2,] -0.5887116318  0.5276459625
```

R Code

```
> A%*%Evecs[,1]-Evals[1]*Evecs[,1]
[,1]
[1,] 0
[2,] 0
```

We end this section by providing two ways for calculating the power of a matrix. It turns out that A^n is not repeated matrix multiplication as it raises each element of the matrix to the power n . It is often the case when you want to do something in R that someone has created a package to do it. In this case the `expm` package solves the problem as the next example illustrates. Take a moment to look at the matrix A in the example and figure out what multiplying A by itself six times will produce. In this example the matrix A is unchanged and if you want to use some power of A then you should do something like `B=A%^%6`. Packages are helpful but sometimes you can do it yourself. In this case calculating powers of a matrix allows us to provide an example of a for loop to solve the same problem.

R Code

```
> library(expm)
> A=matrix(c(1,0,1,1),2,2)
> A%^%6
```

```
[,1] [,2]
[1,] 1 6
[2,] 0 1
```

Our second method for computing a power of a matrix starts by defining the same matrix. Our next line defines a function. The function has two inputs: a matrix A and an integer n . We first set B equal to A since we will need to retain a copy of A throughout. We then begin our for loop by having i range from 1 to $n - 1$. For each value of i we set A to be the matrix multiplication of A times B . The function then returns A . Our last line of the code evaluates **f(A,10)**. We used a matrix A that helps us check our code and we do get what we expect.

R Code

```
> A=matrix(c(1,0,1,1),2,2)
> f=function(A,n){
+ B=A
+ for(i in 1:(n-1)) A=A%*%B
+ return(A)}
> f(A,10)
```

```
[,1] [,2]
[1,] 1 10
[2,] 0 1
```

There are a few items to point out in this code. First there are three + signs in the code. When typing our function in the editor we used four lines to make it easier to follow. When the code was sent to the console the + signs were automatically added to note that the lines are all part of one command. Note that the domain of n is $n \geq 2$. We could add an if statement for the case when $n = 1$, but it isn't worth it here. Finally, note that the matrix A sent to the function is replaced. We leave it as an exercise to fix this. Finally note that the code **f(A,20)[1,2]** makes sense. What should it return? Note that this will only work once since the original A is replaced by a new matrix, a power of A , with the current function. To work with the result of **f(A,20)** set it equal to another variable, say $A.20$.

17.2 Row Operations

It is instructive to know how to perform row operations to row-reduce matrices. For the most part this is easy enough to do except for a little catch in switching rows. We begin by defining a simple matrix A that will be easy to follow through the example.

R Code

```
> A=matrix(c(2,2,1,1),2,2,byrow=TRUE)
> A

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

We next multiply a row by a scalar. Specifically we multiply the first row by 2. The key here is that $A[1,]$ represents the first row because leaving the column entry blank produces the entire associated columns.

R Code

```
> A[1,]=A[1,]*2
> A

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

We now subtract four times row 2 from row 1.

R Code

```
> A[1,]=A[1,]-4*A[2,]
> A

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

Finally, we will swap rows 1 and 2. The key here is that we need to temporarily save the second row and so the first line of code sets B to the second row of A . Our next line sets row 2 to that of row 1. We can now set row 1 to B which was the old row 2.

R Code

```
> B=A[2,]
> A[2,]=A[1,]
> A[1,]=B
> A
```

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

Code Review for the Matrix Chapter

A%*%B is matrix multiplication.

A[j] refers to the entire jth column, whereas **A[i,]** refers to the entire ith row.

A[i,j] refers to the element in the ith row and jth column of *A*.

cbind(A,B) adds the columns *B* to the end of *A*, where *B* can be simply a vector or a matrix. The command can take more than 2 arguments.

det(A) returns the determinant of *A*.

diag(a) creates a square matrix with the vector *a* on the diagonal and 0 elsewhere.

diag(A) returns a vector with the diagonal elements of *A*.

e=eigen(A) calculates the eigenvalues and eigenvectors of *A* and stores them as *e*. Use *e\$val* and *e\$vec* to refer to the eigenvalues and eigenvectors specifically.

matrix(c(, , ,),nrow=a,ncol=b) creates an $a \times b$ matrix with the elements in *c(, , ,)*.

rbind(A,B) is similar to cbind but adds rows at the bottom of matrix *A*.

replicate(i,function()) will create a matrix with *i* columns with rows created by the function. The function is often one that will create random numbers such as `sample(a:b,n)` or `rnorm(n, mean=a, sd=b)`.

rowMeans(A) returns the means of each of the rows of *A*, with **colMeans(A)** returning means for the columns.

rowSums(A) returns the sums of each of the rows of A , with **Sums(A)** returning sums for the columns.

solve(A) returns the inverse of A .

t(A) transposes matrix A .

17.3 Exercises

1. Fix the function to calculate the power of a matrix so that it works for $n = 1$. Hint: Use an if-else statement.
2. Fix the function to calculate the power of a matrix so that the matrix A is preserved. Hint: Very little has to be changed.
3. Use column operations to reduce the matrix that we did by row operations.
4. Find the row reduced form of the matrix created by `matrix(sample(10, 9, replace=TRUE), nrow=3, ncol=3)`.
5. Create a 10 by 10 matrix where each entry is the sum of its row and column.
6. This is a mini monopoly example. Consider a square game board with locations from 1 to 8 starting in the lower right corner and circling clockwise. If you land on square 5 you get sent to square 1. In playing this game you start on square 1 and use a four sided die. What is the longterm distribution of the occupancy of squares? Which square is most popular? Hint: The eigenvector of the transition matrix with the largest eigenvalue.

18

Differential Equations

We present three examples using R to analyze differential equations. We exhibit Newton's law of cooling which has only one parameter; the logistic model which has two parameters; and the predator-prey model, which is a system of equations. Each case has a numerical solution that uses the deSolve package and a phase plane that uses the phaseR package. These three examples build understanding throughout the chapter, so it is expected that the reader not skip around in this chapter.

18.1 Newton's Law of Cooling

Our first example models a cup of tea that starts at boiling temperature and cools in a 70° room. In other words, we consider the differential equation $y'(t) = -k(y - 70)$ where we take $k = 0.1$ and $y(0) = 212$. We load the deSolve package and define the differential equation, Cooling.DE, with **function**. The function has three variables t, y, and parameter and the definition begins with a left brace. The first line sets k to parameter while the second expresses the differential equation. The last line, **list(dy.dt)**, returns the result as list. The function that numerically solves the differential equation is **ode**, and before invoking that function we define the initial value, the set of times for evaluation, and the parameter k. The next three lines do just this. We set $y.0=212$ and define a sequence of values from 0 to 60 in steps of 0.01 using **seq**. We define the parameter k using **parameters=c(k=0.10)**. At this point we could simply use **parameters=0.10**, but the syntax is a good habit and becomes important when there is more than one parameter. We assign Cooling.DE.Results to the output of **ode**. The input of **ode** consists of the initial value, the times, the differential equation function, and the parameter, in that order. We can simplify this by typing **ode(y.0,t,Cooling.DE,parameters)**, but add the extra text for clarity and to attach names to the values. The output is in the form of a matrix — this can be checked with **class(Cooling.DE.Results)** — and we display the first 10 rows with **head**.

R Code

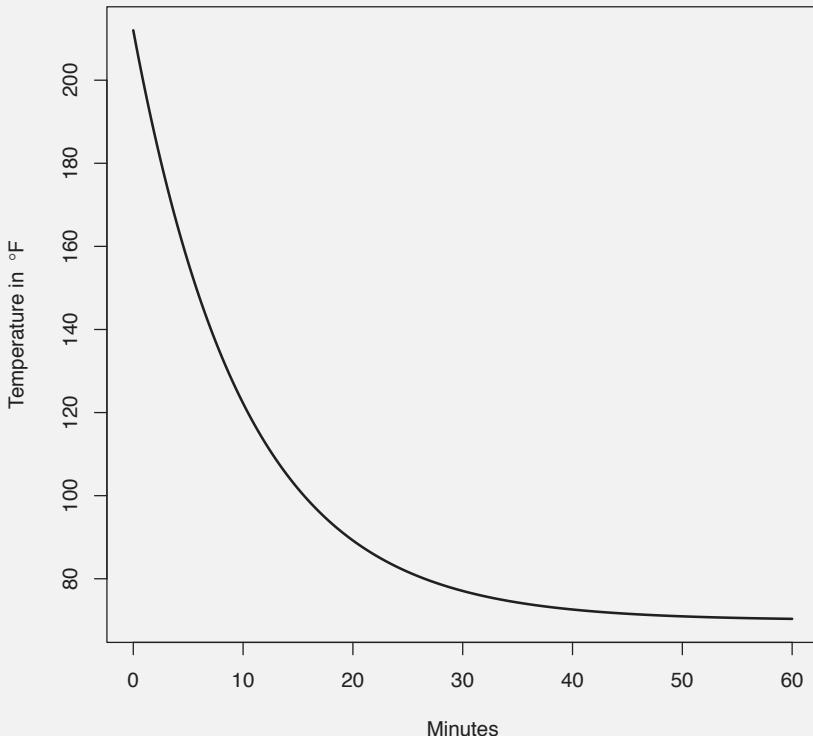
```
> library(deSolve)
> Cooling.DE=function(t,y,parameters){
+ k=parameters
+ dy.dt= -k*(y-70)
+ list(dy.dt)}
> y.0=212
> t=seq(0,60,0.01)
> parameters=c(k=0.10)
> Cooling.DE.Results=ode(y=y.0,times=t,func=Cooling.DE,
parms=parameters)
> head(Cooling.DE.Results,n=10)
```

time	y
[1,]	0.00 212.0000
[2,]	0.01 211.8581
[3,]	0.02 211.7163
[4,]	0.03 211.5746
[5,]	0.04 211.4331
[6,]	0.05 211.2918
[7,]	0.06 211.1506
[8,]	0.07 211.0095
[9,]	0.08 210.8685
[10,]	0.09 210.7277

A graph of the result is presented using **plot**. The **plot** function takes the first column of a matrix as the independent variable and the second as the dependent variable. The line width is set to 2 with **lwd**, the default title is set to blank with **main** equal to empty quotes, and the *x*-axis is labelled with **xlab**. The *y*-axis is labelled using **expression**, to typeset mathematical expression, along with **paste** to concatenate characters and mathematical expressions. Within **paste**, text is in quotes and \sim degree is for the degree symbol. Each of the three components are separated by commas.

R Code

```
> plot(Cooling.DE.Results,lwd=2,main="",xlab="Minutes",
ylab=expression(paste("Temperature in", ~"degree,"F")))
```



We would like to know how long it will be before we can drink the tea, i.e., when the temperature gets to 136° . This will take two steps. First, we store the output from a matrix as a data frame using **as.data.frame** and name the data frame as `Cooling.DE.Results.df`. There isn't necessarily a time when the tea is exactly 136° . If there were we could do `Cooling.DE.Results.df[Cooling.DE.Results.df$y==136,]`, which would return all rows where the temperature is 136° . Instead we create a logical statement using `&` (for and) and search for rows where the temperature is greater than 135.5° and less than 136.5° . The tea is below 136° by 7.67 minutes.

R Code

```
> Cooling.DE.Results=df=
  as.data.frame(Cooling.DE.Results)
> Cooling.DE.Results=df [Cooling.DE.Results=df$y>135.5 &
```

```
Cooling.DE.Results.df$y<136.5, ]
```

time	y
760	7.59 136.4751
761	7.60 136.4086
762	7.61 136.3423
763	7.62 136.2759
764	7.63 136.2097
765	7.64 136.1435
766	7.65 136.0774
767	7.66 136.0114
768	7.67 135.9454
769	7.68 135.8795
770	7.69 135.8136
771	7.70 135.7479
772	7.71 135.6821
773	7.72 135.6165
774	7.73 135.5509

We now turn our attention to using phaseR to create a slope field, with two trajectories, and the nullcline. As such, we begin by loading the phaseR package. We can use the same Cooling.DE function created for use with **ode**. We define our parameter k=0.01 directly, which is slightly different than with **ode**. The slope field is generated with the **flowField** function where the first argument must be the differential equation, in this case Cooling.DE. Many of the arguments to **flowField** are similar to **plot** such as xlim, ylim, xlab, ylab, main, and col. Specific to **flowField** are points for the number of arrows and system to define the DE as one- or two- dimensional. For **flowField**, the default for add is TRUE but here we use it to generate a new graph, so we set add to FALSE. We store the output of **flowField** to Cooling.Flow which suppresses the report generated by flowField; that includes the slopes for each of the arrows. Entering Cooling.Flow will produce the report if desired.

The functions **trajectory** and **nullclines** are similar. For **trajectory** we set y0 to a list of initial values and tlim for the starting and ending times of the trajectories. The use of **nullclines** by default adds the legend in the bottom right of the graph; and may be suppressed with add.legend=FALSE.

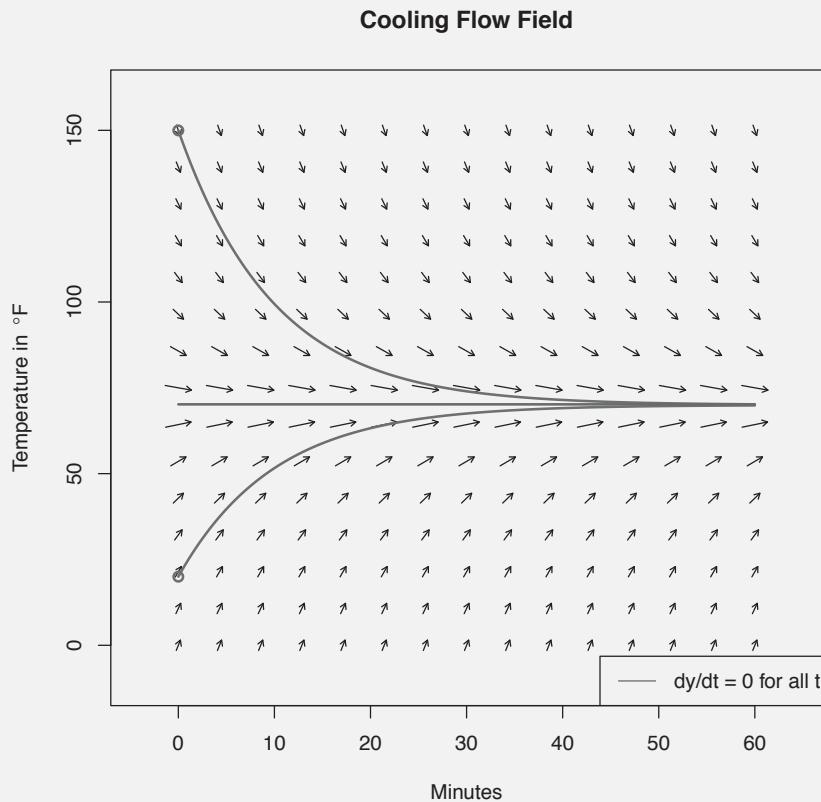
R Code

```
> library(phaseR)
> k=0.1
> Cooling.Flow=flowField(Cooling.DE,xlim=c(0,60),
+ ylim=c(0,150),main="Cooling Flow Field",xlab="Minutes",
+ ylab=expression(paste("Temperature in", ~~degree,"F")),
```

```

parameters=k,points=15,col="black",system="one.dim",
add=FALSE)
> Cooling.Traj=trajectory(Cooling.DE,y0=c(20,150),tlim=
c(0,60),lwd=2,col="red",parameters=k,system="one.dim")
> Cooling.Null=nullclines(Cooling.DE,xlim=c(0,60),
ylim=c(0,200),lwd=2,parameters=k,system="one.dim")

