Chapter 1

Introduction to R

Chaitra H. Nagaraja[1]
Gabelli School of Business & Graduate School of Business Administration, Fordham University, New York, USA
[1]*Corresponding author: e-mail: cnagaraja@fordham.edu*

ABSTRACT
The basics of R programming are discussed from syntax, conditional statements, and control structures, to writing functions. Fundamentals such as data cleaning, exploratory data analysis, hypothesis testing, and regression are introduced. Simulation and random generation and numerical methods are considered as well. Finally, a list of add-on packages and additional resources are listed.

Keywords: CRAN, Workspace, R syntax, Functions, Statistical analysis, Simulation, Numerical methods

1 INTRODUCTION

The statistical programming language R is a free version of the language S initially developed at Bell Laboratories in 1976 by John Chambers. Due to its open source philosophy and versatility, R is now used widely in academia, government, and industry. Its flexibility allows one to link to low-level programming languages, such as C++ and FORTRAN, scripting languages like UNIX, blend with word processors such as LaTeX, and handle large data sets with parallel computing capabilities.

Learning any new language requires an initial investment of time to internalize the syntax and intuition behind its structure. This chapter is designed to get you started in this process, along with illustrating the broad range of possible applications.

We begin with how to install R in Section 2 and introduce the command line environment. In Section 3, various R object and data types are discussed along with functions for common calculations. To help write your own programs, we focus on control statements, function construction, and error correction in Section 4. The remainder of the chapter focuses on applications of R. In Sections 5, 6, and 7 we discuss uploading and writing data files, data cleaning, and exploratory data analysis. Basic statistical inference and regression are

Handbook of Statistics, Vol. 32. http://dx.doi.org/10.1016/B978-0-444-63431-3.00001-2

TABLE 1 Workspace Functions

Operation	Function(s)
Save/load workspace objects	`load()`, `save()`, `save.image()`
Read code into workspace	`source()`
Commands history	`history()`, `loadhistory()`, `savehistory()`
Get working directory	`getwd()`
Set working directory	`setwd()`
Workspace maintenance	`gc()`, `ls()`, `rm()`, `rm(list=ls())`
Help	`?`, `help()`

discussed in Section 8. In Section 9, we describe how to set up simulation studies and sampling schemes. Numerical methods, such as finding roots and integration, are introduced in Section 10. We conclude in Section 11 with some references for R and related applications. To illustrate some of the topics, we sparingly use plots with little clarification regarding the code. Refer to Chapter 2 of this book for more on R graphics. Finally, to eliminate dependence on supplementary materials, we will use the data that comes with the base R package in our examples; see Section 2.1 for a list.

To help the reader, we have compiled tables to summarize many of the commands and R packages introduced within the text: see Table 1 for workspace functions, Table 2 for packages and functions for a wide range of statistical methods, Table 3 for miscellaneous functions, Table 4 for numerical operators, Table 5 for logical operators and conditional statements, Table 6 for data processing functions, Table 7 for summary statistics, Table 8 for basic hypothesis testing functions, Table 9 for probability distributions and random number generation, and Table 10 for numerical methods.

We now introduce the notation for expressing R code within the text of this chapter. In particular, `R_objects` for objects, `function()` for functions, "`name`" in double quotes for function arguments or variable names, an italicized *`R_package`* for add-on packages, and *`USER_INPUT`* in all capital letters for user input such as file and object names. Blocks of code will be written in `this font` and indicated as such. Consequently, when we write code in the R environment, we will begin each line with > to simulate the console (note: do not copy the > symbol); if a particular command continues to a second line, > is replaced by +. Within these blocks of code, to avoid confusion, we will not display the output, only the commands.

TABLE 2 R Packages for Statistical Analysis (*base* Indicates Function in Standard Installation of R); Other Applications: Table 3; Basic Hypothesis Testing: Table 8; Numerical Algorithms: Table 10

Method	R Package: Notes
ANOVA, ANCOVA, MANOVA	*base*: anova(), aov(), lm(), manova(), oneway.test()
Bayesian methods	*arm; bayess; mcmc; MCMCpack; rbugs*; *R2WinBUGS*
Capture–recapture	*Rcapture*
Conjoint analysis	*conjoint*
Copulas	*copula*
Cluster analysis	*base*: hclust(), kmeans(); *cluster*; *mclust*; *pvclust*
Data mining, machine learning	*ada*; *adabag*; *e1071*; *gbm*; *ipred*; *mboost*; *neuralnet*; *nnet*; *party*; *randomForest*; *rpart*; *tree*
Discriminant analysis	*MASS*: lda(), qda()
Experiments	*pcalg*; *experiment*; *stepp*
Factor analysis	*base*: factanal()
Functional data analysis	*fda*; *fda.usc*
Generalized linear models	*base*: glm(); *gam*
Hierarchical/mixed models	lme4: glmer(), lmer(), nlmer(); *hglm*; *nlme*
Instrumental variables	*AER*: ivreg(), ivreg.fit(); *sem*: tsls()
Kernel methods	*base*: density(); *kernlab*; *ks*
Loess/lasso	*basic*: loess(); *genlasso*; *glmnet*; *lars*; *lasso2*
Linear regression	*base*: AIC(), anova(), BIC(), lm(), predict.lm(), residuals.lm(), summary.lm(), plot.lm(), step(); *cars*: vif(); *MASS*: stepAIC(); *leaps*: leaps()
Missing data methods	*mi*; *mice*; *MImix*; *mitools*; *pan*
Network analysis	*igraph*; *network*; *sna*; *statnet*; *statnet.common*
Observational studies	*Matching*; *MatchIt*; *multilevelPSA*; *nonrandom*; *optmatch*; *PSAgraphics*; *rbounds*; *RItools*
Order statistics	*ismev*; *heavy*
Principal components	*base*: princomp()

Continued

TABLE 2 R Packages for Statistical Analysis (*base* Indicates Function in Standard Installation of R); Other Applications: Table 3; Basic Hypothesis Testing: Table 8; Numerical Algorithms: Table 10—Cont'd

Method	R Package: Notes
Psychometrics	*psych*; *psychotools*
Quantile regression	*quantreg*
Queueing	*queueing*
Resampling methods	*boot*; *bootES*; *bootstrap*
Signal processing, wavelets	*signal*; *signalextraction*; *tuneR*; *wavelets*; *waveslim*; *wavethresh*
Spatial statistics	*maptools*; *OpenStreetMap*; *raster*; *rgdal*; *rgeos*; *sp*; *spatial*; *SpatialPack*; *spatial.tools*
Splines	*base*: `smooth.spline()`, `splinefun()`; *polspline*; *pspline*
State space models	*dse*; *dlm*; *KFAS*
Statistical process control	*qat*; *qcc*; *qcr*
Structural equation modeling	*sem*; *SEMID*; *semdiag*; *semGOF*
Survey methods	*EVER*; *sampling*; *samplingbook*; *SamplingStrata*; *samplingVarEst*; *stratification*; *survey*;
Survival analysis	*OIsurv*; *survival*; *survivalMPL*; *survivalROC*
Time series	*base*: class `ts`, `acf()`, `ar()`, `arima()`, `arima.sim()`, `pacf()`, `predict.Arima()`, `tsdiag()`; *forecast*; *gsarima*; *rugarch*; *seasonal*; *timeSeries*; *tseries*; *tseriesChaos*

2 SETTING UP R

In the following section, we describe how to install R and begin coding. We briefly discuss how memory is managed and list additional packages that expand the capabilities of R.

2.1 Installing and Starting R

Install R from the CRAN (Comprehensive R Archive Network) Website: http://cran.r-project.org/; choose the link applicable to your computer system (e.g., Linux, Mac, Windows). R is continually updated; therefore, it is important to check the CRAN Website periodically for new versions.

TABLE 3 Miscellaneous R Functions and Packages

Method	R Package: Notes
Data processing	*plyr*
Data sets	*base*: `library(help = "datasets")`
Date/time stamp	*base*: `date()`, `timestamp()`, `proc.time()`
Debugging	*base*: `browser()`, `debug()`, `trace()`, `untrace()`, `traceback()`; *debug*
Integrate R with LaTeX	`Sweave()`, *knitr*
Parallel computing	*snow*
R colors	*base*: `col2rgb()`, `colors()`, `colours()`, `rgb()`
R color spectra	*base*: `cm.colors()`, `heat.colors()`, `rainbow()`, `terrain.colors()`, `topo.colors()`

The base package (herby referred to as base R or *base*) contains a variety of functions and operations most commonly used in data processing and statistics. Furthermore, a range of data sets are included (type `library(help = "datasets")` for more details); to make this text self sufficient, we will use these data in our examples. Specifically, we use: `airmiles` (commercial airline revenue), `attitude` (Chatterjee–Price Attitude survey data on a firm's employees), `faithful` (geyser data from Yellowstone National Park's Old Faithful), `iris` (iris plant measurements), and `ToothGrowth` (tooth growth in guinea pigs receiving Vitamin C supplements).

Opening the R program loads a console where you can type commands directly into a workspace. The symbol > below simulates the R interface; if a command continues to a second line, it is indicated by +, as in the R console. Now, try the following commands (output omitted below):

```
> # R will ignore everything after the symbol #
> x              # error: x is not a defined object yet
> x <- 4         # assign the number 4 to the object x
> y <- sqrt(x)   # assign square root of x to new object y
> x <- x+5       # save the value of x plus five in x
> x              # x now defined; print value(s) of x
```

Observe that all text after # in a given line is ignored by the computer; this feature allows us to add comments to our code. Comments are a crucial component of any programming language. It is easy to forget what your code signifies after time has passed. Do not learn this lesson the hard way. Comment your code!

TABLE 4 Basic Numerical Operations

Operation	Function name	Matrix operation	Function
Arithmetic	`+-*/`	Transpose vector/ matrix	`t()`
Modulo	`%%`	Solve linear system, invert matrix	`solve()`
Exponent	`^`	Eigenvalues, eigenvectors	`eigen()`
Square root	`sqrt()`	Determinant	`det()`
Logarithm	`log()`	Decompositions	`chol`, `svd()`, `qr()`
Absolute value	`abs()`	Vector/matrix multiplication	`%*%`
Exponential	`exp()`	Diagonal matrices	`diag()`
Sum/product of vector	`sum()`, `prod()`	Outer product	`%o%`
Cumulative sum/product of vector	`cumsum()`, `cumprod()`	Cross product	`crossprod()`
Sort vector	`sort()`	Trace	`sum(diag())`
Rounding	`ceiling()`, `floor()`,		
	`round()`, `signif()`,		
	`trunc()`		
Integer division	`%/%`		
Trigonometry	`sin()`, `cos()`, `tan()`,	Miscellaneous	
	`acos()`, `atan()`, `asin()`	Infinity	`Inf`, `-Inf`
Combinations	`choose()`	π	`pi`
Factorial	`factorial()`	Missing data	`NA`
Complex numbers	`complex()`, `Re()`, `Im()`,	Not a number	`NaN`
	`Mod()`, `Arg()`, `Conj()`	Null (empty)	`NULL`

TABLE 5 Components for Conditional Statements; for Examples, Let `x <- 5`, `y <- c(7,4,2)`, **and** `z <- c("a","b","c")`

Symbol/Function	Description	Example	Example Output
==	Equal to	`x == 4`	`FALSE`

		`z == "c"`	`FALSE`
			`FALSE`
			`TRUE`

!=	Not equal to	`x != 4`	`TRUE`

		`z!="b"`	`TRUE`
			`FALSE`
			`TRUE`

<, >	Less/greater than	`x<4`	`FALSE`

<=, >=	Less/greater than or equal to	`y >= 5`	`TRUE`
			`FALSE`
			`FALSE`

&	And	`(x>4) &` `(x <=10)`	`TRUE`
&&	& Expression `TRUE` for all elements in vector	`(y<4) && (y>=6)`	`FALSE`

|	(Inclusive) or	`(z=="a") |`
		`(z=="b")`
			`TRUE`
			`TRUE`
			`FALSE`

||	| Expression `TRUE` for at least one element in vector	`(y<4) || (y>=6)`	`TRUE`
!	Not	`!(x>4)`	`FALSE`
`any()`	Expression `TRUE` for at least one element in vector	`any(z=="b")`	`TRUE`
`all()`	Expression `TRUE` for all elements in vector	`all(x==3)`	`FALSE`
`is.element()`,	Are the elements of `x` in `y`?	`is.element(x,y)`	`FALSE`
`%in%`		`x%in%y`	`FALSE`

