Foundational and Applied Statistics for Biologists using R
Electronic Appendix (Introduction to R)

Ken Aho
updated 10/16/2019
# Table of Contents

## 1 Introduction to R

1.1 A very brief overview
1.2 The R language
1.3 R copyrights and licenses
1.4 R and reliability

## 2 Basics

2.1 First operations
2.2 Use your scroll keys
2.3 Note to self
2.4 Unfamiliar commands
2.5 Basic options
2.6 Saving and loading your work

## 3 Getting help in R

3.1 help
3.2 Manuals and additional information

## 4 Expressions and assignments

4.1 Naming objects
4.2 Combining objects with c

## 5 R objects and R classes

5.1 Listing objects
5.2 Mathematical operators and functions
5.3 Statistical operators
5.4 Logical commands
5.5 Mathematical functions
5.6 Mathematical constants
5.7 Random number generation

## 6 Mathematical operations

6.1 Mathematical operators and functions
6.2 Statistical operators

## 7 R packages

7.1 Datasets in packages
7.2 R and spreadsheets

## 8 R and spreadsheets

8.1 attach/attach
8.2 with
8.3 remove
8.4 Cleaning up
8.5 External files
8.6 Functions

## 9 R graphics

9.1 plot
9.2 Scatterplots
9.3 Graphical devices
9.4 par

## 10 Data structures

10.1 Vectors
10.2 Matrices
10.3 Arrays
10.4 Dataframes

## 11 Data entry at the command line

11.1 scan, readline, readline
11.2 Facilitating command line entry

## 12 Importing data into R

12.1 read.table
12.2 read.csv
12.3 scan
12.4 Easy import of use of file.choose()

## 13 Exporting data from R

13.1 Exporting matrix, dataframe and array components

## 14 Subsetting matrix, dataframe and array components

15 Operations on matrices and dataframes

## 16 Logical commands

16.1 ifelse
16.2 switch
16.3 if, else, and elif

## 17 Simple functions for data management

17.1 replace
17.2 which
17.3 sort

## 18 Testing and coercion

18.1 NA, NaN, and NULL

## 19 Binary numbers, bits and bytes

20.1 Floating point arithmetic
20.2 Binary operators

## 21 Writing functions

21.1 An introductory example
21.2 Global variables versus local variables
21.3 unlist
21.4 switch
21.5 Triple dot
21.6 Looping
21.7 Loops without for
21.8 Calling and receiving code from other languages
21.9 Functions with animation
21.10 Building GUIs
21.11 Functions with class and method
21.12 Documenting functions and workflow

## 22 Exercises

22.1 Exercises

## References

145

Index
148
I believe that R currently represents the best medium for quality software in support of data analysis.

John Chambers, a developer of S

R is a real demonstration of the power of collaboration, and I don’t think you could construct something like this any other way.

Ross Ihaka, original co-developer of R

1 Introduction to R

R is a computer language and an open source setting for statistics, data management, computation, and graphics. The outward mien of the R-environment is minimalistic, with only low-level interactive facilities. This is in contrast to conventional statistical software consisting of extensive, high-level, often inflexible tools. The simplicity of R allows one to easily evaluate, edit, and build procedures for data analysis.

R is useful to biologists for two major reasons. First, it provides access to a large number of existing statistical, graphical, and organizational procedures, many of which have been designed specifically for biological research. Second, it allows one to “get under the hood”, look at the code, and check to see what algorithms are doing. If, after examining an R-algorithm we are unsatisfied, we can generally modify its code or create new code to meet our specific needs.

1.1 A very brief history

R was created in the early 1990s by Australian computational statisticians Ross Ihaka and Robert Gentleman to address scoping and memory use deficiencies in the language S (Ihaka and Gentleman 1996). At the insistence of Swiss statistician Martin Maechler, Ihaka and Gentleman made the R source code available via the internet in 1995. Because of its relatively easy-to-learn language, R was quickly extended with code and packages developed by its users (§ 7). The rapid growth of R gave rise to the need for a group to guide its progress. This led, in 1997, to the establishment of the R-development core team, an international panel that modifies, troubleshoots, and manages source code.

Because of its freeware status, the overall number of R-users is difficult to determine. Nonetheless, experts in the computer industry estimated that between 1 and 2 million individuals were actively using R in 2009. This number has undoubtedly increased as the number of R-packages has continued to increase exponentially (Fox 2009). Other reports indicate that the internet traffic for R topics exceeds discussions for all other types of statistical software. In addition, recent searches using Google scholar® (http://scholar.google.com/) show that the number of citations of R and its packages in scientific articles has increased dramatically since 2005 while citations for software giants SAS® and SPSS® have declined. R is currently the most popular programming language in the world among data scientists, and is ranked 6th in world by the Institute of Electrical and Elec-

---

4. For additional information on the historical and technical development of R see Hornick (2009) and Venebles et al. (2008).
Among other things, this difference means more available pre-reserved computer memory, and fewer amount of memory at startup. This is in contrast to S-engines which adjust available memory to session needs. The big five are, from first to last, Python, C, Java, C++, and C#.

Lexical scoping are further addressed in § 21. This allows functions in R to access to variables that were in effect when the function was defined in a session. S, however, does not allow such free variables. The characteristics of R functions and details concerning lexical scoping are further addressed in § 21.

1.3 R copyrights and licenses

R is free, and as a result there are no warranties on its environment or packages. As its copyright framework uses the GNU (a recursive acronym for GNU is not Unix) General Public License (GPL). This allows users to share and change R and its functions. The associated legalese can read by typing RShowDoc("COPYING") in the R-consoled. Because its functions can be legally (and easily) recycled and altered we should always give credit to developers, package maintainers, or whoever wrote the R functions we are using.

1.4 R and reliability

The lack of an R warranty has frightened away some scientists. But be assured, with few exceptions R works as well or better than “top of the line” analytical commercial software. Indeed, SAS® and SPSS® have recently made R applications accessible from within their products (Fox 2009). For specialized or advanced statistical techniques R often outperforms other alternatives because of its diverse array of continually updated applications.

The computing engine and packages that come with a conventional R download (see § 1.5) meet or exceed U.S. federal analytical standards for clinical trial research (R Foundation for Statistical Computing 2008). In addition, core algorithms used in R are based on seminal and trusted functions. For instance, R random number generators include some of the most venerated and thoroughly tested functions in computer history (Chambers 2008). Similarly, the LAPACK algorithms (Anderson et al. 1999), used by R for linear algebra, are among the world’s most stable and best-tested software (Chambers 2008).

1.5 Installation

To install R, first go to the website http://www.r-project.org/. On this page specify which platform you are using (Figure 1, step 1). R can currently be used on Unix-like, Windows and Mac operating systems. This appendix has been created specifically for Windows users of R. In almost every case, however, it will also be applicable to other platforms. Once an operating system has been selected one can click on the “base” link to download the precompiled base binaries if R currently exists on your computer. If R has not been previously installed on your computer click on “Install R for the first time” (Figure 1, step 2). You will delivered to a window containing a link to download the most recent version of R. Click on the “Download” link (Figure 1, step 3).

Two new versions of R are generally released each year, one in April and one in October. New packages (§ 7), and newer versions of older packages are released on a continual basis at the discretion of their developers. Archived versions of R and its packages are also available from http://www.r-project.org/.

---

1. http://spectrum.ieee.org/computing/software/the-2017-top-ten-programming-languages accessed 9/1/2017 (currently, the big five are, from first to last, Python, C, Java, C++, and C)
2. Virtual pagination is a memory management scheme that allows a computer to store and retrieve data from secondary storage for use in main memory.
2 Basics

Upon opening R two things that will appear in the console of the Graphical User Interface (R-GUI). These are the license disclaimer and the command line prompt, i.e., > (Figure 2). The prompt indicates that R is ready for a command. All commands must begin at >. R user commands are colored red, and output, including errors and warnings are colored blue.

We can exit R at any time by typing q() in the console, closing the GUI window (non-Unix only), or by selecting “exit” from the pull-down File menu (non-Unix only).

![Figure 2 The R console. R version 2.15.1, "Roasted Marshmallows"](image)

2.1 First operations

As an introduction we can use R as a calculator. Type 2 + 2 and press Enter.

2 + 2
[1] 4
>

The [1] means, “this is the first requested element”. In this case there is just one requested element, 4, the solution to 2 + 2. If the output elements cannot be held on a single console line, then R would begin the second line of output with the element number comprising the first element of the new line. For instance, the command rnorm(20) will take 20 random samples from a standard normal distribution (see Ch. 3 in the Foundational and Applied Statistics text). We have:

rnorm(20)
[1] 0.73704627 0.06572694 -0.19401659 1.55767302 -0.66656940 -0.63375586
[7] -0.38926816 0.46596203 -0.35382023 0.72729659 0.42944759 -0.50559415
[13] 0.95743654 0.49844963 0.01833043 -0.29257820 -0.56753070 0.25374833
[19] -0.27808670 -0.83199069
>

The reappearance of the command line prompt indicates that R is ready for another command.

The R commands are generally insensitive to spaces. This allows the use of spaces to make code more legible. The command 2 + 2 is simply easier to read and debug than 2+2.

2.2 Use your scroll keys

As with many other command line environments, the scroll keys (Figure 3) provide an important shortcut in R. Instead of editing a line of code by tediously mouse-searching for an earlier command to copy, paste and then modify it, you can simply scroll back through your earlier work using the upper scroll key, i.e., ↑. Accordingly, scrolling down using ↓ will allow you to move forward through earlier commands.

![Figure 3 Direction keys on a keyboard.](image)

2.3 Note to self: #

R will not recognize commands preceded by #. As a result this is a good way for us to leave messages to ourselves.

# Note at beginning of line
2 + 2
[1] 4
2 +  # Note in middle of line
+ 2
[1] 4

In the "best" code writing style it is recommended that one place a space after # before beginning a comment, and to insert two spaces following code before placing # in the middle of a line. This convention is followed
2.4 Unfinished commands

R will be unable to move on to a new task when a command line is unfinished. For example, type

```
2 +
```

and press Enter. We note that a continuation prompt, by default +, is now in the place the command prompt should be. R is telling us the command is unfinished. We can get back to the command prompt by finishing the function, clicking Misc>Stop current computation or Misc>Stop all computations (non-Unix only), typing Ctrl-C (Unix only), or by hitting the Esc key (Windows only).

2.5 Basic options

To enhance our experience, we can adjust the appearance of the R-console and customize options that affect function output. These include the characteristics of the graphics devices, the width of print output in the R-console, and the number of print lines and print digits. Changes to some of these parameters can be made by going to Edit>GUI Preferences in the R toolbar. Many other parameters can be changed using the options function. To see all alterable options one can type:

```
options()
```

To change default options one would simply define the desired change within parenthesis following the options function name. For instance, to see the default number of digits in R output I would type:

```
options("digits")
```

To change the default number of digits in output from 7 to 5, in the current session I would type:

```
options(digits = 5)
```

To establish user defaults for all future sessions one will need to change the R-console file located (currently) in the etc directory of the version of R you are using. This can either be done manually, or by using Edit>GUI Preferences>Save in the R toolbar.

By default the R working directory is set to be the home directory of the workstation. The command getwd() shows the current file path for the working directory.

```
getwd()
```

The working directory can be changed with the command setwd(filepath), where filepath is the location of the directory, or by using pulldown menus, i.e., File>Change dir (non-Unix only).

An .Rprofile file exists in your Windows R/R-version/etc directory. R will silently run commands in the file upon opening. By customizing the .Rprofile file one can set options, install frequently loaded non-default packages (§ 7), define your favorite package repository, and create aliases and defaults for frequently used functions.

Here is the content of my current .Rprofile file.

```
options(repos = structure(c("http://ftp.osuosl.org/pub/cran/")))
.First <- function(){
  library(asbio)
  cat("\nWelcome to R Ken! ", date(), "\n")
}
.Last <- function(){
  cat("\nGoodbye Ken", date(), "\n")
}
```

The options(repos = structure(c("http://ftp.osuosl.org/pub/cran/"))) command defines my favorite R-repository. The function .First() will be run at the start of the R session and .Last() will be run at the end of the session. As we go through this primer it will become clear that these functions force R to say hello and to load my package asbio when it opens, and to say goodbye when it closes (although the farewell will only be seen when running R from a command line interface).

The .Rprofile file in the /etc directory is the so-called .Rprofile.site file. Additional .Rprofile files can be placed in the working and user directories. R will check for these and run them after running the .Rprofile.site file.

One can create .Rprofile files, or any type of R extension file using the function file.create. For instance, the code:

```
file.create("defaults.Rprofile")
```

places an empty, editable, .Rprofile file called defaults in the working directory.

2.6 Saving and loading your work

As noted earlier, an R session is allocated with a fixed amount of memory that is managed in an on-the-fly manner. An unfortunate consequence of this is that if R crashes all unsaved information from the work session will be lost. Thus, session work should be saved often. Note that R will not give a warning if you are writing over another file from the R console with the same name! The old file will simply be replaced.

Saving the R History

To view the history (i.e., the commands that have been used in a session) one can use history(n) where n is the number of old command lines one wishes to see. For instance, to see the last three commands in a (somewhat
To save the session history in Windows one can use **File>Save History** (non-Unix only) or the function `savehistory()`. For instance, to save the session history to the working directory under the name `history1` I could type:

```r
device(file.choose())
```

This will allow one to browse interactively for files using dialog boxes.

Detailed procedures for importing and exporting (saving) data with a row and column format, and an explicit delimiter (e.g., `.csv` files) are described in §12 and 13, respectively.

### Saving R Scripts

To save R code as executable source file, one can save the code in the `R`-script editor (go to **File>New script**, or simply type `fix()`), or within some other `R`-compatible text editor (e.g., ESS, TinnR, etc.) as an `.R` extension file. `R` scripts can be called and executed using the function `source()`. For instance to go looking interactively for source code to execute I could type:

```r
source(file.choose())
```

or go to **File>Source R code**.

To load the history from a previous session one can use **File>Load History** (non-Unix only) or the function `loadhistory()`. For instance, to load `history1` I would type:

```r
loadhistory(file = "history1.Rhistory")
```

To save the session history at the end of (almost) every interactive Windows or Unix-alike R session, one can alter the `.Rprofile` file. Last function to include:

```r
.Last <- function() if(interactive()) try(savehistory("~/Rhistory"))
```

### Saving R Objects

To save all of the objects (see §4) available in the current R-session one can use **File>Save Workspace** (non-Unix only), or simply type:

```r
save.image()
```

This procedure saves session objects (see §4) to the working directory as a nameless file with an `.RData` extension. The file will be opened, silently, with the inception of the next R-session, and cause objects used or created in the previous session to be available. Indeed, `R` will automatically execute all `.RData` files in the working directory for use in a session. `.RData` files can also be loaded using **File>Load Workspace** (non-Unix only). One can also save the `.RData` objects to a specific directory location and use a specific file name using **File>Save Workspace**, or with the `file` argument in `save.image`, or with the flexible function `save` (see §2.6).

R data file formats, including `.rda`, and `.RData`, (extensions for `R` data files), and `.R` (the format for `R` scripts), can be read into R using the function `load()`. Users new to a command line environment will be reassured by typing:

```r
load(file.choose())
```

This will allow one to browse interactively for files using dialog boxes.

Importantly, the functions `savehistory()`, `loadhistory()` and `history()` are not currently supported for Mac OS X. There are ways around this. For instance, in RStudio (§21) the Mac OS X command history can be obtained by clicking the History icon history that appears on the tool bar at the top of the console window. As an additional issue, Windows and Unix-alike platforms have different implementations for `savehistory()` and `loadhistory()`. Get the help pages for these particular functions within your platform for particulars.
3 Getting help in R

There is no single perfect source for information/documentation for all aspects of R. Detailed information concerning base operations and package development are described at the R website http://www.r-project.org/, but is generally intended for those familiar with both Unix/Linux systems, and command based formats. They may not be especially helpful to new R users used to point-and-click formats, and no programming history.

3.1 help

A comprehensive help system is built into R. The system can be accessed question mark (?) and help functions. For instance, if I wanted to know more about the plot function I could type:

?plot
or
help(plot)

Documentation for functions will include a list of arguments for functions, and a description of variables for datasets, and other pertinent information. Quality of documentation will generally be excellent for functions from packages in the default R download (see § 7), but will vary from package to package otherwise.

A list of arguments for a function, and their default values, can (often) be obtained using the function formals.

formals(plot)

$x
$y
$...

For help and information regarding programming metacharacters used in R (for instance $, ?, [[, [, or ^), one would enclose the metacharacters with quotes. For example, to find out more information about the logical operator & I could type help("&") or ?"&".

Placing two question marks in front of a word or function will cause R to search for help files concerning with respect to all packages in a workstation. For instance type:

??lm
or, alternatively
help.search(lm)

for a huge number of help files on linear model functions identified through fuzzy matching. Help for particular R-questions can often be found online using the search engine at http://search.r-project.org/. This link is provided in the Help pulldown menu in the R console (non-Unix only).

The function demo allows one access to examples developers have worked out (complete with R code) for a particular function or topic. For instance, type:

demo(graphics)

for a brief demonstration of graphics or demo(persp) for a demonstration of 3D perspective plots. Typing:

demo(Hershey)

will provide a demonstration of available of modifiable symbols from the Hershey family of fonts (see Hershey 1967; 9.6). Typing:

demo()

lists all of the demos available in the loaded libraries for a particular workstation. The function example usually provides less involved demonstrations from the man directories (short for user manual) in packages, along with code requirements. For instance, type:

e.example(plotmath)

for a demonstration of mathematical graphics complete with code.

R packages often contain vignettes. These are short documents that generally describe the theory underlying algorithms and guidance on how to correctly use package functions. Vignettes can be accessed with the function vignette. To view all available vignettes for packages attached for a current work session type:

vignette(all = FALSE)

To view all vignettes for all installed packages type:

vignette(all = TRUE)

To view all vignettes for the installed package asbio:

vignette(package = "asbio")

To see the vignette simpson in package asbio I would load asbio, then type:

vignette(simpson)

The function browseVignettes() provides an HTML-browser that allows interactive vignette searches.

3.2 Manuals and additional information

Because the Foundational and Applied Statistics for Biologists using R textbook is primarily about statistics and not R, this appendix is not intended to be an exhaustive operators manual. General guidance for R-programming can be found in Chambers (2008), or more recently Matloff (2011). In addition to Foundational and Applied Statistics, good resources for R in an applied statistical context include Venables and Ripley (2002),
Expressions and Assignments

All entries in R are either expressions or assignments. If a command is an expression it will be evaluated, printed and discarded. Examples include: 2 + 2. Conversely, an assignment evaluates an expression, and assigns the output to an R-object (§ 5), but does not automatically print the results.

To convert an expression to an assignment we use the assignment operator, <-, which represents an arrow. The assignment operator can go on either side of an object. For instance, if I type:

```
x <- 2 + 2
```

or

```
2 + 2 <- x
```

then the sum 2 + 2 will be assigned to the object x. To print the result (to see x) I simply type:

```
x
```

```
[1] 4
```

or

```
print(x)
```

```
[1] 4
```

Note that we could have also typed \texttt{x = 2 + 2} with the same assignment results. However in this document we will continue to use the arrow operator for assignments, and save = for specifying arguments in functions.

4.1 Naming objects

When assigning names to R-objects we should try to keep the names simple, and avoid names that already represent important definitions and functions. These include: TRUE, FALSE, NULL, NA, NaN, and Inf. In addition, we cannot have names:

- beginning with a numeric value
- containing spaces, colons, and semicolons
- containing mathematical operators (e.g., *, +, -, /, =)
- containing important R metacharacters (e.g., @, #, ?, !, %, &).

Names should also be self explanatory. Thus, for a object containing 20 random observations from a normal distribution, the name \texttt{rN20} is far superior to the, albeit easily-typed, name \texttt{n}. Finally, with assignment commands we should also remember that, like most software developed under Unix/Linux, R is case sensitive. That is, each of the following 2^4 combinations will be recognized as distinct:

```
namE, Name, nAmE, naMe, NAmE, naMe, NaMe, NaMEn, NaMe, NaME, NaMEn, NaMEn, NaME, NaME, NaME, NaME
```
Before proceeding further it is important to make two observations.

• First, everything created or loaded in R is an object. This means, in the sense of object oriented programming (OOP), that they possess certain traits that describe them.

• Second, every R-object belongs to at least one class. This requirement means that R-objects can be classified in a way that allows them to be recognized and used correctly by both users and other objects. Object classes allow for custom utility functions, e.g., `plot`, `print`, `summary`, to be created for particular classes. These can be called by simply using generic function names e.g., `plot`, `print`, `summary` (Section 21.11). Among other things, this means fewer function names to memorize.

For the object `x` below:

```r
x <- 2 + 2
```

we have the following class:

```r
class(x)
[1] "numeric"
```

Thus, the object `x` belongs to class `numeric`. Objects in class `numeric` can be evaluated mathematically.

Objects can also be distinguished by their storage mode or type (the way R caches them in its primary memory). For `x` we have:

```r
typeof(x)
[1] "double"
```

This means that `x` is stored with 64 bit floating point double precision: a computer number format usually occupying 64 bits in computer memory. This topic is readdressed in §20. Object type names and class names are often identical.

Many R-objects will also have attributes (characteristics particular to the object or object class). Typing:

```r
attributes(x)
NULL
```

indicates that `x` does not have additional attributes. However, using coercion (§18) we can define `x` as an object of class `matrix` (a collection of data in a row and column format):

```r
x <- as.matrix(x)
```

*OOP languages include R, C++, C++, Objective-C, Smalltalk, Java, Perl, Python and PHP. C is not considered an OOP language.
Now `x` has the attribute `dim` (i.e., dimension).

```r
attributes(x)
$dim
[1] 1 1
```

Now `x` is a one-celled matrix. It has one row and one column.

Amazingly, classes and attributes allow R to simultaneously store and distinguish objects with the same name\(^{10}\). For instance:

```r
mean <- mean(c(1, 2, 3))
mean
[1] 2
mean(c(1, 2, 3))
[1] 2
```

In general it is not advisable to name objects after frequently used functions. Nonetheless, the function `mean` is distinguishable from the new user-created object `mean` because these objects have different identifiable characteristics.

The function `str` attempts display the internal structure of an R object. It is extremely useful for succinctly displaying the contents of lists (§ 10).

```r
str(x)
 num [1, 1] 4
```

Thus, `x` is a `1 x 1` matrix structure containing the number 4.

### 5.1 Listing objects

By typing:

```r
ls()
```

or

```r
objects()
```

one can see a list of the available R-objects in a particular R session. Currently, we have:

```r
ls()
[1] "mean" "x"
```

\(^{10}\) Note, however, naming conflicts with global variables often causes issues with package development. The code:

```r
mean <- NA;
rm(mean)
```

will remove the offending user-created object `mean` above, leaving behind the global variable...
6 Mathematical operations

6.1 Mathematical operators and functions

Usual (and unusual) mathematical operators and functions are easily specified in R. In applications below, x represents either a user-specified scalar (single number) or a vector (collection of numbers).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
</tr>
<tr>
<td>%%</td>
<td>modulo (find the remainder in division)</td>
</tr>
</tbody>
</table>

As with functions from all programming languages, R functions generally require a user to specify arguments (in parentheses) following the function name.

