The Stat 390 R Primer

Christopher G. Green
cggreen@stat.washington.edu
http://www.stat.washington.edu/cggreen/

Revision date: **September 12, 2003**

Contents

List of Figures

List of Tables

Preface

This book is based upon material from many sources. My primary reference was the introductory R manual included with the R distribution (*An Introduction to R*, W. N. Venables, D. M. Smith, and the R Development Core Team). Other sources include the excellent help files built into R, the R-Help mailing list, and some notes from Garrett Hellenthal, my predecessor as 390 TA.

My intended audience for this book is students in introductory statistics classes, primarily Statistics 390. The initial inspiration for this book was the lack of R material devoted to students with little or no computing background. I have written this book with these students in mind.

Acknowledgements

I would like to thank David Rose, for whom I TA'd Stat 390 for four great quarters. He gave me the freedom to work on this book, and readily adopted it as the official R manual for the class. I would also like to thank Garrett Hellenthal, from whose notes I picked up many R tips.

I must, of course, thank the developers of R for producing such a useful piece of software and *giving it away*.

Finally, I would like to thank all the 390 students for using the manual and offering helpful suggestions (and catching my mistakes!).

Legal Notice

0.1 Disclaimer

This book contains information derived from on-line instruction manuals, newsgroups, mailing lists, and other outside sources. This book is provided as a convenience to assist students enrolled in Statistics 390 at the University of Washington (the "students") in installing the R programming environment ("R") on the students's home computers. Chris Green adds information to this book whenever he deems it necessary. Chris Green provides no guarantees that this book will contain up-to-date information. The date listed in the front of the book as "Last updated" is the last date on which Chris Green modified this book.

Chris Green has access to a limited number of computers, operating systems, and configurations. Chris Green therefore does not and cannot guarantee or certify the accuracy of this information. The information may contain errors and omissions, including, but not limited to, technical inaccuracies and typographical errors. The information in this book is provided "as is" without representations or warranties of any kind, either express or implied. Each person obtaining information from this resource assumes full responsibility and all risks arising from the use of and reliance on the information contained on this book. In particular, Chris Green assumes no responsibility for any losses and/or damages to computer equipment incurred by following the instructions on this book.

Chris Green reserves the right to make additions, deletions or modifications to the information contained on this book.

If you do not accept these terms, then you must not use this book.

Chapter 1

Installing R on your home computer

R is available for free on the Comprehensive R Archive Network (CRAN) website. For most systems, the base R package is available in a single, ready-to-install file.

1.1 Installing R

First, open your favorite web browser and make your way to the R Project homepage (`http://www.r-project.org/`). Click on the "CRAN" link on the menu on the left-hand side of the page. This will take you to a page listing servers from which you can download R. You should pick the server that is (geographically) closest to you. (If you are a Statistics 390 student here at the University of Washington, the CRAN mirror at UCLA (`http://cran.stat.ucla.edu/`) is probably the best one for you.)

Once you have clicked the link for your mirror of choice, you will be greeted with a page listing "Precompiled Binary Distributions" and "Source Code". You should find your operating system (Windows, Mac, Linux, etc.) under "Precompiled Binary Distributions". These are the ready-to-install files containing the base R package, and are the easiest way to get R up and running on your machine.

(The "Source Code" contains all the computer code used to create R. It must be assembled into a executable file before it can be used. Do not download the source code unless you are absolutely sure you know how to compile and set up a package from a tarball.)

Now follow the platform-specific instructions below.

1.1.1 Windows

1. Click on the "Windows (95 and later)" link.

2. Click on "base".

3. Click on "rw1071.exe". Save it to a file on your hard drive. NOTE: the installation file is about 21 Megabytes. If you are on a slow Internet connection, ask your professor to loan you an installation CD.

4. Find "rw1071.exe" on your hard drive. Although the maintainers of R regularly scan their software for viruses, it is a good idea to scan this file with your virus scanning software, just to be safe.

5. Close all other programs before beginning the installation. You should also disable your virus-protection software now, as it may interfere with the installation.

6. If you are using Windows 95, 98 or ME, you may simply double click on the "rw1071.exe", and follow the directions. If you are using Windows NT 4.x, 2000, XP, or later, you will need to run the installation program as "Administrator". If you do not have administrative privileges on your computer, ask your system administrator to install R for you.

7. After you have installed R, don't forget to reenable your virus protection software!

For additional assistance read the file "ReadMe.rw1071", also located under the "base" directory on CRAN.

Installation Notes

- CAUTION. You only need the file "rw1071.exe" to install R. *You do not need the "mini*" files.* These are designed for making a set of R installation floppies. Installing R from floppies is *really* slow. We have made installation CD's for those of you on slow Internet connections; you should borrow one of these from your professor if you find yourself in this situation.

- The current version of R, as of this writing, is version 1.7.1. It is still relatively new, so there may still be minor problems with the installation procedure. The developers of R only have access to a small number of machines, and cannot test the software on every possible platform. If you encounter any difficulties installing R, contact your TA.

1.1.2 Mac OS 8.6-9.2, OS X (without Darwin)

NOTE: According to the README file on CRAN, Mac OS Versions 8.6-9.2.2 will not be supported from R version 1.8.0 onward.

1. Click on the "MacOS (System 8.6 to 9.1 and Mac OS X)" link.

2. Download the "stuffed" file "rm171.sit". It contains the base distribution and several recommended packages. Save it to a file on your hard drive. NOTE: the installation file is about 11 Megabytes. If you are on a slow Internet connection, ask your professor to loan you an installation CD.

3. Find "rm171.sit" on your hard drive. It is a good idea to scan this file with your virus scanning software, just to be safe.

4. Drag the archive "rm171.sit" from where you downloaded it and drop it on the Alladin Stuffit Expander (TM) icon.

5. When the expander has finished you should move the folder to where you prefer to put R.

You should be ready to go. It is a good idea to read the User's Guide "rmac-FAQ.html" (which is also located on CRAN in the same directory as the installation file) before you start.

1.1.3 Mac OS X (with Darwin)

At the time of this writing, the Mac OS X Darwin version of R is only available in source code form, so there is a bit more work involved in installing R here. Luckily, the configuration scripts have been extensively tested, so installation should proceed smoothly.

1. From the starting page, go to the "Source Code for the Latest Release" link. Download the tarball "R-1.7.1.tgz" to a directory on your computer. (A *tarball* is a group of files that have been collected into one big file (a "tape archive" or "tar" file) and then compressed (usually with the `gzip` program). If you've never seen these before just think of them as the equivalent of a "zip" file or a "Stuffit" file.)

2. Decompress the file by typing "`gzip -dc R-1.7.1.tgz | tar xf - `" at the prompt. The `gzip` command will decompress the "tar" file, and the `tar` command will extract the files from the archive.

3. Run the configuration script by typing "`./configure`" at the prompt. This will create a "`Makefile`" containing the commands needed to automatically build R from the source code.

4. Assuming the `configure` script did not run into any errors, you should now build R using the command "`make`". This will process the `Makefile` you made in the previous step and assemble the source code into a working program.

1.1.4 Linux

1. Click on the "Linux" link.

2. Click on your distribution.

R has been ported to several popular distributions. If you don't see your distribution, you may be able to get one of the listed ones to work. Otherwise, you'll have to get the source tarball and build it on your machine. This is not terribly hard, as the configuration scripts have been tested on many platforms. The instructions for this are identical to those given in the Mac OS X Darwin section above. Be sure to run the installation procedure as the "root" user; otherwise, some parts of the installation may fail. If you do not have "root" priviledges on your system, ask your system administrator to install R for you.

Chapter 2

Getting Started

Now that you've got R installed on your machine, let's talk about how to do the simple things in R.

2.1 Running R

2.1.1 Starting R

Under Windows you will probably have an entry for R somewhere on your Start Menu or an icon on your desktop. Double-clicking on this icon will start the R GUI (Graphical User Interface) and bring up the R console.

Under UNIX you should be able to start R by typing 'R' at the command prompt. (If you are not sure what the exact command is, ask your system administrator.) This doesn't provide the fancy GUI like Windows does, but you can accomplish all the same tasks.

In either case, you should see the R console, the command-line interface to R. This is where you will type commands, and where R's textual output will appear.

After starting R, you should see a startup message followed by the R prompt, ">":

```
R : Copyright 2002, The R Development Core Team
Version 1.5.0  (2002-04-29)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

>
```

The exact message you see may be different that the one above (it is from the version of R installed on the UNIX system used in the production of this manual); this is no cause for concern.

2.1.2 File Management

You should create a directory to hold your R files, data, etc. R will prompt you to save your workspace upon exiting, so it is best to keep this workspace separate from your files for other classes. You may even want to create separate directories for each homework, project, etc.

It will be easier to keep things organized if you always remember to start R in the directory you have chosen to hold all your R-related files. This is easy under UNIX: simply change to that directory using `cd`, then start R as you would normally.

Under Windows, it's a tad more complicated.

1. First, find your shortcut to R. This could be an icon on your desktop or an entry on your Start Menu.
2. Right-click on the icon or entry (this should work even for an entry on your Start Menu). You should get a "context" menu.
3. Find the "Properties" entry and left-click on it. This will bring up a dialog box.
4. Find the line that says, "Start in". There will be a text box next to this line.
5. Change the entry in the text box to your R directory. Be sure to type the full path to the directory.
6. Click "Apply", then "Ok".
7. Start R, and at the prompt type `getwd()`. You should see the path of your R directory. If not, go back to Step 2, and double-check what you typed in the "Start in" text box.

