

MICHAEL CLARK
CENTER FOR SOCIAL RESEARCH
UNIVERSITY OF NOTRE DAME

INTRODUCTION TO R

A FIRST COURSE IN R

Contents

<i>Goals</i>	3
<i>What You Will Need</i>	3
<i>What is R?</i>	4
<i>Getting Started</i>	5
<i>Installation</i>	5
<i>Layout</i>	6
<i>Console</i>	6
<i>Script Editor & Related Programming Functionality</i>	6
<i>Output</i>	7
<i>Menus</i>	7
<i>Working with the Language</i>	8
<i>Basics</i>	8
<i>Coding Practices</i>	9
<i>Functions</i>	9
<i>Conditionals, Control Flow, Operators</i>	9
<i>Loops & Alternative Approaches</i>	10
<i>Conditional Statements</i>	10
<i>Importing & Working with Data</i>	10
<i>Creation of Data</i>	11
<i>Importing Data</i>	12
<i>Working with the Data</i>	12
<i>Indexing</i>	12
<i>Data Sets & Subsets</i>	14
<i>Merging and Reshaping</i>	14
<i>Miscellaneous</i>	15

Initial Data Analysis & Graphics 16*Initial Data Analysis: Numeric* 16*Initial Data Analysis: Graphics* 18*The Basic Modeling Approach* 18*Visualization of Information* 21*Examples* 21*Enhancing Standard Graphs* 22*Beyond R* 24*Getting Help* 24*Basics* 24*A Plan of Attack for Diving In* 25*Some Issues* 26*Summary* 26*Why R?* 27*Who is R for?* 27*R as a solution* 27*References* 28*General Web* 28*Manuals* 28*Graphics* 28*Commercial Versions* 28*Miscellaneous* 28

Goals

The goal of this tutorial and associated short course is fairly straightforward- This handout draft is dated April 10, 2013. The code, which can be directly copied from this pdf into an R console, was last tested with R 2.15.1 and 2.15.2

I wish to introduce the program well enough for one to understand some basics and feel comfortable trying things on their own. In this light the focus will be on some typical applications (colored by a social science perspective) with simple examples, and furthermore to identify the ways people can obtain additional assistance on their own in order to eventually become self-sufficient. It is hoped that those taking it will obtain a good sense of how R is organized, what it has to offer above and beyond the package they might already be using, and have a good idea how to use it once they leave the course.

What You Will Need

You will need R and it helps to have something like Rstudio to do your programming. Some exposure to statistical science is assumed but only the minimum that would warrant your interest in R in the first place. Programming experience is useful but not really needed for this course.

What is R?

R is a dialect of the S language¹ ('Gnu-S') and in that sense it has a long past but a shorter history. R's specific development is now pushing near 20 years². From the horse's mouth:

"R is a language and environment for statistical computing and graphics" and from the manual, "The term *environment* is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software."

This will be the first thing to get used to- R is not used in a manner like many such as SPSS, Stata, etc. It is a programming environment within which statistical analysis and visualization is conducted, not merely a place to run canned routines that come with whatever you have purchased. This is not intended as a knock on other statistical software, many of which accomplish their goals quite well. Rather, the point is that it may require a different mindset in order to get used to it depending on the kind of exposure you've had to other programs.

The initial install of R and accompanying packages provides a fully functioning statistical environment in which one may conduct any number of typical and advanced analyses. You could already do most of what you would find in the other programs, but also have the flexibility to do much more. However, there are over 4300³ user contributed *packages* that provide a vast amount of enhanced functioning. If you come across a particular analysis, it is unlikely there wouldn't already be an R package devoted to it. Unfortunately, there is not much sense to be made of a list of all those packages just by looking at them, and I don't recommend that you install all of them. The [CRAN Task views](#) provides lists of primary packages in an organized fashion by subject matter, and is a good place to start exploring.

R is a true object-oriented programming language, much like others such as C++, Python etc. *Objects* are manipulated by *functions*, creating new objects which may then have more functions that can be applied to them. Objects can be just about anything: a single value, variable, datasets, lists of several types of objects etc. The object's *class* determines how a generic function (like `summary` and `plot`) will treat the object, which may be numeric, factor, data frame, matrix etc. It may be a little confusing at first, and it will take time to get used to, but in the end R can be much more efficient than other stat languages for many applications.

Typically there are often several ways to do the same thing depending on the objects and functions being used and the same function may do different things for different objects. One can see below how to cre-

¹ S was developed primarily by John Chambers at Bell Labs in the 70s.

² R was initially written by Robert Gentleman and Ross Ihaka 1993, and version 1.0.0 came out in 2000.

³ As of early 2013 and not counting other sources such as R-Forge and Bioconductor.

For example the `summary` function can work on single variables, whole data sets, or specific models. The `plot` function can be used for e.g. a simple scatterplot, or have specific capabilities depending on what type of object is supplied to it.

ate an object and three different ways to produce the same variable the console yourself.

```
UScapt = "D.C."
UScapt

## [1] "D.C."

myvar1 = c(1,2,3)
myvar2 = 1:3
myvar3 = seq(from = 1, to=3, by=1)
myvar1

## [1] 1 2 3

myvar2

## [1] 1 2 3

myvar3

## [1] 1 2 3
```

This code creates an object called UScapt that is just the string "D.C.". After that is demonstrated three different ways in which to produce an object that myvar* that is the series of numbers from 1 to 3.