Continued

TABLE 5 Components for Conditional Statements; for Examples, Let `x <- 5`, `y <- c(7,4,2)`, and `z <- c("a","b","c")`—Cont'd

Symbol/Function	Description	Example	Example Output
`which()`	Which elements of `x` satisfy a certain condition	`which(z=="b")`	`2`
`which.min()`, `which.max()`	Position of first min/max value in a vector	`which.max(y)`	`1`
`complete.cases()`, `is.finite()`, `is.infinite()`, `is.na()`, `is.nan()`	Checking for missing and/or infinite data	`is.na(y)`	`FALSE FALSE FALSE`
`is.character()`, `is.factor()`, `is.integer()`, `is.numeric()`	Checking data types	`is.factor(z)`	`FALSE`
`is.data.frame()`, `is.list()`, `is.matrix()`, `is.vector()`	Checking object types	`is.vector(y)`	`TRUE`

After entering the five lines of code above, the workspace now has two objects, `x`, which is equal to 9, and `y`, equal to 2. Note that changing the value of `x` after `y` is defined does not alter the value of `y`. Nearly any name can be used to label an object, provided that it does not start with a number; however, contextually meaningful names help when deciphering your code. The operator `<-` assigns the value on the right to the object on the left. The equal sign (=) can technically also be used, but should be avoided as it has alternate interpretations in R. Finally, `sqrt()` denotes the square root function. In R, arguments for a function are listed within the parentheses. Typing the function name without the parentheses generally prints the code for the function. To obtain details about `sqrt()`, type `?sqrt` or `help(sqrt)`.

As you code, your workspace will be filled with objects. To keep it organized, especially if you run out of space, the following commands are helpful (also see Table 1):

```
> # define objects x, y, and z
> x <- 10
> y <- log(5)
```

```
> z <- "a"
>
> ls()              # list everything in the workspace
> rm(x, z)          # remove objects x and z, but not y
> rm(list=ls())     # remove all objects from the workspace
> gc()              # list current memory usage
```

By defining `x <- 10`, we have overwritten the previous command `x <- 4` from the previous section of code; `y` has been replaced by the natural log of 5.

TABLE 6 Input/Output and Processing Functions

Operation	Function(s)
Read data into console	`scan()`, `read.table()`, `read.csv()`; *`foreign`*; *`xlsx`*
Write to a file	`write()`, `write.table()`
Formatting output	`cat()`, `format()`, `paste()`, `round()`, `truncate()`, `ceiling()`, `floor()`
Data summaries	`colnames()`, `dim()`, `head()`, `names()`, `ncol()`, `nrow()`, `rownames()`, `tail()`
Reverse/sort vector	`rev()`, `sort()`
Sort data frame	`order()`
Subset data frame	`subset()`
Separate data frame by category	`split()`
Training and test sets	`sample()`
Categorical variables	`chartr()`, `cut()`, `grep()`, `strsplit()`, `substr()`, `tolower()`, `toupper()`
Construct/append to a data frame	`cbind()`, `data.frame()`, `rbind()`
Merge data frames	`merge()`
Computations by row or column	`apply()`
Computations for each element of a list	`lapply()`
Computations on a data frame by category	`aggregate()`
additional data processing functions in package *`plyr`*	

TABLE 7 Summary Statistics and Plots

Statistic (R Package)	Function	Summary Plot	Function
Mean, trimmed mean	`mean()`	Histogram	`hist()`
Weighted mean	`weighted.mean()`	Box plot	`boxplot()`
Mean absolute deviation	`mad()`	Bar graph	`barplot()`
Median	`median()`	Scatter plot/time series	`plot()`
Mode	`mode()`	Stem and leaf plot	`stem()`
Variance	`var()`	Pie graph	`pie()`
Standard deviation	`sd()`	Mosaic plot	`mosaicplot()`
Max/min	`max()`, `min()`	Dot chart	`dotchart()`
Cumulative max/min	`cummax()`, `cummin()`	Matrix plotting	`matplot()`
Range	`diff(range())`	Contour plot	`contour()`
Quantile	`quantile()`	Plot a function	`curve()`
Six number summary	`summary()`	Normal quantile plot	`qqnorm()`, `qqline()`
Covariance	`cov()`		
Correlation	`cor()`	Quantile–quantile plot	`qqplot()`, `qqline()`
Skewness (*moments*)	`skewness()`		
Kurtosis (*moments*)	`kurtosis()`		
Contingency table	`table()`		

2.2 Memory

The computing speed in R depends heavily on your computer. We will not go into much detail here regarding memory in R. Use `?Memory` for information on memory usage in R and `?.Machine` for details on the largest and smallest numerical values possible on your machine. As mentioned above, `gc()`, which stands for garbage collection, lists current memory usage. If you run out of memory, or want to clear your workspace, remove unnecessary objects using `rm()`.

TABLE 8 Hypothesis Tests

Test	Function
One-, two-sample, paired *t*-tests for means	`t.test()`
One-, two-sample *z*-tests for proportions	`prop.test()`
Wilcoxon rank sum/Mann–Whitney tests	`wilcox.test()`
Kruskal–Wallis rank sum test	`kruskal.test()`
Bartlett test for equal variances	`bartlett.test()`
χ^2-test for contingency tables, goodness-of-fit tests	`chisq.test()`
Fisher's exact test for independence	`fisher.test()`
Kolmogorov–Smirnov goodness-of-fit tests	`ks.test()`
Shapiro–Wilk test for normality	`shapiro.test()`
t-Test for correlation	`cor.test`
t-Test for regression slopes	`summary()`
Overall *F*-test, partial-*F* test for linear models	`anova()`
One-way ANOVA	`oneway.test()`
Tukey's HSD for multiple comparisons	`TukeyHSD()`
Durbin–Watson test for autocorrelation	*`lmtest`*: `dwtest()`

As with all programming languages, code can be written inefficiently. To reduce this issue, check to see if built-in functions can be applied and decrease your reliance on loops where possible (see Section 4.6). Using the command `date()` before and after a piece of code can help monitor the time to run a program. Furthermore, if your data sets are large, parallel computing methods may lower computing times as well. Use the package *`snow`*. See the CRAN Website for more details.

2.3 Saving Your Code and Workspace

Each time you open R, files are uploaded from and downloaded to a directory (e.g., a folder on your computer); `getwd()` will print this directory file path. To change the working directory, specify the file path in `setwd()` or use the toolbar at the top of your screen. In this section, we will discuss how to save and load code and workspaces (also see Table 1). For information on reading and writing data files, refer to Section 5.

There are two ways to preserve your code. If you have entered your commands directly into the console, you can save them by using `savehistory(file = "`*`FILE_NAME`*`.Rhistory")` where *`FILE_NAME`* is the name of your file

TABLE 9 List of Common Probability Distributions

Distribution (R Package)	Density	Distribution	Quantile	Sampling
Beta	`dbeta()`	`pbeta()`	`qbeta()`	`rbeta()`
Binomial	`dbinom()`	`pbinom()`	`qbinom()`	`rbinom()`
Burr (*actuar*)	`dburr()`	`pburr()`	`qburr()`	`rburr()`
Cauchy	`dcauchy()`	`pcauchy()`	`qcauchy()`	`rcauchy()`
χ^2	`dchisq()`	`pchisq()`	`qchisq()`	`rchisq()`
Exponential	`dexp()`	`pexp()`	`qexp()`	`rexp()`
F	`df()`	`pf()`	`qf()`	`rf()`
Gamma	`dgamma()`	`pgamma()`	`qgamma()`	`rgamma()`
Hypergeometric	`dhyper()`	`phyper()`	`qhyper()`	`rhyper()`
Log-normal	`dlnorm()`	`plnorm()`	`qlnorm()`	`rlnorm()`
Multivariate normal (`mvtnorm`)	`dmvnorm()`	`pmvnorm()`	`qmvnorm()`	`rmvnorm()`
Multivariate t (*mvtnorm*)	`dmvt()`	`pmvt()`	`qmvt()`	`rmvt()`
Negative binomial	`dnbinom()`	`pnbinom()`	`qnbinom()`	`rnbinom()`
Normal/Gaussian	`dnorm()`	`pnorm()`	`qnorm()`	`rnorm()`
Pareto (*actuar*)	`dpareto()`	`ppareto()`	`qpareto()`	`rpareto()`
Poisson	`dpois()`	`ppois()`	`qpois()`	`rpois()`
t	`dt()`	`pt()`	`qt()`	`rt()`
Uniform (continuous)	`dunif()`	`punif()`	`qunif()`	`runif()`
Weibull	`dweibull()`	`pweibull()`	`qweibull()`	`rweibull()`
Discrete random sampling				`sample()`
Random number generation				`RNGkind()`
Setting random number seed				`set.seed()`

TABLE 10 Table of Numerical Calculation Functions (*base* Indicates Function in Standard Installation of R)

Method	R Package: Notes
Fourier transforms	*fftw*, *fftwtools*
Roots of a polynomial	*base*: polyroot()
Roots of a function	*base*: uniroot(); *rootSolve*
Optimization	*base*: optimize(), optimise(), optim(), constrOptim(), nlm()
Numerical derivatives	*numDeriv*: grad(), jacobian(), hessian()
Numerical integration	*base*: integrate(); *cubature*: adaptIntegrate()
Differential equations	*deSolve*; *ReacTran*; *bvpSolve*; *rootSolve*; *sde*; *pomp*

and ".Rhistory" denotes a history file. If you just include a file name and not a file path, the file will be saved in the current working directory. To load the saved history, use `loadhistory(file = "FILE_NAME.Rhistory")`. However, it is more efficient to write your code in a separate file. A text file is sufficient, although additional features such as color coding and parenthesis matching can help. For Apple computers, a text editor is included when downloading R; alternatives include GNU Emacs, WinEdt, and Tinn-R (see Section 11 for more details). Save your R code with a ".R" file: `"FILE_NAME.R"`. To enter the code from an entire file, use `source("FILE_NAME.R")`.

All objects in your workspace can be saved to an ".RData" workspace file by using `save.image(file = "FILE_NAME.RData")`. To save only a subset of objects, use `save()` and specify those objects in the "`list`" argument. To load the workspace objects, use `load(file = "FILE_PATH.RData")`.

Finally, you can save your R session (i.e., what you see on your console) as a text file by choosing "Save" from the "File" menu from the toolbar menu at the top. All commands that were typed in that R session and the output printed will be saved, but cannot be uploaded back into R.

2.4 R Packages

Apart from base R, you can expand R's capabilities by installing a variety of packages written by fellow users. These bundles contain functions

(i.e., routines) and/or data sets, which can be downloaded from CRAN or from the toolbar at the top of your screen. If you do not have administrative privileges on your computer, download the ZIP or TGZ package file to a local folder instead and install it through your console.

Once installed, a package is loaded using the command `require(PACKAGE_NAME)` or, equivalently, `library(PACKAGE_NAME)`. Saved workspaces retain only the objects you created; consequently, packages must be loaded again each time you open a new R session. Each package has a help manual on the CRAN Website, which lists and describes its components. These help pages can also be accessed through the console using `?` and `help()` after loading a package.

Check functions in an R package before use to ensure they are operating as you expect them to. The list of packages is dynamic; consequently, updating your computer's operating system or your version of R may cause problems. One way to circumvent this issue is to manually install an older version of a package. See the CRAN Website for instructions on creating your own R packages.

A brief list of packages for a range of statistical methods available at the time of writing is provided in Table 2; functions and packages for miscellaneous applications are in Table 3. Note that these lists are in no way exhaustive.