You can also change directories while in R. From the command line, the commands `getwd()` will print the current directory, and the command `setwd(<directory>)` will set the current directory to <directory>.[1] Under Windows you can also use the menu: go to "File"-> "Change Dir...". You will be prompted to enter a directory or to select one using the browser (click "Browse").

2.1.3 Ending Your R Session

To quit R, use the `quit()` or `q()` commands. R will ask you if you want to save your workspace. Your workspace contains all the variables you have used so far. This is helpful if you want to come back to an assignment later.

2.2 How to Find Help in R

Now that you've started R, you'll need to know how to find help. You can get help on a specific command by typing `help(command)` at the R prompt. For instance, to find out what the `median` function does, try the following.

```
> help(median)
```

(`median` computes the median of a set of numbers.)

A synonym for the `help` command is the "?" operator. Use it by prefixing the name of the function on which you seek help:

[1] Be careful when entering directories in R. Under Windows backslashes need to be escaped, so you should type "C:\\foo\\bar.txt" instead of "C:\foo\bar.txt".

```
> ?(median)
```

To get help on special characters, enclose them in quotes and use `help`. For example:

```
> help("<-")
```

will tell you what the $<-$ operator does (it is an assignment operator).

On most systems you can type `help.start()` at the R prompt to call up help pages in your web browser. (On Windows this will typically work correctly after installation (note that it defaults to calling Internet Explorer); on other systems you might need to set an environment variable for this to work.)

Another useful function is the `example` command. It runs the examples listed in the help file for a given function.

```
example(<command name>)
```

will give you examples of *<command name>*. Continuing with our above example, let us see some examples of `median`.

```
> example(median)

median> median(1:4)
[1] 2.5

median> median(c(1:3, 100, 1000))
[1] 3
```

It is important to note that *the `example` command can only give you the examples found in the help files.* Not all commands have examples.

Finally, if you forget what the arguments to a command are, try the `args` command.

```
> args(median)
function (x, na.rm = FALSE)
NULL
```

Hence `median` takes 2 arguments. To see what each argument represents, refer back to the help page for `median`.

2.2.1 What To Do When You're Lost...

Sometimes you don't know the exact command for what you are trying to do. Fortunately, the developers of R have built some search capabilities into R that you can try out. The function `help.search()` can be used to find commands by keywords. Enclose the keyword in quotes; for example, to find the R command to compute the standard deviation, you might try

```
> help.search("standard deviation")
```

This will bring up a list of R commands containing the keywords "standard deviation" in their description.

```
Help files with alias or title matching 'standard deviation',
type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE':

sd(base)                Standard Deviation
pooledSD(nlme)          Extract Pooled Standard Deviation
```

Here we see that the command for the standard deviation is `sd()`. The "base" in parentheses means that `sd()` is part of the base R distribution, i.e, it is not located in an add-on package. (In Statistics 390, we will rarely make use of functions in add-on packages. Hence when you are searching for a command you can restrict your search to functions in the "base" package.)

Now we look up `sd()` using the ordinary help utility to find the actual syntax for the command

```
sd                    package:base                    R Documentation

Standard Deviation

Description:

     This function computes the standard deviation of the values in
     'x'. If 'na.rm' is 'TRUE' then missing values are removed before
     computation proceeds. If 'x' is a matrix or a dataframe, a vector
     of the standard deviation of the columns is returned.

Usage:

     sd(x, na.rm = FALSE)

Arguments:

       x: a numeric vector, matrix or data frame.

   na.rm: logical. Should missing values be removed?

See Also:

     'var' for its square, and 'mad', the most robust alternative.

Examples:

     sd(1:2) ^ 2
```

A word of advice: your search is only as good as the keyword you use, so be sure to choose a good one!

2.3 The Rules—R Conventions

Now let us move on to the day-to-day use of R.

Like any computer package, R has certain syntactical rules. It is important that you know these—it will save you many headaches!

1. R is case-sensitive, i.e., "a" and "A" are different.
2. Variable names, function names, etc., should contain only alphanumeric characters (A-Z, a-z, 0-9) and the period ".". Take note, *a name cannot start with a digit.*[2]
3. Unlike other computing packages (MATLAB, for instance), R does not print the results of an assignment. Hence, if you assign the value 2 to the variable x (we'll talk about how to do this shortly), R will do the assigment, but will not print anything to reflect its action.

[2]You should also avoid using the name of a built-in R function as the name of a variable. So that means you should avoid variable names like `mean`, `data`, etc.

4. Commands are separated by semicolons (";") or by a newline. You can group commands using curly braces ("{ }").

5. R is an interpreted language. This essentially means that your code will be executed one line at a time. (See the "R versus C" document for more information on this.) It is possible to extend R using C/C++/Fortran, but this is beyond the scope of this document.

6. Lines starting with "#" are comments. This is useful for documenting your code, *which you should always do*. (If you're new to programming, you'd be surprised how quickly you can forget what a piece of code does.)

7. If a command is not complete at the end of a line, R will give a continuation prompt, "+", on subsequent lines until the command is complete. This is especially useful when you are typing a really long command or a command with several parentheses.

8. You can recall previous commands you have entered using the up and down ARROWS.

Now that you know the rules of the game, it's time to play. In the next chapter we will discuss how to do basic computations in R.

Chapter 3

Computation

Now let's talk about how to compute with R. First, a brief discussion of how to declare variables and assign values to them. Then, we will talk about vectors, the simplest data structure in R. Most of our work in Statistics 390 will be done with vectors. Finally, we will present the mathematical and statistical functions you will need for Statistics 390.

3.1 Variables

3.1.1 Assignment

As we noted earlier, you may only use alphanumeric characters in the names of your variables. Furthermore, a name cannot start with a number. Thus, `x`, `price.total`, and `foobar1` are all legal variable names in R, while `3x`, `price_total`, and `foobar!` are not legal and will result in a "syntax error".

To set the value of a variable, use one of R's assignment operators. In more recent versions of R, you can use the equals sign "=" for assignment. This may be easiest for those of you coming from other languages. You can also assign values using the "<−" operator for assignment. The value goes where the arrow points! For example, both `x = 3` and `x <- 3` will set `x` to 3. It would also be okay to write `3 -> x`, although this is not a common practice (and can have rather unpleasant consequences if you are not careful).[1]

Previous versions of R also allowed one to use the "_" operator for assignment, as in `x_3`. The "_" operator is now depreciated (i.e., it will not be supported in future versions of R), so *don't use it.* We mention it here only so that you can decipher any old R code you run into.

3.2 Data Structures

R is an *object-oriented* language. Essentially, this means that certain concepts (such as vectors and matrices) and the relationships among them (for instance, a matrix can be thought of a row vector of column vectors) have been encapsulated in abstracts called *objects*. (This is admittedly a poor definition, but a precise definition of "object-oriented" is best left to a computer science class.) In R it is often useful to adopt the viewpoint that "everything is an object"—from data vectors to complicated data structures to functions. It is this viewpoint that makes R well-suited for statistical computations.

The data structure you will use most frequently in Statistics 390 is the vector. The vector is the simplest data structure in R. You will become very familiar with basic vector operations in this class.

[1] Even though `=` and `<-` accomplish the same thing, the `<-` is preferred. The reasons for this are rather technical; we refer the interested reader to http://developer.r-project.org/equalAssign.html.

3.2.1 Vectors

A *vector* is an ordered collection of elements of the same type. Many of R's functions are designed to be used with vectors (i.e., they are *vectorized*). Not only does this make programming in R a bit more natural (instead of writing a `for` loop to add two vectors you can just type "$x+y$"), it also makes your code slightly faster, as the vectorized routines have been optimized for your particular computer.

As we mentioned above, all the elements of a vector must be of the same type (*mode*, in the R parlance). The available modes in R are character, complex, logical, and numeric.

- The character mode is for strings, such as "one".
- The complex mode is for complex numbers. We won't have much use for complex numbers in this class, so this is the last time we'll mention them.
- The logical mode is for TRUE/FALSE expressions, such as the results of comparisons ("is x > 4?").
- The numeric mode is for integers and real numbers.

Creating Vectors

The most general way to create a vector in R is to use the concatentation operator, `c()`. For example,

```
> x <- c(1,2,3,4,5)
```

creates a numeric vector of 5 entries, {1,2,3,4,5}, and assigns this vector to `x`. Type `x` at the prompt to see the values of `x` (remember, R won't print the result of an assignment).

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

The "[1]" on the left-hand side of the output means that the entry "1" is the first element of the vector. When a vector is too long to fit on a single line its entries are continued on subsequent lines; the index of the first entry on each line is printed in brackets (as "[1]" was above) as handy reference.

To create a character vector, use `c()` with quoted strings.

```
> ord <- c("one","two","three","four","five")
> ord
[1] "one"   "two"   "three" "four"  "five"
```

R also has an easy way to generate numeric sequences: the colon operator ":". We could generate the previous example more succinctly as

```
> x <- c(1:5)
```

More generally, one can use the *sequence* command `seq()` and the *repeat* command `rep()` to generate arbitrary sequences. Here are some examples.

```
> seq(from=1,to=10,by=0.1)
 [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0
[12]  2.1  2.2  2.3  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1
[23]  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.0  4.1  4.2
[34]  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3
[45]  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4
[56]  6.5  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5
[67]  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4  8.5  8.6
[78]  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7
[89]  9.8  9.9 10.0
```

produces a vector containing the numbers 1 through 10, incremented by 0.1. Note once again the indices of the first entries on each line of the output are printed for your reference.