Getting Started

Installation

Getting R up and running is very easy. Just head to the website⁴ and once done basking in the 1997 nostalgia of the front page, click the download link, choose a mirror⁵, click your operating system of choice (for us Windows), choose 'base', and voila, you are now able to download this wonderful statistical package. The installation process after that is painless though- R does not 'infect' your machine like a lot of other software does; you can even just delete the folder to uninstall it. Note also that it is free as in speech and beer⁶ and as such you actually own your copy, no internet connection is required or extra software that has to run to tell the parent company you aren't a thief. It is not a trial or crippled version.

It should be noted again that this base installation is fully functioning and out of the box it does more than standard statistical packages, but you'll want more packages for additional functionality, and I suggest adding them as needed⁷. When you upgrade R⁸ you'll be downloading and installing the program as before, after which you can copy your packages from the old installation library folder over to the new folder, and then update the packages from within the new installation once you start it up. If you find you use one or two packages very regularly you can alter the Rprofile.site file in your R/etc folder. It's just a text file so you can open it with something like Notepad and just

⁴ One can just google R and it may be the first return.

⁵ It doesn't matter where really, though download speeds will be different.

⁶ Freedom!

⁷ `install.packages(c("Rpack1", "Rpack2"))`

⁸ Always upgrade to the latest available. A new version is available every few months. Don't forget to update your packages. As one who uses R regularly, I do every week or so. If you use it infrequently, just get into the habit of updating every time you use it.

insert `library(myfavepack)` on a new line. Otherwise you will normally be calling packages at the point you need to use them, and this is generally preferable.

The *workspace* is your current R working environment and includes any and every user-defined object (vectors, matrices, functions, data frames, lists, even plots). In a given session one may save all objects created as .Rdata file, but note that this is not a data set like with other packages. When you load an .Rdata file you will have everything you worked on during the session it was saved in, and readily available with no need to rerun scripts. All objects from single values to lists of matrices, analysis results, graphical objects etc. is at the ready to be called up again.

Layout

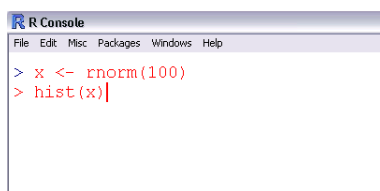
I'll spend a little time here talking about the basic R installation, but there is typically little need on your own to not work with an IDE of your choice to make programming easier. The layout for R when first started shows only the console but in fact you'll often have three windows open⁹. The console is where everything takes place, the script file for extended programming, and a graphics device. As I will note shortly though, an IDE such as Rstudio is a far more efficient means of using R.

⁹ You have the choice of whether you want a single document interface (SDI), which has separate windows to deal with, or multi-document interface (MDI), where all windows open within one single window (think of the Stata layout).

Console

The console is not too friendly to the R beginner (or anyone for that matter) and best reserved for examining the output of code that has been submitted to it.

One can use it as a command line interface similar to Stata's with single line input, but it's far too unforgiving and inflexible to spend much time there. I still use it for easy one-liners like the `hist()` or `summary()` functions, but otherwise do not do your programming there.



```
R Console
File Edit Misc Packages Windows Help
> x <- rnorm(100)
> hist(x)
```

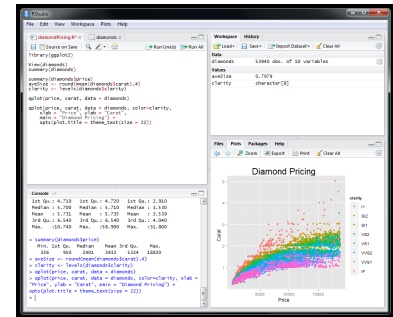
Script Editor & Related Programming Functionality

Rather than use the console, one can write their program in the script editor that comes with R¹⁰. This is just a plain text editor not unlike Notepad, but far easier than just using the console. While it gets the job done I suggest something with script highlighting and other fea-

¹⁰ File/New Script... Ctrl+R will send the current line or or selection to the console.

tures, and in that vein I would suggest **Rstudio**, an interactive development environment built specifically for R. It is highly functional, and quite simply it will make R a great deal easier to work with. You get syntax highlighting, more plot options, project management, better data views, and a host of other useful features. See the pic at the right.

One might also check out **Revolution Analytics**. Although this tool is geared toward the high-performance computing environment, it also provides a host of nifty features and is free to the academic community.



Output

Unlike some programs (e.g. SPSS), there is no designated output file/system with R. One can look at it such that everything created within the environment is "output". All created objects are able to be examined, manipulated, exported etc., graphics can be saved as one of many typical file extensions e.g. jpeg, bmp etc. Results of any analysis may be stored as an object, which may then be saved within the .Rdata file mentioned previously, or perhaps as a text file to be imported into another program (see e.g. **sink**). There are even ways to have the results put into a document and updated as the code changes (Sweave). Thus rather than have a separate output file full of tables, graphics etc., which often get notably large and much of which is disregarded after a time, one has a single file that provides the ability to call up anything one has ever done in a session very easily, to go along with other possibilities for capturing the output from analysis.