3 BASIC R OBJECTS AND COMMANDS

Information in R is generally one of three variable types: numeric, character string, or logical (i.e., Boolean). These bits of information are organized using R objects such as vectors, matrices, and data frames. We begin this section with these data types, followed by a discussion of R object types. We then introduce some basic mathematical operations that can be applied to numeric data. Finally, we focus on character strings and factors, which are a special way of handling categorical data. Note that R has additional object types, such as linear model objects or spatial objects, associated with more advanced data structures. To check the types, use `class()` or `mode()`. You can check for or force a specific class by using, for example, `is.numeric()` or `as.numeric()`, respectively. This applies to both data and object types (i.e., numeric, integer, double, complex, character, logical, matrix, array, data frame, list, and factor).

3.1 Numbers, Character Strings, and Logicals

Numeric values are real numbers. The largest and smallest number allowed depend on your computer; use the command `?.Machine` for more details. Infinite values are denoted by `Inf` and `-Inf` and indeterminate forms by `NaN`. R can also handle complex numbers written symbolically such as `3+4i`, where `i` is treated not as an R object name but the imaginary number $i = \sqrt{-1}$. Operations on numerical data are discussed further in Section 3.2 and listed in Table 4.

Character strings treat words (or numbers) as abstract symbols. They must always be enclosed in double quotes, otherwise R treats the "word" as the name for an R object. We consider strings, and a related object type, factors, in Section 3.4.

The third data type, logical (i.e., Boolean), is generally used in conditional statements and to subset data. You can either use `TRUE`/`FALSE` or simply `T`/`F` to denote this type. See Section 4.1 for more details.

Below are some examples. Observe that missing data is denoted by `NA`.

```
> w <- 5          # numeric
> x <- 3+4i       # complex number
> y <- "hello"    # character string
> z <- TRUE       # logical/Boolean (TRUE/FALSE, T/F)
>
> NA              # missing data
```

3.2 Scalars, Vectors, Matrices, and Arrays

The most basic R objects are vectors and matrices that are related to the mathematical conceptions of the words. While these objects can contain any data type (e.g., character, numeric), they cannot mix data types within the object itself. For example, a vector initially created with both character string and numeric elements will be automatically converted into a character string vector. Operations can be applied element-wise or for, numeric objects, to the entire object (e.g., matrix algebra).

Up until this point, all of our examples have been with scalars. To construct a column vector from scratch use the concatenate function `c()`. For numerical vectors, three additional options exist: use `:` for integer sequences, `seq()` for sequences with regular spacing, and `rep()` for replicating a vector a set number of times. We examine a few examples next:

```
> ### scalar object saved to object u
> u <- TRUE
>
> ### vector: c() function concatenates values
> v <- c("hello","world",NA)          # note: 3rd element missing
> w <- -1:10                          # integer sequence: -1, 0, 1,
>                                     # 2,..., 10
> x <- seq(from=-1, to=10, by=2)      # sequence in increments of 2
> y <- seq(from=-1, to=10, length=10) # sequence from -1 to 10 with ten
>                                     # values in equal intervals:
>                                     # -1, 0.222, 1.444,...,8.777,10
> z <- rep(c(4,5), times=6)           # repeat vector c(4,5) six times
>                                     # result: 12x1 column vector
```

Once a vector is constructed, new elements can be appended by reusing `c()`. To determine the number of elements, use `length()`. In R, vectors are

indexed starting from 1. To extract elements from a vector, specify which elements between square brackets `[]`:

```
> v <- c("hello","world",NA)      # note: 3rd element missing
> v <- c(v, "R")                  # append value to end of v
> length(v)                       # length of vector v
>
> v[2:4]                          # extract 2nd-4th elements
> v[c(FALSE, TRUE, TRUE, FALSE)]  # extract 2nd-4th elements
```

Each vector is essentially a column vector, as in linear algebra, and `t()` is the transpose function. To construct a matrix from individual vectors, use `rbind()` to append by row or `cbind()` to append by column. The function `matrix()` allows one to construct a matrix directly, similar to `vector()`. As with vectors, all elements of a matrix must be of the same data type. You can name or extract the column and row names using `colnames()` and `rownames()`, respectively.

Matrix dimensions, like in mathematics, are expressed as (rows×columns). The functions `dim()`, `nrow()`, and `ncol()` return the dimensions, number of rows, and number of columns, respectively. Both rows and columns are indexed beginning from 1. You can extract certain rows, columns, or elements again by using square brackets, but separating the row and column indices by a comma:

```
> ### matrix
> A <- matrix(c(4,2,7,8,4,6), nrow=3, byrow=TRUE)
> dim(A)     # dimensions of matrix A: rows x columns
> A[1,1]     # A[row_number, column_number]
> A[1,]      # first row, all columns
```

A related object is an array, a multidimensional matrix, which can be constructed with the function `array()`. Be careful when constructing an array to ensure the elements are in the correct order:

```
> ### array: multidimensional matrix
> B <- array(1:8, dim=c(2,2,2))
>
> dim(B)
> B[1,1,1]
> B[1,,]
```

There are many numerical operations in base R, many of which we list in Table 4. Most of the functions are applied element-wise to a vector or matrix on the left side of the table. On the top-right, the matrix algebra functions are applied to the object as a whole. Some miscellaneous, but useful, representations are provided at the bottom-right of the table.

Use `help()` for details on any of these functions. For functions applied to character strings, see Section 3.4; for logical operators, see Section 4.1.

3.3 Data Frames and Lists

Data frames are what statisticians generally think of as a data table where each column represents a variable. Columns can contain data of different variable types, but must be of the same length. Lists are the more general form of a data frame, where elements are not required to have equal lengths or even the same object types.

The function `data.frame()` binds vectors together, possibly with column headings. The functions `colnames()`, `names()`, and `rownames()` can be used to attach and extract column and row names for these objects. You can extract a given row, column, or specific cell from a data frame using the column/row index value or by name using the operator `$`:

```
> ### data frame (can combine variable types among
> ### columns unlike vectors and matrices)
> y <- data.frame("a"=c(1,2,3), "b"=c("hi","hi","bye"),
>            "c"=c(TRUE,TRUE,FALSE))
>
> # 3 ways to extract first column (vector) "a"
> y$a
> y[,1]
> y[,"a"]
>
> dim(y)    # dimensions of data frame, nrow() by ncol()
```

Lists are constructed very similarly using the `list()` function; however, the data is not structured like a data table as the combined objects may differ. Extraction of elements differs slightly as well. In the next example, the list `z` has two elements: a vector and a data frame.

```
> ### list (general form of data.frame--components
> ###        can be of varying object type)
> z <- list("d"=c(1,2), "e"=data.frame("v1"=c("is","hi","bye"),
+                                      "v2"=c(4, 6, 3)))
> z$d             # extract vector "d"
> z[[1]]          # extract vector "d" (first list component)
> (z[[1]])[2]     # extract element 2 of vector "d"
>
> names(z)        # extract (or assign) list names
> length(z)       # number of components in z
> unlist(z)       # remove list structure; convert to vector
```

To remove the list structure, use the function `unlist()`; this collapses all components into one vector. Note that this conversion may change the variable type of the data. Lists are extremely useful when writing functions as multiple results can be returned in the output. More details are provided in Section 4.5.

3.4 Strings and Factors

Character strings can be used to represent text and categorical variables. Some functions are: `nchar()` to calculate the number of characters, `substr()` to extract a subset of characters within a string, `paste()` to concatenate strings, `print()` and `cat()` to print output, `strsplit()` to separate a string based on a specified character (or string), `chartr()` to replace specified characters, `grep()` for pattern matching, and `tolower()`/`toupper()`/`casefold()` to change the case of the letters. These functions are useful when cleaning data (see Section 6). Some examples are given below:

```
> w <- c("hello", "this", "is", "R")
> nchar(w)                          # output: 5 4 2 1
> substr(w, start=1, stop=2)        # output: "he" "th" "is" "R"
>
> x <- paste("Hello", " what is your name?", sep=",")
> strsplit(x, split=",")            # split x before and after comma
>
> print(x)                          # print output to console
> cat("Hello", " what is your name?", sep=",")
```

Factors are an R object type for categorical data which are treated differently from character strings. They have an additional attribute, levels, that describes all possible values a factor object can take (this can exceed the number of categories which actually exist in the data). To construct a factor, use `factor()`. The argument "`levels`" can be omitted if the levels should be extracted from the data itself. If not all categories are represented within the data, use "`levels`" to specify them. In the following example, the data contains categories `a`, `b`, and `c`, but there is a fourth category `d` that, while not observed in the data, is a possible outcome for the variable. The function `levels()` extracts the levels and `table()` produces a count of each category.

```
> y <- factor(c("a","a","b","c","c"), levels=c("a", "b", "c", "d"))
> levels(y)         # extract levels: "a", "b", "c", "d"
> table(y)          # print contingency table (note 0 counts
>                   # for level "d")
>
> as.character(y)   # remove "factor" property
> as.numeric(y)     # convert to numeric labels; remove
>                   # "factor" property
```

Factors are tricky in R as it is possible to convert objects, sometimes unknowingly, into factors causing problems with analysis. The function `is.factor()` can check for this object type. To convert factors back into character strings, use `as.character()`; factors can also be converted into numerical categories by using `as.numeric()`. This issue is discussed further in Section 4.6.

One useful application of factors is to convert a quantitative variable into a categorical one. For instance, say you have ages of people and you want to use age ranges in your analysis instead. The `cut()` function can handle this conversion:

```
> z <- runif(n=20, min=0, max=100)      # generate 20 random
>                                       # numbers from 0 to 100
>
> cut(z, breaks=c(0, 25, 50, 75, 100))  # ranges & labels:
>                                       # (0,25], (25,50],...
> cut(z, breaks=c(0, 25, 50, 75, 100), labels=c("a", "b", "c", "d"))
```

The argument "`breaks`" specifies what the cut points should be; make sure to include the minimum and maximum value in this vector. If labels are not specified, then R assigns a label for each range. Note that `cut()` has some optional arguments which allow you to set additional specifications.

4 WRITING PROGRAMS

Among base R and the additional packages, there are a wide range of available functions; however, you may need to write your own code. In this section, we describe the components of such programs beginning with conditional statements, continuing to loop structures, and ending with developing your own functions. These constructions allow you to write generalized statistical methods that you can later apply to multiple data sets and settings (or perhaps turn into your own R package). On a practical note, following good form when writing functions is critical to both readability and debugging: indent for nested expressions and comment liberally.

4.1 Conditional Statements

We introduced logical (or Boolean) types in Section 3.1 that take two values: `TRUE` or `FALSE` (the abbreviations `T` and `F` work too). To generate logical vectors, we need conditional statements that determine whether the specified conditions are true. In this section, we introduce these types of expressions and show their utility when writing functions. In Section 6, we will use them to clean and process data.

Most logical operators, such as checking for equivalence (==), nonequivalence (!=), "not" (!), inclusive "or" statements (|), and finally "and" statements (&) can be applied to both numerical and character objects. Those such as <, >, <=, and >= are interpreted mathematically and are appropriate for numerical objects only. More complex expressions can be constructed by combining operators. All of the expressions to construct conditional statements, along with examples, are listed in Table 5. Note that these operators will produce an `NA` if applied to a missing data value (e.g., `NA`).

The functions in Table 5, `any()`, `all()`, and `is.element()` apply specifically to vectors and are related to operations for sets: `union()`, `intersect()`, `setdiff()`, and `setequal()`. Finally, functions such as `is.na()` or `is.factor()` can be used to check for certain object or data types.

4.2 `if/else` Statements

If/else statements and loops are control structures; they can change the direction of program flow. We start with if/else statements in this section. These have the following structure: `if(`*CONDITION*`){`*PROCESS_A*`}else{`*PROCESS_B*`}`. If *CONDITION* is true, then *PROCESS_A* is implemented, otherwise (if applicable) *PROCESS_B* is executed. The `else` component of the statement is optional. If the processes are simple (i.e., one command), then the more efficient `ifelse()` can be used: `ifelse(`*CONDITION*`,` *COMMAND_A*`,` *COMMAND_B*`)`. An alternative is `switch()`, which is convenient if there are multiple "else" statements.