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

repeats the vector (3,4,5) four times to yield the vector (3,4,5,3,4,5,3,4,5,3,4,5). On the other hand,

```
> rep(c(3,4,5),c(1,2,1))
[1] 3 4 4 5
```

repeats the elements of (3,4,5) according the elements of (1,2,1), i.e., "3" occurs once, "4" occurs twice, and "5" occurs once. So the output is (3,4,4,5). We can furthermore combine `seq()` and `rep`.

```
> rep(seq(0,20,2),seq(0,20,2))
  [1]  2  2  4  4  4  4  6  6  6  6  6  6  8  8  8  8  8  8
 [19]  8  8 10 10 10 10 10 10 10 10 10 10 12 12 12 12 12 12
 [37] 12 12 12 12 12 12 14 14 14 14 14 14 14 14 14 14 14 14
 [55] 14 14 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
 [73] 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
 [91] 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
[109] 20 20
```

Note that `seq()` and "`:`" can only be used for numeric sequences, while `rep()` can be used for character and logical vectors.

Indexing Vectors

You can extract data from a vector with a subscript. In R subscripting is performed using square brackets ("[" and "]") and an index vector. (Note that in R vectors are indexed starting with 1, i.e., the first element has index 1.)

For example, to access the fifth element of a vector `x`, we would type `x[5]` at the R prompt. To access a range of entries, use the colon operator ":" to specify a range of indices:

```
> x <- c(0,1,2,3,4,7,8,9,10)
> x[3:6]
[1] 2 3 4 7
```

The index vector need not be contiguous; you can select any subset of a vector with indicies:

```
> x[c(1,2,5,7)]
[1] 0 1 4 8
```

You can also *exclude* elements from a vector using a set of negative indices:

```
> x[-(2:4)]
[1] 0 4 7 8 9 10
```

The negative indices specify the elements to be left out of the result.

More general and powerful subsetting of vectors can be accomplished using logical vectors. For example, if `x` is the numeric vector used in the above examples, `x[x < 5]` yields a vector containing those elements of `x` that are less than 5. This is because the "`x < 5`" is a logical vector: for each entry in `x` the condition "< 5" is evaluated as true or false, and the result is a vector of TRUE's and FALSE's.

```
> x
[1]  0  1  2  3  4  7  8  9 10
> x < 5
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Indices must be numeric, though, so the logical values TRUE and FALSE are *coerced* into 1 and 0, respectively. The "1"'s indicate which elements to keep, while the "0"'s indicate the elements to omit. Thus we have

```
> x[x < 5]
[1] 0 1 2 3 4
```

Logical indices can be quite complicated, as in the following code snippet:

```
> x[ (x >= 3) & (x <= 9) & (x != 7)]
[1] 3 4 8 9
```

Here we have used the elementwise "and" operator, `&`, to combine multiple logical conditions. Each logical expression is applied to `x` to produce a logical vector. The logical "and" function is then applied elementwise to each vector to produce the final index vector. (Recall that X AND Y is TRUE if and only if X and Y are TRUE.)

The next example illustrates one of the more common uses of logical indices in Statistics 390.

Example 3.2.1 (Estimating Probabilities). Suppose Z is a standard normal random variable. What is the probability that $Z > 2$? While this could be calculated directly, let us try to estimate the answer using R. First, we draw a large random sample from the standard normal distribution.

```
> z <- rnorm(1000)
```

Next, notice that a logical vector could be used to select the entries in `z` that satisfy our condition of interest, "> 2":

```
> y <- z[z > 2]
```

Now we must compute the observed frequency of numbers greater than 2. This is simply the number of elements in `z` that are larger than 2, divided by our sample size (1000):

```
> length(y)/1000
[1] 0.022
```

A slick way to do this is to sum the logical vector `z > 2`. If you list the vector `z > 2`, you will see it consists of `TRUE` and `FALSE` values. The R function `sum` operates on numeric vectors, so the logical values TRUE and FALSE in the vector `z > 2` will be coerced to 1 and 0, respectively. The sum of the logical vector, therefore, is precisely the number of entries of `z` that satisfy the condition "z > 2". The sample mean of `z > 2` is therefore the observed frequency of number greater than 2:

```
> mean(z > 2)
[1] 0.022
```

For comparison, the true probability $P(Z > 2)$ can be found using `pnorm`:

```
> pnorm(2,lower.tail=F)
[1] 0.02275013
```

+, −, ∗, /	Arithmetic operators.
^	Exponentiation.
>, <, >=, <= ==, ! =	Comparisons.
&, |, !	Elementwise logical operators.
log(), exp()	Natural log/exponential.
sin(), cos(), tan()	Trigonometric functions.
sqrt()	Square root.
abs()	Absolute value.
sum()	Computes the sum of the entries in a vector.
cumsum()	Cumulative sum.
prod()	Computes the product of the entries in a vector.
length()	Length of a vector.
sort()	Sort a vector.

Table 3.1: Basic mathematical functions.

3.2.2 Other Data Structures

Vectors are not the only data structure offered in R. R also provides *matrices* and *arrays* (higher-dimensional versions of matrices) for your computational needs. Matrices and arrays, like vectors, can only contain elements of the same mode. For those times when you need to group dissimilar types of data, R provides *lists*. The elements of a list can be vectors, matrices, or even other lists. Lists are used in R to return multiple values from a function. A specialized type of list, called a *data frame*, is available for representing data sets. A data frame can hold numerical and character data, complete with variable names.

For the moment, we don't have much use for these other data structures in Statistics 390. We will say more about them in later chapters as appropriate.

3.3 Mathematical and Statistical Functions

3.3.1 Basic Mathematical and Statistical Functions

See Tables 3.1 and 3.2. These commands are pretty self-explanatory. Also, all of these commands work on vectors, i.e., `mean` computes the mean of the entries of a vector. For the exact syntax of each command, consult the on-line help.

3.3.2 Probability Distributions

R contains functions for dealing with many common probability distributions. All of these functions follow a specific naming convention: a one-letter keyword (r,p,q,d) followed by an abbreviated name of the distribution. The abbreviations are usually pretty obvious; we've already seen "norm" for the normal distribution. The keywords have the following meanings.

- "r" denotes the random number generator for the distribution;
- "p" denotes the probability function (or cumulative distribution) for the distribution (e.g., `pnorm(1,0,1)` is $P(Z \leq 1)$ for a standard normal random variable Z;

mean()	Computes the mean of a vector.
weighted.mean()	Weighted mean.
max()	Computes the maximum entry in a vector.
min()	Computes the minimum entry in a vector.
median()	Computes the median of a vector.
sd()	Computes the standard deviation of a vector.
var()	Computes the variance of a vector.
range()	Computes the range of a vector.
IQR()	Computes the interquartile range.
quantile()	Computes quantiles.
table()	Makes a small frequency table.
summary()	Returns the minimum, maximum, median, mean, and 1st and 3rd quartiles of the data.

Table 3.2: Basic statistical functions.

- "q" denotes the quantile function, i.e., given $0 \leq p \leq 1$ it returns the quantile x such that $P(X \leq x) = p$; and
- "d" denotes the density or mass function for a distribution.

A complete list of the R functions for the distributions we will use in this class can be found in Appendix A.

The above commands work well for commonly encountered distributions, but what about those pesky "unnamed" distributions? Well, in this class, we wouldn't ask you to do any computations with a continuous distribution not found in R. But occasionally we will give you discrete distributions that do not correspond to an available R command. With a little thought, these are no more difficult to handle in R than the built-in distributions.

The simplest way to handle an arbitrary discrete distribution is to form two vectors: one to hold the possible values, and one to hold their corresponding probabilities. For example, suppose we have the distribution given in Table 3.3. In R we would create two vectors, one for "X" and "$p(X)$".

X	0	1	2	3
$p(X)$	1/2	1/3	1/12	1/12

Table 3.3: Example Probability Distribution

```
> x <- c(0,1,2,3)
> px <- c(1/2,1/3,1/12,1/12)
```

We can do now computations with this distribution by going back to definitions.

