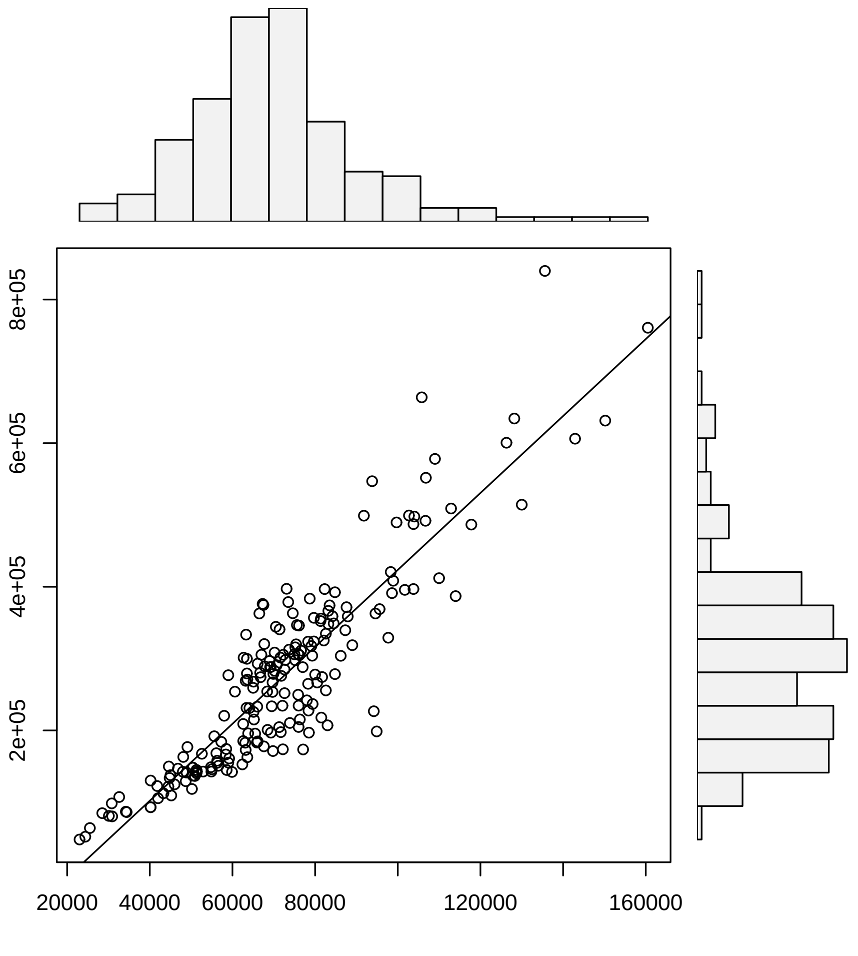
simpleR – *Using R for Introductory Statistics*

John Verzani



Preface

These notes are an introduction to using the statistical software package `R` for an introductory statistics course. They are meant to accompany an introductory statistics book such as Kitchens *"Exploring Statistics"*. The goals are not to show all the features of `R`, or to replace a standard textbook, but rather to be used with a textbook to illustrate the features of `R` that can be learned in a one-semester, introductory statistics course.

These notes were written to take advantage of `R` version 1.5.0 or later. For pedagogical reasons the equals sign, `=`, is used as an assignment operator and not the traditional arrow combination `<-`. This was added to `R` in version 1.4.0. If only an older version is available the reader will have to make the minor adjustment.

There are several references to data and functions in this text that need to be installed prior to their use. To install the data is easy, but the instructions vary depending on your system. For Windows users, you need to download the "zip" file , and then install from the "packages" menu. In UNIX, one use the command `R CMD INSTALL packagename.tar.gz`. Some of the datasets are borrowed from other authors notably Kitchens. Credit is given in the help files for the datasets. This material is available as an `R` package from:

`http://www.math.csi.cuny.edu/Statistics/R/simpleR/Simple_0.4.zip` for Windows users.
`http://www.math.csi.cuny.edu/Statistics/R/simpleR/Simple_0.4.tar.gz` for UNIX users.

If necessary, the file can sent in an email. As well, the individual data sets can be found online in the directory

`http://www.math.csi.cuny.edu/Statistics/R/simpleR/Simple`.

This is version 0.4 of these notes and were last generated on August 22, 2002. Before printing these notes, you should check for the most recent version available from

the CSI Math department (`http://www.math.csi.cuny.edu/Statistics/R/simpleR`).

Copyright © John Verzani (`verzani@math.csi.cuny.edu`), 2001-2. All rights reserved.