```



18.2 The Logistic Equation

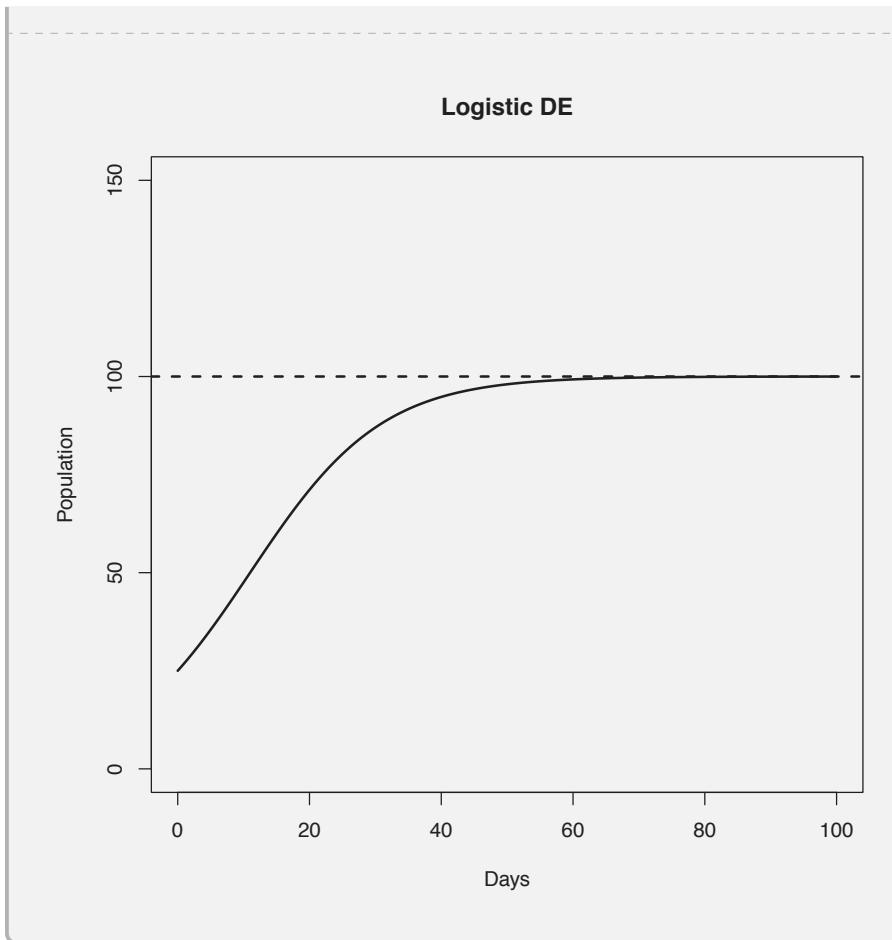
The differential equation that represents the logistic model is a classic two-parameter differential equation and our the next example. Similar to above, we define the model and graph a solution with given values of the parameters and

initial condition. The Logistic.DE function defined here has three variables, t, y, and parameters, where parameters is a vector of two values. Within the function definition we set k to the first value of parameters and r to the second. Note that square brackets are used to identify the location in a vector. The differential equation is defined and set to dy.dt and the last line, which is the value the function returns, is **list(dy.dt)**. The variable t is a list of values, so the function returns a list of values given by dy.dt as in the previous example.

The variable t is defined to be a sequence of values from 0 to 100 in steps of 0.01. We set y.0 to 25 as the initial value. The parameters are defined as c(k=100,r=0.1). Note that c(100,0.1) also works, but the previous definition adds clarity. The **ode** function generates the numerical solution to the differential equation. The inputs to the function in order are the initial value, the t-values, the function, and the parameters. We can simplify this by entering **ode(y.0,t,Logistic.DE,parameters)**, but we prefer clarity and attaching names to the values. The output, set to Logistic.DE.results, is a two-column matrix with the first column t and the second the values of y. The **plot** function interprets this output for graphing. We set the y-axis range to 0 to 50, a line width of 2, label axis, and provide a title. The last line of code adds a horizontal line at 100 of type 2, dashed, and width 2.

R Code

```
> Logistic.DE=function(t, y, parameters){
+ k=parameters[1]
+ r=parameters[2]
+ dy.dt= r*y*(1-y/k)
+ list(dy.dt) }
> t=seq(0,100,0.01)
> y.0=25
> parameters=c(k=100,r=0.1)
> Logistic.DE.results=ode(y=y.0,times=t,
func=Logistic.DE,parms=parameters)
> plot(Logistic.DE.results,ylim=c(0,150),lwd=2,main=
"Logistic DE",xlab="Days",ylab="Population")
> abline(h=100,lty=2,lwd=2)
```



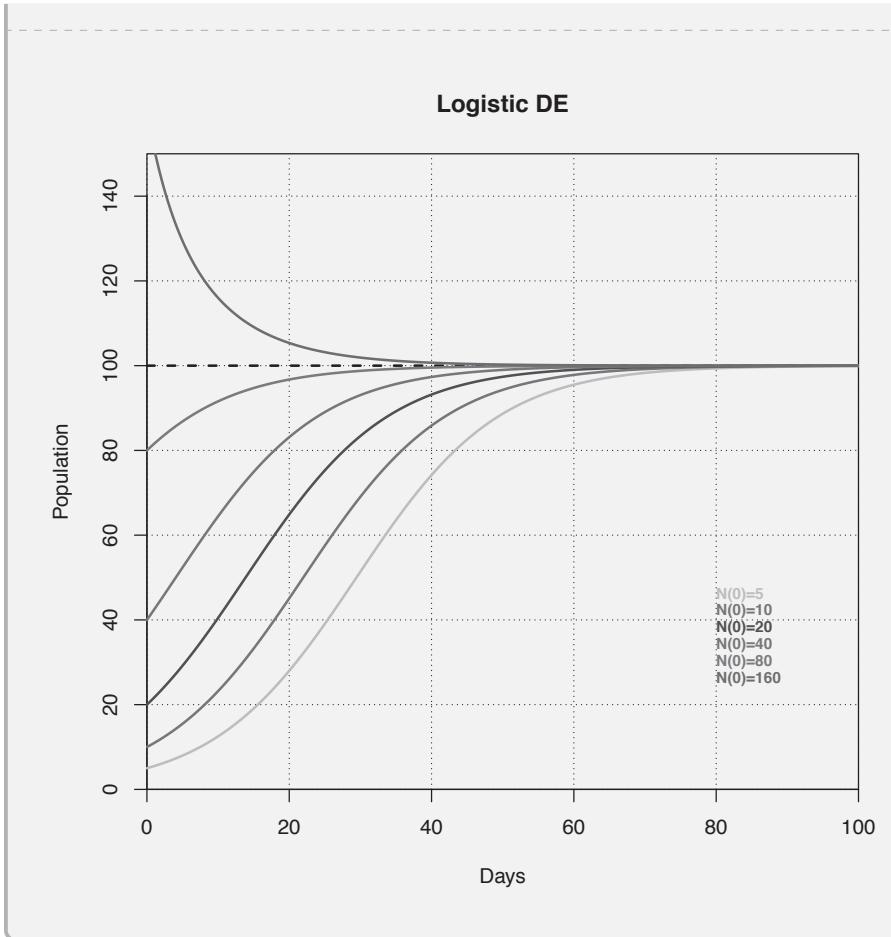
The next example with the logistic model creates a graph with results based on different initial values. We begin with **plot** to initiate a graphing window. We plot Logistic.DE.results but set the type to “n” to set up a graph frame but plot nothing. The *y*-axis range is set with ylim and the *x*-axis is set by default since we are plotting Logistic.DE.results. Setting xaxs and yaxs to “i” removes the extra space around the graph. The line width is set to 2 and a title and axis labels are set using main, xlab, and ylab. A grid is added with **grid** where NULL adds grid lines at the tick marks for the respective axes. A dashed horizontal line at 100 is added with **abline** using lty=2 for a dashed line. The line width is set to 2. We set colors to 6 colors from the **rainbow** palette in the range from cyan, start=3/6, to red, end=6/6. The variable *j* will be used as a counter and is initially set to 0.

The for loop has *i* range through the values 5, 10, 20, 40, 80, and 160, which will be the initial values for the model. The variable *j* is increased by one with *j=j+1*. Logistic.DE.results is set to the output of **ode** with the initial

value y set to i , and the rest is the same as the previous example. The result of the output is added to the graph with **lines**. The line width is set to 2 and the color is the j th color in the vector colors. The **text** function is used to create a legend of initial values. Text is added to the graph at $(80, 50 - 4j)$, which was determined by trial and error. The **paste** function is used to concatenate the text in quotes and the current value of i . The option sep is set to empty quotes so that no spaces are added between the text and variable i . The font size is scaled to 0.75 with cex, the font is bold with font=2, and adj=0 left justifies the text. The color is the j th color in the vector colors.

R Code

```
> library(phaseR)
> plot(Logistic.DE.results,type="n",ylim=c(0,150),
+ xaxs="i",yaxs="i",lwd=2,main="Logistic DE",xlab="Days",
+ ylab="Population")
> grid(NULL,NULL,col="black")
> abline(h=100,lty=2,lwd=2)
> colors=rainbow(6,start=3/6,end=6/6)
> j=0
> for (i in c(5,10,20,40,80,160)){
+ j=j+1
+ Logistic.DE.results=ode(y=i,times=t,func=Logistic.DE,
+ parms=parameters)
+ lines(Logistic.DE.results,lwd=2,col=colors[j])
+ text(80,50-4*j,paste("N(0)=",i,sep=""),cex=0.75,
+ font=2,col=colors[j],adj=0) }
```



We conclude this example as we did the previous one: we graph the slope field and add three trajectories based on different initial values. Recall that this requires loading the `phaseR` package. The code begins by defining `k` and `r`. The `flowField` function creates the slope field for the `Logistic.DE` model. The `xlim`, `ylim`, `xlab`, `ylab`, and `main` arguments are used to set the axis ranges and label the axes and title. The number of arrows is set by `points`. The system is one dimensional and the color of the arrows is set to black. The slope field is not being added to another graph and so `add=FALSE`. Note that assigning `flowField` to `Logistic.Flow` suppresses output in the console as it is stored in the variable.

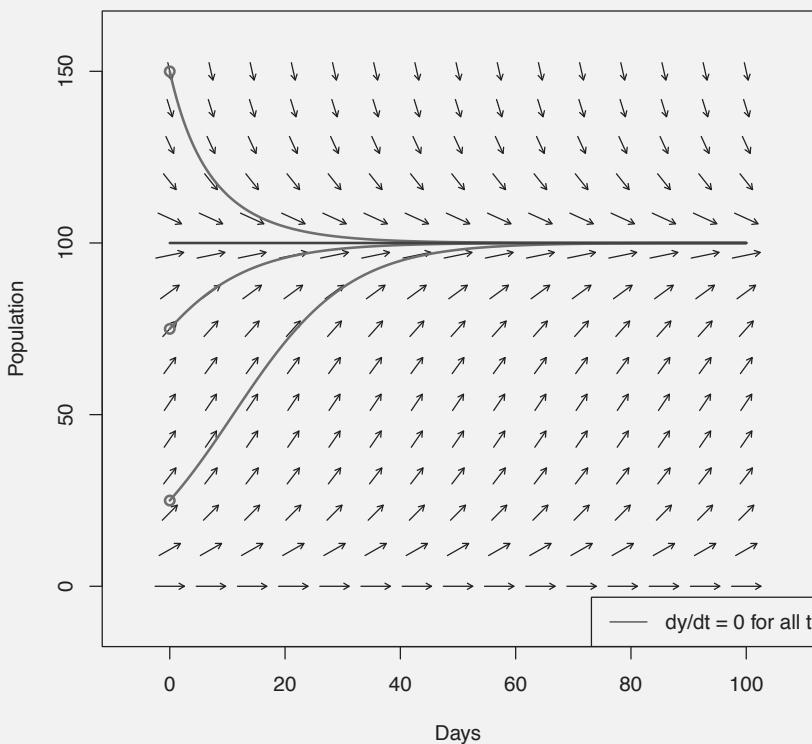
The trajectories are added to the graph with the `trajectory` function. The inputs are similar to `flowField` except that the initial values used are set to `y0` and `tlim` is set to the beginning and end of the trajectory. For the same reasons the output of `trajectory` is assigned to `Logistic.Traj`. Nullclines are

added with the **nullclines** function. The **nullclines** function adds the legend in the bottom right and can be suppressed with `add.legend=FALSE`.

R Code

```
> k=100
> r=0.1
> Logistic.Flow=flowField(Logistic.DE,xlim=c(0,100),
  ylim=c(0,150),main="Logistic DE",xlab="Days",ylab=
  "Population",parameters=c(k,r),points=15,system=
  "one.dim",col="black",add=FALSE)
> Logistic.Traj=trajectory(Logistic.DE,y0=c(25,75,150),
  tlim=c(0,100),lwd=2,col="red",parameters=c(k,r),
  system="one.dim")
> Logistic.Null=nullclines(Logistic.DE,xlim=c(0,100),
  ylim=c(0,150),parameters=c(k,r),lwd=2,system="one.dim")
```

Logistic DE



Two more function are worth noting. The **findEquilibrium** functions returns the fixed point and its classification. It needs a reasonable starting point for the algorithm and here we set $y0=90$ as an example. We assign the results to `Logistic.Eq` to limit the output to the console. To see what is stored in the variable use **names(Logistic.Eq)**. For example, `Logistic.Eq$classification` will return “stable” while `Logistic.Eq$ystar` is the value of the equilibrium. The function **stability** will return the stability of an equilibrium point passed to `ystar`.

R Code

```
> Logistic.Eq=findEquilibrium(Logistic.DE,y0=90,
parameters=c(k,r),system="one.dim")

1 101.25
2 100.0152
3 100
4 100
Fixed point at y = 100

Discriminant: -0.1    Classification: Stable
```

R Code

```
> names(Logistic.Eq)

[1] "classification" "deriv"           "discriminant"
[4] "h"              "max.iter"        "parameters"
[7] "plot.it"        "summary"         "system"
[10] "tol"            "y0"              "ystar"
```

R Code

```
> Logistic.Stability=stability(Logistic.DE,ystar=100,
parameters=c(k,r),system="one.dim")

Discriminant: -0.09999999    Classification: Stable>
```

18.3 Predator-Prey Model

Our final example of this chapter illustrates the classic predator-prey model. The model has prey, x , that grows exponentially and dies at a rate proportional

to the interaction between prey and predator, y , so that $\frac{dx}{dt} = ax - bxy$. The predators grow proportional to the interaction with prey and die at a exponential rate so that $\frac{dy}{dt} = cxy - dy$. We begin by defining the model with **function** and setting it to PredPrey.DE. The function has three variables t, y, and parameters. The definition of the function begins with a left brace. The values a, b, c, and d are assigned to the first four values of parameters. The two differential equations are implemented by dy[1] ($\frac{dx}{dt}$) and dy[2] ($\frac{dy}{dt}$). In general, the variable t will be a list of time values; the function returns the list that is defined by dy. After the function definition is ended with a right brace, the variables are defined. We assign y.0 to a pair of initial values. Note we could just use c(2,2), but including y1 and y2 makes the code easier to follow. We define t to be a sequence of values from 0 to 100 in steps of 0.01 using the **seq** function. Last, parameters defines the values of a, b, c, and d. As above we could have used c(20,5,2,10) but adding the variable names adds clarity. The solution is computed using the **ode** function as in the previous sections. The results are assigned to PredPrey.DE.result. The first 10 lines of output are exhibited with **head**.