To compute the median value of a numeric vector, we can use an if/else statement. If the vector has an odd number of elements, the median is simply the middle value, otherwise, the average of the two "middle" values must be computed. We can determine the parity of a number using the modulo operator, `%%`. Note also how we use indentation within the if/else statement to make the program easier to read.

```
> # computing the median
> x <- c(1, 3, 5, 2, 7, 8)
> n <- length(x)
>
> if(n%%2==0){
>       # n is even
>       median.x <- sum(sort(x, decreasing=FALSE)[(n/2):(n/2+1)])/2
> }else{
>       # n is odd
>       median.x <- sort(x, decreasing=FALSE)[ceiling(n/2)]
> } # end if/else
> median.x
>
> # checking the code using built-in R functions:
> median(x)
> quantile(x, probs=0.5)
>
> # solution: median is 4
```

4.3 `for` Loops

Especially useful for resampling methods and simulation studies, a `for` loop repeats a process for a preset (fixed) number of iterations. The basic format is `for(`*INDEX* `in` *VECTOR*`){`*PROCESS*`}`. *INDEX* is the variable name that is the counter for the loops, *VECTOR* is a vector object or a vector itself which gives

the counter values to be looped through, and *PROCESS* is the set of operations to be repeated. For example, to compute the sample mean, we must start by adding the elements of a vector, best done by a loop. In the following code, we use the data `faithful` that contains observations on Old Faithful geyser activity at Yellowstone National Park in the United States. The first column lists the length of each eruption, and the second, the time until the next eruption, both in minutes. The code below utilizes a `for` loop to compute and print the sample mean of each column; observe that we again use indentation for clarity:

```
> for(i in names(faithful)){ # loop through each column of faithful
>
>    sum.x <- 0
>    for(j in 1:nrow(faithful)){ # loop through each row of column i
>       # compute cumulative sum
>       sum.x <- sum.x + faithful[j,i]
>    } # end j for loop (through rows)
>
>    # compute and print column average
>    print(paste("sample mean of column", i, "is:",
+       round(sum.x/nrow(faithful), digits=3), sep=" "))
> } # end i for loop (through columns)
>
> # checking our code:
> apply(faithful, 2, mean)
>
> # eruptions mean: 3.488
> # waiting mean: 70.897
```

We have two loops that are nested in the example above. For loop `i`, *VECTOR* is composed of character strings, the column names, whereas loop `j` is indexed by the row numbers.

The function `paste()` was used above to format the output generated by `print()` (or `cat()`). A second use is to print how many iterations have been completed. For instance, say the loop is indexed by `i` through a numerical vector:

```
> i <- 100
> if(i%%100==0) print(paste("iter",i,"is complete", sep=" "))
> # sep=" " indicates a space should separate each component in paste()
```

This prints `i` after every 100 iterations using the modulo operator `%%`. For loops with time intensive processes, generating such a statement is convenient to monitor your progress.

4.4 `while` Loops

All `for` loops can be written as `while` loops, the more general type of loop. The general format of a `while` loop is: `while(`*CONDITION*`){`*PROCESS*`}`, where *PROCESS* is repeated until *CONDITION* is false. We rewrite our inner `for` loop

in the previous section with a `while` loop to compute the average length of Old Faithful eruptions (from data `faithful`):

```
> x <- faithful$eruptions
>
> # average the numbers in vector x using a while() loop
>
> sum.x <- 0
> index <- 1 # set index
> while(index <= length(x)){ # loop repeated as long as
>                            # index is less than or equal to length(x)
>
>       # compute cumulative sum
>       sum.x <- sum.x + x[index]
>
>        if(index%%50==0) print(index) # print index value (for debugging)
>
>        # increase index
>      index <- index+1
>
> } # end while loop
>
> # compute average
> sum.x/length(x)
>
> # check code using built-in R function
> mean(x)
>
> # eruptions mean: 3.488
```

Prior to executing the code in *PROCESS*, the condition `index <= length(x)` is checked. The moment `index`, the counter, is greater than the number of observations in `x`, the loop ends. Note, again, that we use indentation to highlight the body of the loop. When the number of iterations may be large or each iteration is time consuming, `print(index)` which prints the number of completed iterations is helpful (in this case, we print after every 50 iterations). See the end of Section 4.3 for another example.

For a known number of iterations, a `for` loop is easier to write; however, this number is not always known in advance. Optimization algorithms are an example of such a situation. Then, a `while` loop or a `repeat` loop with a `break` clause is necessary. As a second example, we write a `while` loop which finds the first 10 prime numbers starting from 2:

```
> num.prime <- 0                    # number of primes
> int <- 2                          # first number to check if prime
>
> prime.list <- rep(NA, times=10)   # empty vector to hold prime
>                                   # numbers as we find them
>
```

```
> while(num.prime < 10){              # check whether we've found
>                                     # the 10th prime
>
>     num.divisor <- 0                # to count number of divisors
>                                     # for int
>     for(i in 2:int){
>         if(int%%i == 0) num.divisor <- num.divisor + 1
>     } # end for loop
>
>     # if number of divisors is 1, int is prime
>     if(num.divisor == 1){
>        num.prime <- num.prime + 1
>        prime.list[num.prime] <- int
>     } # end if statement
>
>     int <- int+1                    # need to test next integer
> } # end while loop
>
> prime.list # print prime numbers
> # solution: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

In this example, the `while` loop repeats nearly 30 times to extract the first 10 prime numbers; we could not use a `for` loop to run this program. It is possible (and easy) to accidentally construct an infinite `while` loop, therefore, testing your code is key. The escape key is useful to end such an infinite loop.

Above, we first constructed an empty vector, `prime.list`, then filled it within the `while` loop. This is more efficient than starting with a vector of length one and continuously appending to it as the loop progresses. See Section 4.6 for more details.

A final note: it is best to avoid loops where possible as they tend to utilize more memory in R. For simple processes, vectorization can be used as a substitute. Instead of looping through a vector or columns of a matrix, for instance, writing code that can directly implement a function element-wise is more efficient. Examples of vectorized functions include `nchar()` or `log()`; in contrast, `sum()` and `det()` are applied to entire objects. The functions `apply()`, `lapply()`, and `aggregate()`, which we describe in Section 6, are essential for vectorization as well.

4.5 Functions

Functions are self-contained pieces of code creating a subenvironment from which operations are run. They allow for more flexible programs. We are already using many such functions such as `length()`, `data.frame()`, and `nchar()`. In this section, we will learn how to write our own functions.

Let us start by studying a function already available to us `matrix()`. If we read the help page for this function, we find that `matrix()` has five arguments:

"data," "nrow," "ncol," "byrow," and "dimnames." Four of these arguments have default settings. For example, if you do not specify the number of columns, "ncol," it is automatically set to 1. Some arguments may not be applicable to all uses of the function; for such a case, the default value is set to NULL as in the argument "dimnames."

By calling the function matrix() without any specified argument, R prints a 1×1 empty matrix (i.e., NA) with no row or column names. If we forget to add the parentheses and type matrix, we get the function code; for most functions in the base R package, this may not be useful, however, within your own R code or when customizing add-on packages, this can be helpful.

To call (i.e., use) a function, specify the appropriate arguments. You can define them by argument name, as in matrix x below, or not, as in matrix y. If you omit the argument name, R assumes that you have listed the arguments in the same order as described in the help page.

```
> ?matrix   # load help function
> matrix()  # constructing a matrix with all preset values
> matrix    # print function code
>
> # x and y are identical matrices
> x <- matrix(data=c(5, 6, 3, 4), nrow=2, ncol=2, byrow=FALSE)
> y <- matrix(c(5, 6, 3, 4), 2)
```

To write our own functions, we follow the format:

```
FUNCTION_NAME <- function(ARGUMENT_1=DEFAULT_1, ARGUMENT_2=
DEFAULT_2,...){
   PROCESS
   return(OUTPUT)
} # end function
```

The arguments listed between the parentheses (possibly with default or NULL settings) are then used within the curly brackets. The return() function contains your R output; if you skip this step, R will return the results of the last command executed. If you want to include more than one object as output, specify them in list form using list() within return().

For our first example, we convert the code for computing the mean of each column of a data frame from Section 4.3 into a function. This will allow us to apply this program to any data frame, not just the Old Faithful data, faithful. We will call this function, mean.data.frame() and it will accept one argument, "data.set" with no default settings. Much of our initial program remains only with the generic "data.set" replacing the specific "faithful." At the end, we return the result as a data frame, each column representing one of the variables in data.set. To check our code, we apply the function to two data sets: faithful and attitude.

```
> mean.data.frame <- function(data.set){
>
>     # data.set: data frame with all numerical values
>
>     mean.col <- rep(NA, times=ncol(data.set))
>     n <- nrow(data.set)
>
>     for(i in 1:ncol(data.set)){ # loop through each column of data.set
>         sum.x <- 0
>            for(j in 1:n){ # loop through each row of column i
>                # compute cumulative sum
>                sum.x <- sum.x + data.set[j,i]
>            } # end j for loop (through rows)
>
>     mean.col[i] <- sum.x/n
>     rm(sum.x)
>  } # end i for loop (through columns)
>
>  # format output
>  result <- data.frame(t(mean.col))
>  colnames(result) <- names(data.set)
>  return(result)
> } # end function
>
> # testing function
> mean.data.frame(faithful) # Old Faithful geyser data
> mean.data.frame(attitude) # Chatterjee-Price Attitude Data
```

As with loops, it is important to use indentation and comments to make your code more readable. Furthermore, you can call other functions or even write functions within a function. It is helpful to divide a large task into smaller ones, writing a function per task. A final wrapper function can then execute the tasks in order, calling internal functions as needed. This is another technique that increases the readability of your code.

When a function is called, it opens a local environment in R. The objects defined within it are unavailable to the global environment (i.e., outside of the function) unless returned as output. However, global objects, even if not specified within the argument list, are accessible within the function environment. Apart from calling other functions within your function, it is generally inadvisable to refer directly to global objects within the local environment of the function. This limits the flexibility of the function. It is also good practice not to use similar object names within a function when they already exist as global objects; this may lead to confusion as a writer (and reader) of code along with leaving open the possibility of unintentionally altering those global objects.

As mentioned earlier, we often need to return multiple objects as output from a function; we can do this using `list()` within `return()`. In our second example, we use the code for listing consecutive prime numbers from Section 4.4. We will generalize the program to list a user-specified number of prime numbers beginning from a user-specified starting point.

Our function, `prime.number()` accepts two arguments: "`num.of.primes`" indicating how many prime numbers to list and "`start.num`" the smallest number at which to start checking for primes, the default being 2. Again, the code is essentially the same as before, but we replace "10" and "2" with the function arguments. We want to return both the vector of prime numbers and the number of values checked for the prime number property as a list. It is helpful to add names to the elements in the list to remind the user what each component represents. As with all lists, the operator `$` can be used to extract particular list elements.