Contents

Introduction **1**
What is `R` 1
A note on notation 2

Data **2**
Starting `R` 2
Entering data with `c` 3
Data is a `vector` 4
Problems 9

Section 1: Introduction

What is R

These notes describe how to use R while learning introductory statistics. The purpose is to allow this fine software to be used in ”lower-level” courses where often MINITAB, SPSS, Excel, etc. are used. It is expected that the reader has had at least a pre-calculus course. It is the hope, that students shown how to use R at this early level will better understand the statistical issues and will ultimately benefit from the more sophisticated program despite its steeper “learning curve”.

The benefits of R for an introductory student are

- R is free. R is open-source and runs on UNIX, Windows and Macintosh.
- R has an excellent built-in help system.
- R has excellent graphing capabilities.
- Students can easily migrate to the commercially supported S-Plus program if commercial software is desired.
- R’s language has a powerful, easy to learn syntax with many built-in statistical functions.
- The language is easy to extend with user-written functions.
- R is a computer programming language. For programmers it will feel more familiar than others and for new computer users, the next leap to programming will not be so large.

What is R lacking compared to other software solutions?

- It has a limited graphical interface (S-Plus has a good one). This means, it can be harder to learn at the outset.
- There is no commercial support. (Although one can argue the international mailing list is even better)
- The command language is a programming language so students must learn to appreciate syntax issues etc.

R is an open-source (GPL) statistical environment modeled after S and S-Plus (`http://www.insightful.` The S language was developed in the late 1980s at AT&T labs. The R project was started by Robert Gentleman and Ross Ihaka of the Statistics Department of the University of Auckland in 1995. It has quickly gained a widespread audience. It is currently maintained by the R core-development team, a hard-working, international team of *volunteer* developers. The R project web page

`http://www.r-project.org`

is the main site for information on `R`. At this site are directions for obtaining the software, accompanying packages and other sources of documentation.

A note on notation

A few typographical conventions are used in these notes. These include different fonts for `urls`, `R commands`, dataset names and different typesetting for

```
longer sequences of R commands.
```

and for

```
Data sets.
```

Section 2: Data

Statistics is the study of data. After learning how to start `R`, the first thing we need to be able to do is learn how to enter data into `R` and how to manipulate the data once there.

Starting R

`R` is most easily used in an interactive manner. You ask it a question and `R` gives you an answer. Questions are asked and answered on the command line. To start up `R`'s command line you can do the following: in Windows find the `R` icon and double click, on Unix, from the command line type `R`. Other operating systems may have different ways. Once `R` is started, you should be greeted with a command similar to

```
R : Copyright 2001, The R Development Core Team
Version 1.4.0  (2001-12-19)

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 > is called the prompt. In what follows below it is not typed, but is used to indicate where you are to type if you follow the examples. If a command is too long to fit on a line, a + is used for the continuation prompt.

Entering data with c

The most useful R command for quickly entering in small data sets is the c function. This function combines, or concatenates terms together. As an example, suppose we have the following count of the number of typos per page of these notes:

```
2 3 0 3 1 0 0 1
```

To enter this into an R session we do so with

```
> typos = c(2,3,0,3,1,0,0,1)
> typos
[1] 2 3 0 3 1 0 0 1
```

Notice a few things

- We assigned the values to a variable called typos
- The assignment operator is a =. This is valid as of R version 1.4.0. Previously it was (and still can be) a <-. Both will be used, although, you should learn one and stick with it.
- The value of the typos doesn't automatically print out. It does when we type just the name though as the last input line indicates
- The value of typos is prefaced with a funny looking [1]. This indicates that the value is a vector. More on that later.

Typing less

For many implementations of R you can save yourself a lot of typing if you learn that the arrow keys can be used to retrieve your previous commands. In particular, each command is stored in a history and the up arrow will traverse backwards along this history and the down arrow forwards. The left and right arrow keys will work as expected. This combined with a mouse can make it quite easy to do simple editing of your previous commands.

Applying a function

R comes with many built in functions that one can apply to data such as typos. One of them is the mean function for finding the mean or average of the data. To use it is easy

```
> mean(typos)
[1] 1.25
```

As well, we could call the median, or var to find the median or sample variance. The syntax is the same – the function name followed by parentheses to contain the argument(s):