For instance, `sqrt` and `factorial` each require one argument, a call to data. Thus, to solve $1/\sqrt{22!}$, one type:

```r
1/sqrt(factorial(22))
[1] 2.982749e-11
```

and to solve $\Gamma\left(\sqrt{23\pi}\right)$, one type:

```r
gamma((23 * pi)^(1/3))
[1] 7.410959
```

The `log` function can also compute logarithms to a particular base by specifying the base in an optional second argument called `base`. For instance, to solve the operation: $\log(3, 10) + \log(5, 3)$, one would simply type:

```r
log(3, 10) + log(5, 3)
[1] 1.942095
```

By default the function `log` computes natural logarithms, i.e.,

```r
log(exp(1))
[1] 1
```

That is, by default `base` = Euler's constant $e = 2.718282...$

Arguments can be specified by the order that they occur in the list of arguments in the function code, or by calling the argument by name. In the code above I know that the first argument in `log` is a call to data, and the second argument is the base. I may not, however, remember the argument order in a function, or may wish to only change certain arguments from a large allotment. In this case it is better to specify the argument by calling its name and defining its value with an equals sign.
**Mathematical expressions**

\[
\log(x = 3, \text{ base } = 10) + \log(x = 5, \text{ base } = 3)
\]

\[
[1] 1.942095
\]

**Trigonometry**

R assumes that the inputs for the trigonometric functions are in radians. Of course degrees can be obtained from radians using \(\text{Deg} = \text{Rad} \times 180 / \pi\), or conversely \(\text{Rad} = \text{Deg} \times \pi / 180\). Thus, to find \(\cos(45^\circ)\) I type:

\[
\cos(45 * \pi/180)
\]

\[
[1] 0.7071068
\]

This is correct because \(\cos(45^\circ) = \sqrt{2}/2 = 0.7071068\).

\[
sqrt(2)/2
\]

\[
[1] 0.7071068
\]

**Derivatives**

The function \(\text{D}\) finds symbolic and numerical derivatives of simple expressions. It requires two arguments, a mathematical function specified as an \textit{expression} [i.e., a list (§ 10) that can be evaluated with the function \textit{eval}], and the denominator in the difference quotient. Here is an example of how \textit{expression} and \textit{eval} are used:

\[
eval(\text{expression}(2 + 2))
\]

\[
[1] 4
\]

Of course we wouldn’t bother to use \textit{expression} and \textit{eval} in this simple application.

**Example 1**

To demonstrate \textit{expression} and \(\text{D}\) (the latter calls \textit{eval}) we will evaluate the following derivatives:

\[
\frac{d}{dx} 0.5x^4, \quad \frac{d}{dx} \log(x), \quad \text{and} \quad \frac{d}{dx} \cos(2x).
\]

\[
\text{D}(\text{expression}(0.5 * x^4), "x")
\]

\[
0.5 * (4 * x^3)
\]

\[
\text{D}(\text{expression}(\log(x)), "x")
\]

\[
x^{-1}
\]

\[
\text{D}(\text{expression}(\cos(2 * x)), "x")
\]

\[
-\sin(2 * x) * 2
\]

Symbolic solutions to partial derivatives are also possible. For instance, consider

\[
\frac{\partial}{\partial x} 3x^2 + 4xy.
\]

We have:

\[
\text{D}(\text{expression}(3 * x^2 + (4 * x * y)), "x")
\]

\[
3 * (2 * x) + 4 * y
\]

**Integration**

Integration requires specifying an integrand in the form of a \textit{function}. Recursively speaking, a function is defined by a function named \textit{function}. Arguments will be contained in a set of parentheses following the call to \textit{function}. The function contents follow, generally delineated by curly brackets. Thus, we have the following form for a function called \"example\" with \(n\) arguments:

\[
\text{example} <- \text{function}(\text{argument}_1, \text{argument}_2, \ldots, \text{argument}_n)(\text{function contents})
\]

Recall that \(<-\) is the assignment operator (see § 4). Functions are explained in detail in § 21.

The R function for integration, \textit{integrate}, requires, as its 1st argument, a user-defined function describing the integrand. The 2nd and 3rd arguments for \textit{integrate} will be the lower and upper bounds for definite integration. The integrand function, in turn, must have as its 1st argument the name of the variable to be integrated.

**Example 2**

To solve for:

\[
\int_1^3 3x^2 dx
\]

we can use the following code:
For instance, consider the dataset \( x \) below:

\[
x <- c(1, 2, 3, 2, 1)
\]

The mean is

\[
mean(x)
\]
\[
[1] 1.8
\]

and the median is:

\[
median(x)
\]
\[
[1] 2
\]

The formulae and underlying theory for point estimators such the mean, variance, and median are dealt with in Chapter 4 in the *Foundational and Applied Statistics for Biologists* text. Interval estimators are addressed in Ch. 5.
## 7 R packages

An R package contains a set of related functions, documentation, and data files that have been bundled together. As of 6/2016 packages provided in the basic R distribution are those shown in Table 1. R distribution packages are directly controlled by the R core team and are extremely well-tested and trustworthy.

Table 1 “distributed” core packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Maintainer</th>
<th>Topic(s) addressed by package</th>
<th>Current Citation/Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>R Core Team</td>
<td>Base R functions</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>compiler</td>
<td>R Core Team</td>
<td>R byte code compiler</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>datasets</td>
<td>R Core Team</td>
<td>Base R datasets</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>grDevices</td>
<td>R Core Team</td>
<td>Graphics devices for base and grid graphics</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>graphics</td>
<td>R Core Team</td>
<td>Functions for base graphics</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>grid</td>
<td>R Core Team</td>
<td>Grid graphics layout capabilities</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>methods</td>
<td>R Core Team</td>
<td>Formally defined methods and classes for R objects</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>parallel</td>
<td>R Core Team</td>
<td>Support for parallel computation</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>splines</td>
<td>R Core Team</td>
<td>Regression spline functions and classes</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>stats</td>
<td>R Core Team</td>
<td>R statistical functions</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>stats4</td>
<td>R Core Team</td>
<td>Statistical functions with S4 classes</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>tcltk</td>
<td>R Core Team</td>
<td>Interface and language bindings to Tcl/Tk GUI elements</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>tools</td>
<td>R Core Team</td>
<td>Tools for package development and administration</td>
<td>R Core Team (2017)</td>
</tr>
<tr>
<td>utils</td>
<td>R Core Team</td>
<td>R utility functions</td>
<td>R Core Team (2017)</td>
</tr>
</tbody>
</table>

Packages in Table 2 constitute recommended packages. These are not necessarily controlled by the R core team, but are also extremely useful, well-tested, and stable. Like R distribution packages (Table 1), recommended R packages are included in conventional downloads of R.

Aside from distributed and recommended packages, there are a large number of user-defined contributed packages that are useful to biologists and biometrists (> 10000 as of 9/5/2017). Table 1 lists a few. Contributed R packages can be downloaded from CRAN (the Comprehensive R Archive Network). To do this, one can go to Packages>Install package(s) on the R-GUI toolbar, and choose a nearby CRAN mirror site to minimize download time (non-Unix only). Once a mirror site is selected, the packages available at the site will appear. One can simply click on the desired packages to install them. Packages can also be downloaded directly from the command line. To install the package vegan (Table 3), I would simply type:

```r
install.packages("vegan")
```

Of course, both of these approaches require that your computer has web access. If local web access is not available, libraries can be saved from the CRAN website or some other source as compressed (.zip, .tar) files which can then be placed manually on a workstation by inserting the package files into the “library” folder within the top level R directory, or into a path-defined R library folder in a user directory. This package can then be loaded for a particular session.

Once a package is installed it never needs to be re-installed. However, for use in an R session one will need load the package using the `load` function. For instance, to load the vegan package I would type:

```r
load(vegan)
```

Or one could go to Packages>Load packages on the R-GUI toolbar (non-Unix only) Notwithstanding user modification, none of packages in Table 1 require loading, as they will be loaded automatically upon opening R. Typing `search()` lets one see all the attached (loaded) packages (and other attached objects) currently available in a work session.

Table 2 “Recommended” R packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Maintainer</th>
<th>Topic(s) addressed by package</th>
<th>Current Citation/Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>KernSmooth</td>
<td>B. Ripley</td>
<td>Functions for kernel smoothing and density estimation</td>
<td>Wand (2015)</td>
</tr>
<tr>
<td>MASS</td>
<td>B. Ripley</td>
<td>Wide variety of important statistical methods</td>
<td>Venables and Ripley (2002)</td>
</tr>
<tr>
<td>Matrix</td>
<td>M. Maechler</td>
<td>Classes and methods and operations for matrices using LAPACK and SuiteSparse</td>
<td>Bates and Maechler (2016)</td>
</tr>
<tr>
<td>boot</td>
<td>B. Ripley</td>
<td>Bootstrapping and analytical extensions</td>
<td>Canty and Ripley (2016), Davison and Hinkley (1997)</td>
</tr>
<tr>
<td>class</td>
<td>B. Ripley</td>
<td>Functions for classification</td>
<td>Venables and Ripley (2002)</td>
</tr>
<tr>
<td>cluster</td>
<td>M. Maechler</td>
<td>Functions for cluster analysis</td>
<td>Maechler et al. (2016)</td>
</tr>
<tr>
<td>codetools</td>
<td>S. Wood</td>
<td>Code analysis tools</td>
<td>Tierney (2016)</td>
</tr>
<tr>
<td>foreign</td>
<td>R core team</td>
<td>Functions for reading and writing data stored by non-R statistical software, e.g. SAS</td>
<td>R Core Team (2016)</td>
</tr>
<tr>
<td>lattice</td>
<td>D. Sarkar</td>
<td>Lattice graphics, an implementation of Trellis graphics functions</td>
<td>Sarkar (2008)</td>
</tr>
<tr>
<td>nlm</td>
<td>R core team</td>
<td>Linear and non-linear mixed effect models</td>
<td>Pinheiro et al. (2016)</td>
</tr>
<tr>
<td>rpart</td>
<td>B. Ripley</td>
<td>Recursive PARTitioning and regression trees</td>
<td>Venables and Ripley (2002)</td>
</tr>
<tr>
<td>spatial</td>
<td>B. Ripley</td>
<td>Functions for kriging and point pattern analysis</td>
<td>Venables and Ripley (2002)</td>
</tr>
<tr>
<td>survival</td>
<td>T. M. Therneau</td>
<td>Functions for survival analysis, including penalized likelihood</td>
<td>Therneau (2019)</td>
</tr>
</tbody>
</table>

As noted in § 3, once a package is installed information its functions can be accessed using `help`. For example, information about the `vegan` package can be accessed by typing `help(package = "vegan")`.

Given web access, newer (updated) versions of packages can be obtained using the command:

```r
update.packages()
```

Note, however, that this will only give you the latest possible package applicable to the version of R that you are running. Acquiring the “very latest” version of a package may require downloading the latest version of R.

To remove `vegan` (and its functions) from a particular session I would type:
Aside from CRAN, there are currently two large repositories of R packages. First, the bioconductor project (http://www.bioconductor.org/packages/release/Software/html) contains a large number of packages for genomic analysis (1,383 as of Sept 2017) that are not found at CRAN. Packages can be downloaded from bioconductor using an R script called biocLite. To download the package rRytoscape from biocondctor I would type:

```r
source("http://bioconductor.org/biocLite.R")
biocLite("RCytoscape")
```

Second, R-forge (http://r-forge.r-project.org/) contains alpha releases of packages that have not yet been implemented into CRAN, and other miscellaneous code. Either of these sources can be specified from Packages>Select Repositories in the R-GUI (non-Unix only)\(^\text{11}\).

<table>
<thead>
<tr>
<th>Table 3 Additional, contributed R packages that may be useful to biologists and biometricians.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
</tr>
<tr>
<td>akima</td>
</tr>
<tr>
<td>coin</td>
</tr>
<tr>
<td>ggplot2</td>
</tr>
<tr>
<td>lmodel2</td>
</tr>
<tr>
<td>lme4</td>
</tr>
<tr>
<td>spatstat</td>
</tr>
<tr>
<td>vegan</td>
</tr>
</tbody>
</table>

7.1 Datasets in packages

At this point it would be helpful to have a dataset to play with. The command: data(package = "base") lets us see all of the datasets available in the R base package while the command: data() lets us see all the datasets available within all the packages loaded onto a particular computer workstation. We will learn how to enter our own data into R shortly.

\[^{11}\text{I do not encourage downloading all of the R packages that have been developed for analysis of biological data, much less trying out all of their functions. It is remarkable, however, to consider the range of topics these packages encompass. There are R packages for bioinformatics including packages devoted only to phylogenetics (e.g. packages such, picante, apTreeshape) or compiling amino acid sequences (package: seqinr) or interpolating with geographic information system software (e.g. GRASS), packages specifically created for animal tracking data (adehabitat), functions for finding spatial patterns in cells (applications in package spatstat), and even algorithms for quantifying the existence of indicator species using count data from benthic environments (package bio.infer), or discerning the effect of fishing methods (package fishingmethods).}^\]

\[^{11}\text{Note that Loblolly still exists, lp is just a copy.}

Typing:

```r
summary(lp)
```

provides a simple statistical summary of lp, i.e. the minimum response, 1\(^{st}\) quartile, median (or 2\(^{nd}\) quartile), mean, 3\(^{rd}\) quartile and maximum response. These statistics (excluding the mean) are known as the 5 number summary.
A data editor is provided by typing `fix(x)` or `View(x)`, when `x` is a dataframe or matrix (see § 10 for information on these object classes). For instance, the command `fix(lp)` will open the Lobolly pine dataset in the R data editor (Figure 4). When `x` is a function or character string, then `fix(x)` opens a script editor containing `x`.

The data editor has limited flexibility compared to software whose interface is a worksheet, and whose primary purpose is data entry and manipulation, e.g., Microsoft Excel®. However, it is possible to access R from Excel® (and vice versa) through use of the RExcel package (see Heiberger and Nuewirth 2009).

Changes made to an object using `fix` will only be maintained for the current work session. They will not permanently alter objects brought in remotely to a session.

8.1 `attach` and `detach`

The function `attach` will let R recognize the column names of a dataframe (more on what a dataframe is later). For instance, typing:

```
attach(lp)
```

will let R recognize the individual columns in `lp` (a dataframe) by name. The function `detach` is the programming inverse of `attach`. For instance, typing:

```
detach(Loblolly)
```

does not remove Loblolly, but will prevent R from recognizing the variables in Loblolly by name.
8.2 with

A safer alternative to attach is the function with. Using with eliminates concerns about multiple variables with the same name becoming mixed up in functions. This is because the variable names for a dataframe specified in with will not be permanently attached in an R-session. For instance, try:

```r
detach(lp)
with(lp, age + height)
```

8.3 remove and rm

Typing:

```r
remove(Loblolly)
or
rm(Loblolly)
```

will remove Loblolly completely from a work session. Happily, the object Loblolly (a dataset in the package datasets) cannot actually be destroyed/deleted using rm or remove (or any other commands) at the R-console.

8.4 Cleaning up

By now we can see that the R-console can quickly become cluttered and confusing. To remove clutter on the console (without actually getting rid of any of the objects in a session) press Ctrl+L or using the Edit pulldown menu to click on “Clear console”12 (non-Unix only). To remove all objects in the current session, including saved objects brought in from the working directory, one can type:

```r
rm(list = ls(all = TRUE))
```

Of course, one should never include this or other equivalent lines of code in a distributed function, since it would cause users to delete their work.

---

3 Many other keyboard shortcuts for R are available including Ctrl+U which deletes all the text from the current line, and Ctrl+K which deletes text from the current character to the end of the line. Conventional shortcuts are also valid, e.g. Ctrl+C = copy and Ctrl+V = paste. For guidance go to Help>Console (non-Linux only).
An enormous number of color choices for plots are possible and these can be specified in at least six different ways. First, we can specify colors with integers as I did in Figure 5. Additional varieties can be obtained by drawing numbers `colors()[number]` (Figure 6).

In the code above the x and y coordinates are both sequences of numbers from 1 to 20 obtained from the command `1:20`. I varied symbol colors and plotting characters (`col` and `pch` respectively) using `1:20` as well. The combination `col = 1` and `pch = 1` results in a black open circle, whereas the combination `col = 20`, `pch = 20` results in a blue filled circle. Note that we need to enclose the axis names in quotations for R to recognize them as text. Symbol numbers 21-26 allow background color specification using the argument `bg`. Many other symbol types are also possible including thousands of unicode options.

Second, we can specify colors using actual color names, e.g. "white", "red", "yellow". For a visual display of essentially all the available named colors in R type `example(colors)`.

Third, we can define colors by requesting red green and blue color intensities with the function `rgb`. Usable light intensities can be made to vary individually from 0 to 255 (i.e., within an 8 bit format). Fourth, we can specify colors using the function `hcl` which controls hues, chroma, and luminescence and transparency. Fifth, we can define colors using hexadecimal codes\(^{13}\), e.g., blue = `#0000FF`. Finally we can specify colors using entirely different palettes. Figure 7 shows six pie plots. Each pie plot uses a different color palette. Each pie slice from each pie represents a distinct segment of the palette.

Figure 5 Some symbol and color plotting possibilities.

![Figure 5](image1.png)

![Figure 6](image2.png)

Figure 6 Color choices from `colors()`.

\(^{13}\)A data coding system that uses 16 symbols: the numbers 1-9, and the letters A-F. Hexadecimals are primarily used to provide a more intuitive representation of binary-coded values (see §20).
Figure 7 Use of color palettes in R. Note that the numbers do not correspond to actual color type designations.

A large number of additional palettes (including color-blind-safe palettes) can be obtained using the R-package `RColorBrewer`.

display.brewer.all(n = 7, colorblindFriendly = T)

Figure 8 `RColorBrewer` color-blind-safe seven category palettes. Top palettes are so-called “sequential” palettes, middle palettes are “qualitative”, and bottom palettes are “divergent”.

Here are the hexadecimal names for the “Set2” palette chunks in Figure 8.

```r
brewer.pal(7,"Set2")
[1] "#66C2A5" "#FC8D62" "#8DA0CB" "#E78AC3" "#A6D854" "#FFD92F" "#E5C494"
```
9.2 Scatterplots

The most common type of graph projects points at the intersection of paired observations describing two quantitative variables. The result is a **scatterplot**. Scatterplots are often presented in conjunction with **regression analyses** in which one models the behavior of one variable (the response) as a function of a second variable (the predictor).

**Example 4 – Further Exploration of the Loblolly Dataset.**

Let’s visualize the relationship of the age and height of Loblolly pines using a scatterplot (Figure 9).

```
with(lp, plot(age, height))
```

![Figure 9 Scatterplot of height and age from the loblolly pine dataset.](image)

Not surprisingly, there appears to be a strong positive correlation between these variables.

Now let’s fit a simple linear regression for loblolly pine height as a function of age. A regression line will show the best possible linear fit for a response variable as a function of an quantitative explanatory variable. The **R** function for a linear model is **lm**. It encompasses and allows a huge number of statistical procedures, including regression (see Chs. 9, 10, and 11 in the *Foundational and Applied Statistics* text).

We have:

```
ha.lm <- lm(height ~ age, data = lp)
```

The tilde lets **R** know that I want height to be a function of age. We note that the function **lm** has a built-in version of **with** which is specified with the argument **data**. Objects of class **lm** have their own summary function. This can be called (in this case) by simply typing:

```
summary(ha.lm)
```

```
Coefficients:
Estimate  Std. Error   t value  Pr(>|t|)
(Intercept) -1.31240    0.62183  -2.111   0.0379 *
age          2.59052    0.04094  63.272   <2e-16 ***
```

The output shows us the Y-intercept, -1.31240, and slope, 2.59052, of the regression line.

The **abline** function allows the plotting of a line over an existing plot. The first two arguments for **abline** are the Y-intercept and slope (Figure 10).

```
with(lp, plot(age, height, pch=2, col=3))
abline(-1.312396, 2.590523)
```

![Figure 10 Scatterplot of height and age with a regression line overlaid.](image)

9.3 Graphical devices

Graphics in **R** are created within **graphics devices** that vary with respect to storage modes, display modes, available typefaces and other characteristics. In the current **R**-windows download, six graphics devices will be available including **windows**, **pdf**, **postscript**, and **X11**. The **X11** device is a windows graphics system for bitmap displays, and is commonly used in Unix-alike operating systems. Six other devices will also exist, although they may return a warning if **R** was not compiled to use them upon installation. These are **cairo_pdf**, **svg**, **png**, **jpeg**, **bmp**, and **tiff**.
Multiple devices (currently up to 63) may exist simultaneously in an R work session, although there will only be one active device. To find the current (active) graphics device's flavor we can type `dev.cur()`. I get:

```
dev.cur()
windows
2
```

R tells me there are two devices open. The current device is a `windows` device. The second device is the so-called `null device`. The null device is always open but only serves as a placeholder. Any attempt to use it will open a new device in R. Occasionally, on purpose or by accident, all graphics devices (except the null device) may become turned off. A new active graphics device can be created at any time by typing:

```
dev.new()
```

The active device can be changed using the function `dev.set()`.

### 9.4 par

Parameters for a graphics device (which may contain several plots) can be accessed and modified using the function `par`. Below are important arguments for `par`. Some of these can also be specified as arguments in `plot`, with different results.

- `bg` gives the background color for the graphical device. When used in `plot` it gives the background color of plotting symbols.
- `bty` is the box-type to be drawn around the plots. If `bty` is one of `"n"` (the default), `"l"`, `"7"`, `"n"`, `"u"`, or `"j"` the resulting box resembles the corresponding upper case letter. The value `"n"` suppresses the box.
- `fg` gives the foreground color.
- `font` is an integer that specifies the font face. 1 corresponds to regular text (the default), 2 to bold face, 3 to italic and 4 to bold italic.
- `las` is the style of axis labels: 0 always parallel to the axis (default), 1 always perpendicular to the axis, 3 always vertical.
- `mar` will have the form `c(bottom, left, top, right)` and gives the number of lines of margin to be specified on the four sides of the plot. The default is `c(5, 4, 4, 2) + 0.1`.
- `mfrow` will have the form `c(number of rows, number of columns)` that indicates the number and position of plots in a graphical layout.
- `oma` specifies the outer margins of a graphical device, given multiple plots, using a vector using a matrix of the form `c(bottom, left, top, right)`.
- `usr` will have the form `c(x1, x2, y1, y2)` giving the extremes of the user coordinates of the plotting region.

When setting graphical parameters, it is good practice to revert back to the original parameter values. Assume that I want to background of the graphics device to be black. To find this I would type:

```
old.par <- par(no.readonly = TRUE) # save default, for resetting...
par(bg = "black") # change background parameter
```

To return to the default settings for background I would type:

```
par(old.par)
```

Defaults will also be reset by closing the graphics device containing the customized parameters, or by opening a new device. For instance, using `dev.new()`.

Other fundamental properties of the default graphics device, such as device `height`, `width` and `pointsize`, can be adjusted using the `dev.new` function. For instance, to create a graphical device 9 inches wide, and 4 inches high, I would type:

```
dev.new(width = 9, height = 4)
```

### 9.5 Exporting graphics

To export R graphics, one can often copy snapshots to a Windows clipboard using pull down menus on the `windows` graphical device. These can then be pasted into Windows programs (e.g., word processors) as bit-
maps (a spatially mapped array of bits) or metafiles, a generic term for a file format that can store multiple types of (generally graphical) data. To create the best possible graphs, however, one should save device output using `postscript` (`ps`), portable document format (`pdf`), or other universal high resolution graphical formats. This can be done directly by:

- Using pulldown menus on a `windows` or X11 graphics device itself (currently only metafile and postscript saves are available).
- Clicking on the extent of a `windows` or X11 graphics device and then going to File>Save as on the R-GUI pullmenus (obviously, non-Linux only).
- Using save-associated arguments included in graphical device functions, i.e., `pdf`, `tiff`, `jpeg`, `bmp`, `gif`, `postscript`, and `win.metafile` (+ metafile).