Menus

Let's just get something out of the way now. The R development team has no interest in providing a menu system of the sort you see with SPSS, Stata and others. They've had plenty of opportunity and if they wanted to they would have by now. For those that like menus, you'd be out of luck, except that the power of open source comes to the rescue.

There are a few packages here and there that come with a menu system specifically designed for the package, but for a general all-purpose one akin to what you see in those other packages, you might want to consider R-commander from John Fox (at right). This is a true menu system, has a notable variety of analyses, and has additional functionality via Rcmdr-specific add-ons and other customization. I think many coming to R from other statistical packages will find it useful for quickly importing data, obtaining descriptive statistics and plots, and familiarizing oneself with the language, but you'll know you are getting a very good feel for R as you rely on it less and less¹¹.

Note also that there are proprietary ways to provide a menu system along with R. S-plus is just like the other programs with menus and

```
load("myworkspace.Rdata")
```



¹¹ After installing the package you load the library just like any other and it will bring up the menus.

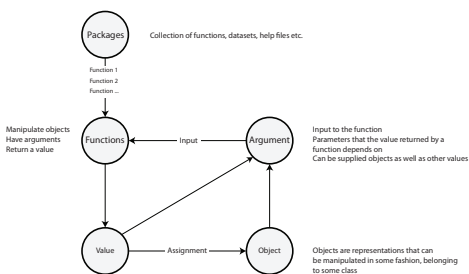
spreadsheet, and most of the R language works in that environment. SPSS, SAS and others have addons that allow one to perform R from within those programs. In short, if you like menus for at least some things, you have options.

Working with the Language

Basics

The language of R can prove to be quite a change for some, even if they have quite a bit of programming experience in other statistical packages. It is more like Python or C++ than it is like SPSS or SAS syntax. This is an advantage however, as it makes for a far more flexible approach to statistical programming.

Recall that the big idea within the R environment is to create objects of certain types and pass them to functions for further processing. Most of the time you will be doing just that- creating an object by means of some function, applying a function to the object to create yet another object, and so forth. As an example you may create a data object through use of the `data.frame` function, use `subset` on the data object to use only a portion of the data for analysis, use `summary` on the subsetted data object for summary statistics etc. Doing things in this manner allows you to go back to any object and change something up and process it further. The following graphic provides an overview of how one can think about dealing with objects in R.



Example. You import a dataset using the `read.csv` function, and this returns a value which is a data frame. However, unless you assign the value to an object, you'd have to repeat the `read.csv` process everytime you wanted to use the dataset. Assigning it to an object, one can now supply it to a function, which can then use elements of that data as values to other arguments.



When looking through texts or websites regarding R or even these notes, you will notice objects assigned with either `<=` or `=`. This used to be of some concern, and you are more than welcome to browse R-help archives for more information. For some users it is a pragmatic issue (one takes fewer keystrokes and they don't typically have to worry with others reading their code), but others find that `<=` makes code easier to read and it technically can be used more flexibly, not to mention it allows them to be snobby toward those who use the `=` sign. Type `?<=` at the console (with quotes) to find out some basic info. There is a bit of a difference but it is typically not an issue except perhaps in writing some functions in particular ways. Normal usage should not result in any mishap. I will flop between both styles so that you are used to seeing it either way.

Typing `ls()` or `objects()` with the empty parentheses will list all objects you've created.

Coding Practices

As it is a true programming language, there are ways to program in general that can be helpful in making code easier to read and interpret. Some coding practices have been suggested specific to R, such as Google's R [style guide](#) and this one from [Hadley Wickham](#). While some of those suggestions are very useful, one should not assume they are commandments from on high¹², nor have any studies been conducted on what would make the best R code. Furthermore, heavy commenting does a lot more for making code legible than any of the little details offered in those links. I tell people to assume they will finish the project only to pick it up again a year later, and that their code needs to be clear and obvious to that future version of them.

¹² Quite frankly, a couple of the suggestions would make coding less legible to me.

Functions

Writing your own user-defined function can enhance or make more efficient the work you do, though it won't always be easy. It's a very good idea to search for what you are attempting first as a relevant package may already be available¹³, and as the code is open source, you can take existing functions and change them to meet your needs very easily. But if you can't find it already available or you need a function specific to your data needs, R's programming language will surely allow you to do the trick and often very efficiently.

The following provides an example for a simple function to produce the mean of an input *x*.

```
mymean = function(x) {
  if (!is.numeric(x)) {
    stop("STOP! Does not compute.")
  }
  return(sum(x)/length(x))
}
var1 = 1:5
mymean(var1)

## [1] 3

mymean("saywhatnow")

## Error: STOP! Does not compute.
```

¹³ No sense reinventing the wheel, though if you're like me this won't stop you from trying.

This code creates a function that takes some input vector *x*, returns an error and message if *x* is not of the *numeric* class (the ! means 'not', so when the *is.numeric* function applied to *x* returns 'FALSE' the error will be returned), and returns the mean of whatever *x* is otherwise. I could have also put `if(is.numeric(x) == FALSE)`.