```
> x.mean <- sum(x*px) ; x.mean   # mean
[1] 0.75
> weighted.mean(x,px) # same thing
[1] 0.75
> x.sd <- sqrt(sum((x^2)*px) - (x.mean^2)) # standard deviation
> x.sd
[1] 0.9242114
```

The cumulative distribution can be obtained using the `cumsum` function:

```
> cumsum(px)
[1] 0.5000000 0.8333333 0.9166667 1.0000000
```

Quantiles can be read off the probability vector. To get random numbers, use the `sample` command:

```
> sample(x,100,replace=T,prob=px)
  [1] 0 1 2 0 0 0 1 0 0 2 0 1 0 0 0 1 0 0 3 2 0 1 1 3 1 1 0
 [28] 0 1 1 1 0 0 1 0 0 2 0 1 1 0 2 1 1 0 1 1 0 1 2 0 1 2 1
 [55] 0 3 1 0 0 0 1 2 0 2 1 1 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0
 [82] 0 0 1 0 0 0 0 1 0 0 3 0 0 0 1 2 1 0 1
```

Take note of the arguments to `sample`:

- the first argument is the vector from which to sample;
- the second argument is the number of samples to take;
- the third argument specifies that we want to sample with replacement; and
- the fourth argument specifies the weights (probabilities) to use.

We will make extensive use of the `sample` command in this class. As a parting example, let us simulate 100 rolls of a fair die:

```
> z <- sample(1:6,100,prob=rep(1/6,6),replace=T)
> z
  [1] 5 3 4 3 6 4 5 4 6 1 5 4 4 4 4 4 2 6 5 4 2 5 5 6 2 4 2
 [28] 5 4 6 1 4 5 5 6 6 5 4 6 1 3 3 1 1 4 4 5 4 4 4 4 4 1 3
 [55] 2 1 1 2 6 1 2 3 2 5 6 5 4 6 1 3 1 3 2 3 3 2 3 6 2 1 1
 [82] 3 3 1 2 4 4 1 6 3 6 1 2 2 4 5 1 4 3 4
```

One more example before we move on: `sample(1:100)` produces a permutation of the integers $1, 2, \ldots, 100$. Thus `z[sample(1:100)]` permutes the elements of `z` according to this permutation.

3.4 Programming Constructs

3.4.1 Logical Operators

The logical operators for R are && (and), || (or), and ! (not). These are short-circuited, i.e., they stop evaluating an expression as early as possible. For instance, if x is FALSE and y is TRUE, then in the expression `x && y` only x will be evaluated, since A AND B can only be TRUE if both arguments are TRUE.

Be careful not to use && and || when you mean & and |. The former are short-circuited while the latter are not. Also, && and || only look at the first entry of a vector. Thus the vector `c(0,1)` will be seen as FALSE by && and || but as the vector (FALSE, TRUE) by & and |.

3.4.2 Control Statements—If-Else

Like most programming languages, R has an if-else construct for building conditionals. It takes the form

```
> if ( expr ) action else action
```

when typed all on one line, and the form

```
> if ( expr ) {
+ action
+ } else
+ {
+ action
+ }
```

The braces are useful (though not necessary) when the action consists of more than one statement. (If the braces are missing, `else` is attached to the most recent `if` statement lacking an `else` clause.) Also note that in the second form of the if-else construct, the `else` statement must appear on the same line as the closing brace of the `if` statement; otherwise, R will evaluate the `if` statement immediately. [2]

The `expr` is evaluated first; if it is TRUE, the related action is performed; otherwise, if there is an else clause, its action is performed. When there is no else clause and `expr` is false, as in the construction

```
> a <- FALSE
> if (a) print(a)
```

no action is taken, and the return value of the whole expression is defined to be NULL.

`expr` should only produce one logical value.

3.4.3 For Loops

As we've mentioned earlier, R is vectorized. That is, many R commands will implicitly loop over the elements of a vector. Sometimes, however, we want to do explicit looping. R provides the `for` construct to do this.

The `for` construct has the form

```
> for ( <index variable> in <index vector> ) {
+ action
+ }
```

The index variable is the "loop counter", while the index vector contains the values over which we want to loop. For example, to loop over the values 1 to 100, you might use the construct

```
> for ( i in 1:100 ) {
+ # do something
+ }
```

The index vector need not be contiguous, however; we can loop over any vector.

```
> for ( i in c(3,7,12,4) ) {
+ # some action
+ }
```

You can loop over any type of vector:

```
> for (car in c("Ford", "Chevy", "Toyota", "Nissan")) {
+ print(car)
+ }
```

One of the most important uses for the `for` loop construct in Statistics 390 is the investigation of sampling distributions via the bootstrap. This is illustrated in the following example.

[2] This has to do with how R parses commands. R evaluates commands as soon as they are syntactically complete, i.e., as early as they can be executed. Thus, if you hit <ENTER> immediately after typing the closing brace of the `if` block, you have created a command that could be fully executed, so R does just that.

Example 3.4.1 (The Bootstrap). Suppose we want to investigate the properties of the sample median of a particular distribution, say, the standard normal distribution. While this can be done analytically, it is a fairly advanced calculation and we do not expect you to be able to do it. You can, however, investigate the sample median experimentally using R.

To bootstrap the distribution of the sample median, we will repeatedly sample from the given distribution, compute the sample median of the *sample*, and store the result. The resulting vector of sample medians can be used to investigate the properties of the sample median of the original distribution.

First, we start by initializing a vector to hold the results from each trial. Let us suppose we will perform 1,000 trials.

```
> x.stat <- rep(0,1000)
```

This creates a vector `x.stat` that is large enough to hold our results.

Next, we set up a `for` loop to perform the trials. At the R prompt we would type

```
> for ( i in 1:1000 ) {
```

Hitting <ENTER> after the opening brace gives us a continuation prompt ("+").

Now we enter the body of the `for` loop. The first step is to draw a random sample from the distribution under investigtion:

```
+ y <- rnorm(10000)
```

This gives us a random sample of size 10000 from the standard normal distribution.

Finally, we compute the sample statistic, and store the result:

```
+ x.stat[i] <- median(y)
}
```

Don't forget the closing brace for the body of the `for` loop!

After you hit <ENTER>, R will execute the `for` loop. When it is done, the vector `x.stat` will contain the sample median for each of the 1000 trials.

```
> x.stat
 [1]  0.0262186806 -0.0018468265 -0.0104944919  0.0276343065
 [5] -0.0102339605 -0.0046865152  0.0069701374 -0.0007195085
 [9] -0.0267888755  0.0041114397  0.0168158083 -0.0060680072
[13]  0.0076626434 -0.0023256773 -0.0069791640  0.0158987440
[17] -0.0160878824  0.0018203985 -0.0167052135  0.0076035924
[21] -0.0073469931 -0.0202203678 -0.0089333843  0.0022119175
[25]  0.0156723709  0.0068493387 -0.0146437820  0.0011926394
[29]  0.0092101143  0.0260376860  0.0051305362  0.0075694974
[33]  0.0306138884 -0.0037979060 -0.0006837109  0.0001682163
[37]  0.0148673917  0.0005019140  0.0035311870 -0.0024472341
[41]  0.0220545558  0.0134181848  0.0116971681 -0.0046658552
[45] -0.0099204343 -0.0199975575  0.0025972221  0.0074276502
[49]  0.0087353238 -0.0015801124 -0.0010855275 -0.0097489003
[output trimmed to save space]
```

We can then compute statistics of the sample median, estimate its distribution, etc.

Chapter 4

Visualizing Data—Graphics in R

Graphics commands in R come in three flavors: high-level, low-level, and interactive.

4.1 High-Level Graphics Commands

High-level graphics commands are plotting commands that can stand on their own—given data, they can produce a plot complete with axes and titles. High-level graphics are used to produce common types of plots, such as histograms, scatterplots, boxplots, and stem-and-leaf plots.

The most commonly used high-level graphics command is the `plot` command. The `plot` command is actually an interface to more specialized plotting commands; `plot` automatically chooses the correct type of plot based on the input data. For example, given two vectors `x` and `y`, `plot(x,y)` will produce a scatterplot (see the next section for an example of this). Similarly, given a single vector `x`, `plot(x)` plots the values of `x` against their index.

We will discuss more specialized applications of the plot command in later chapters.

Other useful high-level plotting commands include

- `boxplot`, which produces a boxplot;
- `hist`, which produces a histogram; and
- `stem`, which produces a stem-and-leaf plot.

In this class we will not have much use for stem-and-leaf plots, so we will say no more about them. The `boxplot` and `hist` commands will be discussed in the sequel.

4.1.1 Scatterplots

Scatterplots can be made with the `plot` command. The ideal way to do this is to put your data into two vectors, x and y. Then to plot x versus y, use the command `plot(x,y)`. For example

```
> x <- seq(0,10,0.1)                    # making some data
> x <- x[sample(1:length(x))]           # randomize for fun
> y <- -1.5 + 4*x + rnorm(length(x),0,2)# linear function + noise
> plot(x,y)                             # make a scatterplot
```

The resulting figure is shown in Figure 4.1.

If you provide only one vector to plot, the entries of the vector will be plotted against their indices. For example, let us plot the results of 100 rolls of a fair die.

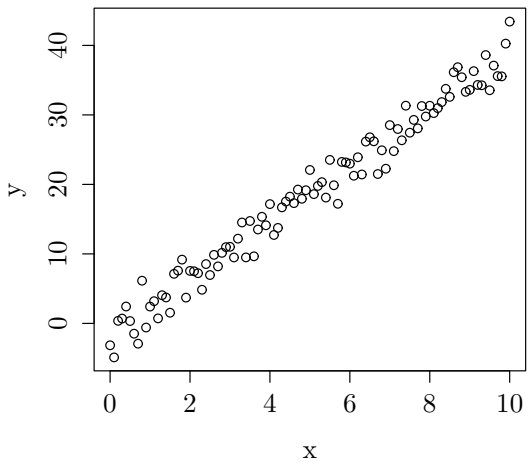

Figure 4.1: Example Scatterplot

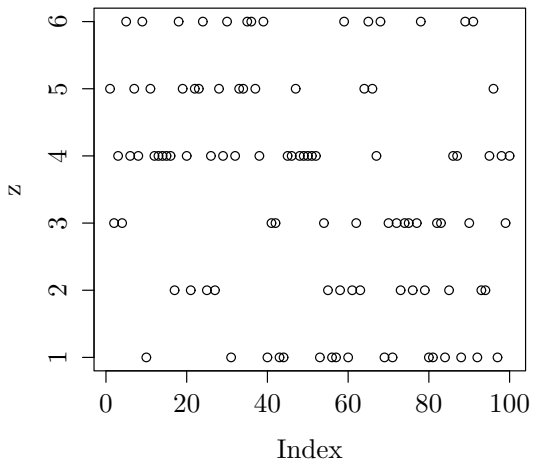

Figure 4.2: Scatterplot of Dice Rolls