```
> prime.number <- function(num.of.primes, start.num=2){
>     # num.of.primes: number of primes to find
>     # start.num: start checking for primes with this number
>
>     num.prime <- 0       # number of primes
>     int <- start.num     # first number to check if prime
>     # empty vector to hold prime numbers as we find them
>     prime.list <- rep(NA, times=num.of.primes)
>
>     while(num.prime < num.of.primes){  # kth prime found?
>
>         num.divisor <- 0        # count number of divisors for int
>         for(i in 2:int){        # check for prime number property
>           if(int%%i == 0) num.divisor <- num.divisor + 1
>         } #end for loop
>
>         # if number of divisors is 1, int is prime
>         if(num.divisor == 1){
>           num.prime <- num.prime + 1
>           prime.list[num.prime] <- int
>         } #end if statement
>
>         int <- int+1         # need to test next integer
>      } # end while loop
>
>    # results in the form of a list
>    return(list("primes"=prime.list, "num.iter"=int-start.num))
> } # end function
>
> # test function
> prime.number(num.of.primes=10)
>
```

```
> x <- prime.number(3, 4)
> x$primes    # extract primes: 5, 7, 11
> x$num.iter  # extract number of iterations
```

While writing functions, there are a few additional useful techniques. First, there are settings where all of the required arguments are not known in advance. For example, say you want to simulate values from a user-specified probability distribution. Every distribution requires a different set of parameters (e.g., for a normal distribution you need the mean μ and standard deviation σ, for Poisson, the rate λ). For your function to handle such variability, after listing all of the arguments, add an ellipsis "..." before ending the parentheses. The "..." represent the additional arguments. Whenever these arguments are required, insert "..."; the contents will be copied to those locations. When using the function, simply add these arguments in your function call in place of "....". R will know which are the extra arguments. See the function `samp.dist()` which simulates the sampling distribution of the sample mean in Section 9 for an example. (An example of a base R function with varying arguments is `list()`.)

Another useful pair of functions are `assign()` and `get()`. The first function allows you to assign names to new R objects dynamically and the second to extract existing R objects dynamically. You can also use `mget()` to obtain multiple R objects:

```
> for(i in 1:2){
>    assign(paste("vector",i,sep="."), faithful[,i])
> } # end for loop
> ls()
>
> mean(get(paste("vector",2,sep=".")))
```

For an example of using `get()` in practice, see `samp.dist()` in Section 9, where the sampling function is extracted for a user-specified distribution.

Finally, to check the run time of your code, use `date()` before and after a function call as in the example below:

```
> date()
> prime.number(num.of.primes=500)
> date()
```

4.6 Debugging and Efficiency

As programs become more complex and involve more lines of code, error checking—both of the syntactical and methodological variety—becomes harder. In this section, we outline some basic practices to reduce errors in your code; we also introduce the debugging functions within R.

Syntax errors will generally produce a warning message from R. Try the following examples:

```
> x <- c(5, 6,          # forgetting a ")"; R thinks you are continuing
>                       # to the next line
> for(i 1:6) print(i)   # forgetting "in" within the for loop
>
> y <- "a"
> y = "c"               # want to check if the value in y is equal to "c",
>                       # instead we have replaced "a" with "c";
>                       # no error message!
```

Methodology errors, however, are much harder to spot and rarely generate an error message. These errors require careful checking of code under a variety of conditions. For example, let us apply the function `mean.data.frame()`, which we wrote in Section 4.5, to two new data sets: air quality measurements in New York (`airquality`) and plant measurements (`iris`):

```
> mean.data.frame(iris)     # output: Species column NA
> mean.data.frame(airquality) # output: Ozone and Solar.R columns are NA
```

In both cases, the output is incomplete. The function stumbles on each data set for different reasons. With `iris`, R generates warnings (type `warnings()`). If we examine the data, we see the final column, "`Species`," is a categorical variable, treated as a factor; the mean is an irrelevant statistic here, hence the errors (and warnings). (See code below.)

With `airquality`, some of the data is missing. We can see that observations have missing data and also whether the missing data occurs within specific variables only. We find that 42 of the 153 observations contain missing values for at least one of the variables; we also see that only "`Ozone`" and "`Solar.R`" have missing values; all other variables are complete. The code is provided below:

```
> head(iris)                          # first 6 rows of iris
>
> head(airquality)                    # first 6 rows of air quality
> nrow(airquality)
> # row numbers containing NAs
> sum(!complete.cases(airquality))
> # determine whether any columns have NAs
> apply(apply(airquality, 2, is.na), 2, any)
```

We did not account for either of these scenarios in `mean.data.frame()`. To do so, we update our function, now calling it `mean.data.frame2()` below. Both errors have been eliminated in our second attempt.

```
> mean.data.frame2 <- function(data.set){
>     # data.set: data frame
>     mean.col <- rep(NA, times=ncol(data.set))
```

```
>     n <- nrow(data.set)
>
>     for(i in 1:ncol(data.set)){ # loop through each column of faithful
>
>           if(is.numeric(data.set[,i])){ # column numeric?
>
>              print(paste("column",names(data.set)[i],
+                    "is numeric"))
>              sum.x <- 0
>
>              for(j in 1:n){ # loop through each row of column i
>                # compute cumulative sum handling NAs
>                sum.x <- sum.x + ifelse(!is.na(data.set[j,i]),
+                                       data.set[j,i], 0)
>              } # end j for loop (through rows)
>
>              mean.col[i] <- sum.x/n
>              rm(sum.x)
>           } # end if statement
>
>         if(i%%2==0) print(paste(i,"th column complete", sep=""))
>       } # end i for loop (through columns)
>
>     # format output
>     # drop categorical variable results
>     result <- data.frame(t(mean.col[!is.na(mean.col)]))
>     # drop categorical variable column names
>     colnames(result) <- names(data.set)[!is.na(mean.col)]
>     return(result)
> } # end function
>
> # testing function
> mean.data.frame(faithful)    # initial function with ideal case
> mean.data.frame2(faithful)   # updated function with ideal case
>                              # (check if answers match)
> mean.data.frame2(iris)       # with categorical data
> mean.data.frame2(airquality) # with missing data
```

There are two features in `mean.data.frame2()` that are helpful with debugging and efficiency tests. The first is `print(paste("column",names(data.set)[j], "is numeric"))`; this command prints a message when the specified column is numeric indicating that the `if` condition is `TRUE`. We know that in the test case of `iris`, the first four columns are numeric and the fifth categorical. Therefore, this command should execute only four times. Printing indicators of progress throughout the function is helpful for following the flow of your program, especially if you have a lot of control structures, and to see when a function stops working. These can be commented out after you finish debugging (use #).

Another useful command is `if(i%%2==0) print(paste(i,"th column complete", sep=""))`. In this example, the column number is printed if the *i*th column is divisible by 2 (i.e., `i%%2==0`). For fast computations, this is unnecessary and slows down the execution of the program. For more complex programs, printing the iteration every so often will allow you to chart progress. Alternatively, you can print messages to a file which, if saved to a cloud, can allow you to monitor your program even if you are away from your computer (more helpful if your computer is not part of a larger server system or you cannot set up a remote desktop application).

Writing output to a file is beneficial especially if your program is memory-intensive. You can use `write.table()` to append output to an existing file as you generate results. In case your program crashes, writing to a file allows you to examine at least some of your results.

For simulated data, as will be described in Section 9, using `set.seed()` allows you to replicate your results when using random number generation. Retaining the same simulated values each time you test your function is helpful for debugging.

A common error occurs when character string vectors (or, for example, a particular column in a data frame) are incorrectly treated as factors. As they add attributes to a vector, some character string operations may execute incorrectly when used with factors. For example, selecting rows based on row names can fail if the vector of row names is treated as a factor. In such a case, using `as.character()` can fix the problem, although it is better to find the command where the problem originated (refer to Section 3.4 for more on strings and factors). Furthermore, when reading in data from a file, character string columns can automatically be converted to factors unless otherwise specified (see Section 5 for more details on avoiding this). A related issue is when factor levels are unintentionally converted into numeric values in place of the original category labels. Checking for these common problems is key if you frequently work with categorical data.

Another issue occurs with row numbers on data frames. Row numbers (row names) on a data frame are essentially attached to the original observations they label. After sorting and manipulating data, these row names will no longer be in order or match the total number of rows actually in the data set. This may cause problems when you execute commands that refer to the row number. Therefore, after the data cleaning step, it is important to renumber your data frame (i.e., `rownames(DATA_FRAME) <- 1:nrow(DATA_FRAME)`).

The techniques just described are *ad hoc*; R, however, has built-in debugging functions: `trace()`, `untrace()`, `browser()`, `traceback()`, and `debug()`. These functions allow you to step through your code to locate errors.

As mentioned earlier, to check for run times, you can use `date()` to time stamp your program. This is helpful if you to identify whether portions of your program are time intensive and may need to be more efficiently written. Loops tend to be slow in R; however, functions such as `apply()`, `lapply()`,

and `aggregate()` can reduce computing times (see Section 6 for details). Vectorization of processes, as described in Section 4.4, can improve efficiency as well.

Another useful trick to reduce run times is to construct empty vectors and load them in as you progress as opposed to continually appending an element to an existing vector. We do this in our function `prime.number()` in Section 4.5 when defining the R object `prime.list` as an empty vector of length `num.of.primes`. This is more efficient than defining `prime.list <- NA` and using `prime.list <- c(int, prime.list)` instead of `prime.list[num.prime] <- int`. For more on how memory is allocated and used in R, see Section 2.2.

For complex programs, it is best to split the code into smaller functions, each with one main task. Then, at the end write a wrapper function which simply calls the sub-functions. This practice also helps with debugging as you can focus on one task in your program at a time. (Note, however, if you are having issues with memory, this suggestion may add to that problem.) Moreover, it is good practice to test your code as you progress, not after you have finished writing your program. Finally, do not use the same object name for global and local environments, as this may cause both computational and testing errors.

Text editors that color code your programs (e.g., orange for comments, green for functions, etc.) and match parenthetical/bracketed expressions can help immensely with syntax errors. Using indentation to indicate nested expressions more clearly is also a worthwhile practice. Finally, it cannot be repeated enough that commenting your code as you program is vital to both the debugging process and the reusability of your functions.

5 INPUT AND OUTPUT

Three file types can be uploaded into R: a workspace, code, and data. For the first two, refer to Section 2.3. As mentioned in the introduction, to be a self-containing text, we use the data already in R. Therefore, we start with file output before considering file input.

For example, let us use the `airmiles` data, a vector containing the number of miles people traveled on commercial U.S. airlines each year between 1937 and 1960. To write this vector to a file, use `write()`. First, specify the R object; second, the file name. (Note: make sure you know which directory your file is being written to as discussed in Section 2.3). Now, we can upload the vector using `scan()`:

```
> as.vector(airmiles)                          # vector of data
> write(as.vector(airmiles), file="airmiles.txt")  # output to a file
> scan("airmiles.txt")                         # input file as vector
```

For data frames, the function `write.table()` is used instead of `write()`. This function has numerous arguments, the most important of which are: "`x`," the

data; "file," the file name; "append," whether or not the data should be added to an existing file (if FALSE, any existing file will be replaced); "sep," the character which separates values for each column within a row (common characters are a comma ",", a tab "\t", or a space " "); "row.names," whether or not row names should be written (best to set this to FALSE); and "col.names," whether the variable names of the data frame should be included or not. The "append" argument is especially useful when writing output as it is generated within a program (refer to Section 4.6).

You can then upload your data to R using read.table() where the basic arguments are the file name, the separating character, and whether the first line is a column heading. We use the Old Faithful geyser data, faithful, below as an example:

```
> head(faithful)      # print first 6 rows of faithful
> write.table(faithful, file="faithful.txt", append=FALSE, sep=",",
+           row.names=FALSE, col.names=TRUE)
> y <- read.table("faithful.txt", sep=",", header=TRUE)
> head(y)             # print first 6 rows of y
```

The function read.table() has additional arguments to specify the data type (character, logical, numerical, factor) for each column ("colClasses"), specifying the number of lines to skip in a file before beginning to read the data ("skip"), and whether character string columns should be treated as factors ("stringsAsFactors," the default is TRUE; this is generally best set to FALSE). The help page can elaborate on these function parameters. For CSV files, where variables are separated by commas (i.e., sep=","), the function read.csv() is an equivalent alternative. To upload files generated by other statistical software packages, such as Minitab, SAS, or SPSS, use the package *foreign*; for Excel files, install *xlsx*.

We end this section with a few miscellaneous functions. Before printing to a file, some formatting of the output may be required; the functions paste (), format(), and the rounding functions may be useful. R can also handle command line input using the function readline() when R is used interactively (see ?interactive for more information). To check whether your data has uploaded correctly, head(), tail(), colnames() and dim() can give you some basic information. Note that as # is the comment indicator in R, make sure the symbol does not show up in your data files. This can occur with geographical data, in particular with street addresses; remove every instance of # before uploading your data. The functions in this section are compiled in Table 6.

6 DATA PROCESSING

After reading in your data as described in Section 5, you may need to clean it. R has a number of functions to help process data. Some of these have been streamlined and improved upon in the package *plyr* (see CRAN for more details and refer to Table 6, for a list of data processing functions).

To work with vectors instead of data frames, use the function `attach()`. This splits the data frame into its component vectors with the column names repurposed as object names. Use `detach()` to undo this step.

To sort a vector, use `sort()`. You can sort in ascending or descending order along with returning a vector of the index using `index.return=TRUE`; this function can also handle missing data, `NA`. To sort your data frame by columns, use `order()` instead:

```
> # order smallest to largest
> ascend.sort <- faithful[order(faithful$eruptions),]
> # order largest to smallest
> descend.sort <- faithful[order(-faithful$eruptions),]
>
> # reassign row numbers
> rownames(ascend.sort) <- 1:nrow(ascend.sort)
```

For sorting by multiple columns, add additional columns within `order()` separated by commas.

In the example above, we sorted the `faithful` data by the column "`eruptions`." This operation shifts the row numbers of the data frame. To avoid problems in later stages, it is prudent to renumber the rows after cleaning your data.

To clean categorical data, commands borrowed from UNIX, such as `grep()`, are available. As described in Section 3.4, numerical variables can be converted into categorical variables by range using `cut()`.

R generally ignores observations containing missing data when running analyses. To check and extract missing data, the functions `is.na()`, `complete.cases()`, and `na.omit()` are useful. For imputation, the packages *mi*, *mice*, *mitools*, *MImix*, and *pan* are available.

If you want to split the data into training and test sets, you can use the `sample()` function (see Section 9). We use the Chatterjee–Price Attitude Data here `(attitude)`:

```
> training <- rep(TRUE, nrow(attitude)) # all observations in training set
> # randomly select 5 observations to be in the test set:
> training[sample(1:nrow(attitude), size=5, replace=FALSE)] <- FALSE
>
> # append vector training to attitude data frame
> attitude2 <- data.frame(attitude, "train.set"=training)
```

Then, use the function `subset()` to select observations you want to use. Alternatively, you can use the `split()` function to separate out the data into a list based on a category:

```
> # extract test set from attitude2
> attitude2[!attitude2$train.set,]
> # extract test set using subset()
> test.set <- subset(attitude2, train.set==FALSE)
```

```
>
> # split attitude2 into list with two components:
> #  'TRUE' (training set) and 'FALSE' (test set)
> attitude3 <- split(attitude2, f=attitude2$train.set)  # split data
> attitude3$'FALSE'                                     # extract test set
```

Note that `subset()` and `split()` can be used to subset data under any set of conditions (see Table 5 for a list of logical operators and related functions).

You can append new columns to your data frame if you, for instance, need to transform your data. Below is an example with the `iris` data on plant measurements:

```
> # add log(Sepal.Length) to data frame
> iris2 <- data.frame(iris, "log.Sepal.Length"=log(iris$Sepal.Length))
```

Provided that two data frames have the same columns (with the same column names), we can attach the rows together to create one larger data frame:

```
> attitude.train <- attitude[attitude2$train.set,]    # extract training set
> nrow(attitude.train)
> attitude.test <- attitude[attitude2$train.set,] # extract test set
> nrow(attitude.test)
>
> # append data frames
```

Method 1

a	b	c
1	10	yes
6	8	no

Method 2

a	b	c
1	10	yes
3	7	<NA>
6	8	no

Method 3

a	b	c
1	10	yes
3	7	<NA>
4	NA	yes
6	8	no

```
> attitude.full <- rbind(attitude.test, attitude.train)
> nrow(attitude.full)
```

Note that `rbind()` works even if the column order is different between the data frames or if columns have different data types.

If you have two data sets with some common columns, you can match observations and combine them into one data frame using `merge()`:

```
> x <- data.frame("a"=c(1, 6, 3), "b"=c(10, 8, 7))
> y <- data.frame("a"=c(1, 6, 4), "c"=c("yes", "no", "yes"))
>
> merge(x,y)                              # method 1
> merge(x,y, all.x=TRUE)                  # method 2
> merge(x,y, all.x=TRUE, all.y=TRUE)      # method 3
```

Observe that the three function calls to `merge()` above produce different results.

With method 1, only the common rows (matched by column "a") are merged. In method 2, we specify that all observations in x should be included. Note that the third observation in x is not in y; consequently, after merging, the value in column "c" for that observation is missing. Similarly, in method 3, we specify that all observations in x and y should be included; consequently, missing values are automatically inserted where appropriate.

Finally, functions can be applied to data frames (or vectors) separately by a category using aggregate(). This function has the following form: aggregate(*R_OBJECT*), by=list(*CATEGORY_LIST*, FUN=*FUNCTION*). In the next example, we use the iris data to compute the maximum value of each quantitative variable for each category in Species:

```
> # compute max for each species
> aggregate(iris[,1:4], by=list(iris$Species), max)
```

Here, *CATEGORY_LIST*, is list(iris$Species); that is, the computations on iris[,1:4] should be completed separately for each species of iris. This must be in list form; to include more than one category, separate terms with a comma within list().

The functions subset(), merge(), apply(), lapply(), and aggregate() can help simplify your code for both data cleaning and computational operations. By reducing your dependence on explicit loops, they also increase efficiency.

7 EXPLORATORY DATA ANALYSIS

After processing your data, exploratory data analysis is the next step. For example, we can calculate the mean of a vector with mean() and the standard deviation with sd(); however, without specifying additional arguments, they will not work with data containing missing values (NA). By adding "na.rm=TRUE" to the function call, the statistic can be calculated:

```
> mean(airquality$Ozone) # vector has NA values
> mean(airquality$Ozone, na.rm=TRUE)
>
> sd(airquality$Ozone, na.rm=TRUE)
```

For categorical data, construct a contingency table using table():

```
> # counts of each species of iris in the data set
> table(iris$Species)
```

When tabulating multiple categorical variables, use a comma between each vector in the arguments for table().

The functions apply() and aggregate() are helpful here too (see Section 6 for use in data processing). To compute a statistic separately for each row or column, use apply():

```
> # compute average for each column in faithful data set
> apply(faithful, 2, mean)
```

For lists, `lapply()` is the analogous function.

Alternatively, to compute a statistic for a numerical variable separately for each concomitant category, use `aggregate()`:

```
> # compute standard deviation of sepal length for each
> # iris species separately
> aggregate(iris$Sepal.Length, by=list(iris$Species), sd)
```

See Table 7 for a list of functions to calculate summary statistics (numerical and categorical) and basic graphs.

8 STATISTICAL INFERENCE AND MODELING

In this section, we cover basic statistical analyses: hypothesis testing and linear regression. See Table 2 for information on more advanced methods.

8.1 Hypothesis Testing

R includes a basic set of hypothesis tests; Table 8 lists the most common ones. In the next example, we focus on the one-sample t-test. Say we want to test whether the average length of an Old Faithful geyser eruption is greater than 2 min:

$$H_0: \ \mu \leq 2 \text{ min.}$$
$$H_a: \ \mu > 2 \text{ min.}$$

We can answer this question using the `faithful` data with a one-sample t-test. The test statistic is $t = (\overline{x} - \mu)/(s/\sqrt{n})$ with $n - 1$ degrees of freedom. In R, function `t.test()` will run the test:

```
> # one-sample t-test, alternative hypothesis: ">"
> x <- t.test(faithful$eruptions, alternative="greater", mu=2,
+             conf.level=0.95)
> x
> names(x)
> x$statistic
```

We save the output in `x`. It includes, among other items, the test statistic, degrees of freedom, p-value, and confidence interval:

```
    One Sample t-test
data: faithful$eruptions
t = 21.498, df = 271, p-value < 2.2e-16
alternative hypothesis: true mean is greater than 2
95 percent confidence interval:
 3.373559   Inf
sample estimates:
mean of x
 3.487783
```

While the output is formatted when `x` is printed, it is actually saved in list form. The list component identifiers are generated with `names(x)`. For instance, to extract the test statistic, enter `x$statistic`.

8.2 Regression

We now introduce linear regression and diagnostic functions. For the first example, we use the `attitude` data (Chatterjee–Price Attitude Data) containing seven variables on employee performance: "`rating`" (y), "`complaints`," "`privileges`," "`learning`," "`raises`," "`critical`," and "`advance`." The variables are numerical with each representing the fraction of positive reviews for an employee on each measure. We start by examining scatterplots of all pairs of variables using `pairs()` as shown in Fig 1. From this plot, rating and complaints seem to have the strongest linear relationship which is confirmed by computing the pairwise correlations with `cor()`.

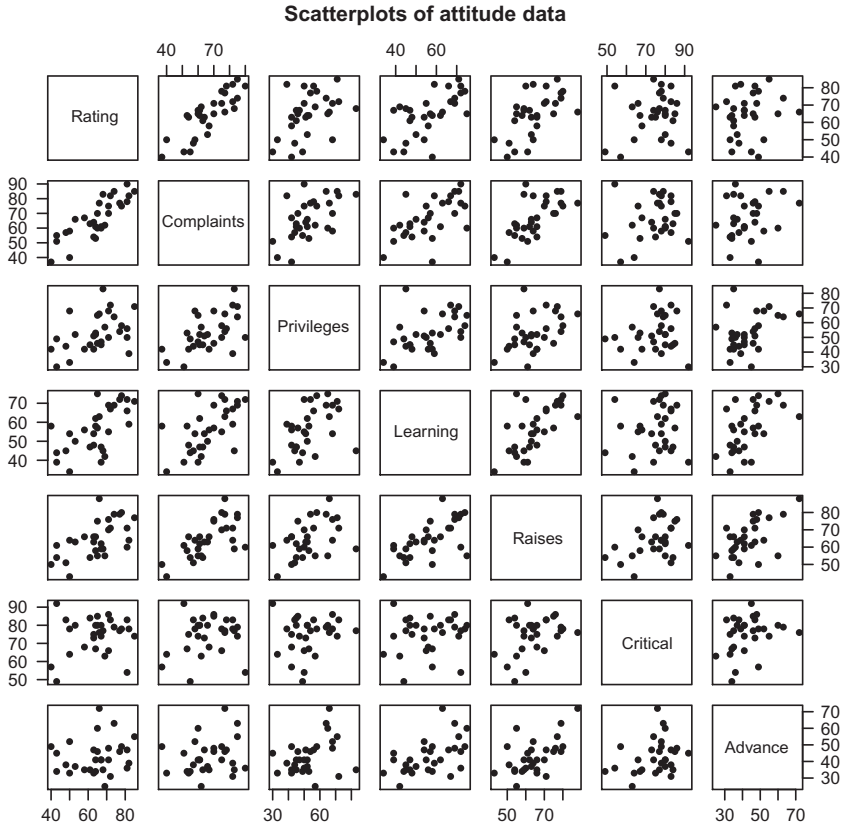

FIGURE 1 Scatterplots of all pairs of variables in `attitude` data.

Consequently, we fit:

$$y_{rating} = \beta_0 + \beta_1 x_{complaints} + \varepsilon. \quad (1)$$

Use `lm()` to run a linear regression; it returns a linear model object in list form as output. The function is called using the following format: `lm(FORMULA, data=DATA_FRAME, subset=LOGICAL_VECTOR, weights=VECTOR)`. The argument *FORMULA* expresses the model to be fit and takes the form: *RESPONSE ~COVARIATE_1 +COVARIATE_2* and so forth. The symbol "~" separates the response variable on the left with the covariates on the right. The remaining arguments are optional. If the response and covariates are not vectors in the workspace (e.g., you did not use `attach()`), specify the data frame in "`data`"; if you want to use only a subset of the observations in your regression (e.g., training and test sets) input a logical vector to the argument "`subset`"; finally, for weighted least squares, add a weight vector to "`weights`." Note that observations with any missing values among the variables selected for your model will be omitted from your analysis. The code for fitting (1) is below:

```
> # scatterplots and correlations for all pairs of variables
> pairs(attitude, las=TRUE, main="Scatterplots of Attitude Data",
+       pch=20)
> cor(attitude)
>
> # rating regressed against complaints
> model.1 <- lm(rating ~ complaints, data=attitude)
> names(model.1)          # components of linear model object model.1
>
> summary(model.1)        # coefficients, t-tests for slopes, etc.
> anova(model.1)          # overall F-test
```

The components of the linear model object, `model.1`, can be listed with `names()` and accessed as you would any list. For a summary of the coefficients and *t*-tests for slopes, use `summary()`. The output of `summary()` varies by the type of object it is applied to. Note that `summary(attitude)` and `summary(model.1)` give very different results; the first yields a six-number summary (see Table 7) while the second generates an overview of a fitted model. There are quite a few functions that behave this way in R; that is, they have multiple purposes. To get information on the use of summary for linear models, type `?summary.lm` instead of `?summary` into the command line. Finally, for the accompanying ANOVA table, use `anova()`.

For a simple regression model, we can plot the regression line on the scatterplot using `plot()` to plot the points, `coef()` to extract the regression coefficients, and `abline()` to plot those coefficients. See Fig. 2 for the graph. Like `summary()`, `plot()` yields a different result when applied to linear model objects. As we see in Fig. 3, `plot(model.1)` graphs model diagnostics (see `?plot.lm` for more details). The code is provided below:

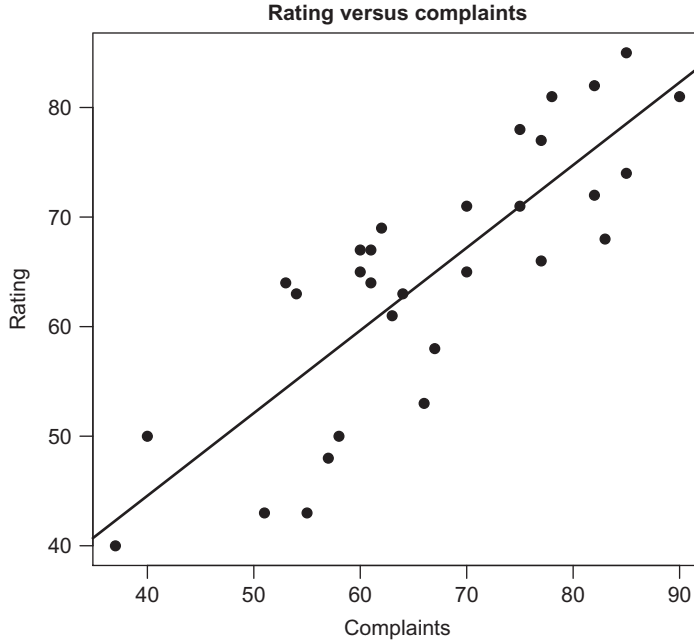

FIGURE 2 Model 1: Plotting the regression line.

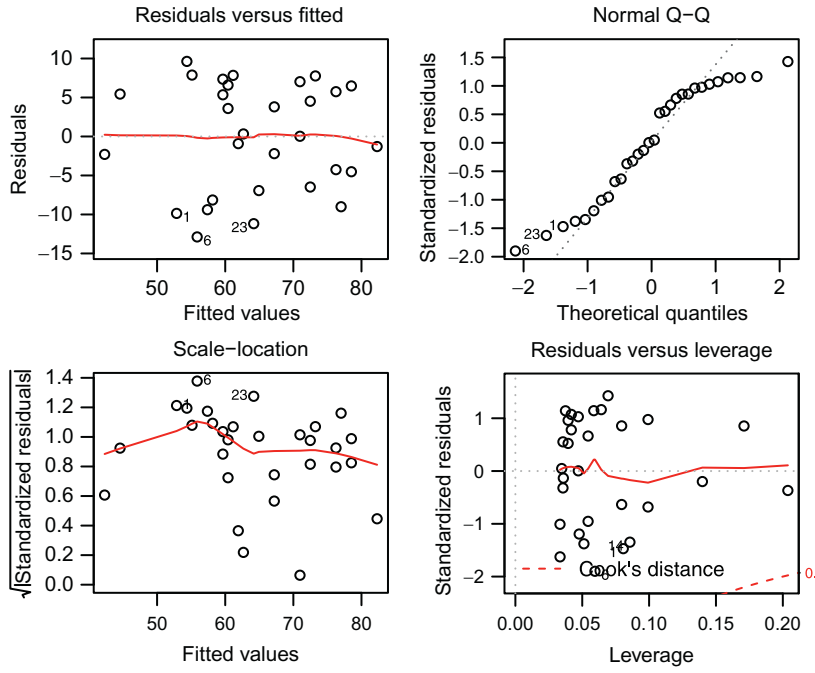

FIGURE 3 Model 1: Diagnostic plots.

```
> # scatterplot with regression line
> plot(attitude$complaints, attitude$rating, pch=19,
+         main="Rating vs. Complaints",
+         xlab="complaints", ylab="rating", las=TRUE)
> abline(coef(model.1), lwd=2)
>
> # model diagnostics
> par(mfrow=c(2,2))
> plot(model.1)
```

Residuals can be extracted one of two ways: directly from the linear model object or from `residuals()`. Other diagnostic measures such Cook's distances can be computed with `influence.measures()`, some of which appear in Fig. 3.

There are two settings to predict the response variable, $\hat{y}$: prediction for the observations used to fit the model and prediction for new observations. For the former case, as with residuals, you can extract them from the linear model object directly or use `predict()`. For new observations, construct a data frame with the relevant covariates and with column names identical to those used by the linear model. Then, use the function `predict()` with this data frame specified in the "`newdata`" argument. Note that we can also obtain standard errors along with confidence or prediction intervals at various α-levels. The code for diagnostics and fitted values are given below:

```
> # residuals, two ways
> model.1$residuals
> residuals(model.1)
>
> # other diagnostic measures
> influence.measures(model.1)
>
> # predictions for data used to fit model, two ways
> model.1$fitted.values
> predict(model.1)
>
> # predictions for new data
> attitude.data <- data.frame("complaints"=c(45, 67, 25))
> predict(model.1, newdata=attitude.data, se.fit=TRUE,
+           interval="prediction", level=0.95)
```

There are a few alternatives when expressing the regression equation. To use all of the variables as covariates, apart from the response, add a period as in `model.2` below. To fit a model without a y-intercept (not recommended), add `-1` along with the covariates as in `model.3`. Finally, you can transform your data directly in the `lm()` formula as in `model.4`. Note that nested models can be compared with partial F-tests by repurposing the function `anova()`: specify the

smaller model first, then the second model separated by a comma. Code for these operations are next:

```
> # all variables used
> model.2 <- lm(rating ~ ., data=attitude)
> anova(model.1, model.2)    # partial-F test
>
> # no y-intercept
> model.3 <- lm(rating ~ -1 + complaints + privileges, data=attitude)
> # transform "privileges"
> model.4 <- lm(rating ~ complaints + log(privileges), data=attitude)
```

Until now, we have considered only numerical covariates; fortunately, categorical covariates are handled identically. However, one must be careful to check whether a column is being treated as a qualitative variable if the categories are numbers. In our second example, we use the data set `ToothGrowth` that records the length of guinea pig teeth (column "`len`") after they receive Vitamin C through orange juice or ascorbic acid ("`supp`") at one of three dosages ("`dose`"). When we examine this data, we see that the response ("`len`") is numeric, the source of the Vitamin C is categorical and listed as a character string, and the dosage is also categorical but listed in the data frame as a numerical variable.

Every statistical software has a slightly different way of expressing regression coefficients with categorical covariates. To see how R conveys the results, we fit a one-way ANOVA model with "`len`" regressed against the categorical "`supp`":

```
> # len is numerical; supp is a category (one-way ANOVA)
> model.5 <- lm(len ~ supp, data=ToothGrowth)
```

We analyze the output from this model `summary(model.5)` next:

```
Call:
lm(formula = len ~ supp, data = ToothGrowth)

Residuals:
     Min       1Q  Median      3Q     Max
-12.7633 -5.7633  0.4367  5.5867 16.9367

Coefficients:
            Estimate Std. Error t value  Pr(>|t|)
(Intercept)   20.663      1.366  15.127 <2e-16 ***
suppVC        -3.700      1.932  -1.915  0.0604 .
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.482 on 58 degrees of freedom
Multiple R-squared: 0.05948,Adjusted R-squared: 0.04327
F-statistic: 3.668 on 1 and 58 DF, p-value: 0.06039
```

There are two categories of "supp": VC and OJ; however, the category OJ is dropped from the output. The *y*-intercept term, 20.663 is the sum of the intercept and the estimate for OJ. The coefficient for VC is then the *difference* between VC and OJ.

Now, let us add in "dose." If we type `lm(len ~supp + dose, data=ToothGrowth)`, R will treat "supp" as a categorical variable but "dose" as a numerical variable. To ensure that `dose` is also treated as a categorical variable, we turn it into a factor (or character string). Finally, to include interaction terms, add the lower order terms and use the multiplication operator `*` for the interaction effects as in `model.6` below:

```
> model.6 <- lm(len ~ supp + as.factor(dose) + supp*as.factor(dose),
+               data=ToothGrowth)
```

To evaluate your models, `AIC()` and `BIC()` are useful. For stepwise regression, use `step()`. Test for multicollinearity with `vif()` in the package *car*. To test for autocorrelation in the residuals, use the Durbin–Watson test, `dwtest()` in the package *lmtest*. Finally, for all subsets regression, use the function `leaps()` within the package of the same name.

As listed in Table 2, R can handle more complex models. For one- or two-way ANOVA models, `lm()` can be used along with `oneway.test()` and `aov()`. Generalized linear models such as logistic regression or Poisson regression can be fit with `glm()`, generalized additive models with `gam()`. Fit mixed effects models with `lmer()` from the package *lme4*. Hierarchical generalized linear models can be fit with the *hglm* package.

9 SIMULATION

The first step in a simulation is the ability to generate random numbers. It is the basis of procedures from drawing a random sample from distributions to the bootstrap and jackknife. Internally in R, the command `.Random.seed` produces the random number generation states. The mechanics behind this function is beyond our scope; type `?.Random.seed` for the available methods (default is Mersenne–Twister).

To sample from a discrete distribution, use `sample()`. The arguments required are "x" to specify the vector to sample from (any data type); "size" for the number of draws; "replace" to indicate whether the sampling should be with or without replacement; and "prob" to indicate the probability of selection for each element in "x" (if unspecified, the distribution is assumed to be uniform). For example, let us simulate 1000 rolls of two dice, one fair and one weighted and plot the results (see Fig. 4).

```
> par(mfrow=c(1,2))
>
> # simulate and graph 1,000 rolls of a fair die
```

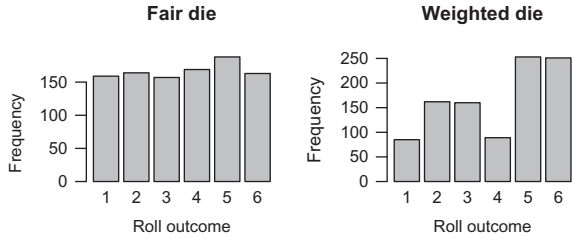

FIGURE 4 Sample simulation results for rolling dice.