```
> median(typos)
[1] 1
> var(typos)
[1] 1.642857
```

Data is a `vector`

The data is stored in `R` as a `vector`. This means simply that it keeps track of the order that the data is entered in. In particular there is a first element, a second element up to a last element. This is a good thing for several reasons:

- Our simple data vector `typos` has a natural order – page 1, page 2 etc. We wouldn't want to mix these up.
- We would like to be able to make changes to the data item by item instead of having to enter in the entire data set again.
- Vectors are also a mathematical object. There are natural extensions of mathematical concepts such as addition and multiplication that make it easy to work with data when they are vectors.

Let's see how these apply to our typos example. First, suppose these are the typos for the first draft of section 1 of these notes. We might want to keep track of our various drafts as the typos change. This could be done by the following:

```
> typos.draft1 = c(2,3,0,3,1,0,0,1)
> typos.draft2 = c(0,3,0,3,1,0,0,1)
```

That is, the two typos on the first page were fixed. Notice the two different variable names. Unlike many other languages, the period is only used as punctuation. You can't use an _ (underscore) to punctuate names as you might in other programming languages so it is quite useful. [1]

Now, you might say, that is a lot of work to type in the data a second time. Can't I just tell `R` to change the first page? The answer of course is "yes". Here is how

```
> typos.draft1 = c(2,3,0,3,1,0,0,1)
> typos.draft2 = typos.draft1  # make a copy
> typos.draft2[1] = 0          # assign the first page 0 typos
```

Now notice a few things. First, the comment character, `#`, is used to make comments. Basically anything after the comment character is ignored (by `R`, hopefully not the reader). More importantly, the assignment to the first entry in the vector `typos.draft2` is done by referencing the first entry in the vector. This is done with square brackets `[]`. It is important to keep this in mind: parentheses `()` are for functions, and square brackets `[]` are for vectors (and later arrays and lists). In particular, we have the following values currently in `typos.draft2`

```
> typos.draft2                  # print out the value
[1] 0 3 0 3 1 0 0 1
> typos.draft2[2]               # print 2nd pages' value
```

[1]The underscore was originally used as assignment so a name such as `The_Data` would actually assign the value of `Data` to the variable `The`. The underscore is being phased out and the equals sign is being phased in.

```
[1] 3
> typos.draft2[4]                # 4th page
[1] 3
> typos.draft2[-4]               # all but the 4th page
[1] 0 3 0 1 0 0 1
> typos.draft2[c(1,2,3)]         # fancy, print 1st, 2nd and 3rd.
[1] 0 3 0
```

Notice negative indices give everything except these indices. The last example is very important. You can take more than one value at a time by using another vector of index numbers. This is called `slicing`.

Okay, we need to work these notes into shape, let's find the real bad pages. By inspection, we can notice that pages 2 and 4 are a problem. Can we do this with `R` in a more systematic manner?

```
> max(typos.draft2)              # what are worst pages?
[1] 3                            # 3 typos per page
> typos.draft2 == 3              # Where are they?
[1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
```

Notice, the usage of double equals signs (`==`). This tests all the values of `typos.draft2` to see if they are equal to 3. The 2nd and 4th answer yes (`TRUE`) the others no.

Think of this as asking `R` a question. Is the value equal to 3? `R`/ answers all at once with a long vector of TRUE's and FALSE's.

Now the question is – how can we get the indices (pages) corresponding to the `TRUE` values? Let's rephrase, *which* indices have 3 typos? If you guessed that the command `which` will work, you are on your way to `R` mastery:

```
> which(typos.draft2 == 3)
[1] 2 4
```

Now, what if you didn't think of the command `which`? You are not out of luck – but you will need to work harder. The basic idea is to create a new vector `1 2 3 ...` keeping track of the page numbers, and then slicing off just the ones for which `typos.draft2==3`:

```
> n = length(typos.draft2)       # how many pages
> pages = 1:n                    # how we get the page numbers
> pages                          # pages is simply 1 to number of pages
[1] 1 2 3 4 5 6 7 8
> pages[typos.draft2 == 3]       # logical extraction. Very useful
[1] 2 4
```

To create the vector `1 2 3 ...` we used the simple `:` colon operator. We could have typed this in, but this is a useful thing to know. The command `a:b` is simply `a, a+1, a+2, ..., b` if `a,b` are integers and intuitively defined if not. A more general `R` function is `seq()` which is a bit more typing. Try `?seq` to see it's options. To produce the above try `seq(a,b,1)`.

The use of extracting elements of a vector using another vector of the same size which is comprised of `TRUE`s and `FALSE`s is referred to as `extraction by a logical vector`. Notice this is different from extracting by page numbers by slicing as we did before. Knowing how to use slicing and logical vectors gives you the ability to easily access your data as you desire.

Of course, we could have done all the above at once with this command (but why?)