```
> z <- sample(1:6,100,prob=rep(1/6,6),replace=T)
> z
  [1] 5 3 4 3 6 4 5 4 6 1 5 4 4 4 4 4 2 6 5 4 2 5 5 6 2 4 2
 [28] 5 4 6 1 4 5 5 6 6 5 4 6 1 3 3 1 1 4 4 5 4 4 4 4 4 1 3
 [55] 2 1 1 2 6 1 2 3 2 5 6 5 4 6 1 3 1 3 2 3 3 2 3 6 2 1 1
 [82] 3 3 1 2 4 4 1 6 3 6 1 2 2 4 5 1 4 3 4
> plot(z)
```

Figure 4.2 shows the resulting scatterplot.

The `plot` command has plenty of options. You can see them all by typing `help(plot)` at the R prompt. Some of the more common options you'll see in this class:

`type` type of plot—change the type of plot. Possible values include "p" to plot points, "l" to plot lines, "b" to plot both points and lines, and "n" to draw the axes for the plot without plotting the input data.

`pch` plot character—can be used to change the plot symbol.

`cex` character expansion—allows you to change the font size used in the plot.

`xlab` x-label—allows you to label the *x* axis.

`ylab` y-label—allows you to label the *y* axis.

`xaxs` x-axis—by default, R leaves some space around the limits of a plot; thus, the "true" x-axis is actually slightly above the labelled x-axis. Use the option `xaxs="i"` to disable this behavior. This is useful if you want to shade the area under a graph, for instance.

`yaxs` y-axis—similar to `xaxs`, but for the y-axes.

`col` color—specify the color to use for plotting.

Most of these options are actually general plotting options. More will be said about these later in this chapter.

4.1.2 Boxplots

Boxplots are useful for comparing related sets of data. For example, if we are running a clinical trial to test whether Drug A is significantly better than Drug B, a boxplot of the data from Drug A and Drug B is a quick way to see if there is a difference. A hypothesis test can then be performed to verify that the difference is significant. Finally, the boxplot also serves as a sanity check—if the result of our hypothesis test contradicts what the boxplot indicates, this alerts us to a possible error in our computations.

To make a boxplot, use the `boxplot()` command. In Figure 4.3 we see a boxplot of the data in the vector `z` used in the above example.

```
> boxplot(z,col="grey")
```

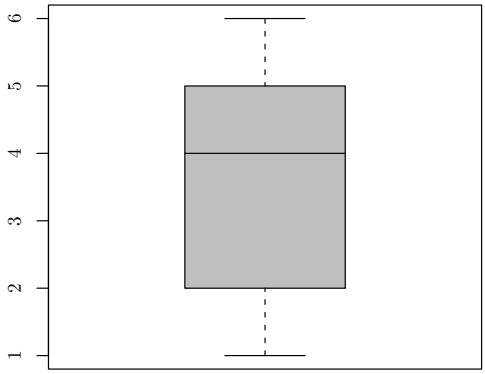

Figure 4.3: Boxplot of data from dice rolls experiment.

4.1.3 Histograms

Histograms in R are created using the `hist` command. For example, a histogram of the dice roll data can be produced by the command:

```
> hist(z)
```

This produces the graph shown in Figure 4.4. Note that this is a *frequency* histogram: the height of each bar is the number of occurences of the corresponding value or class. To get a *density* histogram, in which the *area* of each bar is the relative frequency of the corresponding value or class, use either the `prob=TRUE` argument or the `freq=FALSE` argument to hist:

```
> hist(z,prob=TRUE)
> hist(z,freq=FALSE) # these do the same thing
```

We will obtain the graph shown in Figure 4.5. We should point out that `hist` defaults to putting the

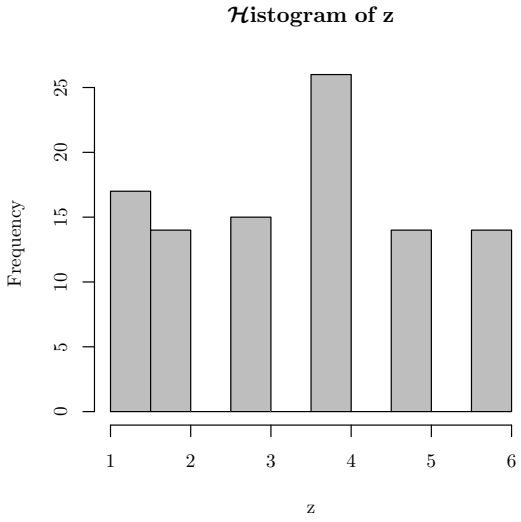

Figure 4.4: Histogram produced using `hist(z)`

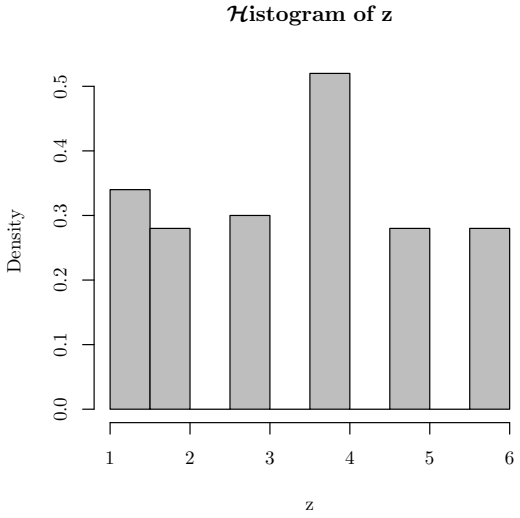

Figure 4.5: Histogram produced using `hist(z,prob=TRUE)`

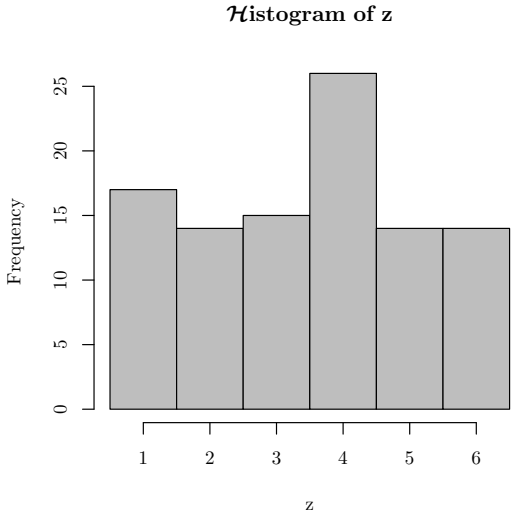

Figure 4.6: Histogram produced using an explicit `breaks` command.

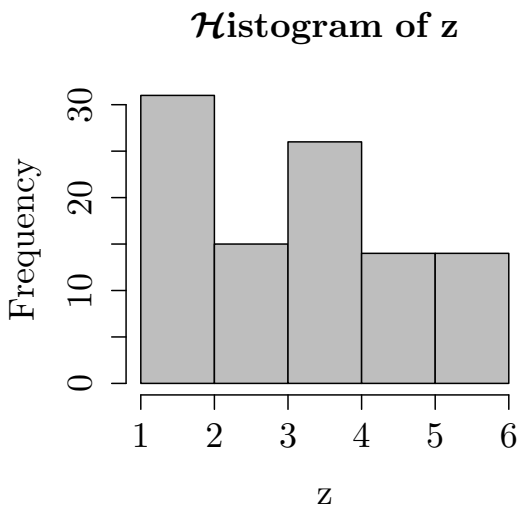

Figure 4.7: Histogram of dice roll data using `br=5` option.

actual data points on the right side of the interval (except the first data point, which is placed on the left). Usually, when we make histograms, we put the actual data point in the middle of the interval. Thus, if your hand-constructed histogram looks very different from your R-constructed histogram, this behavior may be the culprit.

One way to force R to plot histograms in the traditional fashion is to explicitly specify the breakpoints using the `breaks` option to hist (abbreviated here as `br`):

```
> hist(z,br=c(0.5,1.5,2.5,3.5,4.5,5.5,6.5),col="grey")
```

The result of this is shown in Figure 4.6.

You can also use the `breaks` option to stipulate the number of bins that R should use.