Conditionals, Control Flow, Operators

In general, conditionals (*if... then*), flow control (*for*, *while*), and operators (`!=`, `>=`) work much the same as they do in other statistical packages, though some specific functions are available to aid in these matters.

Loops & Alternative Approaches

Looping can allow iteration over indices of vectors, matrices, lists etc. and generally is a good way to efficiently do repetitive procedures. The following example will illustrate how one can do it so they can see parallels with other statistical programs' syntax, and then provide an alternative and better way to go about it.

```
temp = matrix(rnorm(100), ncol = 10)
means = vector("numeric", 10)
for (i in 1:ncol(temp)) {
  means[i] = mean(temp[, i])
}
means

## [1] -0.55837 0.23763 -0.48065 0.01919 -0.03982 0.23390 0.12697
## [8] 0.09914 0.16819 0.15295
```

The above works but is only shown for demonstration purposes, and mostly to demonstrate that you are typically better served using other approaches, as when it comes to iterative processes there are usually more efficient means of coding so in R than using explicit loops. For example it would have been more easy to simply type:

```
means <- colMeans(temp)
```

Note this is not simply using a particular function that does the same loop we just did internally, but performing a vectorized operation, i.e. operating on whole vectors rather than individual parts of a sequence. One can start by looking up the `apply` and related functions `tapply`, `lapply`, `sapply` etc.¹⁴ Using such functions can often make for clearer code and may be faster in many situations.

Conditional Statements

As an example, the `ifelse` function compares the elements of two objects according to some Boolean statement. It can return scalar or vector values for a true condition, and a different set of values for a false condition. In the following code, `x` is compared to `y` and if less, one gets cake, and for any other result, death.

```
x <- c(1, 3, 5)
y <- c(6, 4, 2)
z <- ifelse(x < y, "Cake", "Death")
z

## [1] "Cake" "Cake" "Death"
```

You also have the usual flow control functions available such as the standard `if`, `while`, `for` etc.

The code to the left first creates a 10 column matrix *temp* from 100 random draws of a $N(0,1)$ distribution. Next we create an empty vector that will be filled with column means. Then for each value *i* of 1 through the number of columns in *temp*, we calculate the mean of column *i* of the *temp* matrix, and place it in the means vector at index *i*. Typing *means* afterward produced the result seen, though since I didn't set the seed you will get a different random sample.

¹⁴ I should probably point out that these apply functions do not always work in a way many would think logical. There is the *plyr* package available for an approach that is oftentimes more straightforward.

Importing & Working with Data

Creation of Data

To start with a discussion of dealing with data in the R environment it might be most instructive to simply create some.

One can easily generate data in a variety of ways. The following creates objects that are a simple random sample, random draws from a binomial distribution, and random draws from the normal distribution.

```
x = sample(c("Heads", "Tails", "Edge", "Blows Up"), 5, replace = T, prob = c(0.45,
  0.45, 0.05, 0.05))
x2 = rbinom(5, 1, 0.5)
x3 = rnorm(50, mean = 50, sd = 10)
```

Here is an example simulating data from a correlation matrix using the [MASS](#) library. Code comments are provided in light green¹⁵.

```
cormat = matrix(c(1, 0.5, 0.5, 1), nrow = 2)
# type cormat at the command prompt if you want to look at it after
# creation
library(MASS) # for the mvrnorm function
# empirical = F in the following will produce data that will randomly
# deviate from the assumed correlation matrix
xydata = mvrnorm(40, mu = c(0, 0), Sigma = cormat, empirical = T)
head(xydata) #take a look at it

##          [,1]      [,2]
## [1,]  0.82433  2.0571
## [2,] -1.61582  0.3546
## [3,]  0.06452  0.7776
## [4,]  0.64682 -0.2169
## [5,]  1.06486  1.1749
## [6,]  0.99193  0.5605

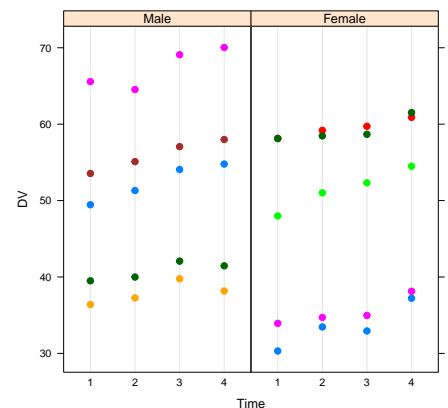
cor(xydata)

##          [,1] [,2]
## [1,]  1.0  0.5
## [2,]  0.5  1.0
```

¹⁵ As in other packages, anything after the comment indicator will be ignored. R uses the pound sign for comments.

The following uses more advanced techniques to create a data set of the kind you'd normally come across. It uses several functions to create a dataset with the variables subject, gender, time and y. Afterwards, the [lattice](#) package is used to create different dot plots. Look at each object as it is created to help understand what's being done on each line and use `?functionname` to find out more about each function used.

```
# Create data set with factor variables of time, subject and gender
mydata <- expand.grid(time = factor(1:4), subject = factor(1:10))
mydata$gender <- factor(rep(0:1, each=4), labels=c("Male", "Female"))
mydata$y <- as.numeric(mydata$time) +
  rep(rnorm(10,50,10),e=4) +
  rnorm(mydata$time,0,1)
```



```
library(lattice)
dotplot(y~time, data=mydata, xlab="Time", ylab="DV") #example plot
# Other plot variations
dotplot(y~time, data=mydata, groups=gender, xlab="Time", ylab="DV",
        key = simpleKey(levels(mydata$gender)))
dotplot(y~time|gender, data=mydata, groups=subject, xlab="Time", ylab="DV",
        pch=19, cex=1)
```

Importing Data

Reading in data from a variety of sources can be a fairly straightforward task, though difficulties can arise for any number of reasons. The following assumes a clean data source.