R Code

```
> PredPrey.DE=function(t,y,parameters) {
+ a=parameters[1]
+ b=parameters[2]
+ c=parameters[3]
+ d=parameters[4]
+ dy = numeric(2)
+ dy[1]=a*y[1]-b*y[1]*y[2]
+ dy[2]=c*y[1]*y[2]-d*y[2]
+ return(list(dy))
+ }
> y.0=c(y1=2,y2=2)
> t=seq(0,100,0.01)
> parameters=c(a=20,b=5,c=2,d=10)
> PredPrey.DE.result=ode(y=y.0,times=t,func=PredPrey.DE,
  parms=parameters)
> head(PredPrey.DE.result,n=10)
```

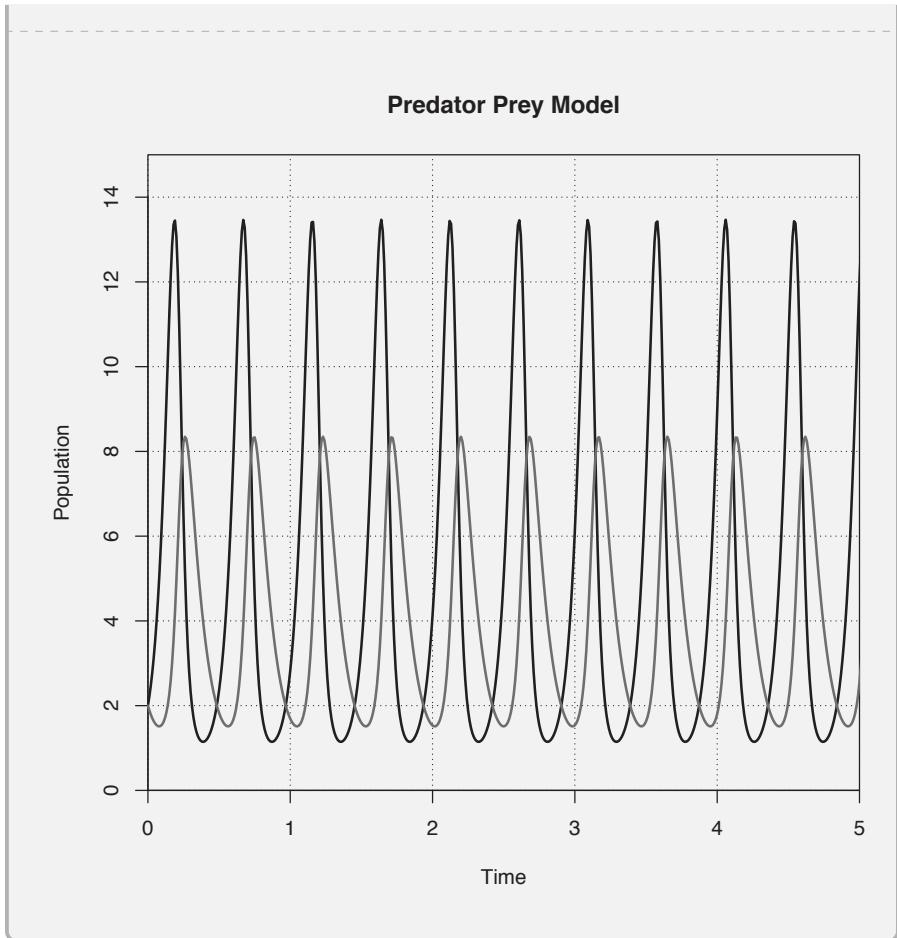
	time	y1	y2
[1,]	0.00	2.000000	2.000000
[2,]	0.01	2.216704	1.887508
[3,]	0.02	2.469837	1.789719
[4,]	0.03	2.764365	1.706300
[5,]	0.04	3.105838	1.637133
[6,]	0.05	3.500327	1.582356
[7,]	0.06	3.954282	1.542427

```
[8,] 0.07 4.474300 1.518205  
[9,] 0.08 5.066713 1.511064  
[10,] 0.09 5.736922 1.523055
```

To graph the results the natural choice is to **plot(PredPrey.DE.result)** and this works, but it makes side-by-side graphs for y1 and y2. It is more instructive to plot the results of y1 and y2 on the same axis. Recall that PredPrey.DE.result[,i] represents all rows, the “empty” first co-ordinate means “all”, and the ith column, in other words, the ith column of the matrix. So, **plot** graphs the first column as the independent variable and the second column as the dependent variable. The type is l for line, and setting xaxs and yaxs to i removes the extra space around the graph. Ranges and labels are specified with xlim, ylim, xlab, ylab, and main. The line width is 2. Using NULL in **grid** places grid lines at the tick marks of the current graph. The grid lines are colored black. The **lines** function adds to the graph the data from y2. The line width is set to 2 and the color is red. The interplay between predator and prey is better represented when graphed on the same set of axes and we now move to creating a phase plane.

R Code

```
> plot(PredPrey.DE.result[,1],PredPrey.DE.result[,2],  
type="l",xaxs="i",yaxs="i",xlim=c(0,5),ylim=c(0,15),  
lwd=2,main="Predator Prey Model",xlab="Time",  
ylab="Population")  
> grid(NULL,NULL,col="black")  
> lines(PredPrey.DE.result[,1],PredPrey.DE.result[,3],  
lwd=2,col="red")
```



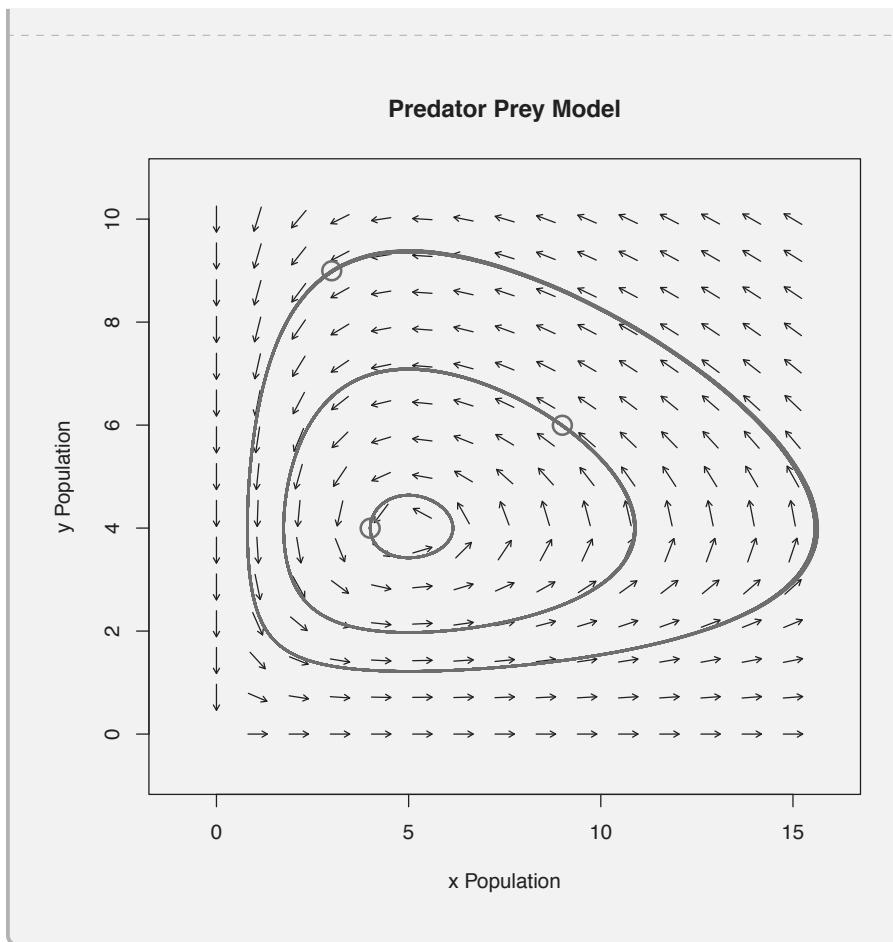
The next graph is a phase plane with three trajectories. We begin by assigning values to the parameters a , b , c , and d . The **flowField** function creates the phase plane of the differential equation defined by PredPrey.DE. The ranges for x and y are set by xlim and ylim. The line width is 2 and a title and axes labels are set with main, xlab, and ylab, respectively. The values of the parameters are set to parameters and points sets the number of arrows drawn. The system is two dimensional, system="two.dim", and the arrows are colored black. The default for **flowField** is add=TRUE, but here it initiates a new graph so add=FALSE. Note that storing **flowField** as PredPrey.Flow suppresses output in the console.

The variable y.ini is a two-column matrix for three initial values for the system. The vector 4, 4, 3, 9, 9, and 6 fills the matrix byrow, set to TRUE. The three rows of the matrix, which are (4,4), (3,9), and (9,6), represent the three initial values. The **trajectory** function adds the trajectories to the graph with the initial values given by y0. Here tlim sets the limits of the independent

variable for plotting the solution. In looking at the previous plot we saw that a time frame from 0 to 1 will cycle through the system more than once. If we used $c(0,0.25)$, for example, the trajectories would not complete a cycle. The use of cex scales the circle representing the initial value.

R Code

```
> a=20
> b=5
> c=2
> d=10
> PredPrey.Flow=flowField(PredPrey.DE,xlim=c(0,15),
  ylim=c(0,10),main="Predator Prey Model",xlab="Time",
  ylab="Population",parameters=c(a,b,c,d),points=15,
  system="two.dim",col="black",add=FALSE)
> y.ini=matrix(c(4,4,3,9,9,6),ncol=2,byrow=TRUE)
> PredPrey.Traj=trajectory(PredPrey.DE,y0=y.ini,
  tlim=c(0,1),lwd=2,col="red",parameters=c(a,b,c,d),
  system="two.dim",cex=2)
```



The find **findEquilibrium** function does just that. A reasonable starting value set to y_0 is necessary. Setting `plot.it` to TRUE will add the equilibrium point to the current graph. Assigning the output to `PredPrey.Eq` limits the output of the function and allows access to output as given by **names(PredPrey.Eq)**. The **stability** function returns stability information. Check **names(PredPrey.Stability)** to see what is stored in `PredPrey.Stability`.

R Code

```
> PredPrey.Eq=findEquilibrium(PredPrey.DE,y0=c(4,3),
parameters=c(a,b,c,d),system="two.dim",plot.it=TRUE)
```

```
1 5.454545 4.363636
2 5.034965 4.027972
3 5.000241 4.000193
```

```

4 5 4
5 5 4
Fixed point at (x,y) = 5 4

T: 0 Delta:200 Discriminant:-800 Classification:Centre

```

R Code

```

> names(PredPrey.Eq)

[1] "classification" "Delta"           "deriv"
[4] "discriminant"   "eigenvalues"     "eigenvectors"
[7] "h"              "jacobian"        "max.ite"
[10] "parameters"    "plot.it"         "summary"
[13] "system"         "tol"             "tr"
[16] "y0"             "ystar"

```

R Code

```

> PredPrey.Stability=stability(PredPrey.DE,ystar=c(5,4),
parameters=c(a,b,c,d),system ="two.dim")

```

T:0 Delta:200 Discriminant:-800 Classification:Centre

Code Review for the Differential Equations Chapter

findEquilibrium(DE,y0=,parameters=,system=) finds the equilibrium value(s) of the differential equation(s) given by DE. The variable y0 defines a starting point for the algorithm to find the equilibrium point(s). Parameters need to be assigned and system assigned as either “one.dim” or “two.dim.”

flowField(DE,xlim=,ylim=,parameters=) creates a phase plane or flow field of the differential equation(s) given by DE, with the graph window define by xlim=c(a,b) and ylim=c(c,d), and any parameters and set to parameters. The number of arrows can be set to points, and define system as either “one.dim” (the defualt) or “two.dim” with system=. If this initiates a plot frame then set add to FALSE. The usual graph options are available.

nullclines(DE,xlim=,ylim=,parameters=,system=) adds nullclines to the graph created by flowField for the differential equation(s) given by DE, over the axis range set by xlim and ylim. Pa-

rameters need to be assigned and system assigned as either “one.dim” or “two.dim.”

ode(y=,times=,func=,parms=) numerically solves a differential equation with initial value assigned to y, over a time interval with time steps assigned to times, with a given equations assigned to func, and parameters assigned to parms.

stability(DE,ystar=,parameters=,system=) returns the stability of the equilibrium point set to ystar of the differential equation given by DE. Parameters need to be assigned and system assigned as either “one.dim” or “two.dim.”

trajectory(DE,y0=,tlim=c(a,b),parameters=) adds trajectories to the flow field graph with the same differential equation(s) given by DE. Initial values for starting points of the trajectories are assigned to y0 and can be a vector for a single equation or a matrix for a system of equations. The variable tlim defines the beginning and end time for the trajectories. Parameters need to be set and use either “one.dim” (default) or “two.dim” as the argument of system=. The width and color of the trajectories can be set with lwd and col.

18.4 Exercises

1. Open your favorite differential equations text, pick a differential equation or system of equations, and create a graphical solution and plot a phase plane.
2. In [Section 18.1](#) suppose the room temperature is 83 degrees and the tea starts out at a temperature of 198 degrees. How long before the tea is 136 degrees?
3. Create a graph of solutions to the logistic equation with $k=100$ and values of r of 0.1, 0.3, 0.5, 0.7, and 0.9.
4. In the predator-prey model we investigated, what is the first time at which there are twice as many prey as predators?

19

Some Discrete Mathematics

This chapter provides a starting point for using R for discrete mathematical ideas. The chapter is divided into three independent sections. Throughout the chapter we provide examples of combinatorics, number theory, set theory, and graph theory. We construct the first dozen iterations of the Cantor set and estimate a random graph question with a simulation. The packages numbers, sets, eulerr, and igraph are highlighted in this chapter.

19.1 Binomial Coefficients, Pascal’s Triangle, and a Little Number Theory

We begin this section by creating the first twelve rows of Pascal’s triangle with the even numbers colored red. We begin by setting $n=12$, the number of rows, and the margins to 0 on all four sides with `par(mar=c(0,0,0,0))`. We create an empty graph frame with `plot` by setting `type="n"` so that the point $(0,0)$ doesn’t appear. The x and y ranges are set with `xlim` and `ylim`, the axes are suppressed with `axes=FALSE`, and both axes labels are blank due to the empty quotes.

We nest two for loops. The first has i ranging from 0 to n , which will be the rows of the triangle, and then j from 0 to i , which will be the position within row i . An if then else statement is used to determine the color of the text. The command `choose(i,j)` produces $\binom{i}{j}$; thus, `choose(i,j)%%2==1` tests to see if the binomial coefficient is odd as $\% \% 2$ is mod 2 arithmetic and two equal signs are the logical test for equality. If the statement is true the color is set to “black”; otherwise it is set to “red”. Outside the if then statement the `text` function places the value of `choose(i,j)` at $(2*j-i,-i)$ with `col` set to color. Note that the rows start at $y = 0$ and go down and that along each row we place values from $-i$ to i . Both of these are choices for simplicity and spacing.