```
> (1:length(typos.draft2))[typos.draft2 == max(typos.draft2)]
[1] 2 4
```

This looks awful and is prone to typos and confusion, but does illustrate how things can be combined into short powerful statements. This is an important point. To appreciate the use of R you need to understand how one *composes* the output of one function or operation with the input of another. In mathematics we call this composition.

Finally, we might want to know how many typos we have, or how many pages still have typos to fix or what the difference is between drafts? These can all be answered with mathematical functions. For these three questions we have

```
> sum(typos.draft2)              # How many typos?
[1] 8
> sum(typos.draft2>0)            # How many pages with typos?
[1] 4
> typos.draft1 - typos.draft2    # difference between the two
[1] 2 0 0 0 0 0 0 0
```

Example: Keeping track of a stock; adding to the data

Suppose the daily closing price of your favorite stock for two weeks is

```
45,43,46,48,51,46,50,47,46,45
```

We can again keep track of this with R using a vector:

```
> x = c(45,43,46,48,51,46,50,47,46,45)
> mean(x)                        # the mean
[1] 46.7
> median(x)                      # the median
[1] 46
> max(x)                         # the maximum or largest value
[1] 51
> min(x)                         # the minimum value
[1] 43
```

This illustrates that many interesting functions can be found easily. Let's see how we can do some others. First, lets add the next two weeks worth of data to x. This was

```
48,49,51,50,49,41,40,38,35,40
```

We can add this several ways.

```
> x = c(x,48,49,51,50,49)         # append values to x
> length(x)                       # how long is x now (it was 10)
[1] 15
> x[16] = 41                      # add to a specified index
> x[17:20] = c(40,38,35,40)       # add to many specified indices
```

Notice, we did three different things to add to a vector. All are useful, so lets explain. First we used the `c` (combine) operator to combine the previous value of `x` with the next week's numbers. Then we assigned directly to the 16th index. At the time of the assignment, `x` had only 15 indices, this automatically created another one. Finally, we assigned to a slice of indices. This latter make some things very simple to do.

R Basics: Graphical Data Entry Interfaces

There are some other ways to edit data that use a spreadsheet interface. These may be preferable to some students. Here are examples with annotations

```
> data.entry(x)                 # Pops up spreadsheet to edit data
> x = de(x)                     # same only, doesn't save changes
> x = edit(x)                   # uses editor to edit x.
```

All are easy to use. The main confusion is that the variable `x` needs to be defined previously. For example

```
> data.entry(x)                  # fails. x not defined
Error in de(..., Modes = Modes, Names = Names) :
    Object "x" not found
> data.entry(x=c(NA))            # works, x is defined as we go.
```

Other data entry methods are discussed in the appendix on entering data.

Before we leave this example, lets see how we can do some other functions of the data. Here are a few examples.

The moving average simply means to average over some previous number of days. Suppose we want the 5 day moving average (50-day or 100-day is more often used). Here is one way to do so. We can do this for days 5 through 20 as the other days don't have enough data.

```
>  day = 5;
> mean(x[day:(day+4)])
[1] 48
```

The trick is the slice takes out days 5,6,7,8,9

```
> day:(day+4)
[1] 5 6 7 8 9
```

and the mean takes just those values of `x`.

What is the maximum value of the stock? This is easy to answer with `max(x)`. However, you may be interested in a running maximum or the largest value to date. This too is easy – if you know that R had a built-in function to handle this. It is called `cummax` which will take the cumulative maximum. Here is the result for our 4 weeks worth of data along with the similar `cummin`:

```
> cummax(x)                      # running maximum
 [1] 45 45 46 48 51 51 51 51 51 51 51 51 51 51 51 51 51 51 51 51
> cummin(x)                      # running minimum
 [1] 45 43 43 43 43 43 43 43 43 43 43 43 43 43 43 41 40 38 35 35
```

Example: Working with mathematics

R makes it easy to translate mathematics in a natural way once your data is read in. For example, suppose the yearly number of whales beached in Texas during the period 1990 to 1999 is

```
74 122 235 111 292 111 211 133 156 79
```

What is the mean, the variance, the standard deviation? Again, R makes these easy to answer:

```
> whale = c(74, 122, 235, 111, 292, 111, 211, 133, 156, 79)
> mean(whale)
[1] 152.4
> var(whale)
[1] 5113.378
> std(whale)
Error: couldn't find function "std"
> sqrt(var(whale))
[1] 71.50789
> sqrt( sum( (whale - mean(whale))^2 /(length(whale)-1)))
[1] 71.50789
```

Well, almost! First, one needs to remember the names of the functions. In this case `mean` is easy to guess, `var` is kind of obvious but less so, `std` is also kind of obvious, but guess what? It isn't there! So some other things were tried. First, we remember that the standard deviation is the square of the variance. Finally, the last line illustrates that R can almost exactly mimic the mathematical formula for the standard deviation:

$$\mathsf{SD}(X) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2}.$$

Notice the sum is now `sum`, $\bar{X}$ is `mean(whale)` and `length(x)` is used instead of n.

Of course, it might be nice to have this available as a built-in function. Since this example is so easy, lets see how it is done:

```
> std = function(x) sqrt(var(x))
> std(whale)
[1] 71.50789
```

The ease of defining your own functions is a very appealing feature of R we will return to.

Finally, if we had thought a little harder we might have found the actual built-in `sd()` command. Which gives

```
> sd(whale)
[1] 71.50789
```

R Basics: Accessing Data

There are several ways to extract data from a vector. Here is a summary using both slicing and extraction by a logical vector. Suppose `x` is the data vector, for example `x=1:10`.

how many elements?	`length(x)`
ith element	`x[2]` ($i = 2$)
all *but* ith element	`x[-2]` ($i = 2$)
first k elements	`x[1:5]` ($k = 5$)
last k elements	`x[(length(x)-5):length(x)]` ($k = 5$)
specific elements.	`x[c(1,3,5)]` (First, 3rd and 5th)
all greater than some value	`x[x>3]` (the value is 3)
bigger than or less than some values	`x[ x< -2 | x > 2]`
which indices are largest	`which(x == max(x))`

Problems

2.1 Suppose you keep track of your mileage each time you fill up. At your last 6 fill-ups the mileage was

```
65311 65624 65908 66219 66499 66821 67145 67447
```

Enter these numbers into `R`. Use the function `diff` on the data. What does it give?

```
> miles = c(65311, 65624, 65908, 66219, 66499, 66821, 67145, 67447)
> x = diff(miles)
```

You should see the number of miles between fill-ups. Use the `max` to find the maximum number of miles between fill-ups, the `mean` function to find the average number of miles and the `min` to get the minimum number of miles.

2.2 Suppose you track your commute times for two weeks (10 days) and you find the following times in minutes

```
17 16 20 24 22 15 21 15 17 22
```

Enter this into `R`. Use the function `max` to find the longest commute time, the function `mean` to find the average and the function `min` to find the minimum.

Oops, the 24 was a mistake. It should have been 18. How can you fix this? Do so, and then find the new average.

How many times was your commute 20 minutes or more? To answer this one can try (if you called your numbers `commutes`)

```
> sum( commutes >= 20)
```

What do you get? What percent of your commutes are less than 17 minutes? How can you answer this with `R`?

2.3 Your cell phone bill varies from month to month. Suppose your year has the following monthly amounts

```
46 33 39 37 46 30 48 32 49 35 30 48
```

Enter this data into a variable called `bill`. Use the `sum` command to find the amount you spent this year on the cell phone. What is the smallest amount you spent in a month? What is the largest? How many months was the amount greater than $40? What percentage was this?

2.4 You want to buy a used car and find that over 3 months of watching the classifieds you see the following prices (suppose the cars are all similar)

```
9000  9500  9400  9400 10000  9500 10300 10200
```

Use `R` to find the average value and compare it to Edmund's (`http://www.edmunds.com`) estimate of $9500. Use `R` to find the minimum value and the maximum value. Which price would you like to pay?

2.5 Try to guess the results of these `R` commands. Remember, the way to access entries in a vector is with `[]`. Suppose we assume

```
> x = c(1,3,5,7,9)
> y = c(2,3,5,7,11,13)
```

1. `x+1`
2. `y*2`
3. `length(x)` and `length(y)`
4. `x + y`
5. `sum(x>5)` and `sum(x[x>5])`
6. `sum(x>5 | x< 3) # read | as 'or', & and 'and'`
7. `y[3]`
8. `y[-3]`
9. `y[x]` (What is `NA`?)
10. `y[y>=7]`

2.6 Let the data `x` be given by

```
> x = c(1, 8, 2, 6, 3, 8, 5, 5, 5, 5)
```

Use `R` to compute the following functions. Note, we use X_1 to denote the first element of `x` (which is 0) etc.

1. $(X_1 + X_2 + \cdots + X_{10})/10$ (use `sum`)
2. Find $\log_{10}(X_i)$ for each i. (Use the `log` function which by default is base e)
3. Find $(X_i - 4.4)/2.875$ for each i. (Do it all at once)
4. Find the difference between the largest and smallest values of `x`. (This is the range. You can use `max` and `min` or guess a built in command.)