To read your basic text file, two functions are likely to be used most- `read.table` and `read.csv` for comma-separated value data sets. For files that come from other statistical packages one will want the *foreign* package, especially useful for Stata and SPSS via `read.dta` and `read.spss` respectively. For Excel things tend to get a bit tricky due to its database properties, but you can use *Rcmdr* menus very easily or `read.xls` in the *gdata* package, although *gdata* requires Perl installation. You can always just save the data as a *.csv file from there anyway.

Examples follow¹⁶. Note the slashes (as in Unix)- if you want to use backslash as is commonly seen in Windows you must double them up as \\.

¹⁶ Save the last, these are just syntax examples and not actually to be run.

```
# not run mydata <- read.table('c:/trainsacrossthe sea.txt', header = TRUE,
# sep=',', row.names='id')

# library(foreign) statadat <- read.dta('c:/rangelife.dta',
# convert.factors = TRUE) spssdat <- read.spss('c:/ourwaytofall.sav',
# to.data.frame = TRUE)

# example read from web source
mydata <- read.table("http://csr.nd.edu/assets/22641/testwebdata.txt", header = T,
  sep = " ")
```

To save the data you can typically use `write` instead of `read` while specifying a file location and other options (e.g. `write.csv(objectname, filelocation)`).

Working with the Data

Indexing

R¹⁷ has a lot of data sets at the ready and more come with almost every package for demonstration purposes. As an example we'll use the state data set which is initially a *matrix* class object¹⁸. To provide

¹⁷ R manual

¹⁸ `Matrix` and `data.frame` are functions for creating matrices and data frames as well the name of their respective object classes. One should check out the *data.table* package for additional data set functionality

some more flexibility we will want to convert it to a *data frame*. The state level data regards population, income, illiteracy, life expectancy, murder rate, high school graduation rate, the number of days in which the temperature goes below freezing, and area¹⁹. Note also there are other variable objects as separate vectors which we might also make use of, e.g. `state.region`.

To begin:

```
state2 <- data.frame(state.x77)
str(state2) #object structure

## 'data.frame': 50 obs. of 8 variables:
## $ Population: num 3615 365 2212 2110 21198 ...
## $ Income : num 3624 6315 4530 3378 5114 ...
## $ Illiteracy: num 2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2 ...
## $ Life.Exp : num 69 69.3 70.5 70.7 71.7 ...
## $ Murder : num 15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 10.7 13.9 ...
## $ HS.Grad : num 41.3 66.7 58.1 39.9 62.6 63.9 56 54.6 52.6 40.6 ...
## $ Frost : num 20 152 15 65 20 166 139 103 11 60 ...
## $ Area : num 50708 566432 113417 51945 156361 ...
```

Next we want to examine specific parts of the data such as the first 10 rows or 3rd column, as well as examine its overall structure to determine what kinds of variables we are dealing with. To get at different parts of typical data frame, type brackets after the data set name and specify the row number left of the comma and column number to its right.

```
head(state2, 10) #first 10 rows

##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost
## Alabama           3615   3624         2.1   69.05   15.1   41.3    20
## Alaska             365   6315         1.5   69.31   11.3   66.7   152
## Arizona           2212   4530         1.8   70.55    7.8   58.1    15
## Arkansas           2110   3378         1.9   70.66   10.1   39.9    65
## California         21198  5114         1.1   71.71   10.3   62.6    20
## Colorado           2541  4884         0.7   72.06    6.8   63.9   166
## Connecticut        3100  5348         1.1   72.48    3.1   56.0   139
## Delaware           579   4809         0.9   70.06    6.2   54.6   103
## Florida            8277  4815         1.3   70.66   10.7   52.6    11
## Georgia            4931  4091         2.0   68.54   13.9   40.6    60
##           Area
## Alabama      50708
## Alaska      566432
## Arizona      113417
## Arkansas      51945
## California   156361
## Colorado     103766
## Connecticut   4862
## Delaware      1982
## Florida     54090
## Georgia      58073

# tail(state2, 10) #last 10 rows (not shown)
state2[,3] #third column

## [1] 2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2.0 1.9 0.6 0.9 0.7 0.5 0.6 1.6
## [18] 2.8 0.7 0.9 1.1 0.9 0.6 2.4 0.8 0.6 0.6 0.5 0.7 1.1 2.2 1.4 1.8 0.8
## [35] 0.8 1.1 0.6 1.0 1.3 2.3 0.5 1.7 2.2 0.6 0.6 1.4 0.6 1.4 0.7 0.6

state2[14,] #14th row

##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost Area
## Indiana           5313   4458         0.7   70.88    7.1   52.9   122 36097
```

¹⁹ Type `?state.x77` at the console for additional detail.

```
state2[3,6] #3rd row, 6th column

## [1] 58.1

state2[, "Frost"] #variable by name

## [1] 20 152 15 65 20 166 139 103 11 60 0 126 127 122 140 114 95
## [18] 12 161 101 103 125 160 50 108 155 139 188 174 115 120 82 80 186
## [35] 124 82 44 126 127 65 172 70 35 137 168 85 32 100 149 173
```

Note that one can also examine and edit the data with the `fix` function (or clicking the data object in Rstudio), though if you want to do this sort of thing I suggest popping over to a statistics package that takes the spreadsheet approach seriously. However it is rare that you can edit sizable data sets in that fashion more quickly than you can via code.