```
> hist(z,br=5,col="grey")
```

The above code instructs R to construct a histogram of the dice roll data using 5 bins. The resulting figure is shown in Figure 4.7. A synonym for this use of the `breaks` option is the `nclass` option. The command

```
> hist(z,nclass=5,col="grey")
```

will produce the same plot as the above code.

Another quirk about `hist`: you cannot make a relative frequency histogram using only `hist`. The developers of R are of the opinion that a density histogram is a more favorable alternative to a relative frequency histogram. If you are really set on constructing a relative frequency histogram, however, it can be done using a combination of the `hist` command and the `axis` command. We will illustrate this after we discuss the `axis` command.

4.1.4 Visualization in More Than Two Dimensions

When several variables are involved, it is often useful to investigate the relationships between them. R provides several commands to do just this.

- The `pairs` command will produce a matrix of all pairwise scatterplots: if `x` is a data frame with three variables A, B, and C, `pairs(x)` will produce a 3x3 matrix containing scatterplots of A versus B, A versus C, and B versus C.

More general multivariate plotting capabilities are provided by the powerful Trellis graphics package (in the `lattice` package).

We will return to higher-dimensional data sets in a later chapter on Regression.

4.2 Low-Level Graphics Commands

Low-level plotting commands are used to add information to an existing plot.

- `points` adds points to an existing plot. Its syntax is identical to that of `plot`; it even accepts many of the same arguments.
- `lines` adds connected lines to an existing plot. It works similarly to `plot`.
- `title(main="<your title>"` adds the title <your title> to a plot.
- `abline` adds a straight line to a plot. `abline` accepts a slope and intercept, or a linear model object (see the chapter on regression)).

The following are some other low-level graphics commands that you may occasionally encounter in demonstrations in this class. They are mostly for making fancy plots and annotations; as such, we do not expect you to use them in your homework.

- `text` adds text at a specified location to a plot.
- `legend` adds a legend to a plot.
- `axis` adds an axis to a plot.
- `segments` adds disconnected lines to a plot. Use `lines` for a connected sequence of lines.
- `polygon` shades regions of a plot.
- `rect` draws a rectangle with specified coordinates.
- `arrows` add arrows to a plot (to point to something of interest, for example).

As an illustration of these "fancy" commands, we now demonstrate how to construct a true relative frequency histogram in R. The resulting figure is shown in Figure 4.8

```
> xx <- rep(c(1:6),c(34,56,22,10,4,5))
> histout <- hist(xx,br=seq(0.5,6.5,1),axes=FALSE,
+ ylab="Relative Frequency",main="Relative Frequency Histogram of xx")
> axis(1)
> ticks <- seq(0,max(histout$counts),1)
> labels <- ticks/length(x)
> labels <- round(labels,2)
> axis(2,ticks,labels)
```

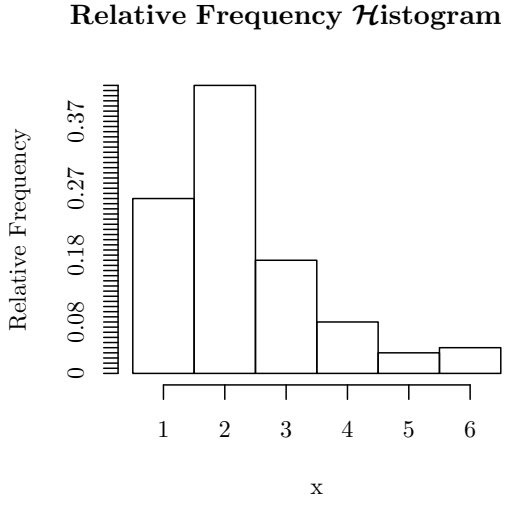

Figure 4.8: Relative Frequency Histogram Produced Using R

4.3 Interactive Graphics Commands

At the moment, we will not have much use for interactive graphics. You may see your TA use these commands in a demonstration, though, so we'll just mention a few.

- `locator` gets the coordinates of a point.
- `identify` identifies points on a plot.

4.4 Graphics Parameters

4.4.1 Making an Array of Plots

It is often useful to place several plots side by side. This is quite easy to do with R, using the `mfrow` and `mfcol` plotting options.

The `mfrow` option to `par` specifies the size of the plotting array. It is used as follows. Suppose we want to plot 6 graphs in 2 rows and 3 columns. We set this up using the call

```
> par(mfrow=c(2,3))
```

We then make our plots as usual. The plot array will be filled from left to right, top to bottom. When the last cell is reached, the next plot will appear in the first cell.

The `mfcol` option to `par` works like `mfrow`, but you specify the columns first.

4.4.2 Other Useful Plotting Options

`pch` plot character—can be used to change the plot symbol. To see the available plotting symbols, use `example(points)` and check out the last graph. Example: `pch=1` uses open circles as plotting characters.

`cex` character expansion—allows you to change the font size used in the plot. Specify a multiple of the default size. Example: `cex=1.2`.

`lwd` line width—change the thickness of the plotting lines. Specify a multiplier of the default line width. Example: `lwd=2`.

`lty` line type—change the type of line. Available types are 0 (blank), 1 (solid), 2 (dashed), 3 (dotted), 4 (dotdash), 5 (longdash), and 6 (twodash). Example: `lty=5`.

`xlab` x-label—allows you to label the *x* axis. Example: `xlab="foo"`.

`ylab` y-label—allows you to label the *y* axis. Example: `ylab="bar"`.

`xaxs` x-axis—by default, R leaves some space around the limits of a plot; thus, the "true" x-axis is actually slightly above the labelled x-axis. Use the option `xaxs="i"` to disable this behavior. This is useful if you want to shade the area under a graph, for instance.

`yaxs` y-axis—similar to `xaxs`, but for the y-axes.

`col` color—specify the color to use for plotting, either by number or by name. Example: `col="red"`.

4.5 Other Graphics-Related Commands

Normally, graphics commands send their output to the currently open graphics windows. Sometimes, though, you would like to send a graph to a new window. To launch a new graphics window, use the one of the following commands:

1. `X11()` if you are on UNIX;
2. `windows()` if you are on Windows; or
3. `macintosh()` if you are on Macintosh.

To move between the windows, click the top border of a window or use the "Window" menu. (On UNIX, you can use the `dev.cur()` and `dev.set()` commands.) To close a window, click the "x" in the upper right corner (or use `dev.off()` on UNIX).

The `colors()` function will give a list of the colors available on a system.

4.6 Getting Plots into Word, Powerpoint, etc.

On Windows, if you right-click on a plot, you will be given some options to copy or to save the graph. "Copy as Metafile" works well with Microsoft Word and Powerpoint. Just paste the graph into an open Word or Powerpoint document.

There are also functions for writing graphics output directly to various file formats, such as Postscript, PDF, JPEG, bitmap, etc. Your TA can help you with these if you are interested. For the most part, though, the cutting-and-pasting method described above is sufficient for your homework. In this class we are not looking for publication-quality graphics, so you shouldn't spend all your homework time trying to make fancy plots. A simple plot that conveys your point is just fine.

Chapter 5

Odds and Ends

There are some additional commands which will be handy for this class. They didn't really fit into the previous chapters, though, so we've collected them together here.

5.1 Input and Output

5.1.1 Reading Commands From a File

Sometimes it is easier to write your code in a separate file all at once, and then run all your commands at once. To read commands from a file, say "foo.txt", use the `source` command:

```
> source("foo.txt")
```

Here the file "foo.txt" is assumed to be in the current directory. If it is not, change directories using `setwd()` (or the menu under Windows) or provide a full path to the file "foo.txt". Remember that on Windows, backslashes must be escaped:

```
> source("A:\foo.txt")
Error in file(file, "r") : unable to open connection
In addition: Warning message:
cannot open file 'A:foo.txt'
> source("A:\\foo.txt")
Hello from foo.txt!
```

The `source` command is also available under Windows on the File menu.

5.1.2 Redirecting Output to Text Files

Another useful thing to do is to redirect R output to a file. To redirect output to a file, use the `sink` command:

```
> sink("bar.txt")
```

To restore normal output to the screen, use sink().

Note that `sink` is only for capturing text output. To write out a data set use one of the formatted methods (such as `write.table`). To write out graphics, use one of the graphics drivers (see the Graphics chapter). On Windows, right-clicking on a plot will give you some options to save a plot to a file.

5.1.3 Reading and Writing Data Files

To read data from a file into a variable, use the `scan` command:

```
> x <- scan("foo.txt")}
```

This command will read the data in the file "foo.txt" into the variable `x`. It is assumed that "foo.txt" contains values delimited by spaces. Remember to give the full path to "foo.txt" if it is not in the current directory.

If you using the GUI version of R, you can use the `file.choose()` function to call up a dialog box instead of giving an explicit filename:

```
> x <- scan(file.choose())
```

This is often easier to use for beginners.

We will generally provide data sets that can be read with a simple `scan()` command, so you shouldn't have to worry about argument to `scan()`.

The functions `read.table` and `write.table` are designed to be used with formatted data sets. (They are just wrappers for the `scan()` command.) In this class we tend to keep things simple, so usually the `scan()` command is the easiest way to read in your data.

5.2 Managing Your Workspace

To see what variables are in your workspace, you can use either the `ls()` command or the `objects()` command.

```
> ls()
 [1] "Pex"          "dd"           "f1"           "f2"           "fun"
 [6] "histout"      "i"            "ipch"         "ix"           "iy"
[11] "k"            "labels"       "last.warning" "np"           "op"
[16] "ord"          "pc"           "pch"          "pu"           "px"
[21] "r"            "rx"           "ry"           "theta"        "ticks"
[26] "u"            "x"            "x.mean"       "x.sd"         "x.stat"
[31] "xx"           "y"            "z"
```

Use the `all=T` option to `ls()` or `objects()` to see all objects include those whose names begin with a period '.'. (These are usually internal R objects.)

Try not to let your workspace get too big, however, as this will slow R down. To delete a variable from your workspace, use the `rm()` command:

```
> objects()
[1] x
> rm(x)
> objects()
character(0)
```

To remove multiple variables from your workspace, use the `list` option to `rm`. To remove *all* variables from you workspace, you can use `list=ls()`:

```
> rm(list=ls(all=T))
```

(The `all=T` option is present to make sure we match files beginning with a period.) Use this with care! Once your data is gone, you will not be able to restore it.

5.3 Files and Directories

To see what files are in the current directory, use `dir()`. You can use the `pattern` argument to `dir` to match only certain files.