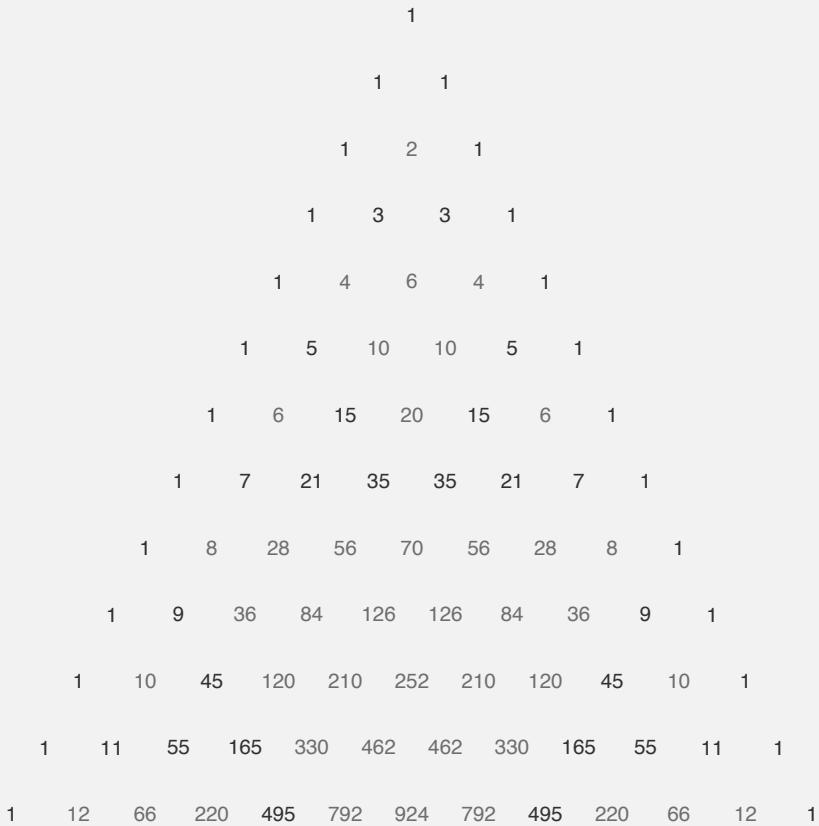
R Code

```
> n=12
> par(mar=c(0,0,0,0))
> plot(0,0,type="n",xlim=c(-n,n),ylim=c(-n,0),
```

```

axes=FALSE,ylab="",xlab="")
> for(i in 0:n){
+ for(j in 0:i){
+ if (choose(i,j)%>2==1){color="black"}
else {color="red"}
+ text( 2*j-i,-i,label=choose(i,j),col=color)
+ }})

```



We pause for a moment to note that **lchoose(n,k)=** $\ln \binom{n}{k}$, **factorial(x)=** $x!$, and **ifactorial(x)=** $\ln(x!)$.

Moving on, we will next compute the sum of the 20th row. We set n=19 and use **sapply**, which returns a vector of values from evaluating the function for each number from 0 to n, 0:n. Note that the first row has n=0 and so the 20th row is with n=19. We have **sapply** nested within **sum** so the value returned is the sum of that vector.

R Code

```
> n=19  
> sum(sapply(0:n,function(x){choose(n,x)}))  
  
[1] 524288
```

The sum of the 20th row is 524288 and it would help to see a pattern if the number were represented as a product of its prime factors. To do this we will use the numbers package. We load the package with library(numbers). We nest our code within the **primeFactors** function which returns the prime decomposition of 524288 or 2^{19} . Another helpful function in the numbers packages is **isIntpower** which decomposes a number into the power of an integer. In this case we get [1] 2 19, which is 2^{19} . This is easier to read but **isIntpower** may not return what we expect. For example, **isIntpower(sum(sapply(0:20,function(x){choose(20,x)})))** is 1024².

R Code

```
> library(numbers)  
> primeFactors(sum(sapply(0:n,function(x)  
{choose(n,x)})))  
  
[1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

R Code

```
> library(numbers)  
> isIntpower(sum(sapply(0:n,function(x){choose(n,x)})))  
  
[1] 2 19
```

The numbers package has numerous functions for exploring number theory. For example, **eulersPhi(25)**, **nextPrime(25)**, and **twinPrimes(10,25)** return 20, 29, and the list of twin primes between 10 and 25.

19.2 Set Theory

We begin by noting that there are a few built-in vectors or sets that are useful. They are LETTERS, letters, month.abb, and month.name, for the upper-case letters, lower-case letters, three-letter abbreviations of the months, and the full names of the months, respectively. The colon command a:b, which is the

numbers starting at a and proceeding by unit steps to b, is also useful. For example, letters[10:22] is the 10th through 22nd lower-case letters.

R Code

```
> letters[10:22]
[1] "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
```

We define four sets to be used in our examples. The first is rainbow as the set of the colors of the rainbow. We let U be all upper-case and lower-case letters, the integers 1 through 20, the abbreviated months of the year, and our colors of the rainbow. We define set.A to be the first ten upper-case letters, the 5th through 15th lower-case letters, the numbers 1 through 10, the abbreviated months from Mar to Jul, and the colors yellow and blue. In a similar manner, set.B is constructed. Note: we did not use A and B for our set names because that may create some confusion with the function we will use below to create Venn diagrams.

R Code

```
> rainbow=c("red","orange","yellow","green","blue",
  "indigo","violet")
> U=c(LETTERS,letters,1:20,month.abb,rainbow)
> set.A=c(LETTERS[1:10],letters[5:15],1:10,
  month.abb[3:7],"red","yellow")
> set.B=c(LETTERS[5:20],letters[10:25],5:20,
  month.abb[6:12],"yellow","blue")
```

NA

The functions **union**, **intersect**, and **setdiff** return expected results. See, for example, **intersect(set.A, set.B)**. The function **setequal** returns either TRUE or FALSE if the sets are equal or not. Note that these functions can take only two sets.

R Code

```
> intersect(set.A, set.B)
[1] "E" "F" "G" "H" "I" "J" "j" "k"
[9] "l" "m" "n" "o" "5" "6" "7" "8"
[17] "9" "10" "Jun" "Jul" "yellow"
```

R Code

```
> setequal(set.A, set.B)
```

```
[1] FALSE
```

We may want to know how many elements are in a set without returning all elements. So, for example, the number of elements in the difference of set.A and set.B is returned by **length(setdiff(set.A, set.B))**. If we want to know if an element is in a set, we can use **is.element** or **%in%**. The query **is.element("blue", set.A)** returns FALSE whereas "blue" %in% rainbow returns TRUE.

R Code

```
> length(setdiff(set.A, set.B))
```

```
[1] 17
```

R Code

```
> is.element("blue", set.A)
```

```
[1] FALSE
```

We can nest these functions as desired. We let set.A.c be the **setdiff(U, set.A)**, which is the complement of set.A. We check to see if the union of set.A.c and set.A is equal to U, with **union** nested within **setequal**. TRUE is returned, as it should be.

R Code

```
> set.A.c=setdiff(U, set.A)
> setequal(union(set.A.c, set.A), U)
```

```
[1] TRUE
```

19.2.1 Venn Diagrams

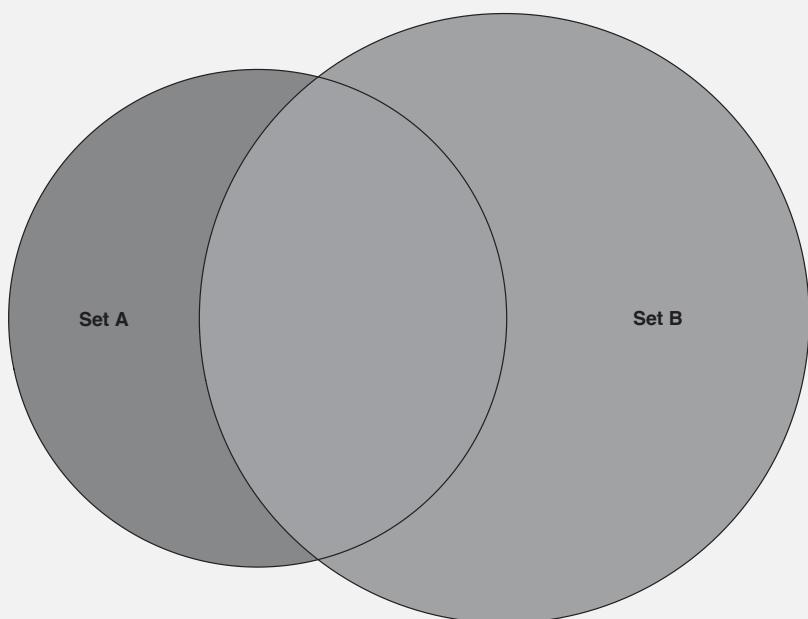
There are a few packages that create Venn diagrams. We will use the **eulerr** package, which creates area-proportional Venn diagrams. The only limitation on the number of sets in the Venn diagram is computing power and the visual usefulness of the diagram. We will provide a simple example with two sets and a more detailed example with three sets.

We begin by loading the **eulerr** package with **library(eulerr)**. The **plot**

function is used to create the Venn diagram with the sets to be graphed organized as a **list** with the **euler** function. We could use A and B for our sets but then the input to **plot** is **plot(euler(list(A=A, B=B))**), which may cause some confusion. Our sets are colored with fill and labelled with labels. The order of colors and names must be consistent with the order of the sets in **list**. If labels is left out the default is to label the diagram with the names given in list. If no labels are desired then use labels=FALSE.

R Code

```
> library(eulerr)
> plot(euler(list(A=set.A,B=set.B)),
  fills=c("firebrick1","cornflowerblue"),
  labels=c("Set A","Set B"))
```



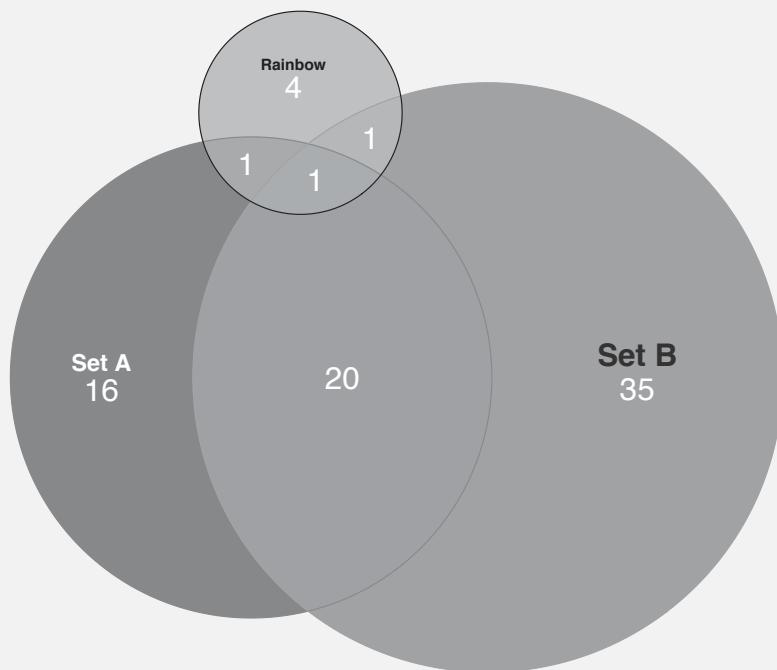
Our next example uses three sets and demonstrates some of the other options available. Within **plot** we have added a third set with C=rainbow. The fills option has a third color listed for our added set. The edges are colored with the edges option. Note how edges is set to a list and within the list we

use col for the color of the edges. Within this list there are other options we can use for the edges including lwd and lty for the line width and the line type. The first two sets have edges colored the same as the fill color, while our third set has a black edge.

The next option is quantities. Within the list TRUE includes the number of elements in the diagram, the numbers are colored white, and the font size is set to 20. As in the last example we set labels, but this time with a list. We first name the labels with labels, set the font size for each label with fontsize, and color each label with col. If the diagram has multiple sets the option to allow for the use of ellipses instead of circles is shape = "ellipse".

R Code

```
> plot(euler(list(A=set.A,B=set.B,C=rainbow)),fills=
  c("firebrick1","cornflowerblue","goldenrod2"),edges=
  list(col=c("firebrick1","cornflowerblue","black")),
  quantities=list(TRUE,col="white",fontsize=20),
  labels=list(labels=c("Set A","Set B","Rainbow"),
  fontsize=c(15,20,10),col=c("white","navy","black")))
```



19.2.2 Power Set, Cartesian Product, and Intervals

In order to do more with sets beyond unions and intersections we introduce the sets package. We load the package with **library(sets)** and create the power set of rainbow with **gset_power**. The output is set to power and **length(power)** returns the number of elements in the set. Entering power will return the entire power set.

R Code

```
> library(sets)
> power=gset_power(rainbow)
> length(power)

[1] 128
```

The sets packages will create the Cartesian product of two sets with **gset_cartesian**. We create the Cartesian product of the set rainbow and the set generated from 1:3 as an example.

R Code

```
> gset_cartesian(rainbow,1:3)

{("red", 1L), ("red", 2L), ("red", 3L), ("blue", 1L),
 ("blue", 2L), ("blue", 3L), ("green", 1L), ("green", 2L),
 ("green", 3L), ("indigo", 1L), ("indigo", 2L), ("indigo",
 3L), ("orange", 1L), ("orange", 2L), ("orange", 3L),
 ("violet", 1L), ("violet", 2L), ("violet", 3L),
 ("yellow", 1L), ("yellow", 2L), ("yellow", 3L)}
```

We can create intervals of real numbers with **reals(a,b,"type")** where type is chosen from **((),[],[])**, or **[]**. For example, **reals(0.5, 1.5, "()")** and **reals(0.1, "[]")** are the intervals $(0.5, 1.5)$ and $(0, 1]$, respectively. We use **&**, **|**, and **-** for the intersection, union, and difference of intervals, respectively. The default for intervals is closed so that **reals(2,4)** returns $[2, 4]$. Logical statements can be asked with intervals as in **reals(1,2) <= reals(1,2,"[]")**.

R Code

```
> reals(0,1,"[]") & reals(0.5,1.5,"()")

(0.5, 1]
```

R Code

```
> reals(1,2) <= reals(1,2,"[]")  
[1] FALSE
```

We can also create intervals of integers and natural numbers. For example, **integers(-2,10.5)** are the integers from -2 to 10 , while **naturals(1.5,11)** are the natural numbers from 2 to 11 .

19.2.3 A Cantor Set Example

We use the sets package to create a graph of the first nine iterations in the construction of the Cantor set. We begin by loading the package sets (if it hasn't already been loaded). We let $n=8$ which will produce nine iterations of the Cantor construction, since we begin counting at 0 . We set the graph frame for four plots in a 2×2 grid with **par(mfrow=c(2,2))**. Our code will create four graphs where each one zooms in on the picture by decreasing the x -axis from 0 to 1 down to 1 to 0.001 with two steps in between. Our margins are set to 2 lines on all four sides with **par(mar=c(2,2,2,2))**.

We begin the first for loop, which has k assuming the values $1, 0.1, 0.01$, and 0.001 . We define the set A as $[0, 1]$ and create an empty graph with **plot**. We use $(0, 0)$ as our point to plot as we can't leave the first two inputs of **plot** empty. The plot type is "n" for none so the point isn't plotted. We set the x -axis to range from 0 to k , which will zoom in on the graph as k changes. The y -axis ranges from 0 to $n+1$. The axes are left blank with $\text{axes}=FALSE$, while the labels for the axes are also blank with empty quotes for xlab and ylab . We plot the interval A with **segments**, with the two points $(\min(A), n+1)$ and $(\max(A), n+1)$. We create tick marks and labels for our x -axis with **axis**. The first option, 1 , is for the bottom axis, whereas $2, 3$, and 4 are for the left, top, and right, respectively. The **seq** function creates a sequence of values from 0 to 1 in steps of $k/10$. This sequence is used for the location of the labels with $\text{at}=$ and the names of the labels with $\text{label}=$.

R Code