Data Sets & Subsets

Often we want to examine specific parts of the data set, and depending on the situation a variety of techniques might be available to help us do so. To begin with, one particular function, `subset`, is expressly devised for this purpose. The basic approach is identify what you want to subset, e.g. a data frame object, and what rule to follow.

```
mysubset = subset(state2, state.region == "South") #note that state.region is a separate object of the same length
```

However, like most situations there are viable alternative approaches, for example:

```
mysubset = state2[state.region == "South", ]
```

Often you might want to subset a collection of variables and there are two typical ways of going about it, either referring to the number or the name like we did previously. Both lines following accomplish the same thing, though the former will probably be a bit troublesome in cases where we have perhaps hundreds of variables.

```
mysubset = state2[, c(1:2, 7:8)]
mysubset = state2[, c("Population", "Income", "Frost", "Area")]
```

One may also drop variables in a likewise fashion for numerically identified variables by placing a minus sign before the column entry or sequence (or before the `c` if combining entries). If your goal is to do something in particular across subsets of a data set, then I again call your attention to the `apply` functions.

Merging and Reshaping

Once you get the hang of knowing how to find the rows and columns of interest, merging different pieces of data can be relatively straightforward. While it may be obvious to some that doing so requires data

sets that have at least something exactly the same about them, this appears to be the biggest hindrance to many data issues we at the CSR come across, often because people don't take the care to do the necessary checks on data integrity from the outset. A simple issue of a v1 in one data set with 100 variables and v.1 that represents the same variable in the other data set will cause problems in merging, and the same goes for ID variables that don't use the same alphanumeric approach.

As an example of how to merge, the following code will first create a demonstration dataset, and then show various ways in which one can attach rows and columns.

```
# Merging and Reshaping
mydat <- data.frame(id = factor(1:12), group = factor(rep(1:2, e = 3)))
x = rnorm(12)
y = sample(70:100, 12)
x2 = rnorm(12)

# add columns
mydat$grade = y #add y via extract operator
df <- data.frame(id = mydat$id, y)
mydat2 <- merge(mydat, df, by = "id", sort = F) #using merge
mydat3 <- cbind(mydat, x) #using cbind

# add rows
df <- data.frame(id = factor(13:24), group = factor(rep(1:2, e = 3)), grade = sample(y))
mydat2 <- rbind(mydat, df)
```

Sometimes you will have to reshape data (e.g. with repeated measurements) from 'long' with multiple observations per unit to wide, where each row would represent a case, and vice versa. This is typically very nuanced depending on the data set, so you'll probably have to play with things a bit to get your data just the way you want. Along with the reshape function one should look into the `melt` and related functions from the `reshape2` package. The following uses `y` from the merge example, but first creates the original data in long format, makes it wide, then reverts to long form.

```
mydat <- data.frame(id = factor(rep(1:6, e = 2)), time = factor(rep(1:2, 6)),
  y)
mydat2 <- reshape(mydat, v.names = "y", direction = "wide")
mydat3 <- reshape(mydat2, direction = "long")
```

Miscellaneous

FACTORS

I want to mention specifically how to deal with factors (categorical variables) as it may not be straightforward to the uninitiated. In general they are similar to what you deal with in other packages, and while we typically would prefer numeric with labels, they don't have

to be. The following shows two ways in which to create a factor representing gender.

```
gender <- c(rep("male", 20), rep("female", 20))
gender <- factor(gender) #levels are now noted, class changes from character to factor

gender <- factor(rep(c(0, 1), each = 20), labels = c("Male", "Female"))
```

At times you may only have the variable as a numeric class, but the R function you want to use will require a factor. To create a factor on the fly use the `as.factor` function on the variable name²⁰.

ATTACHING

Attaching data via the `attach` function can be useful at times, especially to the new user, as it allows one to call variables by name as though they were an object in the current working environment. However this should be used sparingly as one tends to use several versions of a data set, and attaching a recent dataset will mask variables in the previous. As it is very common to use multiple datasets to accomplish one's goal, in the long run it's generally just easier to type e.g. `mydata$myvar` than attaching the data and calling `myvar`. Look also at the `with` function.

²⁰ As an example, the `lm` function in the `rms` package will realize that the numeric variable for the outcome for the ordinal regression is to be treated as a categorical, but the `polr` function in the `MASS` library would need something like `as.factor(y) ~ x1 + x2`.