The third alternative allows fine scale control with respect to resolution (dpi), figure size, and other characteristics. For instance, to save a graphics device image as a pdf under the file name `example.pdf` in the working directory I would type:

```
pdf(file = "example.pdf")
```

I would then make the plot, for instance

```
plot(1:10)
```

The plot will not be shown because the graphical device is engaged with `pdf`. As a final step I close the device.

```
dev.off()
```

The graphics file will now be contained in your working directory. If the file argument is unspecified `pdf` will save a file called `Rplot.pdf`.

### Creating high resolution images in R:

By default, BMP, JPEG, PNG and TIFF graphics devices have a `width` and `height` of 480 pixels, and a 'large' point size (1/72 inch) in R. This results in a rather coarse (72 pp) image resolution. Changing the `res` (resolution) argument in a graphical device function without changing the `pointsize`, `height` and `width` arguments will generally result in unusable figures.
Because $500 \approx 72 \times 6$, one can generate a > 400 ppi TIFF called fig1.tiff by typing:

```r
tiff("fig1.tiff", res = 72 * 6, height = 480 * 6, width = 480 * 6)
plot(1:10)
dev.off()
```

With respect to graphical formats, documentation in the `grDevices` package states:

"The PNG format is lossless (data compression without loss of information) and is best for line diagrams and blocks of color. The JPEG format is lossy (data compression in which unnecessary information is discarded), but may be useful for image plots, for example. The BMP format is standard on Windows, and supported by most viewers elsewhere. TIFF is a meta-format: the default format written by the function `tiff` (`compression = none`) is lossless and stores RGB values uncompressed—such files are widely accepted, which is their main virtue over PNG."

So-called scalar vector graphics (SVGs) can be created with the functions `svg`, `cairo_pdf` and `cairo_ps`. All three scripts apply the `cairographics` application programming interface (API). This device will recognize a large number of symbols and fonts not available for document and image generation in the default setting of the `windows` postscript and pdf devices.

### 9.6 Typeface families

Font typefaces can be changed using a number of graphical functions, including `par`, via the argument `family`. The general typeface families: "serif", "mono", and "sans", and the Hershey family of fonts (type `?Hershey` for more information) are transferable across all graphics devices employed in R. To change the font in a graphical device from the default sans serif (similar to Arial) to serif (similar to Times New Roman) one could type `par(family = "serif")`. To use a Courier-type monospace font one would use `par(family = "mono")`.

Many other typeface families are possible, although they may not be transportable to all graphical devices and graphical storage formats. In the code below I bring in a large number of conventional font families using a function from the book website. These typefaces (and many others) will typically be available on Windows platform machines, although not all will be supported by non-windows graphics devices. The result can be seen in Figure 11.

```r
source(url("http://www2.cose.isu.edu/~ahoken/book/win_fonts.R"))
tiff(file = "fonts.tiff", res = 600, width = 7.7, height = 7.7, units = "in")
x <- rep(c(2.8, 6.4, 9.6), each = 33)
y <- rep(seq(10, 0.25, by = -.2965), 3)
font.type <- paste(rep("f", length(fonts)), 1:length(fonts), sep = "")
par(family = "serif")
par(par(family = "mono")
par(par(family = "sans")
par(par(family = "hershey")
for(i in 1:length(fonts)){
text(x[i],y[i], labels = fonts[i], family = font.type[i])
}
dev.off()
```

The figure displays examples of text from ninety-nine Windows typefaces. To save myself from typing an inordinate amount of code, I use a for loop to place the fonts one at a time in the graphics device (see Section 21.6). Note that I use the function `tiff` to create a high resolution .tiff graphical file. Running the entirety of the preceding code chunk will create the image file fonts.tiff in your working directory.

Importantly, the typefaces imported from the first line of code in the chunk will now be available for graphics functions using the `windows` graphical device. To see the available `windows` fonts one can type:

```r
windowsFonts()
```

Similarly, one can see the available fonts for `postscript` and `pdf` graphics devices using:

```r
names(pdfFonts())
```
To plot a dashed line between the points (0, 2) and (1, 3), I would type:
```
lines(x = c(0, 1), y = c(2, 3), lty = 2)
```
or
```
points(x = c(0, 1), y = c(2, 3), lty = 2, type = "l")
```
To place a red inverted triangle at the point (0, 1), I would type:
```
points(x = 0, y = 1, pch = 6, col = 2)
```
Geometric shapes can be drawn using a number of functions including `rect` (which draws rectangles) and `polygon` (which draws shapes based on user-supplied vertices).

## 9.8 plotmath

R has useful functions for the plotting of mathematical expressions. These include the Greek letters, mathematical operators, italicization, and sub- and super-scripts. These are generally called as an expression in the text argument in the functions `text` or `mtext`. For example, to paste the formula for the sample variance

\[ \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \]

into a plot at coordinates (0, 1) I would type:
```
varexp <- expression(over(sum(paste("","",italic(x[i] - bar(x)),",""""^2), italic(i)==1, italic(n)),(italic(n) - 1)))
text(0, 1, varexp)
```
Complete coverage of `plotmath` mathematical expressions would be unwieldy to summarize here. For more information type `?plotmath`.

## 9.9 axis

The function `axis` can be used to create new axes on a plot or to customize axis characteristics. Its first argument (`side`) specifies the side of the plot that the new axis will occupy: 1=bottom, 2=left, 3=top, 4=right. Other arguments include a vector of axis labels (argument `labels`), and the locations of labels.
For instance, to create a right hand axis I would type:

```r
axis(4)
```

### 9.10 mtext

To place text in the margin of a plot we can use the function `mtext`. For its first argument the function requires the character string to be written into the plot. The 2nd argument defines the plot margin to be written on: 
1=bottom, 2=left, 3=top, 4=right.

For instance, to place the text “Axis 2” on the right hand axis I would type:

```r
mtext("Axis 2", 4)
```

### Example 5 --A complex multiplot example

Consider a rather comprehensive `par` example. The object `C.isotope` in the library `asbio` is a dataset describing variations in the quantity δ^{14}C over time in La Jolla California. δ^{14}C is the ratio of carbon 14 to carbon 12 (14C is unstable, while 12C is a stable isotope of carbon) compared to a standard ratio. We will create a figure with four subplots (Figure 12).

- It will have dimensions 8" x 7".
- The outer margins (in number of lines) will be bottom = 0.1, left = 0.1, top = 0, right = 0.
- The inner margins (for each plot) will be bottom = 4, left = 4.4, top = 2, right = 2. The plot margins will be light gray. We can specify gray gradations with the function `gray`. We will use `gray(0.97)`.
- The first plot will show δ^{14}C as a function of date. The plotting area will be dark gray, i.e., `colors()[181]`. Points will be white circles with a black border.
- The second plot will be a line plot of atmospheric carbon as a function of date. It will have a light green plotting area: `colors()[363]`.
- The third plot will be a scatterplot of δ^{14}C as a function of atmospheric carbon. Points will be yellow circles with a black border. The plotting area will be light red: `colors()[580]`.
- The fourth plot will show the sample variance of atmospheric carbon in the time series. It will have a custom (albeit meaningless) axis with the labels: a, b, c, and d. It will also have a horizontal axis label inserted with `mtext`.

```r
library(asbio) # loads the library asbio
data(C.isotope) # dataset in asbio
dev.new(height = 8, width = 7)
op <- par(mfrow = c(2, 2), oma = c(0.1, 0.1, 0, 0), mar = c(4, 4.4, 2, 2), bg = gray(.97))
```
9.11 Histograms

Histograms are vital for considering the distributional characteristics of data. They consist of rectangles whose area is proportional or equivalent to the frequency of particular numeric intervals (bins) describing that variable.

Example 6 – Exploration of Data from Bryce Canyon National Park

The brycesite dataset from library labdsv consists of environmental variables recorded at, or calculated for, each of 160 plots in Bryce Canyon National Park in Southern Utah.

```r
install.packages("labdsv")
library(labdsv)
data(brycesite)
```

Here are the names of the site environmental variables (columns) in the brycesite dataset:

```r
names(brycesite)
```

[1] "annrad" "asp" "av" "depth" "east" "elev" "grorad" "north"

[9] "pos" "quad" "slope"

Let’s look at the distribution of the slope variable (Figure 13). This variable describes slope (in degrees) of sites in the dataset.

```r
with(brycesite, hist(slope, xlab = "Slope (Degrees)", ylab = "Frequency of observations"); main = ">
```
The distribution of slope is strongly right skewed (i.e. most slopes are gradual, and only few are extremely steep).

Consideration of raw aspect values in analyses is problematic because the measurements are circular. As a result the values 1 and 360 are numerically 359 units apart, although they in fact only differ by one degree. One solution is to use the transformation \((1 - \cos(\text{aspect}° - 45))/2\). This index will have highest values on southwest slopes (at 225 degrees), and lowest values on northeast facing slopes (at 45 degrees). This acknowledges the fact that highest temperatures in the Northern Hemisphere occur on Southwest facing slopes because they receive ambient warming during the morning, coupled with late afternoon direct radiation. We have:

\[
\text{asp.val} <- (1 - \cos(((\text{brycesite$asp} - 45) * \pi)/180))/2
\]

The distribution of aspect values in the Bryce Canyon dataset is shown in Figure 14.

We have a bimodal distribution. Specifically, there are a lot of northeast-facing and southwest-facing sites, and fewer northwest and southeast-facing sites.

### 9.12 Subsetting scatterplot arguments using a categorical variable

It is often useful to distinguish points in scatterplots with respect to a categorical variable.

**Example 7**

The plot below shows \textbf{brycesite} radiation (in Langley units) as a function of aspect value. A Langley (Ly) is a measure of energy per unit area, per unit time. To be precise, one Ly = 1 calorie m\(^{-2}\) min\(^{-1}\). In SI units 1Ly = 41840.00 J m\(^{-2}\). The plot also distinguishes five topographic positions using both point color and shape. For clarity I also create a legend. We see that ridgetop sites have mostly northeastern aspect, and hence have lower radiation inputs (Figure 15).

```r
with(brycesite, plot(asp.val, annrad, xlab = "Aspect value", ylab = "Annual radiation (Langleys)", col = as.numeric(pos), pch = as.numeric(pos)))
legend("bottomright", legend = levels(brycesite$pos), pch = 1:5, col = 1:5)
```
To assign colors and plotting characters appropriately, I coerce the categorical topographic position vector, `pos`, to be numeric with `as.numeric`. The result is:

```
as.numeric(brycesite$pos)
```

```
[1] 4 3 3 4 5 3 3 5 3 3 2 2 3 3 3 3 1 2 2 2 5 4 4 3 5 4 3 5 3 2 5 5 4 1
[38] 1 2 4 4 3 3 3 4 3 5 3 3 2 5 3 5 3 5 5 4 3 3 5 3 5 3 2 5 2 2 3 3
[75] 3 2 2 3 3 2 4 3 2 4 2 5 3 2 2 3 5 5 3 3 3 5 3 3 3 3 3 3 3 3 3 1 2 4 1 2
[112] 1 2 3 5 1 5 3 3 3 3 1 3 2 5 2 1 2 1 1 1 1 1 1 1 1 1 4 5 5 5 4 5 2
[149] 2 4 1 5 5 5 3 2 2 1 5 4
```

Ones correspond to the first alphanumeric level in `pos`, `bottom`, whereas fives correspond to the last alphanumeric level, `up_slope`. The color and symbols assignments are made within the plot using:

```
col = as.numeric(brycesite$pos)
pch = as.numeric(brycesite$pos)
```

Legends in R can be created using the function `legend`. The first argument(s) will be a specific X,Y position for the legend, or one of: "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" or "center". The legend argument names the categories to be depicted. The function levels used in this argument lists the categories in a categorical variable alphabetically.

9.13 Plotting variables using additional axes

It may be necessary to add additional axes in order to plot additional variables. In R this will involve laying one plot on top of another, by specifying `par(new = TRUE)`, and defining `axes = FALSE, xlab = FALSE, and ylab = ""` in the arguments of the second plot.

```
Example 8
```

Consider Figure 16 in which both `brycesite` annual radiation and annual growing season radiation are plotted as a function of aspect value.

```
op <- par(mar = c(5,4.5,1,4.5), cex = 1.1)
with(brycesite, plot(asp.val, annrad, xlab = "Aspect value", ylab = "Annual radiation (Langleys)"))
par(new = TRUE)
with(brycesite, plot(asp.val, grorad, pch = 19, axes = FALSE, xlab = "", ylab = ""))
axis(4)
mtext(side=4,"Growing season radiation (Langleys)",line = 3, cex=1.1)
legend("bottomright", pch=c(1,19), legend=c("Annual radiation", "Growing season radiation"),bty = "n")
par(op)
```
Figure 16 Plot of the relationships of annual radiation, growing season radiation, and aspect value.

The line `par(new = TRUE)` tells R to not clean the graphical device before drawing a new plot. The argument `axes = FALSE` in the second plot, suppresses default plot plotting of axis units on the left and bottom axes.

9.14 Barplots

Barplots are frequently used to compare single number summaries (e.g., sum, median, mean, etc.) of categorical levels.

11 Example 9 – Greenhouse gas emissions

Of great concern to both citizens and scientists are rising global levels of atmospheric greenhouse gases. Atmospheric CO$_2$ concentrations have increased approximately 40% since the start of the industrial revolution while more potent greenhouse gases like CH$_4$ and NO$_2$ have increased 150% and 23% respectively (Mann and Kump 2009). We will take a detailed look at recent global patterns of CO$_2$ emissions and human population numbers in this exercise.

The US department of energy has data since 1980 detailing total CO$_2$ emissions from the consumption and flaring of fossil fuels. In addition, midyear population data can be obtained, by country, from the US census bureau. CO$_2$ and population data are available as `world.co2` and `world.pop` respectively in `asbio`.

We will first import these data.

```r
library(asbio); data(world.co2); data(world.pop)
```

To make them easier to call, we will give the datasets shorter names.

```r
co2 <- world.co2
wp <- world.pop
```

The CO$_2$ data has two additional countries (columns) compared to the world population data: Belgium and Ghana. We will get rid of these columns and the "year" column in the CO$_2$ dataset.

```r
co2.1 <- co2[,-c(1, 3, 8)]
```

We will also want to look at the 2006 CO$_2$ data by itself. It is in row 27.

```r
c02.2006 <- co2.1[27,]
```

The names of some of the counties are too long to fit on the X-axis for the barplot we wish to create (Figure 17). We can deal with this in at least four ways. First, we can increase the bottom margin (e.g. `par(mar = c(5, 4, 2, 2))`). Second, we can decrease the font-size of the names using the `cex.names` argument (the default for `barplot` names is `cex.names = 1`). Third, we can make the country names perpendicular to the X-axis instead of parallel using the `las` argument. Lastly, we can simply make the country names shorter by changing the column names in the dataframe or matrix. For instance we could use some variant on:

```r
colnames(co2.1) <- c("Afghan.", "Brazil", "Canada", "China", "Finland", "Italy", "Japan", "Kenya", "Mexico", "S. Arabia", "UAE", "US", "Total")
colnames(co2.2006) <- colnames(co2.1)
```

We use the column names from `co2.1` as the column names for `co2.2006`

```r
barplot(as.matrix(co2.2006),las = 3,ylab = expression(paste("2006 " , CO[2], " Emissions (metric tons x " , 10^6, ")")))
```

We add a horizontal grid to make levels among countries more discernible.

```r
grid(ny = 20, nx = 0)
```
Next we will create a stacked bar graph of carbon emissions from 1980-2006, with countries as bars, and each bar stacked by year (Figure 18). We will leave out world totals to make patterns among countries easier to discern. By default R will impose its own coloring scheme for the stacked bars. We will create a twenty-seven step grayscale coloring scheme of our own by using the argument \texttt{col = gray(seq(0:26)/27)}. Type \texttt{?gray} for more information. We will also include a legend to describe the stacked bars. Type \texttt{?legend} for more information.

\begin{verbatim}
    barplot(as.matrix(co2.1[,1:12]), las = 3, ylab = expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")")), col = gray(seq(0:26)/27))
    legend("topleft", fill = gray(seq(0,26,1)/27), legend = seq(1980, 2006, 1), cex = .5)
\end{verbatim}

By default \texttt{barplot} builds stacked bar plots. To create side-by-side barplots one would use the arguments \texttt{beside = TRUE}. I will make the figure wider to make it easier to view the large number of side-by-side bars (Figure 19).

\begin{verbatim}
    dev.new(height=5,width=8); op <- par(mar=c(5,4.5,2,1))
    barplot(as.matrix(co2.1[,1:12]), las = 3, ylab = expression(paste(CO[2], " Emissions (metric tons x ", 10^6, ")")), col = gray(seq(0:26)/27), beside = T)
    legend("topleft", fill = gray(seq(0,26,1)/27), legend = seq(1980, 2006, 1), cex = .5)
    par(op)
\end{verbatim}
9.15 Multivariate line and scatterplots

As noted in § 9.1, line plots can be generated by specifying `type = "l"` in `plot()`. Lines can also be added to a plot one at a time using `points()` or `lines()` (see § 9.7). A more efficient method, however, uses the function `matplot()`.

Example 10

To illustrate, we will create a line plot with 13 lines (one for each country, and one for world totals) showing the carbon emission variability from 1980 to 2006 (Figure 20). We will make the lines in the plot of different types. We will also include a legend to describe the lines.

```r
matplot(1980:2006, co2.1, ylab = expression(paste(CO[2], " Emissions (metric tons x 10^6)")), type = "l", xlab = "Year", col = c(gray(0:11/15),1), lwd = c(rep(1,12),2), lty = c(seq(1:6),seq(1:6),1), ylim = c(1,200000), log = "y")
```

We see that the US and China have the highest emissions, and that China’s emissions are increasing rapidly. Of additional interest, Afghanistan’s CO2 emissions peaked in the late 80’s, then collapsed as a result of war and political strife.

Note that we log transform the Y-axis to allow better discrimination among low CO2 emitting countries. Finally, we utilize the census data to create a line plot of per capita emissions by country (Figure 21)

```r
wp.1 <- wp[-c(1)] # get rid of the year column
per.cap <- (10^6) * co2.1/wp.1
```
Graphics

58

The United Arab Emirates has, by far, the highest levels of per capita CO₂ emissions.

9.16 Boxplots

Boxplots or box and whisker plots and their variants are an excellent way to quickly summarize and compare the distributions of levels in a categorical variable with respect to a quantitative variable. The function `boxplot` does this by graphically providing a five number summary for factor levels. Specifically, the upper and lower hinges of boxes from `boxplot` show the 1st and 3rd quartiles. The black stripe in the middle of each box shows the median. The whiskers extend to the most extreme data point which is no more than \(\text{coef}\) times the length of the box away from a hinge (by default \(\text{coef} = 1.5\)). Circle symbols are outlying observations (outside the whiskers).

\| Example 11

We can easily create a boxplot of the `slope` variable with respect to shallow or topographic position categories (Figure 17). The data appear to be positively skewed for most topographic categories.

\begin{verbatim}
with(brycesite, plot(slope ~ pos, ylab = "Slope (Degrees)", xlab = "Topographic position"))
\end{verbatim}

9.17 Interval plots

The function `bplot` from `asbio` is a wrapper for `barplot` that creates bars whose heights show location measures (e.g. means, medians, etc.), along with error bars representing, for each factor level, measures of dispersion. Error bar options include standard error (the default), standard deviation, confidence interval, interquartile range, median absolute deviation, or user defined errors. Thus, the function provides a conventional graphical complement to statistical procedures that compare location values of factor levels. Overlaying confidence intervals (the confidence
When considering two quantitative variables as the function of a third quantitative variable, three-dimensional plotting approaches are often useful.

### Example 13 – Taiga/tundra vegetation in Scandinavia

To consider three-dimensional plotting, we will use two datasets from the library `vegan` describing Scandinavian taiga/tundra. Vegetation data are contained in the dataset `varespec` while soil chemistry data for the same sites are contained in the dataset `varechem`.

```r
library(vegan)
```

```r
data(varespec)
data(varechem)
```

Let's look at the distribution of the heath plant *Vaccinium vitis-idaea* (a common species in boreal forest understories) with respect to both pH and % soil nitrogen (Figure 24). Note that we allow symbol sizes to change with the cover of *V. vitis-idaea*.

```r
with(varechem, plot(N, pH, xlab = "% soil N", pch = 16, cex = varespec$Vac.vit/100 + 15))
```

**Figure 24** Cover of *Vaccinium vitis-idaea* with respect to pH and % soil nitrogen. Larger symbols indicate higher percent plant cover.

---

### Example 12

In Figure 23 is an interval plot of the `brycesite` slope variable with respect to topographic position categories. To better distinguish topographies, a gray color gradient is created using `bar.col = gray(1:5/5)`.

```r
library(asbio)
```

```r
with(brycesite, bplot(slope, pos, ylab = "Slope (Degrees)", xlab = "Topographic position", bar.col = gray(1:5/5)))
```

**Figure 23** Slope means and standard errors for topographic levels from the `brycesite` dataset.
Vaccinium vitis-idaea appears to prefer intermediate to low levels of soil N, and acidic soils. The somewhat negative association between soil N and pH is probably due to soil leaching, because H⁺ (and Al³⁺) cations are more strongly adsorbed by soil colloids than bases in poorly drained soils.

A 3D plot of the same associations can be created using the scatterplot3d library (Figure 25).

```r
install.packages("scatterplot3d")
library(scatterplot3d)
Fig <- function(angle = 55){
s3d <- scatterplot3d(cbind(varechem$N, varechem$pH, varespec$Vaccviti),
type="h", highlight.3d = TRUE,
angle = angle, scale = .7, pch = 16, xlab = "N", ylab = "pH", zlab = expression(paste(italic(Vaccinium), " ", italic(vitis-idaea), " % cover"))
lm1 <-lm(varespec$Vaccviti ~ varechem$N + varechem$pH)
s3d$plane3d(lm1)
}
Fig()
```

I define the figure to be a function (named Fig) to allow the angle of rotation for the 3D scatterplot to be easily changed using the angle argument in Fig. By stipulating highlight.3d = TRUE objects that are closer to us in the X plane are given warmer colors. A regression “plane” is also overlaid on the graph. The fitted plane is produced from a linear model (the command lm).

There are a large number auxiliary packages in R specifically for graphics. Several of these, including the popular package lattice (Sarkar 2008) depend on the Trellis graphical system, so-called because it often utilizes a rectangular array of plots, resembling a garden trellis. The lattice package is implemented by grid low-level graphics (Murrel 2005; 2017) whose approach and arguments are non-analogous to R-base functions like plot(), discussed so far. Despite these differences, auxiliary graphics packages generally utilize the base graphics devices described in § 9.3, although see Murrell (2005).

### Example 14

Figure 26 provides an example of graphics generation using the lattice package. The functions levelplot(), contourplot(), and wireframe() are lattice approaches for making three dimensional scatterplots and surfaces. The functions are most easily used when data are in a spatial grid format with row and column numbers defining evenly spaced intervals from some reference point, and cell responses themselves constituting “heights” for the Z (vertical) axis.

The popular volcano dataset, used here, describes the topography of Maungawhau / Mount Eden, a scoria cone in the Mount Eden suburb of Auckland, New Zealand. In this case, rows and columns represent 10m Cartesian intervals. The first row contains elevations (in meters above sea level) for Northernmost points, whereas the first column contains elevations of Westernmost points.

The argument split is a vector of 4 integers, c(x,y,nx,ny) used to position plots in a lattice multiple plot presentation.

```r
library(lattice)
plot(levelplot(volcano, col.regions = heat.colors, xlab = "x", ylab = "y"), split = c(1, 1, 1, 3), more = TRUE, panel.width = list(x = 6,units = "inches"))
plot(contourplot(volcano, cuts = 20, label = FALSE, xlab = "x", ylab = "y", col = "green"), split = c(1, 2, 1, 3), more = TRUE, panel.width = list(x = 6,units = "inches"))
plot(wireframe(volcano, panel.aspect = 0.7, zoom = 1, lwd = 0.01, xlab = "x", ylab = "y", zlab = "z"), split = c(1, 3, 1, 3), more = FALSE, panel.width = list(x = 6,units = "inches"))
```

Figure 25  Cover of Vaccinium vitis-idaea in a 3D plot with respect to pH and % soil nitrogen.