```
> n=8  
> par(mfrow=c(2,2))  
> par(mar=c(2,2,2,2))  
> for (k in c(1,.1,.01,.001)){  
+ A=reals(0,1,"[]")  
+ plot(0,0,type="n",xlim=c(0,k),ylim=c(0,n+1),  
+ axes=FALSE,ylab="",xlab="")  
+ segments(min(A),n+1,max(A),n+1,lwd=2,col="blue")  
+ axis(1, at=seq(0,1,k/10),label=seq(0,1,k/10))
```

```

+ for(i in 0:n) {
+ A.union=interval()
+ for(j in 1:(2^i)){
+ B=reals(min(A[j])+(max(A[j])-min(A[j])/3,min(A[j])+
2*(max(A[j])-min(A[j]))/3,"()")
+ A.temp=interval_complement(B, A[j])
+ segments(min(A.temp[1]),n-i,max(A.temp[1]),n-i,lwd=2,
col="blue")
+ segments(min(A.temp[2]),n-i,max(A.temp[2]),n-i,lwd=2,
col="red")
+ A.union=A.union|A.temp}
+ A=A.union}}

```



The next for loop is nested within the for loop on k and has i ranging from 0 to n . We initialize $A.union$ as an empty **interval**. We begin another for loop where j ranges from 1 to 2^i . For the first time through the loop on j , the

interval B is the middle third of the interval $[0, 1]$ because the interval A is $[0, 1]$. The first time through this loop j ranges from 1 to $2^0 = 1$ and A[1] is still $[0, 1]$. Hence **min(A[j])=0**, **max(A[j])=1**, and B is the open interval which is the middle third of $[0, 1]$. The interval A.temp is the complement of B relative to A[j]. Again, the first time through gives A.temp= $[0, 1/3] \cup [2/3, 1]$. We plot the two parts of A.temp with **segments**, with the first one colored blue and the second colored red.

We let A.union be the union of itself and A.temp, and end the loop on j. Outside the loop on j we set A to A.union. The second time through the loop on i, when $i=1$, A is now $[0, 1/3] \cup [2/3, 1]$ and the loop on j removes the middle third from each of those intervals and plots the four remaining intervals.

What is the length of the set A once this process is complete? We use **length(A)** to determine the number of disjoint sets of A. The use of **sapply** will return a vector with the length of each piece of A by evaluating the function for each value from 1 to **length(A)**. This vector is then summed with **sum**.

R Code

```
> sum(sapply(1:length(A),function(x)
+ {max(A[x])-min(A[x])}))
```

[1] 0.02601229

19.3 Graph Theory

This section uses the package igraph. We begin this section by constructing some standard graphs while illustrating some of the options for displaying graphs. We then construct our own graph. After that we demonstrate the use of random graphs and finish off by calculating some graph invariants. In general, the examples focus on graphs that are not directed. There are options for directed graphs.

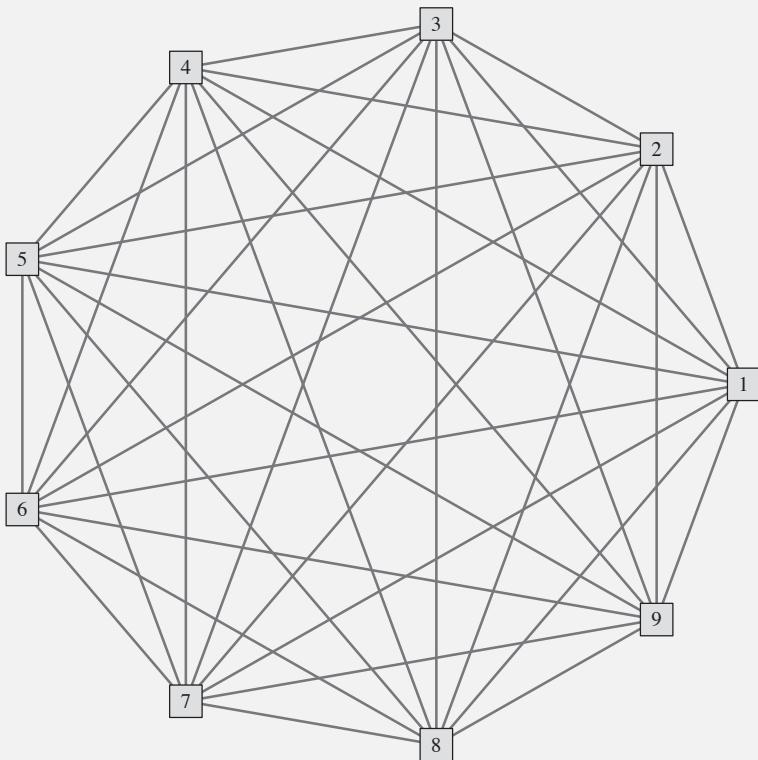
19.3.1 Creating and Displaying Graphs

We begin by loading the igraph package. We use the **make_full_graph** function to produce complete.9, the complete graph on nine vertices. At this point **plot(complete.9)** will display the graph, but it can be improved. In particular, it puts a vertex in the center. Try it. We first set the margins to one line on all sides with **par(mar=c(1,1,1,1))**. We use **plot** to display complete.9 and take advantage of a number of options. We set layout to layout_in_circle, which locates all vertices on a circle. We set the shape, color, size, and border color of

the vertices with vertex.shape, vertex.color, vertex.size, and vertex.frame.color. The vertex number is colored with vertex.label.color. The color and width of the edges are set with edge.color and edge.width. The options for the shape of vertices are circle, square, csquare, rectangle, crectangle, vrectangle, pie, sphere, and none, with size2 options for shapes that require two values such as a rectangle.

R Code

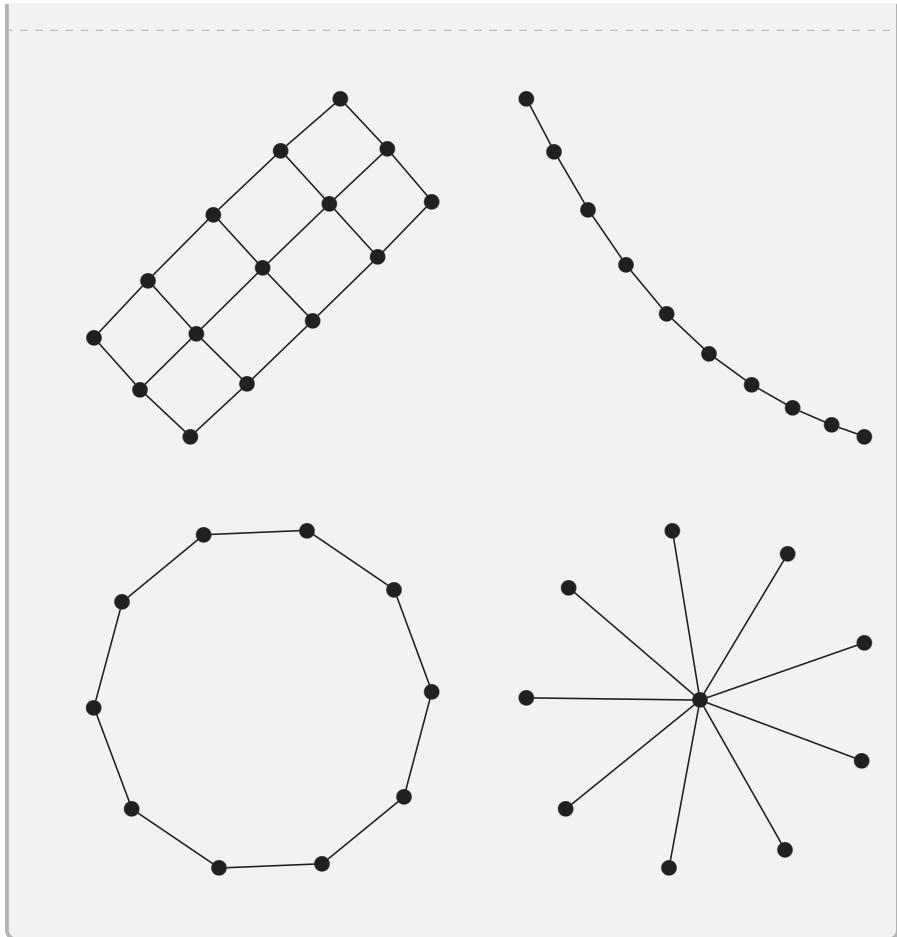
```
> library(igraph)
> complete.9=make_full_graph(9)
> par(mar=c(1,1,1,1))
> plot(complete.9,layout=layout_in_circle,vertex.shape=
"square",vertex.color="lightgoldenrod",vertex.size=9,
vertex.frame.color="navyblue",vertex.label.color=
"navyblue",edge.color="purple",edge.width=2)
```



The next example defines and displays four graphs; a lattice, a path, a ring, and a star. Again we set the margins to one line on all sides and then create a 2×2 grid for our graphs with **par(mfrow=c(2,2))**. Graph lattice.3.5 is defined as a 3×5 lattice with **make_lattice**. We display lattice.3.5 with **plot** setting the vertices and edges to black, the vertex size and shape to 9 and circle, and removing the frame color and labels for the vertices. The graph path.10 is a path of length 10 created with **make_ring** by setting circular=FALSE. Note that directed=False is the default for **make_ring**. We display path.10 in the same way as lattice.3.5. We create a ring, ring.10, by using **make_ring** with circular=TRUE and then display the graph. The last graph in this set is a star, star.10, generated by **make_star** with mode set to undirected. The other options for mode are in, out, and mutual. The number of the vertex in the center is set with center.

R Code

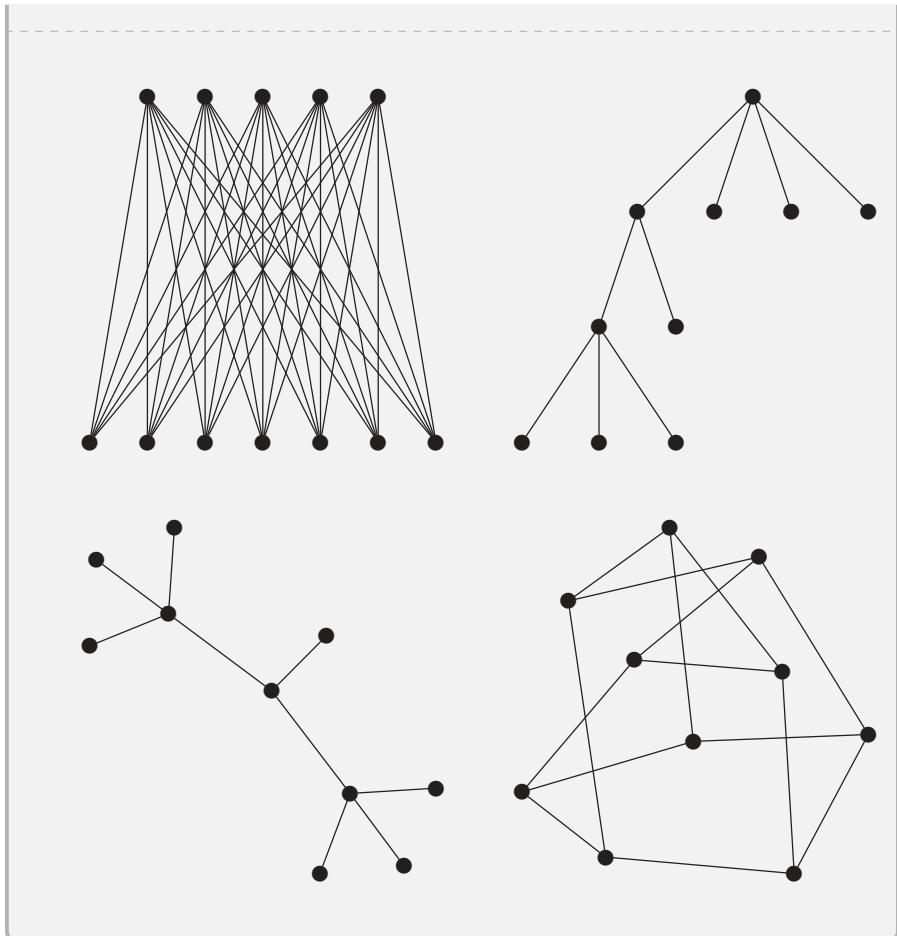
```
> par(mar=c(1,1,1,1))
> par(mfrow=c(2,2))
> lattice.3.5=make_lattice(c(3,5))
> plot(lattice.3.5,vertex.color="black",edge.color=
"black",vertex.size=9,vertex.frame.color=NA,
vertex.shape="circle",vertex.label=NA)
> path.10=make_ring(10,circular=FALSE)
> plot(path.10,vertex.color="black",edge.color="black",
vertex.size=9,vertex.frame.color=NA,
vertex.shape="circle",vertex.label=NA)
> ring.10=make_ring(10,circular=TRUE)
> plot(ring.10,vertex.color="black",edge.color="black",
vertex.size=9,vertex.frame.color=NA,
vertex.shape="circle",vertex.label=NA)
> star.10=make_star(10,mode="undirected",center=1)
> plot(star.10,vertex.color="black",edge.color="black",
vertex.size=9,vertex.frame.color=NA,
vertex.shape="circle",vertex.label=NA)
```



We illustrate three more graphs in the next example: a full bipartite graph, a tree (displayed in two ways), and the Petersen graph. Again we set the margins to one line each and create a 2×2 frame for our plots. The graph `bipartite.5.7` is a full bipartite graph, `make_full_bipartite_graph`, on 5 and 7 vertices with `directed=FALSE`. We display `bipartite.5.7` with the same setting as above but with `layout` set to `layout_as_bipartite`. Graph `tree.10.3` is a tree with 10 vertices and the number of children for each vertex, except leaves, is set to 3. The first plot has `layout` set to `layout_as_tree` while the second removes the `layout` options to allow for the default layout. The package has a number of notable graphs predefined, such as the Petersen graph, which we set to `petersen`. Unfortunately, even with `layout` set to `layout_nicely` we don't get the display of the Petersen graph we would expect.

R Code