Initial Data Analysis & Graphics

Many applied researchers seem to ignore the importance of what goes by Exploratory Data Analysis, Initial Examination of Data, Initial Data Analysis (IDA)²¹, Descriptive Statistics etc. However, if you're doing things correctly, you'll likely be spending more time here than at the analysis stage (which should go much more smoothly because of the time spent in this initial stage), and it is with IDA that one gets a feel for the data they're dealing with, finds problems, and develops solutions so that the main analysis goes smoothly.

²¹ I prefer this terminology as it emphasizes it as an essential part of the overall analytic process.

Initial Data Analysis: Numeric

Obtaining basic numeric information about specific variables or whole data sets can be done with various functions from the base package or enhanced with other packages. I like to use the `describe` function from the `psych` package for numeric variables.

```
mean(state2$Life.Exp)

## [1] 70.88

sd(state2$Life.Exp)
```

```
## [1] 1.342

summary(state2$Population)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      365   1080   2840   4250   4970  21200

table(state.region) #region is a separate vector

## state.region
##      Northeast      South North Central      West
##           9          16          12          13

library(psych)
describe(state2)

##           var  n    mean      sd   median trimmed      mad     min
## Population  1 50 4246.42 4464.49 2838.50 3384.28 2890.33 365.00
## Income      2 50 4435.80 614.47 4519.00 4430.07 581.18 3098.00
## Illiteracy  3 50  1.17  0.61  0.95  1.10  0.52  0.50
## Life.Exp    4 50  70.88  1.34  70.67  70.92  1.54  67.96
## Murder      5 50  7.38  3.69  6.85  7.30  5.19  1.40
## HS.Grad     6 50  53.11  8.08  53.25  53.34  8.60  37.80
## Frost       7 50 104.46 51.98 114.50 106.80 53.37  0.00
## Area        8 50 70735.88 85327.30 54277.00 56575.72 35144.29 1049.00
##           max    range skew kurtosis      se
## Population 21198.0 20833.00 1.92  3.75 631.37
## Income     6315.0 3217.00 0.20  0.24 86.90
## Illiteracy  2.8  2.30 0.82 -0.47 0.09
## Life.Exp    73.6  5.64 -0.15 -0.67 0.19
## Murder      15.1 13.70 0.13 -1.21 0.52
## HS.Grad     67.3 29.50 -0.32 -0.88 1.14
## Frost       188.0 188.00 -0.37 -0.94 7.35
## Area       566432.0 565383.00 4.10 20.39 12067.10
```

Often we'll want information across groups. The following all accomplish the same thing, but note the describe function is specific to the psych library.

```
# tapply(state2$Frost, state.region, describe) #not shown by(state2$Frost,
# state.region,describe) #not shown
describeBy(state2$Frost, state.region)

## group: Northeast
##   var n mean      sd median trimmed      mad min max range skew kurtosis se
## 1  1 9 132.8 30.89 127 132.8 35.58 82 174 92 -0.1 -1.46 10.3
## -----
## group: South
##   var n mean      sd median trimmed      mad min max range skew kurtosis se
## 1  1 16 64.62 31.31 67.5 65.71 33.36 11 103 92 -0.46 -1.22
##   se
## 1 7.83
## -----
## group: North Central
##   var n mean      sd median trimmed      mad min max range skew kurtosis se
## 1  1 12 138.8 23.89 133 137.2 20.02 108 186 78 0.58 -1 6.9
## -----
## group: West
##   var n mean      sd median trimmed      mad min max range skew kurtosis
```

It is worth noting how R is dealing with region as illustrative of the power of object-oriented programming. The Region variable and the state2 data set are the same length, and for the purposes of carrying out the function, R is essentially temporarily binding them on the fly to accomplish the goal. An error will result if you substitute state.region with state.region[-1], which will drop the first entry. One can instantly create objects and use them in conjunction with other objects quite seamlessly.

```
## 1 1 13 102.2 68.88 126 103.6 69.68 0 188 188 -0.29 -1.77
## se
## 1 19.1
```

Once we examine univariate information we'll likely be interested in bivariate and multivariate examination. As a starting point try the `cor` function.

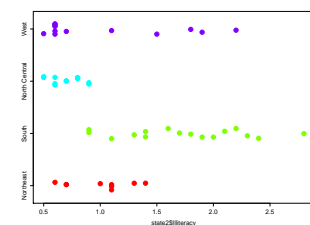
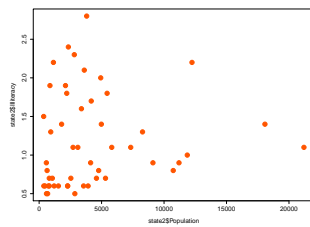
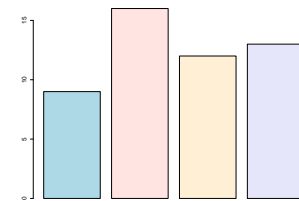
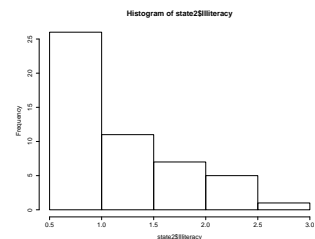
```
cor(state2)

##      Population  Income Illiteracy Life.Exp  Murder  HS.Grad
## Population    1.00000  0.2082  0.10762 -0.06805  0.3436 -0.09849
## Income         0.20823  1.0000  -0.43708  0.34026 -0.2301  0.61993
## Illiteracy      0.10762 -0.4371  1.00000  -0.58848  0.7030 -0.65719
## Life.Exp       -0.06805  0.3403  -0.58848  1.00000  -0.7808  0.58222
## Murder         0.34364 -0.2301  0.70298  -0.78085  1.0000  -0.48797
## HS.Grad        -0.09849  0.6199  -0.65719  0.58222  -0.4880  1.00000
## Frost         -0.33215  0.2263  -0.67195  0.26207  -0.5389  0.36678
## Area          0.02254  0.3633  0.07726  -0.10733  0.2284  0.33354
##      Frost      Area
## Population -0.33215  0.02254
## Income      0.22628  0.36332
## Illiteracy  -0.67195  0.07726
## Life.Exp    0.26207  -0.10733
## Murder     -0.53888  0.22839
## HS.Grad     0.36678  0.33354
## Frost       1.00000  0.05923
## Area        0.05923  1.00000
```