9.19 Auxiliary graphics packages
A great deal of excitement has been generated by the grid-based graphics package ggplot2 (formerly ggplot). Philosophically, ggplot2 emulates Wilkinson’s (2005) “grammar of graphics”, which describes features that underlie all statistical graphics. According to its developer: “ggplot2 ... tries to take the good parts of base and lattice graphics and none of the bad.” A detailed description of ggplot2 attributes can be found in Wickham (2009), and in the ggplot2 manual (https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf).

Probably the easiest to use function in ggplot2 is qplot() (for “quick plot”). The function was intended to be similar to plot() in some respects, and thus acts as a bridge from base R graphics. For instance, both plot and qplot call Cartesian coordinates in initial arguments for creation of scatterplots. The qplot function, however, can implement a wider array of plot types, and facilitate the creation overlays with the argument geom, short for geometric object.

Examples include, for bivariate relationships:

- Scatterplots → geom = “point”, the default graphic if both x and y coordinates are supplied.
- Smoothers → geom = “smooth”, fits a smoother and standard errors to quantitative bivariate data.
- Boxplots → geom = “boxplot”, produces a box-and-whisker plot if x is categorical and y is quantitative.
- Line plot → geom = “line”, Joins points with a line, from left to right.

For univariate displays (in which only x is supplied) useful geoms include:

- Histograms → geom = “histogram”, fits a histogram
- Density plot → geom = “density”, fits relative frequency curves that are constrained, like a probability density function, to have an area of one beneath the curve.

Other geoms can be layered atop a ggplot2 graphic.

Example 15

Figure 27 applies three distinct geoms (“smooth”, “line”, and “point”) to the varchem variables representing soil pH and %N. A curvilinear association between pH and %N is shown by the smoother that was not apparent in a simple scatterplot, i.e., Figure 25.

```r
library(ggplot2)
with(varchem, qplot(N, pH, xlab = "% soil N", geom = c("point", "smooth", "line")))
```
Figure 27 Plot of pH and % soil nitrogen from the varechem dataset. The shaded envelope shows standard errors of the smoother fit. The smooth is generated by the function `loess()`.

`qplot()` assumes that if defined with a vector, point colors (`qplot` argument `colours`) and point shapes (`qplot` argument `shape`), represent a categorical variable, causing the function to print a legend.

**Example 16**
Consider, which ostensibly remakes Figure 15 using `qplot()`.

```r
Position <- brycesite$pos
with(brycesite, qplot(asp.val, annrad, xlab = "Aspect value", ylab = "Annual radiation (Langleys)", colour = Position, shape = Position))
```

One can modify characteristics of `ggplot2` graphics additively using the function `theme()`.

For instance, upon considering Fig. I find, along with other issues, that the margins are too narrow and the axis title and text are too small. Smoothers might be nice as well for each topographic type. Thus, I change the theme in the following ways:

```r
p <- with(brycesite, qplot(asp.val, annrad, xlab = "Aspect value", ylab = "Annual radiation (Langleys)", colour = Position, shape = Position) +
## increase symbol size in plot and legend, disable SEs in smooths
geom_point(size=3) + geom_smooth(se = F))
## change margins
p <- p + theme(plot.margin = unit(c(1,1,1.25,1), "cm"))
## Increase y-axis label size
p <- p + theme(axis.title.y = element_text(size=13, hjust = 0.5, vjust = 7))
```
Data structures

There are five primary data structures in R. We will use each of them repeatedly.

10.1 Vectors

In R a vector is collection of data with order and length, but no dimension. This is very different than the matrix algebra definition of a vector. In this case a row vector with n elements has dimension 1 x n (1 row and n columns), whereas a column vector has dimension n x 1. We can create vectors with the function `c`. Recall that `c` means combine.

```r
x <- c(1, 2, 3)
is.vector(x)
[1] TRUE
length(x)
[1] 3
dim(x)
NULL
dim(as.matrix(x))
[1] 3 1
```

The function `as.matrix` coerces the vector `x` to have a matrix structure with dimension 3 x 1. Thus, in R a matrix (see below) has dimensions, but a vector does not. Elements in vectors must have a single data storage mode: e.g., "integer", "double", "complex", "character". That is, a vector cannot contain both numeric and categorical data.

When an operation is simultaneously applied to two unequal length vectors, R will generate a warning and automatically recycle elements of the shorter vector, beginning with its first element, until it is has the same length as the longer one. For instance:

```r
c(1, 2, 3) + c(1, 0, 4, 5, 13)
[1]  2  2  7  6 15
```

Warning message:

```
In c(1, 2, 3) + c(1, 0, 4, 5, 13): longer object length is not a multiple of shorter object length
```

In this case the result of the addition of the two vectors is 1 + 1, 2 + 0, 3 + 4, 1 + 5, and 3 + 13. This is because the first two elements in the first object are recycled in the addition. R users should obviously be aware of default
presence of recycling and its potentially detrimental effects to analyses.

10.2 Matrices

Matrices are two-dimensional (row and column) data structures whose elements are all comprised of a single type of data: quantitative, categorical, or ordinal. The function `matrix` can be used to create matrices.

```r
a <- c(1, 2, 3, 2, 3, 4)
matrix(ncol = 2, nrow = 3, data = a)
[,1] [,2]
[1,] 1 2
[2,] 2 3
[3,] 3 4
```

10.3 Arrays

Arrays are one, two (matrix), or three or more dimensional data structures whose elements contain a single type of data. The function `array` can be used to create arrays. Below we create a 2 x 2 x 2 array using data using the object `a` from above.

```r
a <- c(1, 2, 3, 4, 5, 6, 7, 8)
array(a, c(2, 2, 2))
, , 1
 [,1] [,2]
[1,] 1 3
[2,] 2 4
, , 2
 [,1] [,2]
[1,] 5 7
[2,] 6 8
```

The first argument in `array` defines the data, while the second argument is a vector that defines both the number of dimensions (this will be the length of the vector) and the number of levels in each dimension (numbers in dimension elements). The function above took the first half of observations in the object `a` and put them in the first level of component 1. It put the other four observations in the second level of component 1. The four elements in each component are arranged into 2 x 2 matrices.

Arrays are useful for containing results made up of multiple matrices or dataframes (see below). For instance, a Markov Chain Monte Carlo (MCMC) analysis of the dataset `cuckoo` requires six parameters, each of which will be described with multiple Markov chains. Below I specify the creation of two chains, each comprised of three steps. The output is an array.

```r
data(cuckoo); mcmc.norm.hier(cuckoo, 3, 2)
, , Chain# 1
 theta1 theta2 theta3 mu s.sq tau.sq
1 23.25000 23.85000 23.85000 23.65000 1.7612736 0.0551980
2 23.71469 23.29092 23.33157 23.33031 1.3925030 2.8243915
3 23.37465 22.92141 22.73779 22.73779 0.8880371 3.9218969
, , Chain# 2
 theta1 theta2 theta3 mu s.sq tau.sq
1 23.85000 23.05000 23.05000 23.31667 0.7906798 1.473440 2
2 23.13697 23.26426 22.78453 24.19186 0.8380785 4.067955 3
3 23.17203 23.25774 22.59440 22.41249 0.5772941 46.341983
```

10.4 Dataframes

Dataframes are two-dimensional data structures whose columns can contain different types of data, e.g. quantitative or categorical. Note that all data in a single column must be of the same class, e.g. `numeric`, `factor`, etc. Each column and row in a dataframe may be given an identifying label. Labels can be assigned in a number of ways, including within the `data.frame` function (see example below). Confusingly, dataframe columns can be named with the function `names`, whereas a parallel approach with a matrix object requires the function `colnames`. The function `rownames` works for assigning row names in both dataframes and matrices. Columns can be called by name in a dataframe using the functions `attach` or `with`, or the operator `$`.

Here we apply the function `data.frame` to create a dataframe called `data`.

```r
data <- data.frame(numeric = c(1, 2, 3), non.numeric = c("a", "b", "c"))
data
numeric non.numeric
1 1 a
2 2 b
3 3 c
```

Here we access the column in `data` called `non.numeric`.

```r
data$non.numeric
[1] a b c
Levels: a b c
```
10.5 Lists

Lists are often used to contain miscellaneous associated objects. Like dataframes, lists need not contain a single type of data (e.g. categorical or quantitative). Unlike dataframes, however, lists can simultaneously include objects with different data classes including strings (i.e. units of character variables), matrices, dataframes and even other lists. Thus, lists do not require a row-column structure, and list components need not be the same length. Recall that the R-object we produced earlier, ha.lm, from the creation of a linear model, was a list. Like dataframes, objects in lists can be called using the expression $\$. The function list can be used to create lists.

```r
data <- list(a = c(1, 2, 3), b = "this.is.a.list")
data
$A
[,1]
[1,]  1
[2,]  2
[3,]  3

$b
[1] "this.is.a.list"
```

List elements can also be identified using double square brackets. Below is the first list component of the list data.

```r
data[[1]]
[1] 1 2 3
```

### Example 17 -- Downs syndrome data

Let’s create a dataframe with three numeric columns using data in Table 4. Note that this is part of a dataset for Down’s syndrome collected in British Columbia by the British Columbia Health Surveillance Registry (Geyer 1991).

<table>
<thead>
<tr>
<th>Mothers age</th>
<th>Number of births</th>
<th>Number of Down’s syndrome cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>13555</td>
<td>16</td>
</tr>
<tr>
<td>20.5</td>
<td>22005</td>
<td>22</td>
</tr>
<tr>
<td>21.5</td>
<td>23896</td>
<td>16</td>
</tr>
<tr>
<td>29.5</td>
<td>15685</td>
<td>9</td>
</tr>
<tr>
<td>30.5</td>
<td>13954</td>
<td>12</td>
</tr>
<tr>
<td>38.5</td>
<td>4834</td>
<td>15</td>
</tr>
<tr>
<td>39.5</td>
<td>3961</td>
<td>30</td>
</tr>
<tr>
<td>40.5</td>
<td>2952</td>
<td>31</td>
</tr>
<tr>
<td>44.5</td>
<td>596</td>
<td>22</td>
</tr>
<tr>
<td>45.5</td>
<td>327</td>
<td>11</td>
</tr>
<tr>
<td>47</td>
<td>249</td>
<td>7</td>
</tr>
</tbody>
</table>

We will give this data subset the name Downs.

```r
Downs <- data.frame(Age = c(17, 20.5, 21.5, 29.5, 30.5, 38.5, 39.5, 40.5, 44.5, 45.5, 47), Births = c(13555, 22005, 23896, 15685, 13954, 4834, 3961, 2952, 596, 327, 249), Cases = c(16, 22, 16, 9, 12, 15, 30, 31, 22, 11, 7))
```

Columns in Downs can be called without attaching the dataframe by using the $ expression.
A question of obvious importance is: "how do I get my data into R?" The answer is: "two ways." First, one can enter data “by hand” at the command line. Second, one can read in data files. We concentrate on command line entry in this section. Data import is described in § 12.

11  scan, cbind, rbind

As we know data can be combined into a single entity with the function c.

\begin{verbatim}
a <- c(1, 2, 3); b <- c(2, 3, 4)
\end{verbatim}

To create an R-object containing character strings, e.g., a categorical variable, we will need to place quotation marks around entries.

\begin{verbatim}
x <- c("low", "med", "high")
x
\end{verbatim}

Command line data entry is made easier with the function scan (which can also be used for file import) because a prompt is given for each data point, and separators are created by the function itself. For instance:

\begin{verbatim}
a <- scan()
1: 1 2 3
4:
Read 3 items
\end{verbatim}

The function will be terminated by a blank line or an end of file (EOF) signal. These will be Ctrl+D in Unix and Ctrl+Z in Windows.

We can use cbind to combine columns,

\begin{verbatim}
cbind(a, b)
a b
[1,] 1 2
[2,] 2 3
[3,] 3 4
\end{verbatim}

while rbind lets us combine rows.

\begin{verbatim}
rbind(a, b)
[,1] [,2] [,3]
a 1 2 3
b 2 3 4
\end{verbatim}
Data import

11.2 Facilitating command line entry: \texttt{seq} and \texttt{rep}

R has a number of functions that can speed up command line data entry. For instance, what if I want to create a sequence from 1990 to 2008? I would type: \texttt{seq(1990, 1998)}, or \texttt{seq(1990 : 1998)}, or simply \texttt{1990 : 2008}

\begin{verbatim}
1990 : 2008
\end{verbatim}

The first two arguments in \texttt{seq} are the start and end of the sequence (unless a sequence is specified in the first argument). The third argument specifies the increment between items in the sequence. For example, if I wanted the vector 1990, 1992, 1994, 1996, 1998, I could simply type:

\begin{verbatim}
seq(1990, 1998, 2)
\end{verbatim}

One can easily create a vector with repeated values using the function \texttt{rep}. For example, to repeat the sequence 1991, 1992, 1993, 1994 ten times. I could simply type:

\begin{verbatim}
\end{verbatim}

The first argument, \texttt{c(1991, 1992, 1993, 1994)}, defines the thing we want to repeat. The second argument, 10, specifies the number of repetitions.

Using these foundations we can create extremely complex sequences. For instance, to compose a sequence in which 1991, 1992, and 1993 were each repeated twice, and then to repeat that sequence three times we have:

\begin{verbatim}
rep(c(1991 : 1993, each = 2, times = 3)
\end{verbatim}

\texttt{Data import}

12  Importing data into R

While it is possible to enter data into R at the command line (§ 11) this will normally be inadvisable except for small datasets. In general it will be much easier to import data. R can import data from many different kinds of formats including .txt, and .csv (comma separated) files, and files with space, tab, and carriage return datum separators. I generally organize my datasets using Excel or some other spreadsheet program (although R can handle much larger datasets than these platforms; § 12.5), then save them as .csv files. I then import the .csv files into R using the \texttt{read.table}, \texttt{read.csv}, or \texttt{scan} functions. As noted in the function \texttt{load} can be used to import data files in rda data formats, or other R objects. The program R studio (§ 21.12) allows menu-driven import of file types from a number of spreadsheet and statistical packages including Excel®, SPSS®, SAS®, and Stata®, making the sections below largely unnecessary.

\begin{verbatim}
read.table("C:/Users/User/Documents/veg.csv", sep = ",", header = TRUE, row.names = 1, na.strings = ".")
\end{verbatim}

Data can also be read directly from the working directory. For instance:

\begin{verbatim}
read.table("C:/Users/User/Documents/veg.csv", sep = ",", header = TRUE, row.names = 1, na.strings = ".")
\end{verbatim}

\begin{verbatim}
read.table("C:\R.data\veg.csv", sep = ",", header = TRUE, row.names = 1, na.strings = ".")
\end{verbatim}

\texttt{Data import}
read.table("veg.csv", sep = ",", row.names = 1, header = T)

12.2 read.csv

The function read.csv has the same arguments as read.table with the exception that data separators are assumed to be commas, precluding the necessity of the sep argument. Thus, for the example above we would have:

read.csv("veg.csv", header = TRUE, row.names = 1, na.strings = ".")

12.3 scan

The function scan can read in data from an essentially unlimited number of formats, and is extremely flexible with respect to character fields and storage modes of numeric data

In addition to arguments used by read.table, scan has the arguments:

• what which describes the storage mode of data e.g., logical, integer, etc., or if what is a list, components of variables including column names (see below), and
• dec which describes the decimal point character (European scientists and journals often use commas).

Assume that veg.csv has column of species names, called species, that will serve as row names, and 3 columns of numeric data, named site1, site2, and site3. We would read the data in with scan using:

scan("veg.csv", what = list(species = "", site1 = 0, site2 = 0, site3 = 0), na.strings = ".")

The empty string species = "" in the list comprising the argument what, indicates that species contains character data. Stating that the remaining variables equal 0, or any other number, indicates that they contain numeric data.

12.4 Easy imports: use of file.choose()

Possibly the easiest way to import data is to use read.csv, read.table, or scan with file.choose function as the file argument. For instance, by typing:

read.csv(file.choose())

we can now browse for .csv files to open.

Other arguments (e.g., header, row.names) will need to be used, when appropriate, to import the file correctly.

12.5 Additional comments

It is generally recommended that datasets imported and used by R be smaller than 25% of the physical memory of the computer. For instance, they should use less than 3GB on a 32-bit operating system. Note that this still equates to a roughly 13,700 x 8000 element data array. R can handle extremely large datasets, i.e. > 10GB, and > 1.2 x 10^10 rows. In this case specific R packages can be used to aid in efficient data handling. Parallel “cluster” computing and workstation modifications may allow even greater efficiency. The actual upper physical limit for an R dataframe is 2 x 10^31-1 elements. Note that this exceeds the latest limits for Excel worksheets by 21 orders of magnitude (Excel 2010 worksheets can handle approximately 1.7 x 10^10 cell elements).

R allows interfacing with a number relational database storage platforms. These include open source entities that express queries in SQL (Structured Query Language). For more information see Chambers (2008, pg. 178) and Adler (2010, pg. 157).
13 Exporting data from R

It is easy to export data from R.

The functions write.table and write.csv let one write output for a large number of formats. For example, the commands below will write the matrix test to the working directory as a .csv file.

```r
test <- matrix(nrow = 2, ncol = 2, data = c(2, 1, 3, 4))
write.csv(test, "test.csv", sep = ",")
```

- The first argument in write.table is the object I wish to export, i.e. test.
- The second argument tells R where to export the object, and the storage name of the object.
- The third argument, sep, tells R what kind of separator to use to distinguish data entries. The function write.csv facilitates the creation of .csv spreadsheets and assumes sep = ",".

Several other optional arguments in write.table are important.

- col.names = TRUE indicates that the first row of data are column names.
- row.names = 1 indicates that the first column of data are row names.
- na = "NA" indicates that missing values are specified in the data matrix with "NA".

We can also use the write.table command to “copy” data to a clipboard. For instance:

```r
write.table(test, "clipboard", sep = "\t", col.names = NA)
```

Now we can go to Excel® or Word® or some other program and paste the information using toolbars or Ctrl+V.

One can save a function, dataframe, or data matrix as a binary .rda or .RData file using the save function. For instance:

```r
save(test,file = "test.RData")
```

saves test.RData to the working directory.

14 Subsetting matrix, dataframe and array components

R allows us to easily specify particular subsets of dataframe, matrix or array using subset brackets, i.e. [ ].

Gaining skills with subsets will greatly enhance one’s ability to manipulate datasets in R.

A dataframe or matrix name followed by brackets with a comma preceding a number inside the brackets, indicates a column number, i.e., [,column number]. For instance, the command `Downs[,1]` specifies column 1 in the dataframe Downs defined on page 73.

```r
Downs[,1]
[1] 17.0 20.5 21.5 29.5 30.5 38.5 39.5 40.5 44.5 45.5 47.0
```

A dataframe or matrix name followed by brackets with a comma following a number inside the brackets, indicates a row number, i.e., [row number,]. For instance, the command `Downs[1,]` specifies row 1 in Downs.

```r
Downs[1,]
     Age Births Cases
1    17 13555    16
```

Brackets without commas can be used to subset individual elements in a data matrix. It should be noted that by default R reads datasets by column. For instance, the command `Downs[16]` indicates the sixteenth element in the Downs dataset, which also happens to be fifth element in column 2.

```r
Downs[16]
[1] 13954
```

Note that in the operation above I convert the Downs dataframe to a matrix. This will force the subset (element 16) to work correctly. For more information on coercing R-objects see § 18.

The command `Downs[1, 1]` or `Downs[c(1, 1)]` specifies element 1 in column 1.

```r
Downs[1, 1]
 Age
17
```

The command `Downs[c(5:8),]` specifies rows 5 through 8 in Downs.
Matrix operations

15 Operations on matrices and dataframes

Operators can be applied individually to every row or column of a matrix using a number of time saving methods. Actual R-applications for linear algebra (e.g. matrix multiplication, matrix inverses, eigenanalysis and matrix decompositions) are described in the mathematical Appendix in the Foundational and Applied Statistics textbook.

Example 18

As a simple example we will plot Down’s syndrome cases per live birth from the Downs dataset. To do this I would simply divide the Cases column by the Births column and plot the result as a function of Age (Figure 30). To accomplish this using variable (column) names we first coerce (§ 18) Downs back into a dataframe.

```r
Downs <- as.data.frame(Downs)  # coerce back to dataframe
with(Downs, plot(Age, Cases/Births))
```

Rows, columns, and layers from arrays can also be accessed using subset brackets. For instance, to obtain all the row and columns from the second layer of the array obj I would type:

```r
obj[, , 2]
```

Note: when a subset results in zero observations for a level in a categorical variable, then it will be useful to use `droplevels()` to remove the empty levels. This is because R will consider these levels to still be an implicit part of the subset data, potentially complicating or preventing analyses.

Dataframe columns can also be accessed by name when using square brackets.

```r
Downs["Age"]
```

Converting back to a data frame allows the column names in Downs to be recognizable. We can see that the rate of Down’s syndrome increases dramatically in older mothers.

Figure 30 The rate of Down’s syndrome births as a function of mother’s age.
Matrix operations

15.1 apply

Operations can be performed quickly on matrices with the function `apply`. The function requires three arguments.

- In the first argument, `X`, we specify a matrix, array, or dataframe to be analyzed.
- In the second argument, `MARGIN`, we specify whether rows or columns are to be analyzed (1 indicates rows, 2 indicates columns, while `c(1, 2)` indicates rows and columns).
- In the third argument, `FUN`, we specify a function to be applied to the margins of the object in the first argument.

```
max.val <- apply(Downs, 2, max)
max.val
 Age   Births   Cases
 47 23896      31
```

In the operation above I created an object called `max.val` that contains the maximum value from each column of `Downs`. Similarly the command below creates a vector made up of the minimum responses at each row.

```
min.val <- apply(Downs, 1, min)
min.val
[1] 16.0 20.5 16.0  9.0 12.0 15.0 30.0 31.0 22.0 11.0  7.0
```

We can use the `apply` command to apply any statistical function (i.e. `mean`, `sd`, `median` etc.) to all the rows and/or columns of a matrix.

```
means <- apply(Downs, 2, mean)
means
 Age     Births      Cases
34.04545 9274.00000   17.36364
```

Several summary statistical functions exist for matrices that can be used in the place of `apply`. These include `rowMeans` and `colMeans` which give the sample means of specified rows and columns, respectively, and `rowSums` and `colSums` which give the sums of specified rows and columns, respectively. For instance:

```
colMeans(Downs)
 Age     Births      Cases
34.04545 9274.00000   17.36364
```

15.2 tapply

Imagine that we have a categorical variable in the `Downs` dataset with two factor levels (categories). The first factor level is associated with the first 6 experimental units, while the second level is associated with the last 5 experimental units. That is,

```
Categories <- factor(c(rep(1, 6), rep(2, 5)))
cbind(Downs, Categories)
Age   Births   Cases Categories
 1  17.0  13555    16          1
 2  20.5  22005    22          1
 3  21.5  23896    16          1
 4  29.5  15685     9          1
 5  30.5  13954    12          1
 6  38.5   4834    15          1
 7  39.5   3961    30          2
 8  40.5   2952    31          2
 9  44.5    596    22          2
10 45.5    327    11          2
11 47.0    249     7          2
```

The mixture of categorical and quantitative variables is allowed because `Downs` is a dataframe.

We can easily summarize our data with respect to the categories in `Categories` by using the function `tapply`. Like `apply`, `tapply` requires three arguments.