```
> par(mar=c(1,1,1,1))
> par(mfrow=c(2,2))
> bipartite.5.7=make_full_bipartite_graph(5,7,directed=
FALSE)
> plot(bipartite.5.7,layout=layout_as_bipartite,
vertex.color="black",edge.color="black",vertex.size=9,
vertex.frame.color=NA,vertex.shape="circle",
vertex.label=NA)
> tree.10.3=make_tree(10,3,mode="undirected")
> plot(tree.10.3,layout=layout_as_tree,vertex.color=
"black",edge.color="black",vertex.size=9,
vertex.frame.color=NA,vertex.shape="circle",
vertex.label=NA)
> plot(tree.10.3,vertex.color="black",edge.color=
"black",vertex.size=9,vertex.frame.color=NA,
vertex.shape="circle",vertex.label=NA)
> petersen=make_graph("Petersen")
> plot(petersen,layout=layout_nicely,vertex.color=
"black",edge.color="black",vertex.size=9,
vertex.frame.color=NA,vertex.shape="circle",
vertex.label=NA)
```

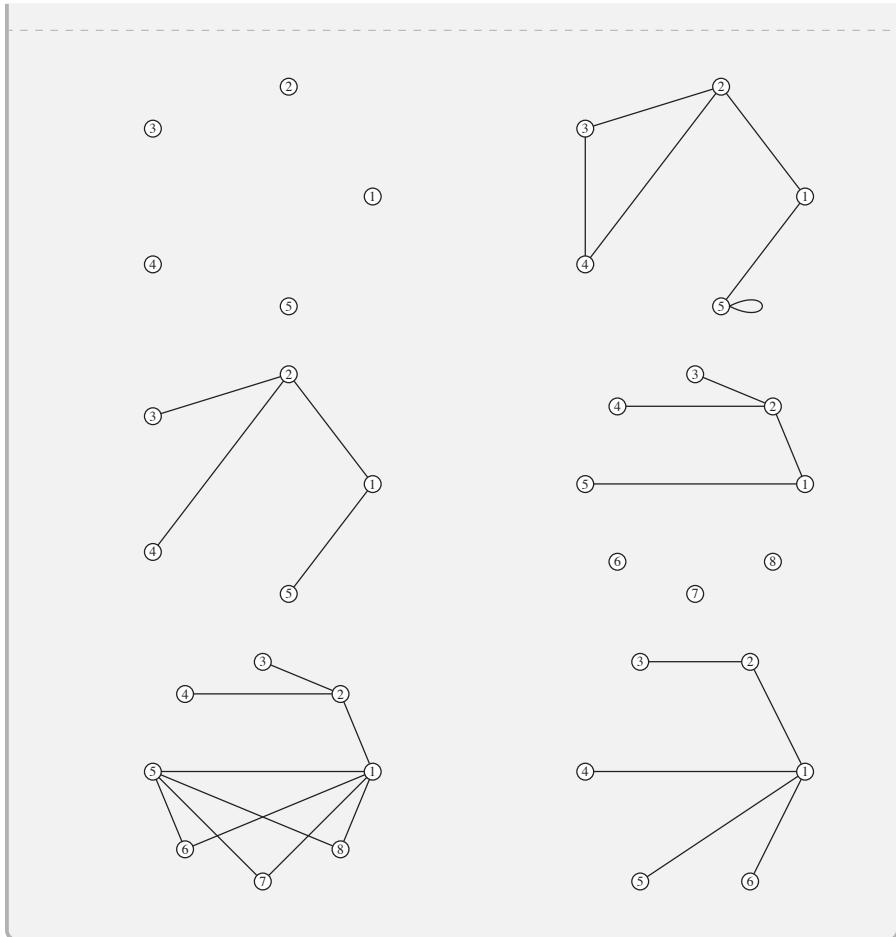


Beyond built-in graphs we can also create or make changes to a graph, which we illustrate in the next example. We begin by defining the graph `g` to be an undirected graph with 5 vertices and no edges with `make_empty_graph`. In order to follow along with the construction and changes to the graph we will display our results at each step. To that end we set the margins of our graph frame to one line on all sides and set up a 3×2 frame for six plots. Given that we will create six plots we define the function `plot.graph`, with input graph `g`, that will plot `g` setting layout to `layout_in_circle`. The vertices are colored white and the edges black. The vertex size is 15 with a black frame. The shape of the vertices is a circle and the label color is black. We next add edges to the graph `g` with `add_edges`. The first option of `add_edges` is the graph to which we add edges. The second option is the list of edges to be added where the vertex numbers are taken in pairs. We added six edges to graph `g` and we set the new graph back to `g`. The new graph `g` is displayed at top right with `graph.plot(g)`.

We now remove two edges from g with **delete.edges**. The first argument is the graph to which edges are removed. The second argument is the list of edges to be removed, where edges are denoted in quotes with a | between the vertex numbers. We again set the new graph to g and plot it. If we would like to add additional vertices to the graph we use **add.vertices**. Here we add three vertices to the graph g , continue to call the graph g , and display the graph. Note how the position of the vertices changes but that vertex 1 is still at 3 o'clock and the vertices are arranged in a circle as requested by the layout in **plot** within the definition of **plot.graph**. We next add six new edges to our graph g and again display the graph. In our last step we delete vertices 5 and 3, which also removes their associated edges, and display this graph.

R Code

```
> g = make_empty_graph(5, directed=FALSE)
> par(mar=c(1,1,1,1))
> par(mfrow=c(3,2))
> plot.graph=function(g){plot(g,layout=layout_in_circle,
+ vertex.color="white",edge.color="black",vertex.size=15,
+ vertex.frame.color="black",vertex.shape="circle",
+ vertex.label.color="black")}
> plot.graph(g)
> g=add_edges(g,c(1,2,1,5,2,3,3,4,4,2,5,5))
> plot.graph(g)
> g=delete_edges(g,c("5|5","3|4"))
> plot.graph(g)
> g=add_vertices(g,3)
> plot.graph(g)
> g=add_edges(g,c(6,1,7,1,8,1,6,5,7,5,8,5))
> plot.graph(g)
> g=delete_vertices(g, c("5","3"))
> plot.graph(g)
```



If at any point we want to list the vertices and edges of a graph we can use **V(g)** and **E(g)**. Our final graph g has 6 vertices and edges from 1 to 2, 2 to 3, 1 to 4, 1 to 5, and 1 to 6. Note that d11b982 will be different.

R Code

```
> V(g)
+ 6/6 vertices, from d11b982:
[1] 1 2 3 4 5 6
```

R Code

```
> E(g)
```

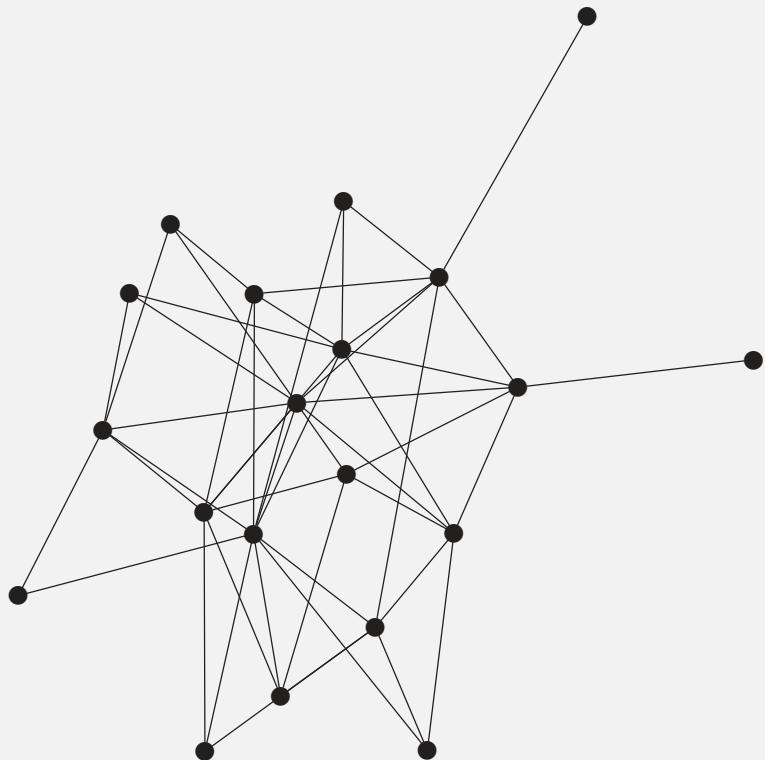
```
+ 5/5 edges from d11b982:  
[1] 1--2 2--3 1--4 1--5 1--6
```

19.3.2 Random Graphs

The igraph package can generate random graphs and we illustrate two such ways this is done. The first is to start with n vertices and add each edge with probability p , **sample_gnp**. The other starts with n vertices and a fixed number of edges m . The set of m edges will be selected uniformly from all possible sets of m edges, **sample_gnm**. For example, we set graph.p to be the random graph on 20 vertices with probability 0.25 of including each edge, with options for directed and loops set to FALSE. Note that results will differ. We set the margins to one line on all sides and display the graph with **plot**, while using the options explained above.

R Code

```
> graph.p=sample_gnp(20,0.25,directed=FALSE,loops=FALSE)  
> par(mar=c(1,1,1,1))  
> plot(graph.p,layout=layout_nicely,vertex.color=  
"black",edge.color="black",vertex.size=5,  
vertex.frame.color=NA,vertex.shape="circle",  
vertex.label=NA)
```

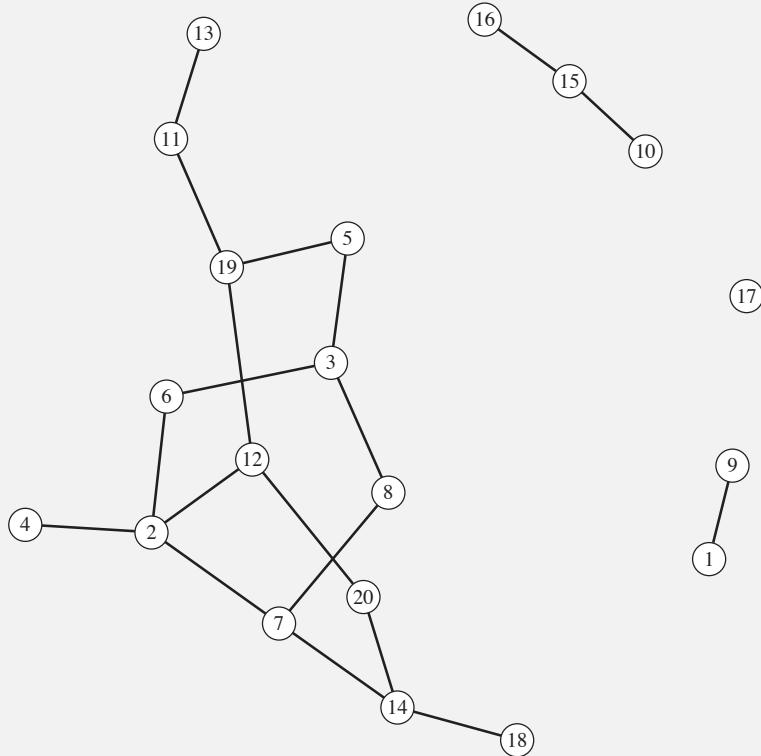


Similarly, we set `graph.m` to be the random graph on 20 vertices and the size of the edge set 1/10 of all possible edges, as there are $\binom{20}{2}$ or **choose(20,2)** possible edges. Again, we set directed and loops to FALSE, the margins to one line on all sides, and display the graph with **plot**. In this case we remove `vertex.label=NA` so that the vertices are numbered. We also color the vertices white, color the vertices frame black, and increase the vertex size to 9. We will use this in the next example.

R Code

```
> graph.m=sample_gnm(20,choose(20,2)/10,directed=FALSE,
loops=FALSE)
> par(mar=c(1,1,1,1))
> plot(graph.m,layout=layout_nicely,vertex.color=
"white",edge.color="black",vertex.size=9,
```

```
vertex.frame.color="black",vertex.label.color=
"black",edge.width=2)
```



This is an opportune time to point out the graph invariant function **components**, which we will use in a simulation in a moment. Reading the output of **components(graph.m)** from the bottom to the top, there are four connected components, the size of each component is listed, and there is a list of which vertices are in which component. For example, vertices 1 and 9 are in component 1 as there is a 1 in the first and ninth location in the membership list. We use the **components** function and random graphs in estimating the probability that a graph with edges added with probability p is not connected.

R Code

```
> components(graph.m)
```

```
$`membership'
[1] 1 2 2 2 2 2 2 2 2 1 3 2 2 2 2 3 3 4 2 2 2

$csizes
[1] 2 14 3 1

$no
[1] 4
```

We set reps to 10000, initialize our counter n to 0, the number of vertices n.vertices to 20, and the probability an edge is included p to 0.2. Our for loop on i ranges from 1 to reps. We set graph.p to be a random graph on n.vertices vertices, probability of including an edge p, with undirected edges, and no loops. The if statement checks to see if the number of components of graph.p is greater than 1, **components(graph.p)\$no**, and if so increase the counter by 1, n=n+1. The results are printed out with **paste** which concatenates characters and variables. In this case there are no spaces between pieces by setting sep to empty quotes and adding spaces as desired inside the quotes of characters. It appears that the graphs are not connected about 26% of the time.

R Code

```
> reps=10000
> n=0
> p=0.2
> n.vertices=20
> for (i in 1:reps){
+ graph.p=sample_gnp(n.vertices,p,directed=FALSE,
loops=FALSE)
+ if (components(graph.p)$no > 1){n=n+1} }
> paste("Probability of disconnected graph with ",
n.vertices," vertices and probability of including
an edge p=", p, " is ", n/reps,".", sep="")
```

[1] "Probability of disconnected graph with 20 vertices
and probability of including an edge p=0.2 is 0.2564."

19.3.3 Some Graph Invariants

There are a number of functions that calculate graph invariants in the igraph package. We used one, **components**, above. In the next few examples we will illustrate some other invariant functions. We let path.6 be a path on six vertices

using **make_ring** and setting circular=FALSE. The graph has four articulation points, the removal of which disconnects the graph; they are vertices 5, 4, 3, and 2.

R Code

```
> path.6=make_ring(6,circular=FALSE)
> articulation_points(path.6)

+ 4/6 vertices, from ff180b1:
[1] 5 4 3 2
```

The degree sequences, the list of degrees or number of adjacent edges of each vertex, is given by **degree(path.6)**. It may be more convenient for this output to be in the form of a table, which is achieved by **table(degree(path.6))**. Note, for example, that **table(degree(path.6))[1]** will return the number of vertices with degree 1.

R Code

```
> degree(path.6)

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

R Code

```
> table(degree(path.6))

1 2
2 4
```

The distribution of degrees is returned with **degree_distribution(path.6)** (0.0000000 0.3333333 0.6666667 in this case, with the first value associated with degree 0 vertices), while the minimum number of vertices whose removal disconnects the graph is given by **min_cut(path.6)** (1 in this case). The diameter of a graph is the greatest distance between two vertices and can be calculated with **diameter(path.6)** (5 in this case). The eccentricity of a vertex is its shortest path distance from the farthest other vertex in the graph. The next example shows the eccentricity of each vertex in path.6. We note that the radius is the minimum eccentricity, which is given by **radius(path.6)**. A matrix of distances between each pair of vertices is given by **distances(path.6)**.