```
> x <- sample(1:6, size=1000, replace=TRUE)
> barplot(table(x), main="Fair Die", las=TRUE,
+          ylab="frequency", xlab="roll outcome")
>
> # simulate and graph 1,000 rolls of a weighted die
> y <- sample(1:6, size=1000, replace=TRUE,
+             prob=c(1/12, 2/12, 2/12, 1/12, 3/12, 3/12))
> barplot(table(y), main="Weighted Die", las=TRUE,
+          ylab="frequency", xlab="roll outcome")
```

Each time we run the code above, a new set of observations are drawn. To replicate results (useful when debugging code), use `set.seed()` to set the random number generation seed.

A number of probability distributions exist within base R; add-on packages increase the available options (see Table 9 for a brief list). There are four possible functions for each distribution: density (`d`), cumulative distribution (`p`), quantile (`q`), and random draws (`r`). The normal distribution, for instance, has `dnorm()`, `pnorm()`, `qnorm()`, and `rnorm()`. For distributions not in base R or an accompanying package, you will have to use other methods such as rejection or importance sampling.

Apart from Bayesian methods, probability distribution functions are useful for simulation studies, especially when exploring asymptotic properties of statistics. In the function `samp.dist()` below, we simulate the sampling distribution of the sample mean, $\overline{x}$, from observations drawn from a given distribution. The results are plotted in Fig. 5. In the code, observe that we use the "..." notation in the function argument list. This allows any distribution to be used without having to preset all of the distribution parameters within the list of arguments for `samp.dist()`. Then "..." is passed on to any function call which requires those additional arguments (in this case, the sampling function). Furthermore, `get()` is used to obtain the sampling function for the specified distribution. (See Section 4.5 for more discussion on these two notes.)

```
> # sampling distribution for mean of a random variable
> samp.dist <- function(n, k,dist,...){
>
```

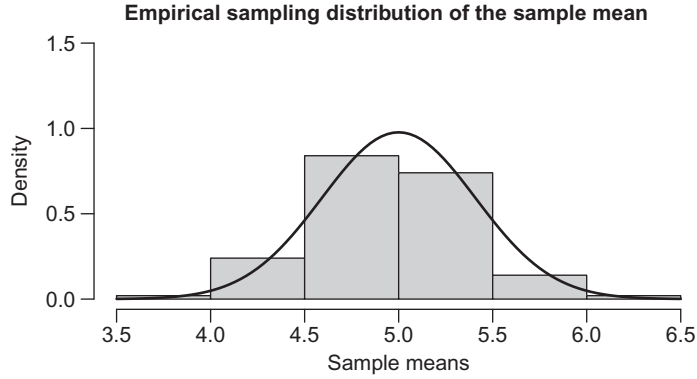

FIGURE 5 Sampling distribution simulation results.

```
>     # n: sample size; k=number of simulation runs,
>     # dist=distribution (norm, pois, etc.),
>     # ... = parameters for distribution in "dist"
>
>     # stop function if input invalid
>     if(any(c(k, n) <= 0)){return("error!")}
>
>     mean.vector <- rep(NA, times=k) # empty results vector
>
>     for(i in 1:k){
>          # extract sampling function
>          random.draws <- get(paste("r", dist, sep=""))
>            x <- random.draws(n=n,...)  # simulate x_1,...,x_n
>            mean.vector[i] <- mean(x)   # compute and save x-bar
>            rm(x)                       # remove x-bar
>     } # end for loop
>
>     return(mean.vector) # function output
> } # end function
>
> # run simulation for Poisson distribution, rate parameter = 5
> z <- samp.dist(n=30, k=100, dist="pois", lambda=5)
>
> # histogram of sample means (density not frequency)
> hist(z, freq=FALSE, ylim=c(0, 1.5), las=TRUE, col="gray80",
+       main="Empirical Sampling Distribution of the Sample Mean",
+       xlab="sample means")
> # add normal density curve to histogram
> curve(dnorm(x, mean=5, sd=sqrt(5/30)), add=TRUE, lwd=2)
```

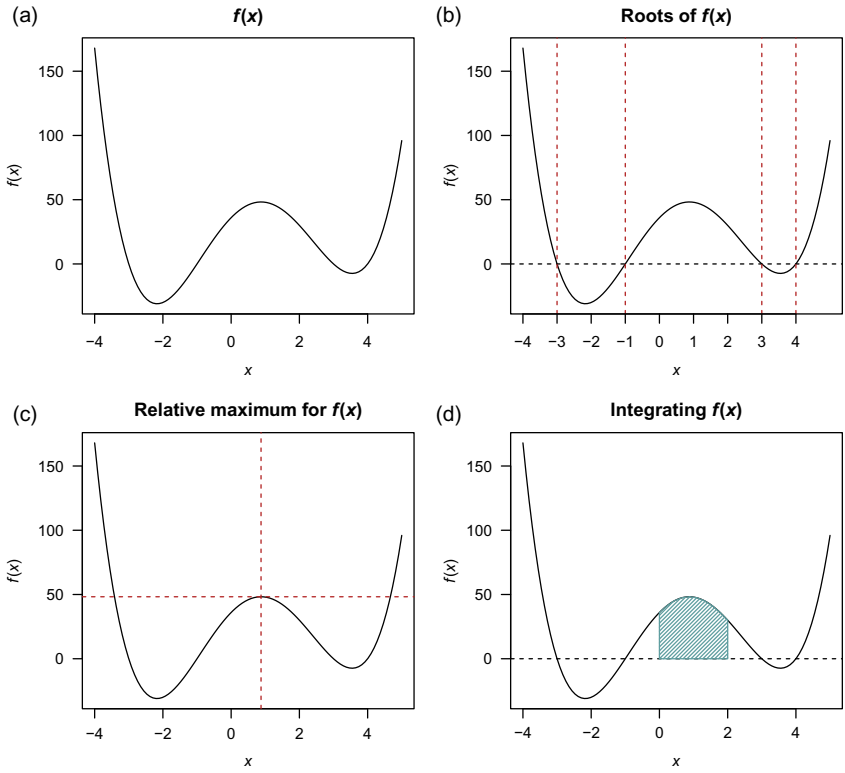

FIGURE 6 Examples of roots, relative maxima, and integration

10 NUMERICAL TECHNIQUES

We now introduce the following methods in R: finding roots of functions, optimization, derivatives, and integrals. These numerical methods are appropriate when analytical solutions are difficult or impossible to compute. Table 10 summarizes these functions.

We will use the polynomial $f(x) = x^4 - 3x^3 - 13x^2 + 27x + 36$ to demonstrate the numerical techniques in this section:

```
> f <- function(x){return(x^4-3*x^3-13*x^2+27*x+36)}
>
> # plot function
> curve(f, from=-4, to=5, main="f(x)", las=TRUE)
```

Figure 6 illustrates the output for each technique, beginning with the graph of $f(x)$.

The function `polyroot()` calculates the complex roots of a polynomial when coefficients are listed in ascending order; however, for a general one-dimensional function, `uniroot()` should be used instead.

```
> # complex roots of a polynomial:
> # enter vector of coefficients in ascending order
> polyroot(c(36, 27, −13, −3, 1))
>
> # for any function: find real root within specified range
> uniroot(f, interval=c(−2, 0))
>
> # graphing
> curve(f, from=−4, to=5, main="Roots of f(x)", las=TRUE, xaxt="n")
> abline(h=0, lty=2)
> abline(v=c(-1, −3, 3, 4), col="firebrick", lty=2)
> axis(side=1, at=−4:4)
>
> # roots at x= -1, −3, 3, 4
```

To use `uniroot()`, you must specify a search interval where the function values at the endpoints are of opposite sign; then, the function will obtain one root within the given interval. For more complex problems, the package *rootSolve* is helpful. Similar to other functions, `uniroot()` structures its output in the form of a list; use `names()` to determine the list components.

Next, we search for relative minima or maxima using the function `optimize()` (equivalently, `optimise()`). This function carries out one-dimensional optimization only. As with `uniroot()`, you must specify a search interval:

```
> # find maximum of function in specified range
> optimize(f, interval=c(0, 2), maximum=TRUE)
>
> # graphing
> curve(f, from=-4, to=5, main="Relative Maximum for f(x)",
+       las=TRUE)
> y <- optimize(f, interval=c(0, 2), maximum=TRUE)
> abline(h=y$objective, col="firebrick", lty=2)
> abline(v=y$maximum, col="firebrick", lty=2)
>
> # maximum at 48.248 when x=0.876
```

The function `optim()` can also handle maximum likelihood estimation among other optimization problems, `constrOptim()` is useful for constrained optimization, and `nlm()` for nonlinear minimization.

The final two methods we will discuss are derivatives and integrals. Let us say we want to determine the derivative of $f(x)$ at $x = 4$. Then, using the function `grad()` within the package *numDeriv*, we obtain

```
> # numerical derivative
> require(numDeriv)
> grad(f, x=4)
>
> # f'(4)= 35
```

which is equivalent to the analytical solution: $f'(x) = 4x^3 - 9x^2 - 26x + 27$, when $x = 4$. Within this package, use `jacobian()` for the Jacobian matrix and `hessian()` for the Hessian matrix to obtain partial derivatives.

To integrate over one dimension, use `integrate()`. For example, $\int_0^2 f(x)\,dx$ can be computed as follows:

```
> # integrate between lower and upper limits
> # (limits can be -Inf or Inf )
> integrate(f, lower=0, upper=2)
>
> # graphing
> curve(f, from=-4, to=5, main="Integrating f(x)", las=TRUE)
> abline(h=0, lty=2)
> x <- seq(0, 2, length=10)
> polygon(x=c(0,x,2), y=c(0,f(x),0), density=40, col="cadetblue",
+         border=TRUE) # shade in area under curve
>
> # area: 85.733
```

The limits of integration can be `-Inf` or `Inf`; insert these limits instead of a very large number for more accurate results. For multidimensional integration, use `adaptIntegrate()` in the package *cubature*.

11 ANNOTATED REFERENCES

Set Up

- The basic R package can be downloaded for free from the CRAN Website: http://cran.r-project.org/. Additional packages can be installed through your R console if you have administrative privileges on your computer or through the CRAN Website.
- For an "enhanced" version of R, with more workspace design features, install RStudio after installing R: http://www.rstudio.com; this program is also free.
- To connect R with the type-setting program LaTeX, use `Sweave()` or install and use the package *knitr*.

Text Editors

Apart from Notepad on your computer (or the built-in R text editor on Apple computers), there are other useful text editors which make reading and

debugging your code easier through color coding (comments and code in different colors, etc.) and parenthesis matching (useful for nested expressions). As mentioned above, RStudio has this capability but alternatives include GNU Emacs (http://www.gnu.org/software/emacs/), RWinEdit (http://www.winedt.com, then for the R add-on go through CRAN and download package `RWinEdt`), Tinn-R (http://tinn-r.soft112.com), and Vim (http://www.vim.org). Many of these editors can be used for other programming languages as well.

Introductory Resources and Books

The following are the resources used in writing this chapter:

- Typing ? or `help()` with any function in R generates a help page describing the function arguments, some examples, and possibly a few references.
- Within the R console, if you choose "R Help" from the "Help" toolbar menu, a number of basic materials can be found, including the valuable *An Introduction to R* produced by the R Core Team.
- The CRAN Website, http://cran.r-project.org/, contains the help manual for every function (in every package) and additional documentation regarding R; the publication *The R Journal* is especially useful.
- The classic *Modern Applied Statistics with S, 4th ed.* by W.N. Venables and B.D. Ripley (Springer, 2002) is an excellent beginners guide.
- *Introductory Statistics with R, 2nd ed.* by Peter Dalgaard (Springer, 2008) is another classic. It focuses more on the statistical applications of R (summary statistics, hypothesis testing) than programming techniques.
- *Using R for Introductory Statistics, 2nd ed.* by John Verzani (Chapman & Hall/CRC The R Series, 2014) assumes the reader knows no statistics or programming.
- *Introduction to Scientific Programming and Simulation Using R, 2nd ed.* by Owen Jones, Robert Maillardet, and Andrew Robinson (Chapman & Hall/CRC The R Series, 2014), unlike Dalgaard (2008), concentrates less on statistics and more on programming and numerical algorithms.
- For a compilation of R recipes, refer to the *R Cookbook* by Paul Teetor (O'Reilly Cookbooks, 2011).
- Chapman & Hall/CRC Press has a series of books focusing on various R techniques called *The R Series* (http://www.crcpress.com/browse/series/crctherser). Similarly, Springer Press has a series titled *Use R!* (http://www.springer.com/series/6991).