- The first argument, `X`, specifies which vector to evaluate.
- The second argument, `INDEX`, will be vector of categories that can be used to subset `X`.
- The third argument, `FUN`, describes the function to be applied to `X` for each level in `INDEX`.

```
tapply(X = Downs[,2], INDEX = Categories, FUN = mean)
1 15654.83 1617.00
```

15.3 outer

Another important function for matrix operations is the function `outer`. The function lets us create an array that contains all possible combinations of two vectors or arrays with respect to a particular function. For example, suppose I wished to find the mean of all possible pairs of observations from a vector. I would type the following commands:

```
max.val <- apply(Downs, 2, max)
max.val
 Age   Births   Cases
 47 23896      31
```

The mixture of categorical and quantitative variables is allowed because `Downs` is a dataframe.

We can use the `apply` command to apply any statistical function (i.e. `mean`, `sd`, `median` etc.) to all the rows and/or columns of a matrix.

```
means <- apply(Downs, 2, mean)
means
 Age     Births      Cases
34.04545 9274.00000   17.36364
```

Several summary statistical functions exist for matrices that can be used in the place of `apply`. These include `rowMeans` and `colMeans` which give the sample means of specified rows and columns, respectively, and `rowSums` and `colSums` which give the sums of specified rows and columns, respectively. For instance:

```
colMeans(Downs)
 Age     Births      Cases
34.04545 9274.00000   17.36364
```

15.2 tapply

Imagine that we have a categorical variable in the `Downs` dataset with two factor levels (categories). The first factor level is associated with the first 6 experimental units, while the second level is associated with the last 5 experimental units. That is,

```
Categories <- factor(c(rep(1, 6), rep(2, 5)))
cbind(Downs, Categories)
Age   Births   Cases Categories
 1  17.0  13555    16          1
 2  20.5  22005    22          1
 3  21.5  23896    16          1
 4  29.5  15685     9          1
 5  30.5  13954    12          1
 6  38.5   4834    15          1
 7  39.5   3961    30          2
 8  40.5   2952    31          2
 9  44.5    596    22          2
10 45.5    327    11          2
11 47.0    249     7          2
```

The mixture of categorical and quantitative variables is allowed because `Downs` is a dataframe.

We can easily summarize our data with respect to the categories in `Categories` by using the function `tapply`. Like `apply`, `tapply` requires three arguments.

- The first argument, `X`, specifies which vector to evaluate.
- The second argument, `INDEX`, will be vector of categories that can be used to subset `X`.
- The third argument, `FUN`, describes the function to be applied to `X` for each level in `INDEX`.

```
tapply(X = Downs[,2], INDEX = Categories, FUN = mean)
1 15654.83 1617.00
```

15.3 outer

Another important function for matrix operations is the function `outer`. The function lets us create an array that contains all possible combinations of two vectors or arrays with respect to a particular function. For example, suppose I wished to find the mean of all possible pairs of observations from a vector. I would type the following commands:
The command `rnorm(16)` generates 16 random values from a standard normal distribution (Ch. 3). Say that we wish to assign distinct treatments to the columns and stack them. Applying `stack` we have:

```r
st <- stack(s)
```

```
   values  ind
1  -0.04941294 X1
2   -1.37260663 X1
3    2.21704986 X1
4    0.89460072 X1
5    0.25489102 X2
6    0.81068275 X2
7    1.04691132 X2
8   -0.83447166 X2
9    2.72308681 X3
10   0.48653472 X3
11   0.12804772 X3
12    0.89494172 X3
13   -0.54162277 X4
14   -1.11219031 X4
15   -1.60632784 X4
16    0.32180719 X4
```

Unstacking we have:

```r
unstack(st)
```

```
     X1        X2        X3        X4
[1,] -0.04941294  0.2548910  -0.5416228  0.3218072
[2,] -1.37260663 -1.1121903 -1.6063278 -1.0469113
[3,]  2.21704986  1.0469113  0.1280477  0.8949417
[4,]  0.89460072  0.8106827  0.8949417  0.3218072
```

The upper and lower triangles of the matrix `o` contain the pairwise means while the diagonal contains the means of the objects with themselves. That is, the diagonal contains the original data in `x`.

### 15.4 lower.tri, upper.tri and diag

We can use the commands `lower.tri`, `upper.tri` and `diag` to examine the upper triangle, lower triangle, and diagonal parts of a matrix. For instance,

```r
o[upper.tri(o)]
```

```
[1]  1.5  2.0  2.5  3.0  3.5  4.0  2.5  3.0  3.5  4.5
```

```r
o[lower.tri(o)]
```

```
[1]  1.5  2.0  3.0  2.5  2.5  3.5  3.0  4.0  3.5  4.5
```

```r
diag(o)
```

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

Note that I use square brackets to subset the data in `o`.

### 15.5 stack and unstack

When manipulating matrices and dataframes it is often useful to stack and unstack columns. These operations are handled with the functions `stack` and `unstack`. Consider the 4 x 4 dataframe below.

```r
s <- data.frame(matrix(nrow=4, ncol=4, rnorm(16)))
```

```r
s
```

```
[1,] -0.08879353  0.5075496  0.8077366  0.6685438
[2,] -0.84689275 -1.1091304 -0.1530534 -1.7209322
[3,]  2.33278611  1.1339444 -0.3599148  0.2705621
[4,]  0.06433390  0.2264004  0.2190580  0.3001240
```

The command `rnorm(16)` generates 16 random values from a standard normal distribution (Ch. 3). Say that we wish to assign distinct treatments to the columns and stack them. Applying `stack` we have:

```r
st <- stack(s)
```

```
   values  ind
1  -0.04941294 X1
2   -1.37260663 X1
3    2.21704986 X1
4    0.89460072 X1
5    0.25489102 X2
6    0.81068275 X2
7    1.04691132 X2
8   -0.83447166 X2
9    2.72308681 X3
10   0.48653472 X3
11   0.12804772 X3
12    0.89494172 X3
13   -0.54162277 X4
14   -1.11219031 X4
15   -1.60632784 X4
16    0.32180719 X4
```

Unstacking we have:

```r
unstack(st)
```

```
     X1        X2        X3        X4
[1,] -0.04941294  0.2548910  -0.5416228  0.3218072
[2,] -1.37260663 -1.1121903 -1.6063278 -1.0469113
[3,]  2.21704986  1.0469113  0.1280477  0.8949417
[4,]  0.89460072  0.8106827  0.8949417  0.3218072
```
16 Logical commands

Computer languages like R that can dichotomously classify true and false statements are called logical or Boolean. R uses the following logical operators:

- `>` “greater than”
- `>=` “greater than or equal to”
- `<” “less than”
- `<=” “less than or equal to”
- `==” “equal to”
- `!=” “not equal to”
- `&” “and”
- `|” “or”

In R logical queries, comparisons, or commands will return the Boolean categories TRUE and FALSE.

// Example 19
We will demonstrate the use of logical commands with the Downs dataset.

```r
attach(Downs)
Age >= 30
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
Age != 30.5
[1] TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE
Age != 30.5 & Age < 40
[1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
Age < 30.5 | Age == 47
[1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

We can subset data with logical commands by using square brackets. For instance the following code requests age data in the Downs dataset less than 30.5, or equal to 47.

```r
Age[Age < 30.5 | Age == 47]
[1] 17.0 20.5 21.5 29.5 47.0
```

Using the function `subset` we have:

```r
subset(Age, Age < 30.5 | Age == 47)
[1] 17.0 20.5 21.5 29.5 47.0
```

16.1 ifelse

A number of functions can be used in combination with a logical argument to evaluate a vector and provide outcomes if the argument is true or false. One example is the `ifelse` function. It requires three arguments

- The first argument, *test*, gives the logical test to be evaluated.
- The second argument, *yes*, provides the output if the test is true.
- The third argument, *no*, provides the output if the test is false. For instance:

```r
ifelse(Age < 25, "Young", "Not so young")
[1] "Young"  "Young"  "Young"  "Not so young" "Not so young"
[6] "Not so young" "Not so young" "Not so young" "Not so young" "Not so young"
[11] "Not so young"
```

16.2 if, else, any, and all

A more generalized approach to providing a condition and then defining the consequences uses the functions `if` and `else`. For instance:

```r
if(any(Age < 25)) "Young" else "Not so Young"
[1] "Young"
```

The `any` function looks through the vector Age to see if any of the elements meet the specified criterion. Conversely, the function `all` will look through a vector or vectors to see if all of the specified elements meet the specified criterion.

```r
if(all(Age < 25)) "Young" else "Not so Young"
[1] "Not so Young"
```
17 Simple functions for data management

An attractive attribute of R is its capacity to efficiently manage large, complex datasets. In this section I list a few functions and approaches useful for data management.

17.1 replace

We can replace elements in a vector with the function `replace`. The function requires three arguments.

- The first argument, `x`, specifies the vector to be analyzed.
- The second argument, `list`, tells R which elements need to be replaced. A logical argument can be used here as a replacement index.
- The third argument, `values`, tells R what these elements need to be replaced with.

For instance:

```r
replace(Age, Age < 25, "R is Cool")
```

```
[1] "R is Cool" "R is Cool" "R is Cool" "29.5"    "30.5"    "38.5"
[7] "39.5"    "40.5"    "44.5"    "45.5"    "47.0"
```

Recall that a vector is not allowed to contain both quantitative and categorical data. As a result R made all of the output from `replace` categorical.

17.2 which

The function `which` can be used with logical commands to subset data. For instance:

```r
which(Age > 30)
```

```
[1]  5  6  7  8  9 10 11
```

The result tells us which elements in Age were in agreement with the logical argument `Age>30`.

To find which element in Age is closest to 32 I type:

```r
which(abs(Age - 32) == min(abs(Age - 32)))
```

```
[1]  5
```

According to R, the 5th element in the Age is closest to 32. We can see that this is true.

```r
abs(Age - 32)
```

```
[1] 15.0 11.5 10.5  2.5  1.5  6.5  7.5  8.5 12.5 13.5 15.0
```

Three other functions related to `which` are `sort`, `rank`, and `match`.

17.3 sort

The function `sort` sorts alphanumeric data from a vector into an ascending order.

```r
sort(Age)
```

```
[1] 17.0 20.5 21.5 29.5 30.5 38.5 39.5 40.5 44.5 45.5 47.0
```

```r
sort(c("a", "d", "c", "Inf"))
```

```
[1] "a"   "c"   "d"   "Inf"
```

Data can be sorted in a descending order by specifying `decreasing = T`.

```r
sort(Age, decreasing = T)
```

```
[1] 47.0 45.5 44.5 40.5 39.5 30.5 29.5 21.5 20.5 17.0
```

17.4 rank

The function `rank` gives the ascending alphanumeric rank of elements in a vector. Ties are given the average of their ranks. This operation is important to rank-based permutation analyses (see Ch. 6 in the Foundational and Applied Statistics text).

```r
x <- c(1, 2, 3, 4, 4, 3)
rank(x)
```

```
[1] 1.0 2.0 3.5 5.5 5.5 3.5
```

17.5 order

The function `order` is more difficult to understand than `sort` and `rank`. It alphanumerically sorts a vector and returns the original element order overlaid on the sorted vector. This allows us to sort a vector, matrix or dataframe into an ascending or descending order, based on one or several vectors. For instance:

```r
x <- c(1, 3, 2, 4, 0.5)
o <- order(x)
o
```

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

The 5th element in x has the smallest value, thus the number five is placed first in vector o. The 1st element in x has the next smallest value, thus the number one is placed second in o. The 3rd element is the next smallest, and so on.

We see here how we would utilize the vector o.
list of species codes from a bird survey on islands in Southeast Alaska. Notice that here are a large number of repeats.

```
```

Using `unique` we have:

```
unique(AK.bird)
```

```
[1] "GLGU" "MEGU" "DOCO" "PAJA" "COLO" "BUFF" "COGO" "WHSC" "TUSW" "GRSC" "GRTE"  
[12] "REME" "BLOY" "REPH" "SEPL" "LESA" "ROSA" "WESA" "WISN" "BAEA" "SHOW" "MADU"  
[23] "COSN" "ROHA"
```

17.7 `match`

Given two vectors, the function `match` finds where objects in the second vector appear in the elements of the first vector. For instance:

```
x <- c(6, 5, 4, 3, 2, 7)
y <- c(2, 1, 4, 3, 5, 6)
match(y, x)
```

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

The number 2 (the 1st element in `y`) is the 5th element of `x`, thus the number 5 is put 1st in the vector `m` created from `match`. The number 1 (the 2nd element of `y`) does not occur in `x` (it is NA). The number 4 is the 3rd element of `y` and `x`, thus number 3 is placed in the third element of `m`, and so on.

The value of this function may seem unclear at first, but consider a scenario where I want to convert field data with species codes into a dataset containing species names. Consider the following species list (which includes species not in the field data from § 17.5).

```
```

17.6 `unique`

To find unique values in dataset (and eliminate unwanted repeats) we can use the function `unique`. Here is
Functions for data management

• The first argument, `x`, specifies the character string to be analyzed.
• The second argument, `split`, is a character criterion that is used for splitting. Letting this argument equal `NULL` (§ 20.219) results in spaces being placed between every character in the string.

```
noquote(strsplit("Achillea millefolium", NULL))
```

\[
[[1]] \text{ Achillea millefolium}
\]

To split the string whenever the letter "l" occurs, I have:

```
noquote(strsplit("Achillea millefolium", "l"))
```

\[
[[1]] \text{ Achi} \quad \text{ea mi} \quad \text{efo ium}
\]

The function `noquote()` removes quotes when printing.

The function `strtrim` is useful for extracting characters from vectors. For instance, for the species codes in the `plant` character vector below, the first capital letter indicates whether the species are flowering plants (anthophytes) or mosses (bryophytes). Assume I want to create a new categorical variable distinguishing anthophytes from bryophytes by extracting the first letter. This is defined by specifying `1` in the second `strtrim` argument, `width`.

```
plant <- c("A_CAAT", "B_CASP", "A_SARI")
strtrim(plant, 1)
```

\[
[1] \text{ A} \quad \text{B} \quad \text{A}
\]

17.8 which and `%in%`

We can use the commands `%in%` and `which` together to achieve the same results as `match`. Under the current example we have:

```
m <- which(species.list[,1] %in% field.data[,1])
field.data[,1] <- species.list[,2][m]
field.data
```

<table>
<thead>
<tr>
<th>code</th>
<th>names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Achillea millefolium</td>
<td>12</td>
</tr>
<tr>
<td>Elymus scriberi</td>
<td>13</td>
</tr>
<tr>
<td>Carex elynoides</td>
<td>14</td>
</tr>
<tr>
<td>Taraxacum ceratophorum</td>
<td>11</td>
</tr>
</tbody>
</table>

%in% returns a logical vector, indicating if a match was located for each element in user-supplied vectors. Thus, unlike `match`, returned values are `TRUE` or `FALSE` but never `NA`.

17.9 strsplit and strtrim

Elements within a text string can be rearranged, identified or extracted using a number of functions. The function `strsplit` splits a character string into substrings based on user defined criteria. It contains two important arguments.

```
strsplit("Achillea millefolium", NULL)
```

\[
[[1]] \text{ Achillea millefolium}
\]

Here I give the correct species names to the field codes using the `match` function.

```
m <- match(field.data[,1], species.list[,1])
field.data[,1] <- species.list[,2][m]
field.data
```

<table>
<thead>
<tr>
<th>code</th>
<th>cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Achillea millefolium</td>
<td>12</td>
</tr>
<tr>
<td>Elymus scriberi</td>
<td>13</td>
</tr>
<tr>
<td>Carex elynoides</td>
<td>14</td>
</tr>
<tr>
<td>Taraxacum ceratophorum</td>
<td>11</td>
</tr>
</tbody>
</table>

17.10 Complex pattern matching: `gsub`, `grep`, and `metacharacters`

The functions `grep` and `gsub` are specifically designed for pattern matching and replacement within a character vector comprised of multiple text strings. Consider the following character object made up of five strings:

```
sample <- c("amy", "joe", "fred", "mike", "betty")
```

If for some reason we wanted to convert every occurrence of a lower-case ‘m’ to an upper case ‘M’ we could use:

```
gsub("m", "M", sample)
```

\[
[1] \text{ aMy} \quad \text{Joe} \quad \text{fred} \quad \text{Mike} \quad \text{betty}
\]

Here the function `grep` tells us which elements in `sample` contain the letter ‘m’.

```
grep("m", sample)
```

\[
[1] \text{ TRUE TRUE TRUE TRUE}
\]
A metacharacter is a keyboard character that has a special (non-literal) meaning to a computer program. In R these include the symbols: \ | () [ ] ^ $ * + ? . We can use metacharacters in conjunction with gsub and grep for the purpose of complex pattern matching and replacement. In using this approach we can also call the Perl Compatible Regular Expressions (PCRE) library, which has been incorporated into R. The PCRE library implements the computer language Perl.

As an example of how metacharacters can be used by the PCRE library we will tell R to capitalize the first letter of each element in the object sample.

gsub("([\w])(\w*)", "\U\1\L\2", sample, perl=TRUE)

In the first argument, the string ([\w]) indicates that I want the function to consider the first letter in each element of sample, and all later letters, separately. In the second argument, the string \U\1\L\2 indicates that I want the first letter of each element in sample to be replaced with an upper case version of itself, and that I want the other letters to be lower case.

The functions toupper and tolower can be easily used to readily make all the letters in text string upper-case or lower-case, respectively.

toupper(sample)

Here I ask R to give me the elements in sample which are four or more letters long.

grep("([[:alnum:]]{4,})", sample, value = T)

The string [[[:alnum:]]] indicates all alphanumeric characters. I am combing this expression with the metacharacter expression {n,}. This combination tells R to report all elements that have four or more alphanumeric units. The argument value = T tells R to provide the actual contents of the elements in sample, and not the element indices.

For more information type ?grep.

18 Testing and coercing

There are a number of functions that are designed to test whether an R object has particular characteristics, or to coerce an R object to have a desired class. For instance, the function is.numeric tests whether an object is numeric while the function as.numeric coerces an object to be of class numeric.

```r
x<-c("a", "b", 4)
is.numeric(x)
[1] FALSE
as.numeric(x)
[1] NA NA 4
Warning message:
NAs introduced by coercion
```

We note that in as.numeric(x) the non-numeric parts of x are discarded. Testing and coercing functions exist for all important R-classes. These include: vector, matrix, dataframe, array, list, factor, numeric, function, dist, double, character and many others.

Coercion of data to categorical and ordinal classes can be managed using the functions factor and ordered respectively. For instance:

```r
x <- c(1, 2, 1, 2, 1, 2)  #quantitative
is.numeric(x)
[1] TRUE
as.numeric(x)
[1] 1 2 1 2 1 2
Levels: 1 2
y <- factor(x)  #categorical
y
[1] 1 2 1 2 1 2
Levels: 1 2
z <- ordered(c(1, 2, 3, 4, 5))  #ordinal
z
[1] 1 2 3 4 5
Levels: 1 < 2 < 3 < 4 < 5
```

All numeric objects in R are stored in double-precision format, meaning that the object occupies two adjacent locations in computer memory.

```r
a <- 1
is.double(a)
[1] TRUE
```

Objects coerced to be integers will be stored with double precision, although one of the storage locations will not be used. As a result integers are not conventional double precision data. This is explained in greater detail.
19 **NA, NaN, and NULL**

R identifies missing values (empty cells) as `NA`, which indicates “not available”. Hence the R function to identify a missing value is: `is.na`.

For example:

```r
x <- c(2, 3, 1, 2, NA, 3, 2)
is.na(x)
[1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE
```

To say “identify all parts of a vector except missing values”, we set a logical test to be true when values are not missing. Because the unary (single argument) operator for “not” in R is `!`, the correct command is:

```r
!is.na(x)
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
```

There are a number of functions to get rid of missing values. These include `na.omit`.

```r
na.omit(x)
[1] 2 3 1 2 3 2
attr(, "na.action")
[1] 5
attr(, "class")
[1] "omit"
```

We see that R omitted the missing observation and then told us which observation was omitted.

Functions in R are often set up to handle missing data. In many cases, however, it will be desirable to have a complete set of observations for a dataframe, matrix, or vector. Consider the following dataframe which contains plant percent cover for four plant species which are identified with codes.