```
> dir(pattern="*.R")
[1] "Rcommands.R" "regress1.R"  "regress2.R"  "relhist.R"
```

5.4 Options

5.4.1 Changing the Number of Digits Printed

To view the current setting, use `getOption("digits")`. To change the setting, use `options(digits=nn)`, where `nn` is a number between 1 and 22.

5.5 Printing

From the command line you can print the value of a variable or object simply by typing its name and hitting <ENTER>. Inside of a script, you must explicitly call the `print()` function or use the `echo=TRUE` option to `source`.

For formatted printing à la C's `printf()` statment, try `formatC()`.

Chapter 6

Regression and ANOVA

6.1 Simple Linear Regression with `lm`

Suppose we have a data set of the form (X_i, Y_i), where the X_i's are the predictor variables (i.e., the independent variables) and the Y_i's are the response variables (i.e., the dependent variables). We hypothesize a linear relationship between X_i and Y_i of the form $Y_i = a + bX_i$ for some unknown parameters a and b. We would like to "fit" this linear model to the data. (Here "fitting" the model means calculating the parameters a and b of the model.)

The hypothesized model is more accurately

$$Y_i = a + bX_i + \varepsilon_i, \tag{6.1}$$

where X_i is the independent variable, Y_i is the dependent (random) variable, a is the *intercept*, b is the *slope*, and ε_i is a random variable with a Normal(0, σ) distribution for some unknown σ. Linear regression involves computing the "line of best fit" to the data, where "best fit" means minimal residual sum of squares error

$$\sum_{i=1}^{n} |Y_i - (a + bX_i)|^2 \tag{6.2}$$

with respect to a and b.

A linear model is defined in R using the $\sim$ (tilde) operator. Thus

```
Y ~ X
```

tells R that Y is an unknown linear function of X. (Here X and Y are vectors.)[1]

This only specifies the model, however; it does not compute anything. The actual computation of the least-squares estimates of a and b is done with the `lm` command:

```
lmobj <- lm(Y ~ X)
```

`lm` returns an object (simply called "an `lm`-object"). You should store its value for future use.

Here is a simple (albeit contrived) example of how to use `lm`.

```
R> X <- sort(runif(100, -10, 10))
R> Y <- -5 + 1.6 * X + rnorm(100, 0, 3)
```

[1] You should read the $\sim$ operator as "is modeled by". Technically, the $\sim$ operator asserts that the left side of the $\sim$ is an unknown linear combination of the terms on the right side of the $\sim$.

```
R> lmobj <- lm(Y ~ X)
```

The object returned by `lm` (which we have stored in the variable `lmobj`) contains a great deal of useful information. To see a summary of this information, use the `summary` command.

```
R> summary(lmobj)

Call:
lm(formula = Y ~ X)

Residuals:
    Min      1Q  Median      3Q     Max
-5.7414 -1.8281 -0.1790  2.0010  6.0346

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.92879    0.26956  -18.29   <2e-16 ***
X            1.56094    0.04911   31.78   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.666 on 98 degrees of freedom
Multiple R-Squared: 0.9116,        Adjusted R-squared: 0.9107
F-statistic:  1010 on 1 and 98 DF,  p-value: < 2.2e-16
```

Let us walk through the output of the `summary` command line by line.

- The "Call:" line of the `summary` output reminds us how we called `lm`.
- The "Residuals:" line gives us some summary statistics of the residuals of the model.
- The "Coefficients:" section gives us estimates of a and b plus their associated statistics. Take note of R's notation: a is called "(Intercept)", and b is called "X". Thus, instead of referring to the coefficient, R refers to the model term. This is true of most statistical packages, so be aware!

  The column labeled "Estimate" contains the computed values of a and b. The "Std. Error" column contains the standard errors of a and b, respectively. "t value" is the value of the test statistic for testing the hypotheses

  $$H_0 : a = 0 \qquad H_0 : b = 0 \tag{6.3}$$
  $$H_A : a \neq 0 \qquad H_A : b \neq 0. \tag{6.4}$$

  Finally, "Pr(>|t|)" contains the p-value associated with the test statistic.
- The "Signif. codes:" line is merely a legend for the p-values. The number of asterisks (*) tells you how significant the p-value is.
- The "Residual standard error:" line gives the estimate of σ, the standard deviation of the residuals.
- The "Multiple R-Squared:" line tells us the value of R^2, the coefficient of determination.
- The "Adjusted R-Squared:" line tells us the value of the adjusted R^2. (You should consult your statistics text if you do not know the difference between the two statistics.)

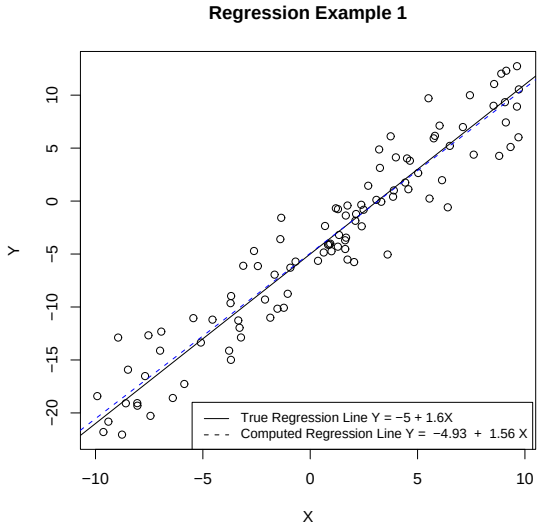

Figure 6.1: Scatterplot of Regression Example. The Fitted Line is shown in blue.

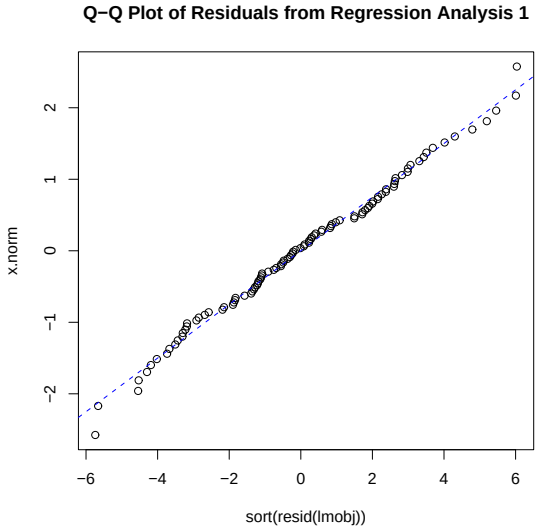

Figure 6.2: Q-Q Plot of Regression Example.

- The "F-statistic:" line gives us the value of the F-statistic associated with the model, and the "p-value:" line gives the p-value of the F-statistic.

Often we do not need so much information, so there are other regression commands you should know. The `coefficients` command, which may be abbreviated as `coef`, returns the computed coefficients a and b.

```
R> coef(lmobj)
```

```
(Intercept)           X
  -4.928787    1.560943
```

We can use `coef` with `abline` to add the fitted line to a scatterplot of the data (Figure 6.1).

```
R> plot(X, Y)
R> abline(-5, 1.6, lty = 1)
R> title("Regression Example 1")
R> abline(coef(lmobj), lty = 2, col = "blue")
R> legtxt1 <- "True Regression Line Y = -5 + 1.6X"
R> legtxt2 <- paste("Y = ", format(coef(lmobj)[1], digits = 3))
R> legtxt2 <- paste(legtxt2, " + ", format(coef(lmobj)[2], digits = 3),
+       "X")
R> legtxt2 <- paste("Computed Regression Line", legtxt2)
R> legend(-5.5, -19, c(legtxt1, legtxt2), lty = c(1, 2), cex = 0.85)
```

The `residuals` command, which you can shorten to `resid`, gives us the residuals of the model. We could use this, for instance, to create a Q-Q plot of the residuals (Figure 6.2).

```
R> q.s <- ((1:length(resid(lmobj))) - 0.5)/length(resid(lmobj))
R> x.norm <- qnorm(q.s)
```

```
R> plot(sort(resid(lmobj)), x.norm)
R> abline(lsfit(sort(resid(lmobj)), x.norm), lty = 2, col = "blue")
R> title("Q-Q Plot of Residuals from Regression Analysis 1")
```

Finally, we should mention the `anova` command. The `anova` command is for performing an Analysis of Variance on a model. In the context of a simple linear regression, `anova` will compute the sums of squares associated with the model and with the residuals.

```
R> anova(lmobj)

Analysis of Variance Table

Response: Y
          Df Sum Sq Mean Sq F value    Pr(>F)
X          1 7178.2  7178.2  1010.1 < 2.2e-16 ***
Residuals 98  696.4     7.1
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The table contains the sums of squares, means square errors, and F statistics associated with both the model and the residuals.

This concludes our discussion of simple linear regression in R. In the next section we will discuss multiple linear regression.