R Code

```
> eccentricity(path.6)

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

R Code

```
> distances(path.6)

 [,1] [,2] [,3] [,4] [,5] [,6]
[1,] 0 1 2 3 4 5
[2,] 1 0 1 2 3 4
[3,] 2 1 0 1 2 3
[4,] 3 2 1 0 1 2
[5,] 4 3 2 1 0 1
[6,] 5 4 3 2 1 0
```

For our next set of examples, we define `complete.5` to be the complete graph on 5 vertices. The clique number is the size of the largest complete subgraph and calculated by `clique_num(complete.5)`. The number of maximum cliques is given by `count_max_cliques(complete.5)` (1 in this case).

R Code

```
> complete.5=make_full_graph(5)
> clique_num(complete.5)
```

```
[1] 5
```

We can use `cliques(complete.5,min=3,max=4)` to list all cliques of size from 3 to 4, but for this example that isn't necessary since we can count the number of them by using `length`. For this example, this is simply $\binom{5}{3} + \binom{5}{4} = 15$.

R Code

```
> length(cliques(complete.5, min = 3, max = 4))
```

```
[1] 15
```

The adjacency matrix is given by `as_adjacency_matrix` and the type can also be set to upper or lower.

R Code

```
> as_adjacency_matrix(complete.5,type="both")
```

```
5 x 5 sparse Matrix of class "dgCMatrix"
```

```
[1,] . 1 1 1 1
[2,] 1 . 1 1 1
[3,] 1 1 . 1 1
```

```
[4,] 1 1 1 . 1  
[5,] 1 1 1 1 .
```

For our last example, we can list all simple paths from one vertex to another with **all_simple_paths**. We show only the first three paths. If all we want is the number of them we can use **length(all_simple_paths(complete.5,1,5))**, which will return 16.

R Code

```
> all_simple_paths(complete.5, 1, 5)  
  
[[1]]  
+ 5/5 vertices, from e362b79:  
[1] 1 2 3 4 5  
  
[[2]]  
+ 4/5 vertices, from e362b79:  
[1] 1 2 3 5  
  
[[3]]  
+ 5/5 vertices, from e362b79:  
[1] 1 2 4 3 5  
  
etc.
```

Code Review for the Binomial Coefficients Section

choose(n,k) returns $\binom{n}{k}$.

eulersPhi(n) returns the number of positive integers up to n that are relatively prime to n . This is Euler's phi function. This requires the numbers package.

factorial(n) returns $n!$.

isIntpower(n) returns a and b where $a^b = n$. This requires the numbers package.

lchoose(n,k) returns $\ln(\binom{n}{k})$.

lfactorial(n) returns $\ln(n!)$.

nextPrime(n) returns the first prime number greater than n . This requires the numbers package.

primeFactors(n) returns the prime factors of n . This requires the numbers package.

twinPrimes(a,b) returns the list of twin primes between a and b .
This requires the numbers package.

Code Review for the Set Theory Section

gset_cartesian(A,B) returns the Cartesian product of the sets A and B . This requires the sets package.

gset_power(A) returns the power set of A . This requires the sets package.

integers(a,b) returns the interval of integers contained in $[a, b]$. This requires the sets package.

intersect(A,B) returns the intersection of sets A and B .

is.element(a,A) returns TRUE if $a \in A$, and otherwise returns FALSE.

letters is the vector of lower-case letters.

LETTERS is the vector of upper-case letters.

month.abb is the vector of three letter abbreviation of the months.

month.name is the vector of months.

naturals(a,b) returns the interval of natural numbers contained in $[a, b]$. This requires the sets package.

reals($x,y,type$) returns the real interval from x to y with options for type (), (), [], and []. Use &, |, and – for the intersection, union, and difference of intervals. This requires the sets package.

setdiff(A,B) returns the set difference of sets A and B .

setequal(A,B) returns TRUE if the sets are equal, and otherwise returns FALSE.

union(A,B) returns the union of sets A and B .

Code Review for the Graph Theory Section (requires the igraph package)

add.edges($G,c(a_1,a_2,\dots,a_{2n})$) returns the graph G with the additional edges $a_1a_2, \dots, a_{2n-1}a_{2n}$.

add.vertices(G,i) returns the graph G with i new vertices.

all_simple_paths(G, a, b) returns all simple paths in G from vertex a to vertex b .

articulation_points(G) returns the list of articulation points of G .

as_adjacency_matrix($G, \text{type}=$) returns the adjacency matrix of G where type can be set to both, upper, or lower.

clique_num(G) returns clique number of G .

components(G) returns the number of connected components, their size, and which vertices are in which component of G .

degree(G) returns a vector of the degrees of the vertices of G .

delete_edges($G, c("a_1|a_2", \dots, "a_{n-1}|a_n")$) returns the graph G with the edges $a_1a_2, \dots, a_{n-1}a_n$ deleted.

delete_vertices($G, c("a_1", \dots, "a_n")$) returns the graph G with the vertices a_1, \dots, a_n deleted.

distances(G) returns a matrix giving the distance between of G .

eccentricity(G) returns a vector of the eccentricities of the vertices of G .

make_full_bipartite_graph($i, j, \text{directed}=$) returns a complete bipartite graph with vertex sets of size i and j , where directed can be set to TRUE or FALSE.

make_empty_graph($n, \text{directed}=$) returns a graph with n vertices and no edges, where directed can be set to TRUE or FALSE.

make_full_graph(n) returns a complete graph on n vertices.

make_lattice($c(i, j)$) returns an $a \times b$ lattice graph.

make_ring($n, \text{circular}=$) returns a path on n vertices if circular is FALSE, and otherwise returns a ring.

make_star($n, \text{mode}=, \text{center}=a$) returns a star graph with n vertices with the vertex in the center labelled a .

make_tree($n, i, \text{mode}=\text{"undirected"}$) returns a tree with n vertices and the number of children for each vertex, except leaves, set to i , where mode can be set to out, in, or undirected.

sample_gnm($n, m, \text{directed}=, \text{loops}=$) returns a random graph with n vertices and m edges selected uniformly from all possible edge sets of size m , where directed, and loops can be set to TRUE or FALSE.

`sample_gnp(n, p , directed=, loops=)` returns a random graph with n vertices and each edge added with probability p , where directed, and loops can be set to TRUE or FALSE.

19.4 Exercises

1. Create 12 lines of Pascal's triangle with the values colored mod 3.
2. Sum every other value in the 11th row of Pascal's triangle. Hint `seq()`.
3. Create a Venn diagram with sets $1, \dots, 10$; $8, \dots, 17$; $15, \dots, 24$; and $22, \dots, 28$, $1, 2, 3$. Color each set differently, with distinct colors for their names, and different types of edges other than just changing color.
4. Illustrate the difference between $\cap_{n=1}^{\infty} (0, 1/n)$ and $\cap_{n=1}^{\infty} [0, 1/n)$
5. Construct the Petersen graph by using `make_empty_graph` and `add_edges`.
6. Use `sapply` instead of the for loop in the example estimating the probability that a graph with edges added with probability p is not connected.
7. With 20 vertices what value of p yields a 50/50 chance of a connected graph? Answer the same question for m .
8. Construct a graph that requires the removal of two vertices to disconnect the graph; verify the result.
9. Construct a graph with one clique of size three and four of size 2; verify the result.

A

Loops, Vectors, and Apply

As with any language there are multiple ways to accomplish a computation in R. In a mathematics class the choice of how to solve a computational problem should take into consideration mathematical learning. The goal of this book is to enhance mathematics with the computational and graphical power of R, so code choices were made with this in mind. Here we present three examples to illustrate the point. In general, the choice is often between a for loop and the use of an apply function. We will not get into a debate about whether or not a for loop is archaic or inefficient.

How would we find the sum of the first 100 terms of $1/n^2$? From a mathematical perspective we think of adding the terms one at a time, which leads us to a for loop. We begin by initializing the variable total to 0. The for loop has i range from 1 to 100, where the colon command a:b produces the numbers starting at a, increasing by steps of 1, and stopping when it exceeds b. The content of a for loop is enclosed with braces. The one line in our loop adds $1/i^2$ to total and stores it as total; in effect this increases total, which represents the partial sum, by $1/i^2$. The value of total is returned in the last line. The use of a for loop closely aligns with the indexing of a summation.

R Code

```
> total=0  
> for (i in 1:100){  
+ total=total+ 1/i^2 }  
> total
```

```
[1] 1.634984
```

Another method to sum of the first 100 terms of $1/n^2$ takes advantage of how functions act on vectors. In the first line we define the function **f** as $1/x^2$. Starting from the inside of the second line **f(1:100)** evaluates the numbers from 1 to 100 and returns the vector $(1, 1/4, 1/9, \dots, 1/100^2)$. The function **sum** returns the sum of this vector. The four lines of code required by the for loop are reduced to two lines and uses a function, but the idea of indexing on i is lost.

R Code

```
> f=function(x){1/x^2}
> sum(f(1:100))

[1] 1.634984
```

We can reduce the length of the code to one line using the **sapply** function. In short, **sapply(v,f)** returns the vector $(f(v_1), f(v_2), \dots, f(v_n))$ where $v = (v_1, v_2, \dots, v_n)$. The vector returned by **sapply**, $(1, 1/4, 1/9, \dots, 1/100^2)$, is summed with **sum**.

R Code

```
> sum(sapply(1:100,function(x){1/x^2}))

[1] 1.634984
```

The next example is similar: we want to compute the first 100 partial sums of $\sum_n^\infty 1/n^2$. The first method is by way of a for loop, which begins by initializing vector1 as a numeric vector of length 100. At this point vector1 contains all zeros. We set the first value, vector1[1], to 1. The for loop has i range from 2 to 100. The one line body of the loop assigns the ith value of the vector to the i-1 value plus $1/i^2$. The last line of code returns the first and last three values of the vector.

R Code

```
> vector1=numeric(100)
> vector1[1]=1
> for (i in 2:100){
+ vector1[i]=vector1[i-1]+ 1/i^2}
> vector1[c(1:3,98:100)]

[1] 1.00000 1.25000 1.36111 1.63478 1.63488 1.63498
```

As in the first example the second method takes advantage of R's ability to evaluate a vector by a function. The function **f** is defined to be $1/x^2$. As above **f(1:100)** returns $(1, 1/4, 1/9, \dots, 1/100^2)$, which is the input to **cumsum**. The cumulative sum of $(1, 1/4, 1/9, \dots, 1/100^2)$ is returned and stored as vector2. Again, the last line of code returns the first and last three values of the vector.

R Code

```
> f=function(x){1/x^2}
> vector2=cumsum(f(1:100))
> vector2[c(1:3,98:100)]
```

```
[1] 1.00000 1.25000 1.36111 1.63478 1.63488 1.63498
```

The third method is almost exactly the same as the third method in the first example. The difference is simply that the output of **sapply** is the input to **cumsum** instead of **sum**.

R Code

```
> vector3=cumsum(sapply(1:100,function(x){1/x^2}))
> vector3[c(1:3,98:100)]
```

```
[1] 1.000000 1.250000 1.361111 1.634782 1.634884 1.634984
```

For our last example, suppose we are given a 10-sided die numbered one through 10. If we would like to know if even numbers occur half of the time we would need to count the number of even values. We run a small experiment where we roll the die five times, record the number, and repeat this 10 times. We first use **set.seed** so that the random sample will be the same. Matrix A is created by using the **sample** function, where we sample the number from 1 to 10, 50 times, and with replacement. The 50 values fill the matrix with 10 columns. Our goal now is to count the number of even values in each column of the matrix. Method one will use nested for loops.

R Code

```
> set.seed(42)
> A=matrix(sample(1:10,50,replace=TRUE), ncol=10)
> A
```

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

We begin by initializing the numeric vector `count1` of length 10. This vector will store the results of the number of even values in each column. The outer for loop has `col.j` ranging from 1 to 10, for the columns of A. The inner for loop then lets `row.i` range from 1 to 5, for the rows of A. The **ifelse** function checks whether the `A[row.i,col.j]` element is even by checking whether it is equal to 0 mod 2; `%%2` is mod 2 and two equal signs is logical. If true, the **ifelse** returns 1 and 0 otherwise. The line of code inside the loops adds 1 to `count[col.j]` if `A[row.i,col.j]` is even. The results of `count1` are returned.

R Code

```
> count1=numeric(10)
> for(col.j in 1:10){
+ for(row.i in 1:5) {
+ count1[col.j]=count1[col.j]
+ ifelse(A[row.i,col.j]%%2==0,1,0)}}
> count1
```

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

The nested for loops can be reduced to one line of code by using the **apply** function. In this example, the first variable in **apply** is our matrix. The second indicates that the columns of A will be used for the input of the function. A 1 would process the rows. Each column of A is a vector passed to the **ifelse** function. We evaluate mod 2 of each element of the vector and return a 1 if it is 0 and 0 otherwise. The **sum** function then sums the vector and the function returns this sum, which is the number of even values in a column of A. The **apply** function does this for each column of A and the results are assigned to count2. The results are the same as above, as they should be.

R Code