```r
da <- structure(c(98, 99, NA, NaN, and NULL), .SDclass = "data.frame")
a <- 1
is.double(as.integer(a))
[1] FALSE
d <- data.frame(a = c(1, 2, 3, 4, 5), b = c(NA, 10, 20, 30, 40), c = c(50, 60, 70, 80, 90))
d[2,]
  a b c
1 NA 10 50
2 10 20 60
3 20 30 70
4 30 40 80
5 40 50 90
```
field.data <- data.frame(code=c("ACMI", "ELSC", "CAEL", "CAPA", "TACE"), cover = c(12, 13, 14, NA, 11))
field.data

code cover
1 ACMI 12
2 ELSC 13
3 CAEL 14
4 CAPA NA
5 TACE 11

I can check for completeness of the data, i.e. the appearance of missing data for any experimental unit (row) using the complete.cases function.

complete.cases(field.data)
[1]  TRUE  TRUE  TRUE FALSE TRUE

I can omit rows with missing data using na.omit.

na.omit(field.data)

code cover
1 ACMI 12
2 ELSC 13
3 CAEL 14
5 TACE 11

One shouldn’t worry too much about memorizing a million R functions. Instead you should develop a basic vocabulary of functions that you will use frequently. For instance, instead of using na.omit the example above, I could have gotten the same result using brackets and logical commands.

field.data[!is.na(field.data[,2]),]

code cover
1 ACMI 12
2 ELSC 13
3 CAEL 14
5 TACE 11

The final comma in the command above tells R that I want to go through the field.data dataframe row by row and eliminate rows with missing values.

The designation NaN is associated with the current conventions of the IEEE 754-2008 (IEEE is an acronym for Institute of Electrical and Electronics Engineers, and pronounced “I triple E”) arithmetic used by R. It means “not a number.” Mathematical operations which produce NaN include:

0/0
[1] NaN

Inf - Inf
[1] NaN

sin(Inf)
[1] NaN

Occasionally one may wish to specify that an R-object is NULL, meaning that it is absent. A NULL object can be included as an argument in a function without requiring that it have a particular value or meaning. As with NA the NULL specification is easy:

x <- NULL

It should be emphasized that R-objects or elements within objects that are NA or NULL cannot be identified with the logical commands == or !=. For instance:

x <- NA; y <- NULL
x == NA
[1] NA
y == NULL
logical(0)

Instead one should use is.na (as above) or is.null to identify NA or NULL elements or objects.

is.na(x)
[1] TRUE
is.null(y)
[1] TRUE
20 Binary numbers, bits and bytes

To acquire a deeper understanding of the workings of \( \mathbb{R} \) we must gain some familiarity with basic principles of computer science.

Computers are designed around bits and bytes. A bit is a binary piece of information, i.e. a 0 or a 1, while a byte consists of a collection of eight bits\(^{14} \). This convention occurs because conventional computer systems use electronic circuits that exist in only one of two states, on or off. In most programs on most workstations the results of computations are stored as 32 bits (4 bytes) or as 64 bits (8 bytes) of information. Double precision storage (used by \( \mathbb{R} \)) requires 64 bits. This allows expression and storage of numerical quantities between \( 5.0 \times 10^{-324} \) and \( 1.8 \times 10^{308} \), and a precision of at least 15 - 17 significant digits (see below).

We can see that the current upper numerical limit in \( \mathbb{R} \) (ver 2.15.1) is somewhere between:

\[
\begin{align*}
1.8 \times 10^{307} \\
1.8 \times 10^{308}
\end{align*}
\]

And that the (non-negative) lower limits are between

\[
\begin{align*}
5.0 \times 10^{-323} \\
4.940656e-323
\end{align*}
\]

With a single bit we can describe only \( 2^1 = 2 \) distinct digital objects. These are, an object represented by a 0 and an object represented by a 1. It follows that \( 2^n \) distinct objects can be described with \( n \) bits, \( 2^8 = 8 \) objects can be described with three bits, and so on\(^{15} \). We count to ten in binary using: \( 0 = 0 \), \( 1 = 1 \), \( 10 = 2 \), \( 11 = 3 \), \( 100 = 4 \), \( 101 = 5 \), \( 110 = 6 \), \( 111 = 7 \), \( 1000 = 8 \), \( 1001 = 9 \), \( 1010 = 10 \). Thus, we require four bits to count to ten. Note that the binary sequences for all positive non-zero integers start with one. We have the following guidelines:

\[\begin{array}{c}
1)\text{ For the smallest integer, given a particular number of bits, the leftmost placeholder will be a one, and all other bits will be zeros (e.g., } 100 = 4). \\
2)\text{ For the next largest integer, a one is placed in the rightmost placeholder occupied by a zero in the previous step (e.g., } 101 = 5). \\
3)\text{ For the next largest integer the one inserted in the previous step is moved one placeholder to the left if that bit is zero (e.g. } 110 = 6). \text{ If the bit to the left is non-zero, then go to step 2 (e.g., } 111 = 7). \text{ If all bits are ones following (or preceding) step 2, then add a bit, and begin again at step 1 (e.g., } 1000 = 8).
\end{array}\]

The addition of a binary digit represents an increasing power of 2. As a result we say that the rightmost digit in a set of binary digits represents \( 2^0 \), the next represents \( 2^1 \), then \( 2^2 \), and so on. This can be defined with the equation

\[a_0 2^0,\]

where \( a \) is quantity known as the significand, that describes the number of significant digits followed by a modifying number, \( b \) (usually 2), and \( k \) is called (appropriately) the exponent. The number of bits in the significand determines the precision of a binary expression, while the exponent determines minimum and maximum possible number. In a 64 bit double precision format 1 bit is allocated to the sign of the stored item, 53 bits are assigned to the significand, and 11 bits are given to the exponent.

Equation 1 actually represents a dot product, i.e., it is the sum of the piecewise multiplication of two vectors. For instance, to find the decimal number version of a single binary bit (consisting of only the right-most digit) we use

\[
\bin2dec(010101)
\]

The function \( \bin2dec \) in \( \text{asbio} \) does the calculation for us.

\[
\bin2dec(010101)
\]

20.1 Floating point arithmetic

Depicting undefinable real numbers (e.g. irrational number), and certain rational fractional numbers in binary, requires binary approximation (Goldberg 1991). One approach is to express numeric values with non-zero fractional parts using floating point arithmetic. This framework is used by all conventional software (although its mechanisms are easily revealed in \( \mathbb{R} \)).

As with binary decimal numbers, binary fractional numbers are expressed with respect to a decimal, and the number of digits (often) be dictated by the significand. Given \( 13 \) bits we have the following binary translations for decimal numbers: \( 1/1 = 1 \), \( 1/2 = 0.1 \), \( 1/3 = 0.01010101... \), \( 1/4 = 0.01 \), \( 1/5 = 0.00110011... \), \( 1/6 = 0.0010101... \), \( 1/7 = \)

\[
\frac{1}{2} = 0.1 \]

\[
\frac{1}{3} = 0.01010101... \]

\[
\frac{1}{4} = 0.01 \]

\[
\frac{1}{5} = 0.00110011... \]

\[
\frac{1}{6} = 0.0010101... \]

\[
\frac{1}{7} = \]
To obtain decimal fractions from binary fractions we multiply the bits by decreasing inverse powers of base two, starting at 0 (excluding 2^-1), and find the sum. For example, back-calculating the decimal value 0.14 from the binary value 0.011 we have:

\[ (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) = 0 + 0.5 + 0.25 = 0.75. \]

Floating point arithmetic may give unexpected results. For instance:

```r
options(digits = 20)
1/10
[1] 0.10000000000000000555
```

Due to the character of the binary floating point 1/10 = 0.1 can only be approximated by a binary number. The approximation is very close (out to the 18th significant digit), but not exactly equal to 0.1. Many other fractional quantities cannot be expressed exactly. Indeed, real (terminal) binary fractions will only exist if the only prime number is in the decimal fraction denominator, and it is 2 (although exception handling allows terminality for many fractions). As other examples, consider

```r
1/3
[1] 0.3333333333333333148296
```

```
bin2dec(0.01)
[1] 0.25
```

```
rawToChar(as.raw(1:128), multiple = TRUE)
[1] "\001" "\002" "\003" "\004" "\005" "\006" "\a" "\b" "\t" "\n" "\v"
[12] "\f" "\r" "\016" "\017" "\020" "\021" "\022" "\023" "\024" "\025"
[23] "\026" "\027" "\030" "\031" "\032" "\033" "\034" "\035" "\036" "\037"
[34] " " " !" #" $" %" &" '"" (" )" *" +" ," -" ." /" "0" "1" "2" "3" "4" "5" "6"
[56] "7" "8" "9" ":" ";" "<" =" >" ?" @" A" B" C" D" E" F" G" H" I" J" K" L" M"
[67] "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z" "[" "]" ^" _" `" a" b" c"
[100] "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" ":n" "o" "p" "q" "r" "s" "t"
[111] "u" "v" "w" "x" "y" "z" "{" "|" "}" "~" "\177" ";

```

Note that the exclamation point is character number 33. Its 16 bit binary code is:

```r
rawToBits(as.raw(33))
[1] 01 00 00 00 00 00 00 00
```

Obviously, such “rounding errors” may result in user-function failures. 

**References**

16 Alternatives to binary floating point arithmetic that address this problem exist, but are rarely implemented because: 1) they are less efficient, and 2) currently no IEEE standards have been specified. In order of increasing precision and decreasing efficiency alternative systems include Limited-Precision Decimal, Arbitrary-Precision Decimal, and Symbolic Calculation systems.

17 This is the number of characters that can be defined with 16 bits. The current number of Unicode characters, however, is actually much larger than 65,536. The 16-bit Unicode Transformation Format (UTF-16) handles these additional characters by using two sixteen bit units.
21 Writing functions

Perhaps the most useful thing about R is its capacity to implement user-defined functions. At some point you may even wish to compile your functions into an R-package. No more will be said on such matters here. Interested readers are directed to the reference for R-package construction “Writing R Extensions”, available at http://cran.r-project.org/doc/manuals/R-exts.html.

Programs like RWinEdt (an R package plugin for WinEdt), Tinn-R (a recursive acronym for Tinn is not Note-pad, http://www.sciviews.org/Tinn-R), ESS (Emacs Speaks Statistics, http://ess.r-project.org) and RStudio ([http://rstudio.org]) have been developed primarily to facilitate function writing in R. Text editors from these programs provide syntax highlighting of the R (and S) code and (often) similar handling for other programming languages, e.g. C/C++, Java, Perl, Pascal, Visual Basic, and Fortran. In addition, some editors, including RWinEdt, Tinn-R, ESS and RStudio allow direct interaction with an R-console running on the same computer, or can open R and generate an R-console. R-scripts, accessed with File-New script (non-Unix only) or with Ctrl+M+N on the command line prompt, provide another excellent way to edit code. A single line of code (or selected lines of code) can be sent to the R-console from an R-script interface using the shortcut Ctrl+R. The entire contents of a script can be selected using Ctrl+A, and then sent to the R-console using Ctrl+R (Figure 31).

![Figure 31 Script editor in R](image)

In all the text editors described above, functions or groups of functions can be saved as .R files. These files can then be read as source code directly in R or opened as scripts in the script editor. It is strongly recommended that word processing programs (e.g., MS Word) not be used to create R scripts and functions as the resulting code may contain hidden formatting characteristics that may affect its implementation in R.

21.1 An introductory example

At the risk of sounding repetitive, we specify a function using the function function. The arguments for function are completely user-defined and will be the argument list for one’s personalized function. For example, in the excerpt example <- function(x) the function example contains a single argument named x (naming conventions for arguments must follow the conventions defined in 4.1). The code for the function itself follows the call to function, and the argument list, generally delimited by braces. This results in the format:

```r
function.name <- function(argument1, argument2,...,argumentn){function contents}
```

Consider the function Exp.growth below that solves the function \( N_t = N_0 \lambda^t \), where \( N_0 \) is initial number of individuals, \( \lambda \) the rate of increase, and \( t \) the number of time intervals or generations.

```r
Exp.growth <- function(N.0, lambda, t){ 
  Nt <- N.0 * lambda^t 
  return(Nt) 
}
```

The function requires three arguments: \( N_0, \lambda, \) and \( t \). The first line of code following the arguments solves the function \( N_t = N_0 \lambda^t \) utilizes the argument definitions and stores the solution as \( N_t \). The last line gives the object I want returned by the function, \( N_t \). Note that if I didn't specify some ‘return value’, nothing would be returned by the function. If a function has multiple return objects, then one can place them in single suitable container like a list.

To increase clarity one should place the first curly bracket on same line as the arguments, and place last curly bracket on its own line. This convention is followed above. Readability can also be improved by use of spaces. Note that I have inserted two spaces to begin lines containing related operations. This distinguishes these lines from the first (argument) line and the end (return) line. Note also that spaces are placed after commas, and before and after operators, including the assignment operator.

Once I have read the function Exp.growth into R (by scanning, reading, typing, or pasting it), I will be able to run it by typing the function name and specifying arguments.

Here we run function for \( N_0 = 100, \lambda = 0.8, \) and \( t = 10 \)

```r
Exp.growth(N.0 = 100, lambda = 0.8, t = 10) 
[1] 10.73742
```

The population size is decreasing because \( \lambda \) is less than one.
21.2 Global variables versus local variables

It is important to note that the arguments and objects created in a function are local variables. That is, they only exist within the confines of the function.

We can use `Exp.growth` to demonstrate that the variable \(N_t\), which was defined in the function, is local.

```r
N_t <- 100; lambda <- 0.8; t <- 3
Exp.growth(N_0 = 100, lambda = 0.8, t = 10)
```

Conversely, R allows one to define the free variables in the text of the function, e.g.,

```r
Exp.growth <- function(N_0, lambda, t){
  N_t <<- N_0 * lambda^t
}
```

Global variables can be assigned within functions using the so-called superassignment operator, `<<-`.

```r
Exp.growth <- function(N_0, lambda, t){
  N_t <<- N_0 * lambda^t
  N_t
}
```

Global variables are (generally) defined outside of functions, and thus can be called within a function as arguments or other sorts of objects. Lexical scoping allows R to distinguish global and local variables.

Languages like S that don’t have lexical scoping would not “know about” the variables \(N_0\), \(lambda\), and \(t\) unless they were defined outside of the function, i.e.,

```r
N_0 <- 100; lambda <- 0.8; t <- 3
Exp.growth(100, 0.8, 3)
```

Below I create a function called `stats` that will simultaneously calculate a large number of summary statistics.

```r
stats <- function(x, digits = 5){
  ds <- data.frame(statistics = round(c(length(x), min(x), max(x), mean(x), median(x), sd(x), var(x), IQR(x), sd(x)/sqrt(length(x)), 
                                      kurt(x), skew(x)), 
                                      digits))
  rownames(ds) <- c("n", "min", "max", "mean", "median", "sd", "var", 
                     "IQR", "SE", "kurtosis", "skew")
  return(ds)
}
```

The function contains two arguments. A call to a vector of data \(x\), and the number of significant digits \(\text{digits}\). Because I have given \(\text{digits}\) the default value 5, only the first argument needs to specified by a user.

Following the arguments, the first three lines of the function create a dataframe called \(ds\). It has one column called “statistics” that will contain eleven statistical summaries of \(x\). The summaries are rounded to the number of digits specified in \(\text{digits}\).

```r
stats <- function(x, digits = 5){
  ds <- data.frame(statistics = round(c(length(x), min(x), max(x), mean(x), median(x), sd(x), var(x), IQR(x), sd(x)/sqrt(length(x)), kurt(x), skew(x)), digits))
  rownames(ds) <- c("n", "min", "max", "mean", "median", "sd", "var", "IQR", "SE", "kurtosis", "skew")
  return(ds)
}
```

The next lines of code define the row names of \(ds\). These are the names of the statistics calculated by the function.

```r
The last command, `return(ds)`, returns \(ds\).
```

Note that the lines beginning `median(x), digits` and "SE" are not indented because they are con-
Here are summary stats for the Age column in Downs.

```r
stats(Downs$Age)
```

<table>
<thead>
<tr>
<th>statistics</th>
<th>n</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>sd</th>
<th>var</th>
<th>IQR</th>
<th>SE</th>
<th>kurtosis</th>
<th>skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>11.00000</td>
<td>17.00000</td>
<td>47.00000</td>
<td>34.0545</td>
<td>10.78994</td>
<td>116.42273</td>
<td>17.00000</td>
<td>3.25329</td>
<td>-1.41039</td>
<td>-0.40977</td>
</tr>
</tbody>
</table>

The output above is a list. The attributes of the list are the column names of Downs. As the result of standardization, three variables now have mean zero, and unit standard deviations and variances.

The function `stats` will work on any R object with a numerical class. For instance, try it on the first two columns of `lp`, the loblolly pine dataset.
Here are some data.

\[
x <- c(2, 1, 4, 5.6, 7, 6.7, 3, 4.5, 10, 12, 2, 5, 6)
\]

I can run the function using any of the four location estimators.

location(x, "mean")

[1] 5.292308

location(x, "geo")

[1] 4.357524

location(x, "huber")

[1] 4.959546

### 21.5 Triple dot ... argument

A particularly handy item for writing wrapper functions (functions which embed other functions) is the triple dot (...) argument. Imagine you wish to create a wrapper for the function `plot` that allows you to automatically display the results from a simple linear regression. We might create the following:

```r
reg.plot <- function(x, y, ...){
  plot(x, y, ...)
  lm.temp <- lm(y ~ x)
  coef <- lm.temp$coefficients
  abline(coef[1], coef[2])
}
```

The triple dots in the arguments of `reg.plot`, and the associated triple dots we included in `plot` let us specify any additional argument from `plot` as an argument in `reg.plot`. For instance we could include specification for X and Y axis labels in `reg.plot` with the plot arguments `xlab` and `ylab`. For instance:

```r
with(Loblolly, reg.plot(age, height, xlab = "age", ylab = "height"))
```

### Example 21–Function writing: quantifying biological diversity

In this extended exercise we examine functions for summarizing a community ecology dataset.

**Alpha diversity** measures the degree of evenness and richness within individual plots in a dataset. Low levels of monodominance and high richness result in high alpha diversity. A large number of alpha diversity indices have been utilized by ecologists (Magurran 1988). The most widely used are **Simpson’s index** ($D_i$, Simpson 1949) and the **Shannon-Weiner index** ($H'$, MacArthur and MacArthur 1961).

\[
D_i = 1 - \sum p_i^2
\]

\[
H' = \sum p_i \ln p_i
\]

where $p_i$ is the proportional abundance of the $i$th species.

Simpson’s index ($D_i$) has a straightforward interpretation. It is the probability of reaching into a plot and simultaneously pulling out two different species. The Shannon-Weiner index ($H'$) does not allow straightforward interpretation, although its values conventionally fall between 1.5 and 3.5, and rarely surpass 4.5 (Magurran 1988, Margalef 1972). $D_i$ and $H'$ will both increase with increasing diversity (i.e., increased richness and evenness).

**Table 5** Part of a community dataset (six species and five sites) from a Scandinavian *Pinus sylvestris* forest (Väre et al. 1995). Responses are percent ground cover.

<table>
<thead>
<tr>
<th></th>
<th>Empetrum nigrum</th>
<th>Vaccinium myrtillus</th>
<th>Vaccinium vitis-idaea</th>
<th>Pinus sylvestris</th>
<th>Vaccinium uliginosum</th>
<th>Betula pubescens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>11.13</td>
<td>0</td>
<td>17.8</td>
<td>0.07</td>
<td>1.6</td>
<td>0</td>
</tr>
<tr>
<td>Site 2</td>
<td>8.92</td>
<td>2.42</td>
<td>10.28</td>
<td>0.12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Site 3</td>
<td>6.45</td>
<td>0</td>
<td>14.13</td>
<td>0.07</td>
<td>0.47</td>
<td>0</td>
</tr>
<tr>
<td>Site 4</td>
<td>9.3</td>
<td>0</td>
<td>8.5</td>
<td>0.03</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Site 5</td>
<td>3.47</td>
<td>0.25</td>
<td>20.5</td>
<td>0.25</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

To read the data into R we can use the `data.frame` function, as we have done before, or the `matrix` function (since the variables are all quantitative). Using the `matrix` command we specify how many rows and columns we want and then simply enter the data by rows or columns.

```r
ps.data <- matrix(nrow = 5, ncol = 6, data = c(11.13, 0, 17.8, 0.07, 1.6, 0, 8.92, 2.42, 10.28, 0.12, 0, 0, 6.45, 0, 14.13, 0.07, 0.47, 0, 9.3, 0, 8.5, 0.03, 0, 0, 3.47, 0.25, 20.5, 0.25, 0, 0), byrow = TRUE)
```

We can give the matrix the correct row and column names.

```r
colnames(ps.data) <- c("Empetrum nigrum", "Vaccinium myrtillus", "Vaccinium vitis-idaea", "Pinus sylvestris", "Vaccinium uliginosum", "Betula pubescens")
rownames(ps.data) <- c("Site 1", "Site 2", "Site 3", "Site 4", "Site 5")
```
Writing functions

Simp.index <- function(x){
  p.i <- apply(x, 1, function(x){x/sum(x)})
  1-apply(p.i^2, 2, sum)
}

To run Simp.index for the Pinus sylvestris data, ps.data, I read or type the function into R and then type:

Simp.index(ps.data)

Site 1    Site 2    Site 3    Site 4    Site 5
0.5265904 0.5956317 0.4586194 0.5006717 0.2778395

Next we will create a function for the Shannon Weiner index.  This measure also utilizes $p_i$; as a result it will be similar to the Simpson's index function.

SW.index <- function(x){
  p.i <- apply(x, 1, function(x){x/sum(x)})
  h <- apply(p.i, 1, function(x){log(x) * x})
  -1 * apply(h, 1, function(x){sum(x[!is.na(x)])})
}

Note that we run into a problem calculating $H'$ when any $p_i = 0$ because $\ln(0)$ is undefined.  As a result zeroes in the data will cause the function to fail.  To account for this I told R to add $p_i$ elements only when they are not undefined.  This is in the line:

h <- -1 * apply(h, 1, function(x){sum(x[!is.na(x)])})

The line works because an undefined (NaN) value is regarded as NA.

Here we run the function on ps.data.

SW.index(ps.data)

Next I need to: 1) square the proportions, 2) take the sum within each row (each site), and 3) subtract this sum from one.

D <- 1 - apply(p.i^2, 2, sum)

It looks like we only really need two lines of code to calculate Simpson's index. We can write the function as:
The method of Euler (the simplest method to find approximate solutions to first order equations) can be specified with the function euler. The arguments for rk4, in order, are the initial population numbers from species 1 and 2, the time frames to be considered, the function to be evaluated, and the parameter values.

```r
out <- as.data.frame(rk4(xstart, time = 1:200, LV, pars))
```

The object out contains the number of individuals in species 1 and 2 for time frames 1-200 (Figure 32).

```r
plot(out\$time, out\$N2, xlab = "Time", ylab = "Number of individuals", type = "l")
lines(out\$time, out\$N1, type = "l", col = "red", lty = 2)
legend("bottomright", lty = c(1, 2), legend = c("Species 2", "Species 1"), col = c(1, 2))
```

**Figure 32** Result of solutions from Lotka Volterra ODEs for t = {1, 2, ..., 200}. Species 1 and 2 coexist, but at levels appreciably below their carrying capacities as a result of interspecific competition.

Functions can obviously be more complicated than these previous examples. Below is a function with some level of complexity, called radiationheatl. Its purpose is to calculate northern hemisphere annual incident solar radiation (MJ cm^{-2} yr^{-1}) and heatload given slope, aspect, and latitude (all measured in degrees). Heatload is a radiation index based on the idea that the highest amounts of radiation occur on southwest facing slopes because they receive late afternoon sun added to ambient heating from earlier daylight hours (McCune and Keough 2002).

We first bring in the package deSolve which contains function for solving ordinary differential equations (ODEs)\textsuperscript{39}.

```r
install.packages("deSolve")
library(deSolve)

We then define starting values for N\textsubscript{1} and N\textsubscript{2} and model parameters

```r
xstart <- c(N1 = 10, N2 = 10)
pars <- list(r1 = 0.5, r2 = 0.4, K1 = 400, K2 = 300, a2.1 = 0.4, a1.2 = 1.1)
```

We then specify the Lotkka-Volterra equations as a function. We will include the argument time = time even though it is not used in the function. This is required by ODE evaluators from deSolve.

```r
LV <- function(time=time, xstart=xstart, pars=pars){
N1 <- xstart[1]
N2 <- xstart[2]
with(as.list(pars),{
  dn1 <- r1 * N1 * ((K1 - N1 - (a1.2 * N2))/K1)
  dn2 <- r2 * N2 * ((K2 - N2 - (a2.1 * N1))/K2)
  res <- list(c(dn1, dn2))
})
}
```

We will run the Lotka-Volterra function with the function rk4 from library deSolve. The rk4 function solves simultaneous differential equations using classical Runge-Kutta 4\textsuperscript{th} order integration (Butcher 1987).

```r
out <- as.data.frame(rk4(xstart, time = 1:200, LV, pars))
```

The object out contains the number of individuals in species 1 and 2 for time frames 1-200 (Figure 32).

```r
plot(out\$time, out\$N2, xlab = "Time", ylab = "Number of individuals", type = "l")
lines(out\$time, out\$N1, type = "l", col = "red", lty = 2)
legend("bottomright", lty = c(1, 2), legend = c("Species 2", "Species 1"), col = c(1, 2))
```

\textsuperscript{39}As opposed to partial differential equations
Writing functions

radiation.heatl <- function(slope, aspect, lat){
  asp.wrap.rad <- (-1 * abs(aspect - 180)) + 180
  asp.wrap.hl <- abs(180 - abs(aspect - 225))
  rad.lat <- (lat/180) * pi
  rad.asp <- (asp.wrap.rad/180) * pi
  hl.asp <- (asp.wrap.hl/180) * pi
  rad.slope <- (slope/180) * pi
  rad <- 0.339 + 0.808 * (cos(rad.lat) * cos(rad.slope)) - 0.196 * (sin(rad.lat) * sin(rad.slope)) - 0.482 * (cos(rad.asp) * sin(rad.slope))
  hl <- 0.339 + 0.808 * (cos(rad.lat) * cos(rad.slope)) - 0.196 * (sin(rad.lat) * sin(rad.slope)) - 0.482 * (cos(hl.asp) * sin(rad.slope))
  list(radiation = rad, heat.load = hl)
}

After reading the function into R I can find the annual incident radiation and heatload for a site with a slope of 20º, with a northeast (30º) aspect, that is located at 40 degrees N latitude, by typing:

radiation.heatl(20, 30, 40)
$radiation
[1] 0.7347784

$heat.load
[1] 0.7183095

This site receives 0.7347784 mega joules of radiation per cm² per year, and has a heat load index of 0.7183095.

21.6 Looping

Loop functions exist in some form in virtually all programming languages. The call to a so-called "for loop" in R is made using the command for. An R for loop requires specification of three entities, in parentheses, following a call to for. These are:
1) an index variable, e.g., i,
2) the statement in, and
3) a sequence that the index variable refers to as loops commence.
The code defining the loop follows, generally (if the loop requires multiple lines) delineated by curly brackets.