6.2 Multiple Linear Regression with `lm`

Now suppose we have several predictors $X_1, \ldots, X_k$. We again hypothesize a linear relationship between the response variable Y and the predictors of the form

$$Y_i = \beta_0 + \beta_1 X_{1i} + \cdots + \beta_k X_{ki} + \varepsilon_i. \tag{6.5}$$

This model can be fit in R as easily as the simple linear model. We once again define the linear model using the $\sim$ operator. For concreteness, let us suppose $k = 3$ and our variables are called $X1$, $X2$, and $X3$.

```
Y ~ X1 + X2 + X3
```

To fit the model (i.e., compute the coefficients β_1, β_2, and β_3), we simply use `lm` as before.

```
lmobj <- lm(Y ~ X1 + X2 + X3)
```

The "`+`" operator here should be read as "and", not as "plus".[2]

The statistics are computed using the same commands as in the simple linear case (`summary`, `anova`, `coef` and `resid`).

Visualization in the higher-dimensional cases is more complicated than in the simple linear case. The `pairs` command takes a matrix and produces all pairwise scatterplots of its columns (i.e., column 1 versus column 2, column 1 versus column 3, etc.)

More powerful visualization functionality can be found in the Trellis graphics packages; Trellis graphics, however, are beyond the scope of this document. See the books in the Appendix "Further Reading".

[2]Note that the order of the terms might be important: $X1+X2$ evaluates the effect of $X1$ by itself and $X2$ given $X1$, while $X2+X1$ evaluates the effect of $X2$ by itself and $X1$ given $X2$. This matters to some of the R modeling facilities, most notably `anova`. The reasons for this beyond this document, so we will say no more about this.

6.3 ANOVA

ANOVA, or ANalysis Of VAriance, can also be performed in R. Since ANOVA is really just a fancy form of linear regression, you might guess that the $\sim$ operator and `lm` are involved. Well, you are almost correct. The $\sim$ operator is once again used to express a model dependence. Instead of `lm`, though, we now use the `aov` command.

The `aov` function fits an analysis of variance model using the specified formula. How does it do this? It calls the `lm` function! Like we stated in the previous paragraph, ANOVA essentially a regression model (from a theoretical standpoint). You could fit the same model using `lm` alone. The `aov` function, however, is easier to use for certain types of ANOVA's. The `summary` function for `aov` is also specialized to produce a traditional ANOVA table, not a regression table.

The syntax of `aov` is identical to that of `lm`. Furthermore, many of the functions you used with `lm` can be applied here.

Let us now take a look at an example. We will analyze an experiment to determine the effect of Vitamin C on tooth growth in guinea pigs. This data set ("ToothGrowth") is one of the example data sets included in R. You can read more about it using `help(ToothGrowth)`.

First, we load the data set using the `data` command.

```
R> data(ToothGrowth)
```

The data set is now available in R as the data frame `ToothGrowth`. Use the `names` command to see what variables are available in `ToothGrowth`.

```
R> names(ToothGrowth)

[1] "len"  "supp" "dose"
```

The help file for the data set explains what each variable means.

```
> help(ToothGrowth)
ToothGrowth               package:base               R Documentation

The Effect of Vitamin C on Tooth Growth in Guinea Pigs

Description:

     The response is the length of odontoblasts (teeth) in each of 10
     guinea pigs at each of three dose levels of Vitamin C (0.5, 1, and
     2 mg) with each of two delivery methods (orange juice or ascorbic
     acid).

...
```

We treat this as a 3×2 factorial treatment design. The interaction effects model is given by

$$\text{len}_{ijk} = \mu + \text{dose}_i + \text{supp}_j + (\text{dose} * \text{supp})_{ij} + \varepsilon_{ijk} \tag{6.6}$$

, where

- len_{ijk} is the k-th observation in the (i, j) "cell";
- μ is an overall mean;
- dose_i is the effect of the i-th level of the treatment *dose*;

- supp_j is the effect of the j-th level of the treatment *supp*;
- $(\text{dose} * \text{supp})_{ij}$ is the interaction effect between the i-th level of *dose* and the j-th level of *supp*; and
- ε_{ijk} is the error term in the (i, j) "cell".

We assume the errors ε_{ijk} come from a $N(0, \sigma)$ distribution for some unknown standard deviation σ.

To fit this model in R, we will need to specify the model using $\sim$. We already know how to specify the dose_i and supp_j terms, but how to we specify the cross-term $(\text{dose} * \text{supp})_{ij}$? R has a special operator for this, the `:` (colon) operator. The expression `A:B` in R specifies a cross term between treatments *A* and *B*.

Thus, our model in R would be

```
len ~ dose + supp + dose:supp
```

This construction occurs so frequently that R has an abbreviation for it: the `*` (asterisk) operator. Thus

```
len ~ dose*supp
```

can also be used to define our model in R.

We now call `aov` on our model.

```
R> aovobj <- aov(len ~ dose * supp, data = ToothGrowth)
```

(The `data=ToothGrowth` argument is necessary to tell `aov` where to find the variables.)

Calling `summary` now gives us a traditional ANOVA table.

```
R> summary(aovobj)

             Df  Sum Sq Mean Sq  F value    Pr(>F)
dose          1 2224.30 2224.30 133.4151 < 2.2e-16 ***
supp          1  205.35  205.35  12.3170 0.0008936 ***
dose:supp     1   88.92   88.92   5.3335 0.0246314 *
Residuals    56  933.63   16.67
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The `anova` command gives the same table.

We can also make a Q-Q plot of the residuals as before. (Figure 6.3).

```
R> q.s <- ((1:length(resid(aovobj))) - 0.5)/length(resid(aovobj))
R> x.norm <- qnorm(q.s)
R> plot(sort(resid(aovobj)), x.norm)
R> abline(lsfit(sort(resid(aovobj)), x.norm), lty = 2, col = "blue")
R> title("Q-Q Plot of Residuals from ANOVA")
```

6.4 Other Models

We have only seen a small fraction of the modeling and analysis capabilities of R. R is capable of fitting and analyzing very complex experiments. If you would like to learn more about statistical analysis with R, consult the references listed in the appendix "Further Reading".

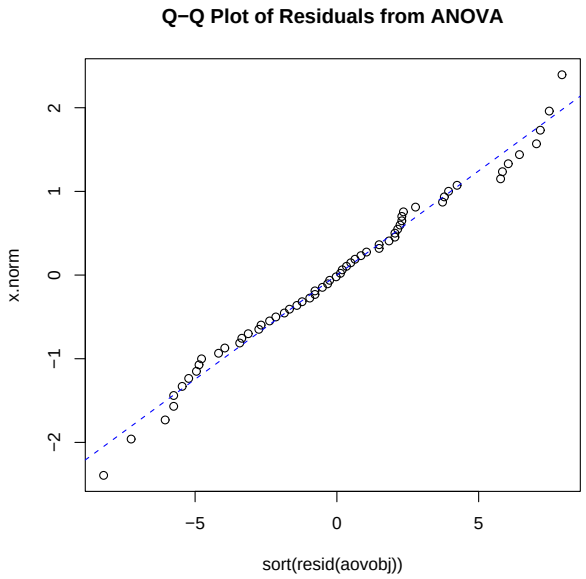

Figure 6.3: Q-Q Plot from ANOVA.

Appendix A

Table of Common Distributions in R

For each distribution in R, there are four commands: a **d-command**, a **p-command**, a **q-command**, and an **r-command**.

1. The d-commands give the probability mass function (for discrete distributions) or the probability density function (for continuous distributions). For example, `dbinom(3,5,0.5)` gives the probability of getting exactly 3 heads on 5 flips of a fair coin, while `dnorm(3,0,1)` evaluates the density function of the standard normal distribution at `x=3`.

2. The p-commands give the cumulative distribution function $F(x) = P(X \leq x)$. For instance, `pexp(4,2)` gives $P(X \leq 4)$ when X has an exponential(rate=2) distribution. Note that if you want the complementary probability $P(X \leq x)$, you can either use `1 - <pcommand>` or use the `lower.tail=F` argument.

3. The q-commands provide quantiles; that is, given a probability p, the quantile commands return the value x such that $P(X \leq x) = p$.

4. The r-commands produce random numbers from the given distribution.

Table A.1 provides the R commands for the distributions you will see in this class. For the exact syntax of each command, use the `help( )` function.

Distribution	d-command	p-command	q-command	r-command
Normal	dnorm	pnorm	qnorm	rnorm
Uniform	dunif	punif	qunif	runif
Exponential	dexp	pexp	qexp	rexp
Binomial	dbinom	pbinom	qbinom	rbinom
Poisson	dpois	ppois	qpois	rpois
Gamma	dgamma	pgamma	qgamma	rgamma
Weibull	dweibull	pweibull	qweibull	rweibull
Student's T	dt	pt	qt	rt
Snedecor's F	df	pf	qf	rf
Chi-Squared	dchisq	pchisq	qchisq	rchisq

Table A.1: R Commands for Common Distributions

Further Reading

[1] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. CRC Press, 1988.

[2] Peter Dalgaard. *Introductory Statistics in R*. Springer-Verlag, 2002.

[3] Andreas Krause and Melvin Olson. *The Basics of S and S-Plus*. Springer-Verlag, third edition, 2003.

[4] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer-Verlag, fourth edition, 2002.

[5] W. N. Venables, D. M. Smith, and the R Development Core Team. *An Introduction to R*. Network Theory Ltd., 2002.

Index