```
> count2=apply(A,2,function(x)
+{sum(ifelse(x%%2==0,1,0))})
> count2
```

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

The three examples here illustrate choices for coding. If coding is being used to support mathematical learning then, in these examples, the for loop may be more appropriate than using higher level functions such as **sapply** or **apply**.

B

Arctic Sea Ice Data

The Arctic sea ice extent in million square kilometers is from the National Snow and Ice Data Center [10]. A direct link to the monthly average extent is <ftp://sidads.colorado.edu/DATASETS/NOAA/G02135/north/monthly/data/>.

year	Years after 1970	March Extent	June Extent	September Extent
1979	9	16.34	12.53	7.05
1980	10	16.04	12.20	7.67
1981	11	15.63	12.43	7.14
1982	12	16.04	12.48	7.30
1983	13	16.09	12.30	7.39
1984	14	15.58	12.15	6.81
1985	15	15.89	12.22	6.70
1986	16	15.91	11.98	7.41
1987	17	15.82	12.49	7.28
1988	18	15.96	11.94	7.37
1989	19	15.42	12.24	7.01
1990	20	15.87	11.64	6.14
1991	21	15.42	12.11	6.47
1992	22	15.48	12.15	7.47
1993	23	15.81	11.87	6.40
1994	24	15.55	12.02	7.14
1995	25	15.26	11.44	6.08
1996	26	15.12	12.08	7.58
1997	27	15.47	11.74	6.69
1998	28	15.60	11.71	6.54
1999	29	15.40	11.78	6.12
2000	30	15.22	11.67	6.25
2001	31	15.52	11.46	6.73
2002	32	15.35	11.58	5.83
2003	33	15.48	11.60	6.12
2004	34	14.99	11.45	5.98
2005	35	14.69	11.16	5.50
2006	36	14.42	10.92	5.86
2007	37	14.54	11.22	4.27
2008	38	15.18	11.21	4.69
2009	39	14.98	11.32	5.26
2010	40	15.14	10.59	4.87
2011	41	14.55	10.75	4.56
2012	42	15.20	10.67	3.57
2013	43	15.03	11.36	5.21
2014	44	14.76	11.03	5.22
2015	45	14.37	10.88	4.62
2016	46	14.40	10.41	4.53
2017	47	14.29	10.76	4.82
2018	48	14.30	10.71	4.71



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Bibliography

- [1] Daniel Adler and Duncan Murdoch. *rgl: 3D Visualization Using OpenGL*. URL <https://cran.r-project.org/web/packages/rgl/rgl.pdf>, R package version 0.99.16, 2018.
- [2] Alan T. Arnholt and Ben Evans. *BSDA: Basic Statistics and Data Analysis*. URL <https://cran.r-project.org/web/packages/BSDA/BSDA.pdf>, R package version 1.2.0, 2017.
- [3] Hans W. Borchers. *pracma: Practical Numerical Math Functions*. URL <https://cran.r-project.org/web/packages/pracma/pracma.pdf>, R package version 2.2.2, 2018.
- [4] Hans Werner Borchers. *numbers: Number-Theoretic Functions*. URL <https://cran.r-project.org/web/packages/numbers/numbers.pdf>, R package version 0.7-1, 2018.
- [5] Centers for Disease Control and Prevention. *National Center for Health Statistics*. URL <https://www.cdc.gov/nchs/fastats/bodymeasurements.htm>, Accessed December 8, 2018.
- [6] Andrew Clausen and Serguei Sokol. *Deriv: Symbolic Differentiation*. URL <https://cran.r-project.org/web/packages/Deriv/Deriv.pdf>, R package version 3.8.5, 2018.
- [7] Gábor Csárdi, et. al. *igraph: Network Analysis and Visualization*. URL <https://cran.r-project.org/web/packages/igraph/igraph.pdf>, R package version 1.2.2, 2018.
- [8] Brian Dennis. *The R Student Companion*. Boca Raton, FL: CRC Press, 2013.
- [9] Brian S. Everitt and Torsten Hothorn. *A Handbook of Statistical Analyses using R*, 2nd Ed. Boca Raton, FL: CRC Press, 2010.
- [10] F. Fetterer, K. Knowles, W. Meier, M. Savoie, and A. K. Windnagel. 2017, updated daily. Sea Ice Index, Version 3. Boulder, Colorado USA. NSIDC: National Snow and Ice Data Center. doi: <https://doi.org/10.7265/N5K072F8>. Accessed November 18, 2018].
- [11] Vincent Goulet, et. al. *expm: Matrix Exponentials, Log, etc.* URL <https://cran.r-project.org/web/packages/expm/expm.pdf>, R package version 0.999-3, 2018.
- [12] Michael J. Grayling and Gerhard Burger. *phaseR: Phase Plane Analysis of One and Two Dimensional Autonomous ODE Systems*. URL <https://cran.r-project.org/web/packages/phaseR/phaseR.pdf>, R package version 2.0, 2018.
- [13] IEA International Energy Agency. *Trends 2017 In Photovoltaic Applications*. URL http://www.iea-pvps.org/fileadmin/dam/public/report/statistics/IEA-PVPS_Trends_2017_in_Photovoltaic_Applications.pdf 2017.

- [14] Owen Jones, Robert Maillardet, and Andrew Robinson. *Introduction to Scientific Programming and Simulation Using R*. Boca Raton, FL: CRC Press, 2014.
- [15] Rob Kabacoff. Quick R. URL <https://www.statmethods.net/>. Accessed Dec 8, 2018.
- [16] Daniel Kaplan. *Start R in Calculus*. Project Mosaic. URL http://www.tf.uns.ac.rs/~omorr/radovan_omorjan_003-prII/r-examples/StartR-master.pdf 2018.
- [17] Alboukadel Kassambara. One-Way ANOVA Test in R. URL <http://www.sthda.com/english/wiki/one-way-anova-test-in-r>, Accessed December 8, 2018.
- [18] Johan Larsson, A. Jonathan R. Godfrey, Tim Kelley, David H. Eberly, Peter Gustafsson, and Emanuel Huber. *eulerr: Area-Proportional Euler and Venn Diagrams with Circles or Ellipses*. URL <https://cran.r-project.org/web/packages/eulerr/eulerr.pdf>, R package version 4.1.0, 2018.
- [19] David Meyer, Kurt Hornik, and Christian Buchta. *sets: Sets, Generalized Sets, Customizable Sets and Intervals*. URL <https://cran.r-project.org/web/packages/sets/sets.pdf>, R package version 1.0-18, 2017.
- [20] Steven P. Millard. *EnvStats: Package for Environmental Statistics, Including US EPA Guidance*. URL <https://cran.r-project.org/web/packages/EnvStats/EnvStats.pdf>, R package version 2.3.1, 2018.
- [21] Duncan Murdoch. *rgl Overview*. URL <https://cran.r-project.org/web/packages/rgl/vignettes/rgl.html>, 2018.
- [22] Paul Murrell. *R Graphics Second Edition*. Boca Raton, FL: CRC Press, 2012.
- [23] David Robinson. *The ‘Deadly Board Game’ Puzzle: Efficient Simulation in R*. URL <http://varianceexplained.org/r/boardgamesimulation>, Accessed Dec 8, 2018.
- [24] D. G. Rossiter. *Technical Note: Curve Fitting with the R Environment for Statistical Computing*. URL http://www.css.cornell.edu/faculty/dgr2/teach/R/R_CurveFit.pdf 2016.
- [25] Karline Soetaert. *rootSolve: Nonlinear Root Finding, Equilibrium and Steady-State Analysis of Ordinary Differential Equations*. URL <https://cran.r-project.org/web/packages/rootSolve/rootSolve.pdf>, R package version 1.7, 2016.
- [26] Karline Soetaert, Thomas Petzoldt, and Woodrow Setzer. *deSolve: Solvers for Initial Value Problems of Differential Equations*. URL <https://cran.r-project.org/web/packages/deSolve/deSolve.pdf>, R package version 1.21, 2018.
- [27] M. Henry H. Stevens. *A Primer of Ecology with R*. New York, NY: Springer, 2009.
- [28] Bill Venables, Kurt Hornik, and Georgi Boshnakov. *PolynomF: Polynomials in R*. URL <https://cran.r-project.org/web/packages/PolynomF/PolynomF.pdf>, R package version 1.0-2, 2018.
- [29] Tian Zheng. *Colors in R*. URL <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>. Accessed December 8, 2018.

Index

- abclines3d, 140, 145
abline, 12, 19, 24, 39, 71, 86, 110, 133, 151, 176, 207, 215, 220, 224, 269
abs, 7, 129
add.edges, 296
adjacency matrix, 304
antiderivative, 132
aov, 240
apply, 184, 309, 312
axis, 7, 25, 38, 42, 110, 207, 289
bar chart, 49
barplot, 29, 51, 166
bartlett.test, 242
binom.test, 212
binomial, 73, 165, 211, 213, 214, 281
bipartite, 294
boxplot, 3, 29, 55, 231, 239
BSDA, 192, 195, 205
Buffon Needle, 175
bxcol, 14
byrow, 47
Cantor set, 289
Cartesian product, 288
cbind, 23, 181, 223, 228, 254
ceiling, 66
Chi-Square, 244
chisq.test, 244
choose, 281, 300
Chuck-A-Luck, 172
circle, 31, 291
clique, 304
coef, 67, 73, 152
colMeans, 256
colnames, 170, 244, 246
colorRampPalette, 113
colSums, 172, 256
combination, 184
combn, 184
components, 301
confint, 222
contour, 136, 138
cor.test, 235
cos, 103
critical points, 78
cumprod, 167
cumsum, 86, 91, 167, 179, 181, 310
curve, 2, 7, 9, 17, 19, 29, 31, 35, 58, 80, 113–116, 121, 122, 124, 129, 133, 152, 154, 156, 158, 160, 162, 221, 226
cylinder, 142, 147
dbinom, 165
degree, 303
degree symbol, 264
Deriv, 79, 103, 105, 106, 108, 114
deSolve, 263
det, 256
determinant, 256
dev.new, 42
diag, 141, 256
diagonal, 256
diameter, 303
differentiation, 76, 79, 82, 92, 103, 105, 106, 108, 112, 114
dim, 255
distances, 303
dnorm, 58, 226
dotchart, 29
dotplot, 42
dunif, 191
eccentricity, 303
edges, 292, 293, 299
eigen, 257
eigenvalue, 256
eigenvector, 256
EnvStats, 248
equations

parametric, 15
 euler, 286
 eulerPhi, 283
 eulerr, 285
 Excel, 5
 exp, 7, 103
 expression, 2, 15, 24, 31, 38, 39, 50, 86, 93, 108, 110, 121, 130, 133, 264
 factorial, 282
 Fibonacci, 85, 97
 findEquilibrium, 273, 278
 fisher.test, 214
 floor, 66
 flowField, 266, 271, 276
 for loop, 20, 22, 44, 49, 50, 53, 54, 71, 74, 79, 81, 82, 98, 113, 120, 123, 125, 161, 168, 171, 173, 177, 189, 206, 208, 215, 258, 269, 281, 289, 299, 302, 309
 format, 191
 function, 7
 butterfly, 15
 function, 9
 polar, 14
 step, 12
 GCD, 69
 gradient, 161
 grid, 2, 12
 gridlines, 37, 39
 gset, 288
 head, 203, 248, 263, 274
 hist, 29, 168, 191, 225
 histogram, 57, 168, 179, 190, 225, 226
 ifelse, 9, 311
 igraph, 291, 299
 Inf, 126
 inflection points, 76, 78, 114
 integers, 289
 integral, 75
 integrate, 126, 127, 129, 132
 integration, 126
 intersect, 284
 iris, 239, 247
 is.element, 285
 isIntpower, 283
 jitter, 56, 231
 Kendall, 235
 Kruskal-Wallis, 236
 kruskal.test, 236
 las, 55
 lattice, 293
 layout, 46, 147, 291
 lchoose, 282
 LCM, 69
 legend, 41, 51, 72, 80, 88, 91, 130, 133, 209, 266, 270, 272
 length, 54, 170, 171, 234, 288, 291, 304
 LETTERS, 283
 letters, 283
 lfactorial, 282
 lines, 78, 275
 lm, 151, 156, 157, 219, 227
 log, 7, 103, 106, 157, 158
 log scale, 159
 logistic, 159, 267, 271, 273
 make_lattice, 293
 make_ring, 293
 make_star, 293
 Mann-Whitney, 234
 mapply, 23
 margin, 29
 outer, 29, 33
 matrix, 22, 47, 98, 135, 138–140, 142, 170, 172, 179, 183, 246, 253, 263, 268, 311, 312
 max, 86, 185, 291
 mean, 2, 3, 31, 58, 86, 184, 189, 195
 median, 86, 233
 midpoint, 119, 122
 min, 86, 88, 291
 min_cut, 303
 modulo, 75
 Monty Hall, 170
 mtcars, 199
 mtext, 31, 33, 39, 55, 209, 215
 multivariate, 106
 names, 5, 154, 193, 204, 212, 219, 233, 241, 248, 278
 nderiv, 105
 nextPrime, 283
 nlevels, 136

- nlm, 154, 160, 162
- nonparametric, 231
- nullcline, 266, 272
- numDeriv, 103
- numeric, 98
- ode, 263, 268, 269, 274
- one, *see also* two
- one-way ANOVA, 236, 239
- open3d, 138
- options
 - scipen, 85
- orthogonal, 156
- outer, 42, 135, 138, 139, 145
- package, 4, 13, 16, 49, 63, 69, 79, 103, 108, 128, 135, 138, 192, 195, 248, 256, 263, 283, 285, 291, 299
- palette, 78
- par, 19, 36, 42, 93
 - bg, 40
 - mar, 31, 33, 42, 55, 86, 94, 108, 121, 231
 - mfrow, 189, 223, 293
 - new, 110
 - oma, 33, 42
 - usr, 24, 31, 33, 110
- parabola, 7, 16
- paraboloid, 139
- parametric, 7
- Pascal's triangle, 73, 281
- paste, 24, 46, 54, 80, 82, 94, 124, 132, 133, 162, 167, 168, 180, 191, 209, 215, 264, 270, 302
- path, 293, 302, 303
- pbinom, 165
- pch, 10
- Pearson, 235
- persp, 135, 138
- persp3d, 138, 145
- Petersen graph, 294
- phase plane, 275, 276
- phaseR, 266
- pie chart, 29, 42, 46, 292
 - 3D, 49
- piecewise, 7
- plot, 12, 15, 29, 33, 35, 40, 65, 68, 71, 73, 78, 86, 93, 95, 110, 156, 160, 162, 174, 176, 179, 207, 215, 223, 264, 266, 268, 275, 281, 285, 289, 291, 293, 299
 - checkerboard, 22
- png, 138, 146
- points, 10, 41, 71, 86, 88, 108, 114, 124, 144, 177, 215
- polar, 7
- polygon, 116, 123, 131, 142, 144
- polylist, 68, 78, 81
- polynom, 64, 80
- PolynomF, 63, 69, 75
- polynomial, 39, 63, 155
 - degree, 64
 - integration, 75, 119
 - Legendre, 81
 - roots, 63, 65, 71
 - Taylor, 35, 79
- pracma, 13
- predator-prey, 273
- predict, 223, 228
- prime, 283
- primeFactors, 283
- print, 54, 82
- prop.test, 211
- punif, 191
- Pythagorean triple, 21
- qbinom, 165
- qqline, 200, 242
- qqnorm, 200, 242
- qqplot, 200, 242
- quantile, 165, 191
- qunif, 191
- radius, 147, 303
- rainbow, 20, 47, 139, 147, 269, 284
- rbind, 51, 98, 254
- rbinom, 165, 168, 215
- read.table, 219
- reals, 288
- rectangle, 292
- regression, 153, 219
 - multiple, 227
- replace, 50
- replicate, 168, 172, 176, 179, 181, 189, 256
- residuals, 224, 241, 242, 245
- result, 202

return, 8
 rev, 139
 rexp, 189
 rgl, 135, 138
 Riemann, 99, 120
 rm, 6
 rnorm, 3, 54, 57, 157, 192, 195, 206
 roots, 16, 108, 128
 real, 71
 rootSolve, 16, 108, 128, 144
 round, 66, 132
 row, 24, 256, 260
 rowMeans, 256
 rownames, 51, 170, 246
 rowSums, 256
 runif, 176, 191, 231

 saddle, 135, 138
 sample, 64, 162, 172, 179, 181, 195, 311
 sample_gnm, 299
 sample_gnp, 299
 sapply, 12, 127, 173, 282, 291, 310
 scatter plot, 5, 39, 41, 86, 220
 segments, 30, 121, 178, 208, 289
 seq, 13, 124, 138, 191, 207, 274, 289
 set.seed, 3, 195
 setdiff, 284
 setequal, 284, 285
 Shapiro-Wilk, 200, 242
 shapiro.test, 200
 sigma, 241
 signif, 66
 sin, 80
 skewed, 168
 slope field, 266, 271
 solids, 147
 solve, 65, 257
 Spearman
 rank correlation, 235
 sphere, 292
 sqrt, 7
 SSlogis, 160, 161
 stability, 273, 278
 stack, 24, 42, 234, 236, 244
 step function, 7
 str, 199, 247
 stripchart, 42, 56, 231
 subgraph, 304
 subintervals, 119

 submatrix, 253
 sum, 46, 86, 90, 119, 167, 282, 291, 309,
 312
 summary, 3, 199
 surface, 135, 138

 t-test, 4, 200
 paired, 205
 table, 303
 tabulate, 165, 173, 179
 tangent line, 112
 tangent plane, 138
 Testing, *see also* one
 text, 39
 three, *see also* Testing
 tickmarks, 42
 title, 156
 tlim, 266, 271, 276
 trajectory, 266, 271, 276
 transpose, 24, 255
 trunc, 66
 tsum.test, 205
 TukeyHSD, 241
 turn3d, 141
 twinPrimes, 283
 two, *see also* three

 undirected, 293, 296
 Uniform, 176, 177, 191, 231
 union, 284, 285
 unique, 247
 uniroot.all, 16, 108, 114, 128, 144
 usr, 124

 variances, 247
 varTest, 248
 vectorize, 17, 132
 Venn, 284, 285
 vertices, 293

 Weibull distribution, 57
 which.max, 185
 wilcox.test, 233–235
 Wilcoxon
 rank sum, 234
 signed rank, 233

 xaxs, 110, 269, 275
 xaxt, 36
 xlab, 7, 29, 37, 41, 42, 51, 55, 86, 93,

110, 135, 138, 140, 142, 151,
156, 160, 162, 167, 168, 174,
179, 191, 215, 264, 266, 269,
271, 275, 276, 289
xlim, 12, 35, 41, 86, 88, 93, 135, 138,
140, 141, 151, 215, 266, 271,
275, 276, 281
yaxs, 110, 269, 275
yaxt, 36
ylab, 7, 9, 17, 29, 37, 41, 55, 80, 86, 93,
135, 138, 140, 142, 151, 156,
160, 167, 168, 174, 180, 215,
266, 269, 271, 275, 276, 289
ylim, 7, 12, 35, 41, 80, 86, 88, 93, 124,
135, 138, 140, 141, 215, 225,
266, 269, 271, 275, 276, 281
z.test, 193, 195
zlab, 135, 138, 140, 142
zlim, 135, 138, 140, 141
zsum.test, 194, 196