In parallel to function writing it is good style to place the first curly bracket on the same line as the call to for, and to place the last curly bracket on its own line. For clarity, spaces should also be used to distinguish the loop contents. Thus, we have the basic form:

for(i in seq){
  loop contents
}

In the loop the ith element of something (e.g., matrix column, vector entry, etc.) is defined (or replaced) as the for sequence commences. The replacement/definition process takes place in the "loop contents".

11 Example 23

One application for a loop is to make functions with scalar input arguments amenable to vector, matrix or dataframe inputs. For example, what if I wanted to apply the radiation.heatl function (which requires scalar inputs) to a three columned dataframe with columns containing slope, aspect, and latitude information, respectively? For instance:

x <- data.frame(slope = c(10, 12, 15, 20, 3), aspect = c(148, 110, 0, 30, 130), latitude = c(40, 50, 20, 25, 45))

The first step in many loop applications is defining an object to contain the results. Because I will obtain a list with two objects from radiation.heatl, I will create two one column matrices to contain my results, one for radiation (I will call this rad.res), and one for heatload (I will call this heat.res). Because I have five observations, each matrix will have five elements.

rad.res <- matrix(ncol = 1, nrow = 5)
heat.res <- matrix(ncol = 1, nrow = 5)

Next I create the for loop.

for(i in 1:5){
  rad.res[i] <- radiation.heatl(x[i,(1)],x[i,(2)],x[i,(3)])$radiation
  heat.res[i] <- radiation.heatl(x[i,(1)],x[i,(2)],x[i,(3)])$heat.load
}

The function loops around on itself letting i = 1 during the first loop, i = 2, during the second loop, up to i = 5 on the final loop. Here are the results.

rad.res
[,1]
[1,] 0.9976634
[2,] 0.8500810
[3,] 0.9302991
[4,] 0.8560357
[5,] 0.9185208
heat.res

Look at the function and try to really understand what I did, using the information from earlier in this document.

Example 24
Another application of a loop is to iteratively build on the results of the previous step in the loop. Consider the following function that counts the number of even entries in a vector of integers.

```r
evencount <- function(x)
{
  res <- 0
  for(i in 1 : length(x)){
    if(x[i]%%2 == 0) res <- res + 1
  }
  res
}
```

Recall that `%%` is the modulus operator in R. That is, it finds the remainder in division. By definition the remainder of any even integer divided by two will be zero. At each loop iteration the function adds one to the number in `res` if the current integer in the loop is even (has remainder zero if divided by two).

Example 25
A third loop application is the summarization of data with respect to levels in a categorical variable. Consider the categorical extension to the `Downs` dataset discussed earlier. We have:

```r
Categories <- factor(c(rep(1, 6), rep(2, 5)))
cbind(Downs, Categories)
```

What if we wished to statistically summarize the variables in `Downs` (Age, Births, and Cases) simultaneously for each level in the categorical variable `Categories`? One solution is a for loop.

We first create an empty list to hold the result:

```r
result <- list()
```

The components in the list will be dataframes of summary statistics. These will be assembled in a for loop that steps through the levels in `Categories`.

```r
for(i in levels(Categories)){
  temp <- Downs[Categories==i,]
  result[[i]] <- as.data.frame(apply(temp,2,stats))
  names(result[[i]]) <- c("Age","Births","Cases")
}
```

Instead of a numeric sequence, the loop will walk through the levels of `Categories`. This is specified with the code: `for(i in levels(Categories))`. Note that in the first line following the `for` specification, the `Downs` dataset is subset by levels in categories. The data are then summarized with the function `stats` that we created in Example 7. Column names for the summary dataframes are given in the last line of the loop.

Here is the loop result:

```
result
$'1'
  Age  Births  Cases
  n     6.00000 6.000000e+00     6.00000
  min   17.00000 4.834000e+03     9.00000
  max   38.50000 2.389600e+04    22.00000
  mean  26.25000 1.565483e+04    15.00000
  median 25.50000 1.481950e+04    15.50000
  sd     7.99844  6.821069e+03    4.38178
  var   63.97500 2.961230e+07    19.20000
  IQR    9.50000  1.976880e+04    3.25000
  SE     3.26535  2.784690e+03    1.78885
  kurtosis -0.79755  2.097100e-01  0.92773
  skew   0.50457  -4.399500e-01  0.36372

$'2'
  Age  Births  Cases
  n     5.00000 5.000000e+00     5.00000
  min   39.50000 2.490000e+04     7.00000
  max   47.00000 3.961000e+04    31.00000
  mean  43.40000 1.617000e+04    20.20000
  median 44.50000 5.960000e+03    22.00000
  sd    3.24848  1.721500e+03    10.89495
  var   10.64870 4.942600e+06   118.70000
  IQR    5.00000  4.783250e+03    19.00000
  SE     1.50000  1.492500e+03    4.87237
  kurtosis -2.48635  2.204480e-01  2.64479
  skew  -0.32174   0.774770e+00  0.28092
```
21.7 Looping without `for`

Looping in R is also possible using the style of the language C which conventionally uses the functions `while`, `repeat`, and `break`.

1 Example 26

Consider an example in which 2 is added to a base number until the updated number becomes greater than or equal to 10: We have:

```
i <- 1
while (i < 10) i <- i + 2
i
[1] 11
```

Or, alternatively

```
i <- 1
while(TRUE) {
i <- i + 2
if (i > 9) break
}  
i
[1] 11
```

Here the variable `i` took on values 1, 3, 5, 7, 9, and 11 as the loop commenced. When `i` equaled 11 the condition for continuation of the loop failed and the loop was halted.

21.8 Calling and receiving code from other languages

Code from C, C++, Fortran, MATLAB, Python and other languages can be linked to R at the command prompt. Indeed, most of the code for R is written in C. R can also be called from a number of different languages including C, Java, or Python (Lang 2005). For instance, the R package RCytoscape allows cross communication between the popular Java-driven software molecular network package Cytoscape and the R-console. We have already learned that Perl applications can be used to identify and manipulate R objects (see § 17.10).

R is an interpreted language, as a result functions written in R must be tacitly translated into binary before they can be executed. This process is in contrast to compiled languages, e.g., C, Fortran, and Java. For these languages a compiler (a translation program) is used to transform the source code into a target "object" language, which is generally binary. The product of the compilation is called an executable file (Figure 33).

Raw Fortran source code is generally saved as an entity called an `.f` file, C source code is saved as an `.c` file, and Java source code is saved as a `.java` file. Windows executable files, compiled from source code, will generally have an `.exe` or `.dll` extension. R contains both Fortran and C compilers. Note, however, that the compilers will only work for Fortran code written as a subroutine and C code written in void formats. As a result, neither will return a value directly.

To write and compile Fortran and C code in Windows one can save files containing the source code in the `bin` directory of the current version of R (where the compilers are located), and then invoke the shell command: `R CMD SHLIB` from a workstation command prompt followed by the name of the file.

The R CMD algorithms located in the R `bin` directory are an important group of functions that are called directly from the command prompt shell (not interactively from the R-console). They include `R CMD check`, which checks user designed packages for errors, and `R CMD batch` which creates batch R output files (e.g., collections of `.pdf` files, dataframes or graphs).

Calling Fortran and C for looping

Because its functions are not compiled, R requires a large amount of system time for iterative procedures like loops. As a result, it may be expedient to call compiled code, originally written in Fortran or C, for complex looping with large datasets.

Below is a simple example (that could easily be handled without a `for` loop) that compares R, Fortran, and C looping. Familiarity with C and Fortran is assumed.

1 Example 27

Here is a Fortran subroutine loop to convert a vector of Fahrenheit measurements to degrees Celsius.

![Diagram of source code compilation and execution steps in compiled languages](Figure 33)
subroutine ftoc(n, x)
  integer n
  double precision x(n)
  integer i
  do 100 i = 1, n
    x(i) = (x(i)-32)*(5./9.)
  100 continue
end

After typing up the code in a text editor we will save it in the R bin directory as the file FtoC.f. We will then compile it by opening up an operating system command line prompt, going to the bin directory (if you are using the 64 bit R, be sure to specify the 64 bit version before the 32 bit version in your Windows system search path), and typing:

R CMD SHLIB FtoC.f

This shell command will create a compiled routine (in a Windows operating system this will be a .dll file, in Unix/Linux it will be an .so file).

Here is an analogous C function for converting °F to °C:

void ftocc(int *nin, double *x)
{
  int n = nin[0];
  int i;
  for (i=0; i<n; i++)
    x[i] = (x[i] - 32) * 5 / 9;
}

We would save the code as the file ftocc.c file in the bin directory and run the shell command:

R CMD SHLIB ftocc.c

will create a compiled routine in the form of a .dll file.

Here is an R-wrapper that can call either the Fortran subroutine, call = "f", or the C function, call = "c".

F2C <- function(x, call = "f"){
  n <- length(x)
  if(call == "f"){
    if(call == "f"){
      dyn.load("C:/Program Files/R/R-3.0.2/bin/x64/ftoc.dll")
      out <- .Fortran("ftoc", n = as.integer(n), x = as.double(x))
    }
    if(call == "c"){
      dyn.load("C:/Program Files/R/R-3.0.2/bin/x64/ftocc.dll", nin = n, x)
      out <- .C("ftocc", n = as.integer(n), x = as.double(x))
    }
  }
  out
}

Let’s convert 1,000,000 randomly generated Fahrenheit temperatures to Celsius.

x <- runif(1000000, 0, 100)

Here is a loop using R alone.

F2CR<-function(x){
  res <- 1 : length(x)
  for(i in 1 : length(x)){
    res[i] <- (x[i] - 32) * (5/9)
  }
  res
}

Let’s compare the run times.

system.time(F2CR(x))
user  system elapsed
2.93    0.00    2.95
system.time(F2C(x, call = "f"))
user  system elapsed
0.03    0.00    0.03
system.time(F2C(x, call = "c"))
user  system elapsed
0.03    0.00    0.03

The Fortran and C loops gave the same transformation results (not shown here), but ran much faster. The Fortran subroutine was particularly fast. This is despite the fact that this language greatly predates C (and certainly R).

21.9 Functions with animation
Writing functions

Animation can be in R used to illustrate a wide range of processes (Xie 2011, Xie and Chang 2008). Functions with animation are generally based on for loops with some method of slowing the loop; usually the function Sys.sleep(). Slowing the loop allows examination of the iterative steps defined in the creation of a plot.

| Example 28 |

Consider the following function:

```r
R.zoom<-function(){ # requires R >= 2.15.1
for(i in seq(30, 1800, 30)/10){
  dev.hold()
  plot(1:10, 1:10, type = "n", xaxt = "n", yaxt = "n", xlab = "", ylab = "")
  text(5, 5, "R", cex = i, srt = i)
  dev.flush()
  Sys.sleep(.1) #Loop stopped for 0.1 seconds at each iteration
}
}
```

Using functions from package animation (Xie and Chang 2008) one can save an R animation as a media file for inclusion in other formats, e.g., webpages, powerpoint presentations, interactive pdf documents, etc.

For instance the code below creates an adobe flash (.swf) file of Rzoom to a temporary directory.

```r
install.packages("animation")
library(animation)
saveSWF(R.zoom(), swf.name = "Rzoom.swf", interval = 0.1, nmax = 30,
ani.dev = "pdf", ani.type = "pdf", ani.height = 6, ani.width = 6)
```

The function saveSWF requires installation of SWF Tools, and a path in the Windows environment to the tools.

The result, embedded in an .flv file, can be shown by clicking on the flashplayer link below.

As before, the rotation can be run by clicking on the flashplayer link below.

Of note, we can also create hand-rotatable 3D figures under the rgl real-time rendering system (Figure 34).

```r
rotate.Fig <- function()
angle <- 0:360
for(i in 1:length(angle)){
  dev.hold()
  Fig(angle = angle[i])
  dev.flush()
  Sys.sleep(.1)
}
```

```r
library(animation)
saveSWF(rotate.Fig(), swf.name = "rotate.swf", interval = 0.1, nmax = 30,
ani.dev = "pdf", ani.type = "pdf", ani.height = 6, ani.width = 6)
```

As before, the rotation can be run by clicking on the flashplayer link below.

As a more practical example we will update Figure 25 to animate a three dimensional plot showing the cover of the plant Vaccinium vitis-idaea as a function of pH and % soil N (see Example 13).
21.10 Building GUIs

GUIs (Graphical User Interfaces) are a mixed bag. A person who is terrified of command line entry will be reassured by their point and click familiarity. However much user flexibility will be lost for controlling functions. In addition, a GUI implementing too many arguments may quickly become visually confusing, and hiding code inside a GUI “black box” is contrary to the “mission statement” of R. Command line entry is valuable for two other reasons. First, one can scroll through earlier commands (using the up arrow), and easily repeat or modify earlier steps. Second, command line entry provides an exact record of everything you have done during a particular R-session. Despite these caveats the package asbio relies on interactive GUIs for pedagogic demonstrations. Type:

```r
library(asbio)
book.menu()
```

Devices like those in asbio can be created in a number of programming languages including Java and Perl. I used the language tcltk (pronounced “tickle tee kay”) to create asbio GUIs, because tcltk GUIs are easy to build, and an interpreter for tcltk is included with a conventional install of R.

tcltk GUIs

An extensive description of the tcltk language and its capacities in R are beyond the scope of this book. I include simple examples here (with little code explanation) merely to show the reader that such devices can be easily constructed in R.

Example 29

Consider a GUI whose only purpose is to destroy itself (Figure 35).

```r
require(tcltk)
tt <- tktoplevel()
DM.but <- tkbutton(tt, text = "Exit", command = function() tkdestroy(tt))
tkgrid(DM.but)
```

Figure 35 An extremely simple radio button GUI. Click “Exit” (in R) to destroy.

- The function `tktoplevel` defines the "toplevel" widget (graphical elements that allow a user to interactively change a function), allowing it to be manipulated.
- The `tkbutton` function creates the "Exit" button. The first argument is name of the widget that the button is to be placed upon. The text argument defines the text on the button. The `command` argument defines the function that the button initiates. In this case it is `tkdestroy`, which closes open tcltk widgets.
- The function `tkgrid` places the button on the widget.

We can also use GUIs to run other R-functions. For instance the GUI below (Figure 36) calls the func-
GUIs can contain a number of widgets, including sliders.

**Example 30**

Below is code that defines the function `plot.me`. The function simply plots a large blue dot. Below it is code for a GUI slider. The dot’s vertical location is defined by the slider (Figure 37).

```r
plot.me <- function(){
  y <- evalq(tclvalue(SliderValue)) # Evaluate the expression
  plot(1,as.numeric(y),xlab = "", ylab = "", xaxt = "n", ylim = c(0,100),
       cex = 4, col = 4, pch = 19)
}
slider.env <<- new.env()
tt <- tktoplevel()
SliderValue <- tclVar("50")
SliderValueLabel <- tklabel(tt, text = as.character(tclvalue(SliderValue)))
tkgrid(tklabel(tt, text = "Slider Value : "), SliderValueLabel, tklabel(tt, text = ":"))
tkconfigure(SliderValueLabel, textvariable = SliderValue)
slider <- tkscale(tt, from=100, to=0,
                  variable = SliderValue, resolution=1,
                  orient = "vertical", command = substitute(plot.me()))
tkgrid(slider)
```

The superassignment operator `<<-` used above initiates a search through the function environment for an existing definition of the variable being assigned. If the variable is found, then its value is redefined, otherwise assignment takes place with respect to the global environment and will be translated in all functions. Other code specifications include:

- `SliderValue <- tclVar("50")` which creates a variable that can be used interactively, and defines the starting value of the slider to be 50.
- `SliderValueLabel <- tklabel(tt,...` which causes the value of the slider to be formatted for projection.
- `tkgrid(tklabel(tt,...` which places the slider values, along with text "Slider Value" and ":", above the slider.
- `tkconfigure(SliderValueLabel, textvariable = SliderValue)` which tells R that the slider output to be written will come from the variable `SliderValue`.
- `slider <- tkscale(tt, from=100, to=0,...` which defines the range of the slider along with other slider characteristics including the name of the function (`plot.me()`) that the slider runs.
- `tkgrid(slider)` which places the slider on the GUI.

For more information on GUIs see the tcltk manual included with a conventional installation of R (but written for a non-R language framework).

Amazingly, one can call and utilize R from a local online server. Further, using shiny apps (http://shiny.rstudio.com/), an implementation of Rstudio, one can control R using sliders and widgets. When using such applications R is run unseen and a user need know nothing about the R language (see https://ahoken.shinyapps.io/seeNorm/).
21.11 Functions with class and method

This is an advanced topic for package developers, but may be of interest to those who wish to embed particular methods for summarizing a family of functions.

It turns out that we are not limited to the pre-existing classes in R (e.g., numeric, factor, etc.). Instead, we can create user-defined classes with associated methods for plotting, printing and summarization. Object classes allow one to customize generic functions, e.g., plot, print, summary, for particular classes. These functions are called with the generic function names e.g., plot, print, summary. Here are functions on my workstation that will be called with plot, depending on the class of the object that is being plotted.

```r
methods(plot)
[1] plot.acf*           plot.data.frame*    plot.decomposed.ts* plot.default
[5] plot.dendrite*      plot.dendrogram*    plot.density*       plot.deSolve*
[9] plot.ecdf           plot.factor*        plot.formula*       plot.function
[17] plot.lm*            plot.medpolish*     plot.mlm*           plot.pairwise
[21] plot.prcomp*        plot.princomp*      plot.profile.nls*
[25] plot.shingle*       plot.spec*          plot.stepfun        plot.stl*
[29] plot.table*         plot.trellis*       plot.ts             plot.tskernel*
[33] plot.TukeyHSD*
```

For example, let \( x \) be an object of class pairw. If I type `plot(x)`, then the command `plot.pairwise(x)` is run.

The package `asbio` contains a number of functions for summary statistics including the function `G.mean` below which calculates the geometric mean (see Ch. 4 in Foundational and Applied Statistics textbook). It might be desirable to create a class called `as.stat` to print such summary statistics in a consistent way.

```r
G.mean <- function(x){
  x <- na.omit(x)
  res <- list(stat = (prod(x))^(1/length(x)), name = "Geometric Mean")
  class(res) <- "as.stat"
  res
}
```

```
x<-c(2, 1, 4, 5.6, 7, 6.7, 3, 4.5, 10, 12, 2, 5, 6)
```

We have gotten rid of some of the clunky default R output (for instance the [1] is gone). `G.mean` calls the function `print.as.stat` to print its results. Within `print.as.stat` the function `cat` concatenates and prints `G.mean` output, the argument "\n" means "put a line break in the output", and the term `invisible` lets additional items be in `print.as.stat` that are not printed, but can still be called. Additional methods can also be made for plotting and other summaries for objects of class `as.stat`. The code for internal functions will not be directly accessible by typing the function name, but can be accessed with the double colon metacharacter (::). For instance a user of `as.stat` can see the internal function `print.as.stat` (if it exists) by typing the package name followed by `::print.as.stat`. The triple colon operator (:::) allows access to so-called "unexported" objects in a package.

It should be noted that the implementation here describes the R S3 system for classes and methods. The so-called S4 system, described in Chambers (2008) requires a much stricter set of conditions for user-classes.

21.12 Documenting functions and workflow

A concern with R is tracking the characteristics of your functions and workflow. I generally use Rstudio or a text editor like TinnR to create .R script files that can be loaded into R, or pasted piecemeal into the R-console. However, even with use of notes using # it may be difficult to deduce what you have done, particularly given an extended long separation from the work.

One solution is to create .Rd document files describing functions, translate them into a Latex style .tex documents, and use these to build a pdf document. Latex is a high quality typesetting freeware system that is the de facto standard for many publishers of scientific documents (see http://www.latex-project.org/).

Consider the function below:

Google has an R style guide: [http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml](http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml) that offers good advice for code writing. The webpage recommends S3 over S4 methods for most applications.
myfunction <- function(text = "Hello world"){
  print(text)
}

After entering function, install and load the package SoDA,

install.packages("SoDA")
library(SoDA)

and type

promptAll("myfunction")

This will create the skeleton of an .Rd documentation file for `myfunction()` that will be placed in your working directory. The document can be completed using any text editor. Now...

• Place the finished file in your R/bin directory,
• Go to a command prompt and navigate to the directory using the MS-DOS command cd
• In the command prompt type `R CMD Rd2pdf myfunction.rd`. This creates a .tex file that is read (in LaTeX) to build a .pdf.

The approach requires that both a LaTeX style repository (e.g., MiKTeX), and a Windows environmental path to this repository exist on your machine. Although normally not a concern for those not developing packages, batches of .Rd files (and other types of R documentation) can be spell-checked using the function aspell. The function requires installation of freeware aspell software.

A more sophisticated approach is to use a LaTeX text editor to create an .Rnw file that can be read with either Sweave or knitr. These programs can write beautiful equations and text (although the LaTeX language can seem daunting initially), distinguish R code from documentation, create graphs and tables from R code, and correctly place all of these into a .pdf document.

Guidance for Sweave can be found at: http://www.stat.uni-muenchen.de/~leisch/Sweave/

Guidance for knitr can be found at: http://yihui.name/knitr/

Examples for some of my consulting work, documented in this way, can be found with the links below.

Zero-inflated generalized linear models and elk
Dall’s sheep population dynamics
Autism neurological genetic networks

Good LaTeX editors include TeXnicCenter, which can be customized to call the R CMD Sweave script using the Define Output Profile dialog box in the Build pulldown menu.
Summary stats for tree heights are shown in Table 1:\
<<echo=FALSE>>=
library("xtable")
@

Two backslashes (i.e. `\`) is a LaTeX carriage return.

We insert the table using:

<<label = tab1, results = "asis", echo = FALSE>>=
mat1 <- xtable(as.matrix(pine.height),
caption = "Loblolly pine height summary", label = "tab:one")
print(mat1, table.placement = "H", caption.placement = "top")
@

The argument `table.placement = "H"` means: place the table exactly at the indicated location with respect to text.

We can also make on-the-fly graphs:

Figure 1 shows a histogram of pine ages
\begin{center}
\begin{figure}
<<out.width="0.8\linewidth">>=
hist(Loblolly$age)
@\caption{Distribution of pine ages.}
\end{figure}
\end{center}

The argument `center` in `\begin{center}` centers the figure. The command `\out.width="0.8\linewidth"` forces the figure width to be 80% of the defined document line width. Figure dimensions will be held constant in this reduction. The quotations and backslashes in this code would be unnecessary in a non-knitr context.

Let's end the document.

\end{document}

We now save the document. Assuming that stats.Rnw is still in the working directory we go back to R and type:

```R
knit2pdf("stats.Rnw")
```

The document stats.pdf will now be in your working directory. The document is shown here. LaTeX/knitr code for the entire example can be found here.

The function `purl` in `knitr` can be used to pull R code from a knitr document and compile an .R file of the code. To create such a file for the current example I would type:

```R
purl("stats.Rnw")
```

RStudio has a number of outstanding features that simplify writing documentation. For instance, generating html R-documentation in RStudio requires no additional LaTeX support. Generation of Sweave and knitr pdfs, however, does require MikTex-alike installation although this process is greatly simplified in RStudio through the straightforward creation of .Rnw and .Rd files, among other features. Implementation of a simple documentation framework called R Markdown is also provided in RStudio, which allows compilation of many output formats including html, pdf, and even MS Word. 

**Exercises**

R and type:

```R
knit2pdf("stats.Rnw")
```

The document stats.pdf will now be in your working directory. The document is shown here. LaTeX/knitr code for the entire example can be found here.

The function `purl` in `knitr` can be used to pull R code from a knitr document and compile an .R file of the code. To create such a file for the current example I would type:

```R
purl("stats.Rnw")
```

RStudio has a number of outstanding features that simplify writing documentation. For instance, generating html R-documentation in RStudio requires no additional LaTex support. Generation of Sweave and knitr pdfs, however, does require MikTex-alike installation although this process is greatly simplified in RStudio through the straightforward creation of .Rnw and .Rd files, among other features. Implementation of a simple documentation framework called R Markdown is also provided in RStudio, which allows compilation of many output formats including html, pdf, and even MS Word.
22 Exercises

1. The following questions concern the history of and general characteristics of R.
   (a) Who were the creators of R?
   (b) When was R first introduced?
   (c) What languages is R derived from/most similar to?
   (d) What features distinguish R from other languages and statistical software?
   (e) What are the three operating systems R works with?

2. Perform the following operations.
   (a) Leave a note to yourself in the console.
   (b) Identify your working directory.
   (c) Change your working directory.
   (d) Create an object called x that contains the numeric entries 1, 2, and 3.
   (e) Examine x in the R console.
   (f) Make a copy of x called y.
   (g) List the current objects in your work station.
   (h) Save your history and open the file in a text editor.
   (i) Save your objects using save.image(). Close and reopen R. Do x and y still exist?

3. Solve the following mathematical operations using lines of code in R. Show code.
   (a) \(1 + \frac{3}{10} + 2\)
   (b) \((1 + 3)/10 + 2\)
   (c) \(\left(\frac{3}{4}\right)^{2/3}\)
   (d) \(\log_2\left(\frac{3}{10}\right)^{1/2}\)
   (e) \(3x^3 + 3x^2 + 2\) where \(x\) is \([0, 1.5, 4, 6, 8, 10]\)
   (f) \(4(x + y)\) where \(x\) and \(y\) are: \(x = [0, 1.5, 4, 6, 8]\) and \(y = [-2, 0.5, 3, 5, 8]\).
   (g) \(\frac{d}{dx} \ln(x) + 2.3e^{3x}\)
   (h) \(\frac{d^{3/2}y}{dx} \frac{3}{4x^2}\)
   (i) \(\int^{15}_0 24x^3 + \ln(x)dx\)
   (j) \(\int^\infty_0 \frac{1}{\sqrt{2\pi}} e^{-x^2} dx\) (i.e., find the area under a standard normal pdf).
   (k) \(\int^\infty_0 \frac{x}{\sqrt{2\pi}} e^{-x^2} dx\) (i.e., find \(E(X)\) for a standard normal pdf).
   (l) \(\int^\infty_0 \frac{x^3}{\sqrt{2\pi}} e^{-x^2} dx\) (i.e., find \(E(X^3)\) for a standard normal pdf).

   (m) Find the arithmetic mean, median, variance, skew and kurtosis of the data \(x\) = \([0, 1.5, 4, 6, 8, 10]\). Functions for skew and kurtosis are in the package asbio.

Exercises

4. Read the section on linear algebra in the mathematical Appendix from the Foundational and Applied Stats textbook. Let:
   \[
   A = \begin{bmatrix} 2 & -3 \\ 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 5 \end{bmatrix}
   \]
   Find:
   (a) \(Ab\)
   (b) \(bA\)
   (c) \(\det(A)\)
   (d) \(A^{-1}\)
   (e) \(A'\)

   (f) We can solve systems of linear equations using matrix algebra with the function \(Ax = b\), and \(A^{-1}b = x\). In this notation \(A\) contains the coefficients from the linear equations (by row), \(b\) is a vector of solutions given in the individuals equations, and \(x\) is a vector of solutions sought in the system of models. Thus, for the linear equations:
   \[
   2x + 3y = 2 \\
   -x + 3y = 4
   \]
   We have:
   \[
   A = \begin{bmatrix} 1 & 1 \\ -1 & 3 \end{bmatrix}, \quad x = \begin{bmatrix} x \\ y \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ 4 \end{bmatrix}
   \]
   And we find: \(x = \begin{bmatrix} 1/2 \\ 3/2 \end{bmatrix}\).

   Solve the system below using linear algebra with R:
   \[
   3x + 2y - z = 1 \\
   2x - 2y + 4z = -2 \\
   -x + 0.5y - z = 0
   \]

   The area of overlap is:
   \[
   x\_new <- x[x>=-2&x<=1] \\
   y1\_new <- y1[which(x %in% x\_new)] \\
   polygon(x\_new, y1\_new, density = 20) \# make sure overlap polygon makes sense \\
   library(geometry) \\
   polyarea(x\_new, y1\_new) \# area of poloygon using triangle method \\
   [1] 4.495
   \]

   Now your turn! Re-engineer the code provided above to draw the following system of inequalities, and to find the area of their intersection.
Exercises

5. The function to the right has the form $f(x, y) = (x^2 + 6y^3)e^{-x^2 - y^2}$.

Find the area under the function surface between $x = 0$ and 2 and $y = -1$ and 2. That is, find:

$$\int_{-1}^{2} \int_{0}^{2} (x^2 + 6y^3)e^{-x^2 - y^2} \, dx \, dy.$$

The solution in R will require the function adaptIntegrate in package cubature.

6. Distinguish expressions and assignments in R.

7. Complete the following exercises concerning R packages:

(a) Install the package asbio.

(b) Load the asbio package for the current work session.

(c) Access the help file for bplot (a function in asbio).

(d) Load the dataset fly.sex from asbio.

(e) Obtain documentation for the dataset fly.sex.

(f) Create a barplot of patient 201 FEV1 responses over time under drug a; i.e., plot row 1 from asthma.

8. The speed of the earth rotating on its axis, $E$, is approximately 1700km/hr, or 1037 mph, at the equator. We can calculate the velocity of the rotation of the earth at any latitude with the equation, $V = \cos(latitude) \times E$. The latitude of Pocatello, Idaho is 42.871 degrees. We can calculate the velocity of the rotation of the earth at this latitude. Remember, the function \( \cos() \) assumes inputs are in radians, not degrees.

9. Make a plot showing the relationship between the speed of the earth's rotation and latitude. See question 8 above for more information.

(a) Create a sequence of numbers (degrees) from 0 to 90. Give this vector a name.

(b) Convert the vector in (a) from degrees to radians. Give this vector a name.

(c) Calculate velocities for the vector in (b). Give this vector a name.

(d) Plot these velocities versus latitude, i.e. the vector from (a). Label axes appropriately.

10. Consider the following variables:

\[ x^2 + y^2 \leq 2 \]
\[ -x + 3y \geq 4 \]

Exercises

5. 8. The speed of the earth rotating on its axis, $E$, is approximately 1700km/hr, or 1037 mph, at the equator. We can calculate the velocity of the rotation of the earth at any latitude with the equation, $V = \cos(latitude) \times E$. The latitude of Pocatello, Idaho is 42.871 degrees. We can calculate the velocity of the rotation of the earth at this latitude. Remember, the function \( \cos() \) assumes inputs are in radians, not degrees.

9. Make a plot showing the relationship between the speed of the earth's rotation and latitude. See question 8 above for more information.

(a) Install the package asbio.

(b) Load the asbio package for the current work session.

(c) Access the help file for bplot (a function in asbio).

(d) Load the dataset fly.sex from asbio.

(e) Obtain documentation for the dataset fly.sex.

(f) Create a barplot of patient 201 FEV1 responses over time under drug a; i.e., plot row 1 from asthma.

8. The speed of the earth rotating on its axis, $E$, is approximately 1700km/hr, or 1037 mph, at the equator. We can calculate the velocity of the rotation of the earth at any latitude with the equation, $V = \cos(latitude) \times E$. The latitude of Pocatello, Idaho is 42.871 degrees. Use R to find the rotational speed of the earth at this latitude. Remember, the function \( \cos() \) assumes inputs are in radians, not degrees.

9. Make a plot showing the relationship between the speed of the earth's rotation and latitude. See question 8 above for more information.

(a) Create a sequence of numbers (degrees) from 0 to 90. Give this vector a name.

(b) Convert the vector in (a) from degrees to radians. Give this vector a name.

(c) Calculate velocities for the vector in (b). Give this vector a name.

(d) Plot these velocities versus latitude, i.e. the vector from (a). Label axes appropriately.

10. Consider the following variables:

\[ x^2 + y^2 \leq 2 \]
\[ -x + 3y \geq 4 \]

Exercises

5. The function to the right has the form $f(x, y) = (x^2 + 6y^3)e^{-x^2 - y^2}$.

Find the area under the function surface between $x = 0$ and 2 and $y = -1$ and 2. That is, find:

$$\int_{-1}^{2} \int_{0}^{2} (x^2 + 6y^3)e^{-x^2 - y^2} \, dx \, dy.$$

The solution in R will require the function adaptIntegrate in package cubature.

6. Distinguish expressions and assignments in R.

7. Complete the following exercises concerning R packages:

(a) Install the package asbio.

(b) Load the asbio package for the current work session.

(c) Access the help file for bplot (a function in asbio).

(d) Load the dataset fly.sex from asbio.

(e) Obtain documentation for the dataset fly.sex.

(f) Create a barplot of patient 201 FEV1 responses over time under drug a; i.e., plot row 1 from asthma.

8. The speed of the earth rotating on its axis, $E$, is approximately 1700km/hr, or 1037 mph, at the equator. We can calculate the velocity of the rotation of the earth at any latitude with the equation, $V = \cos(latitude) \times E$. The latitude of Pocatello, Idaho is 42.871 degrees. Use R to find the rotational speed of the earth at this latitude. Remember, the function \( \cos() \) assumes inputs are in radians, not degrees.

9. Make a plot showing the relationship between the speed of the earth's rotation and latitude. See question 8 above for more information.

(a) Create a sequence of numbers (degrees) from 0 to 90. Give this vector a name.

(b) Convert the vector in (a) from degrees to radians. Give this vector a name.

(c) Calculate velocities for the vector in (b). Give this vector a name.

(d) Plot these velocities versus latitude, i.e. the vector from (a). Label axes appropriately.

10. Consider the following variables:

\[ x^2 + y^2 \leq 2 \]
\[ -x + 3y \geq 4 \]
Exercises

(b) Distinguish points with respect to treatment types by altering plotting shapes and colors. Add an informative legend.

(c) Create a boxplot showing longevity as a function of treatment type.

(d) Create an interval plot (using `bplot()`) showing means and standards errors of longevity for each treatment level.

15. Load the `fly.sex` dataset from `asbio` and use `qplot()` from `ggplot2` to complete the following graphics exercise.

(a) Plot longevity as a function of thorax length.

(b) Distinguish points with respect to treatment types by altering plotting shapes and colors.

(c) Alter axis labels, axis text, panel margins, point size, and legend location using `theme()` and appropriate geoms.

16. Create the following data structures:

(a) A list containing the objects created in (a) and (b).

(b) A dataframe object with two columns; one column containing the numeric entries 1,2,3,4, an one column containing the character entries “a”, “b”, “c”, “d”.

(c) A list containing the character entries “a”,”b”, “c”, “d”.

(d) Show how one would access the fourth element form the third column using `names()`.

(e) Use the functions `sort()`, `order()` and `apply()` to classify heights equal to 58 or 59 as “small”.

17. Access the dataset `cliff.sp` in the library `asbio` which describes cliff vegetation from Yellowstone National Park, USA.

(a) Show how one would access the second column of data using R.

(b) Use the function `which()` to identify which plant heights are greater than or equal to 33.2 dm.

(c) Use `complete.cases()` to eliminate rows with missing data and rerun (b). Discuss the consequences.

18. Consider the data from the table below

<table>
<thead>
<tr>
<th>Plant height (dm)</th>
<th>Soil % N</th>
<th>Water index (1-10)</th>
<th>Management Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.3</td>
<td>12.0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>21</td>
<td>12.5</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>24.7</td>
<td>14.3</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>25</td>
<td>14.2</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>26.3</td>
<td>15.0</td>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>22</td>
<td>14.0</td>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>31 NA</td>
<td>15</td>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>30</td>
<td>13.3</td>
<td>9</td>
<td>E</td>
</tr>
<tr>
<td>42</td>
<td>15.2</td>
<td>10</td>
<td>E</td>
</tr>
<tr>
<td>28.9</td>
<td>13.6</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>33.3</td>
<td>14.7</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>35.2</td>
<td>14.3</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>36.7</td>
<td>16.1</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>34.4</td>
<td>15.8</td>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>33.2</td>
<td>15.3</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>35</td>
<td>14.0</td>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>41</td>
<td>14.1</td>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>43</td>
<td>16.3</td>
<td>9</td>
<td>E</td>
</tr>
<tr>
<td>44</td>
<td>16.5</td>
<td>10</td>
<td>E</td>
</tr>
</tbody>
</table>

19. Create .csv and .txt datasets and read them into R.

20. Export the data frame from Q. 19 to your working directory.

21. Given the following dataset provide solutions and R code.

```r
Q.21 <- data.frame(height.in = c(70, 76, 72, 73, 81, 66, 69, 75, 80, 81, 60, 64, 59, 61, 66, 63, 58, 67, 59), weight.lbs = c(160, 185, 180, 186, 200, 156, 163, 178, 186, 189, 150, 136, 141, 158, 154, 135, 120, 145, 117), sex = c(rep("M", 10), rep("F", 10)))
```

(a) Query wether all heights are less than or equal to 80 inches.

(b) Query wether any heights are more than 80 inches.

(c) Find the mean height of females (i.e. F) greater than or equal to 59 inches but less than 63 inches.

(d) The mean weight of males who are 75 or 76 inches tall.

(e) Use `ifelse()` or `if()` to classify heights equal to 58 or 59 as “small”.

22. Consider the dataset from Q. 18.

(a) Use the function `replace()` to identify samples with soil N less than 13.5% by defining them as “N-poor”.

(b) Use the function `which()` to identify which plant heights are greater than or equal to 33.2 dm.

(c) Sort plant heights using the function `sort()`.

(d) Sort the dataset with respect to ascending values of plant height using the function `order()`.

23. Using `match` or which and `&` in % replace the code column names in the dataset `cliff.sp` from `asbio` with the correct species names from `sp.list` below.

```r
```

24. Create an object, `x`, containing the letters of the alphabet using: `x <- letters`.

(a) Verify that `x` is of character and not classes logical, factor or numeric.

(b) Coerce `x` to be a factor called `y`.

(c) Coerce `y` to be numeric and multiply it by 10.
25. The following questions consider NA, NaN and NULL.
   (a) For the soil N data in Q. 18, identify which elements are not missing using ! = NA. Discuss the results.
   (b) For the soil N data in Q. 18, identify which elements are not missing using !is.na(). Discuss the results.
   (c) Discuss the printed result for log(-2).

26. The following questions concern binary data and floating point applications.
   (a) Find the binary representation of the number 32 by hand.
   (b) Find the digital numbers for the binary numbers 101101, 101010101, and 0.111 by hand. Verify your answer using bin2dec.
   (c) Use bin2dec to evaluate different binary representation of the decimal number 1/3. Use 0.0101, 0.010101, and 0.01010101010101010101. Which provides the most 'precise' representation of 1/3? Why?
   (d) Use R to find the Unicode binary representation of the letter 'A'.

27. Divide the values in the first two columns of the dataset in Q. 18 by their respective column sums by specifying both 10% trimming, and the removal of missing values.

28. Below is McIntosh’s index of diversity (McIntosh 1967):
   \[ U = \frac{1}{S} \sum_{i=1}^{S} n_i^2, \]
   where \( S \) is the total number of species, and \( n_i \) is the number of individuals in the \( i \)th species. Write a function to calculate this index. Run it on a sample from a site that contains five species represented by 5, 4, 5, 3, and 2 individuals respectively.

29. Below is the Satterthwaite formula for approximating degrees of freedom for the \( t \) distribution.
   \[ \nu = \frac{\left( \frac{S_X^2}{n_x} + \frac{S_Y^2}{n_y} \right)^2}{\left( \frac{S_X^2/n_x}{n_x-1} + \frac{S_Y^2/n_y}{n_y-1} \right)^2}, \]
   where \( S_X^2 \) is the sample variance for \( X \), \( S_Y^2 \) is the variance for the variable \( Y \), \( n_x \) is the sample size for variable \( X \), and \( n_y \) is the sample size for variable \( Y \).
   (a) Write a function for this equation that has the variables \( X \) and \( Y \) as arguments.
   (b) Test the function for \( x \leftarrow c(1, 3, 2, 4, 5) \) and \( y \leftarrow c(2, 3, 7, 8, 9, 10, 11) \).

30. Create a function, implementing switch(), that can calculate the first or second derivative of a mathematical expression with respect to \( "x" \) (see Section 6.1). Test it on expression \((x^3)\).

31. Using apply and the triple dot operator (\ldots) create a function call colTrim that calculates trimmed means (trim is an argument in mean; see Section 6.1) and has flexible NA values (na.rm is also an argument in mean) for columns in a quantitative matrix or dataframe. Specify two arguments in your function one for data one for \ldots. Test the function on the first two column of the data in Q. 18 cliff.sp dataset in asiso specifying both 10% trimming, and the removal of missing values.

32. Solve the systems of ODEs below for \( t = [1, 2, ..., 20] \):
   \[ \frac{dx}{dt} = ax + by, \quad \frac{dy}{dt} = cx + dy, \]
   Let \( a = 3, b = 4, c = 5, d = 6 \). Initial values for \( x \) and \( y \) can be anything but \( \{0, 0\} \).

33. Create an R animation loop using your name that changes font-size, color, and string rotation (plot argument srt), as the loop proceeds.

34. Use a for loop to create a 100 x 100 element matrix of random numbers. Accomplish this by completing the following steps.
   (a) Create an empty matrix of the correct dimensions to hold the result, and give it a name. For instance, rand.
   (b) Use a for loop to create columns in rand. Use the function runif(100) to generate 100 random numbers (the contents of one column) based on a uniform probability distribution. Loop to create other columns.

35. More fun with for loops. Here are some classic computer science applications.
   (a) A sequence of Fibonacci numbers is based on the function:
      \[ f(n) = f(n-1) + f(n-2) \quad \text{for} \quad n > 2 \]
      \[ f(1) = f(2) = 1 \]
      where \( n \) represents the \( n \)th step in the sequence.
      Using a for loop, create the first 100 numbers in the sequence, i.e. find \( f(1) \) to \( f(100) \). As a check, the first 5 numbers in the sequence should be: 1, 1, 2, 3, 5.
   (b) An interesting chaotic recursive sequence has the function:
      \[ f(n) = f(n-f(n-1)) + f(n-f(n-2)) \quad \text{for} \quad n > 2 \]
      \[ f(1) = f(2) = 1 \]
      Using a for loop, create the first 100 numbers in the sequence, i.e. find \( f(1) \) to \( f(100) \). As a check, the first 5 numbers in the sequence should be: 1, 1, 2, 3, 5.

36. Below is a mock dataset of plant height with respect to three treatments.

\[
\begin{array}{c|c|c}
\text{height.data} & \text{height.frame} & \text{treatment} \\
\hline
20 & 20 & \text{rep(1:3, each = 5, times = 1)} \\
30 & 30 & \text{rep(1:3, each = 5, times = 1)} \\
40 & 40 & \text{rep(1:3, each = 5, times = 1)} \\
50 & 50 & \text{rep(1:3, each = 5, times = 1)} \\
60 & 60 & \text{rep(1:3, each = 5, times = 1)} \\
\end{array}
\]

(a) Create a function utilizing tapply() that simultaneously calculates means, variances, standard deviations, minimums, maximums and medians with respect to a vector of categories. Use it to find these estimates simultaneously for all three treatments in height.data.
(b) Use the function which() to find which value in height.data is closest to 9.
(d) Use the function `system.time()` to compare the run time for the functions in (a) and (c). Describe your results.

37. Create a `tcltk` GUI to run the animation function in Q. 33.


39. Write R documentation using `Sweave`, `knitr` or R Markdown describing the work done in Q. 8. This will be greatly facilitated through the use of RStudio.

40. Write R documentation using `Sweave`, `knitr` or R Markdown describing the work done in Q. 9. This will be greatly facilitated through the use of RStudio.

41. Perform a statistical and/or graphical analysis of your own data in R and document the workflow using `Sweave`, `knitr` or R Markdown.
References


Index

Symbols

:: 132
::: 132
? 10
... 111
[ ] 80
# 5
<< 107
+ (addition) 17
& (and) 87
<- (assignment operator) 13
/ (division) 17
* (exponentiation) 17
* (function of) 35
> (greater than) 87
>= (greater than or equal to) 87
% (greater than or equal to) 87
% <= (less than or equal to) 87
* (multiplication) 17
*(-) (not equal to) 87
| (or) 87
- (subtraction) 17
A
abline() 36
abs() 17
acos() 17
akima 23, 25
all() 17
alpha diversity 111
animation 125
apply() 83
arguments, also see formals() 17
array() 69
as.array() 96
as.character() 96
cbind() 74
colMeans() 111
colnames() 70, 112
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
colSums() 83
arguments, also see formals() 17
assignments also see formals() 17
as.array() 69
as.data.frame() 69
as.factor() 69
as.function() 69
as.list() 69
as.matrix() 69
as.numeric() 69
aspell assignments also see formals() 17
atan() 17
attach() 17
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RStudio</td>
<td>105</td>
<td>unirout()</td>
</tr>
<tr>
<td>RWineEdt</td>
<td>105</td>
<td>unstack()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update.packages()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upper.tri()</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
<td>Vaccinium vitis-idaea</td>
</tr>
<tr>
<td>S3 system for class and method</td>
<td>132</td>
<td>var()</td>
</tr>
<tr>
<td>SAS</td>
<td>143</td>
<td>varechem</td>
</tr>
<tr>
<td>Satterthwaite formula</td>
<td></td>
<td>varespec</td>
</tr>
<tr>
<td>save()</td>
<td>79</td>
<td>vegan</td>
</tr>
<tr>
<td>savehistory()</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>save.image()</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>scan()</td>
<td>74, 77</td>
<td>View()</td>
</tr>
<tr>
<td>scatterplot</td>
<td>35</td>
<td>vignette()</td>
</tr>
<tr>
<td>scatterplot3d</td>
<td>61</td>
<td>Vim</td>
</tr>
<tr>
<td>Scheme</td>
<td>2</td>
<td>Visual Basic</td>
</tr>
<tr>
<td>SciTE</td>
<td>105</td>
<td>volcano</td>
</tr>
<tr>
<td>scope</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>script editor</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>sd()</td>
<td>21, 37</td>
<td>which()</td>
</tr>
<tr>
<td>setwd()</td>
<td>6</td>
<td>windowsFonts()</td>
</tr>
<tr>
<td>Shannon-Weiner index</td>
<td>111</td>
<td>WinEdt</td>
</tr>
<tr>
<td>shiny</td>
<td>130</td>
<td>win.metafile()</td>
</tr>
<tr>
<td>significand</td>
<td>102</td>
<td>wireframe()</td>
</tr>
<tr>
<td>Simpson’s index</td>
<td>111</td>
<td>with()</td>
</tr>
<tr>
<td>sin()</td>
<td>17</td>
<td>working directory</td>
</tr>
<tr>
<td>skew()</td>
<td>22</td>
<td>world.co2</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>15</td>
<td>world.pop</td>
</tr>
<tr>
<td>sort()</td>
<td>90</td>
<td>write.csv()</td>
</tr>
<tr>
<td>spdep</td>
<td>25</td>
<td>write.table()</td>
</tr>
<tr>
<td>S-Plus</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SPSS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SQL (Structured Query Language)</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>sqrt()</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>stack()</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>strsplit()</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>strtrim()</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>summary()</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>svg()</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>Sweave</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>switch()</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>Sys.sleep()</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tan()</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>tapply()</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>tcltk</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>testing</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>text()</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>TextMate</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>tif()</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>Tinn-R</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>tolower()</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>toupper()</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>trellis graphical system</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unary operator</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>