Initial Data Analysis: Graphics

As an introduction to graphical analysis, to which will return in more detail later, here are some basic plots to start with. However when using these standard plots from the base package, realize there are a great many options available to tweak almost any aspect of the graph both via the function and through other parameters that generally available to plotting functions. See the help for `plot` and `par`.

```
hist(state2$Illiteracy)
barplot(table(state.region), col = c("lightblue", "mistyrose", "papayawhip",
  "lavender"))
plot(state2$Illiteracy ~ state2$Population)
stripchart(state2$Illiteracy ~ state.region, data = state2, col = rainbow(4),
  method = "jitter")
```



The Basic Modeling Approach

R's format for analysis of models may have a bit of a different feel than other statistical packages, but it does not require much to get used to.

At a minimum you will need a formula specification and identification of the data set to which the variables of interest belong.

Typical statistics packages fit a model and output the results in some fashion after the syntax is run. R produces 'model objects' that store all the relevant information. Where other packages sometimes require several steps in order to obtain and subsequently process results, R makes this very easy and is far more efficient. Some commonly used model functions include `lm`²² for linear models, `glm` for generalized linear models, and a host of functions from general purpose packages such as *MASS*, *rms*, and *Zelig*.

The following demonstrates creation of a single predictor model and using `summary` to obtain standard table output. There isn't anything here you haven't seen in other package model summaries, but those new to statistical analysis may find it confusing upon first glance if they've only seen it presented in one other package.

```
mod1 = lm(Life.Exp ~ Income, data = state2)
summary(mod1)

##
## Call:
## lm(formula = Life.Exp ~ Income, data = state2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.9655 -0.7638 -0.0343  0.9288  2.3295
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.76e+01   1.33e+00   50.91  <2e-16 ***
## Income       7.43e-04   2.97e-04    2.51   0.016 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.28 on 48 degrees of freedom
## Multiple R-squared:  0.116, Adjusted R-squared:  0.0974
## F-statistic: 6.28 on 1 and 48 DF,  p-value: 0.0156

mod2 = update(mod1, ~. + Frost + HS.Grad)
summary(mod2)

##
## Call:
## lm(formula = Life.Exp ~ Income + Frost + HS.Grad, data = state2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.0878 -0.6604 -0.0043  0.6791  2.1029
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.59e+01   1.25e+00   52.82  < 2e-16 ***
## Income      -7.32e-05   3.33e-04   -0.22  0.82703
## Frost       1.45e-03   3.32e-03    0.44  0.66495
## HS.Grad     9.68e-02   2.65e-02    3.65  0.00067 ***
```

²² I would also recommend `ols` and other functions from the *rms* package for enhanced approaches to standard models.

Just as an aside, one could use the semi-colon here to put both steps on a single line, and in logical 1-2 steps like this you will sometimes come across it in others' code. However, unlike some programs, the semi-colon is never needed to end a line in the R environment.

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.12 on 46 degrees of freedom
## Multiple R-squared:  0.342, Adjusted R-squared:  0.299
## F-statistic: 7.98 on 3 and 46 DF,  p-value: 0.000217

mod3 = update(mod2, ~. - Frost)
summary(mod3)

##
## Call:
## lm(formula = Life.Exp ~ Income + HS.Grad, data = state2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.0082 -0.6866 -0.0644  0.6186  2.2306
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.59e+01   1.24e+00   53.34 < 2e-16 ***
## Income       -7.34e-05   3.30e-04   -0.22  0.82501
## HS.Grad       1.00e-01   2.51e-02    3.99  0.00023 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.11 on 47 degrees of freedom
## Multiple R-squared:  0.34, Adjusted R-squared:  0.312
## F-statistic: 12.1 on 2 and 47 DF,  p-value: 5.81e-05

anova(mod1, mod2)

## Analysis of Variance Table
##
## Model 1: Life.Exp ~ Income
## Model 2: Life.Exp ~ Income + Frost + HS.Grad
##   Res.Df  RSS Df Sum of Sq    F Pr(>F)
## 1      48 78.1
## 2      46 58.1  2        20  7.93 0.0011 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

anova(mod3, mod2)

## Analysis of Variance Table
##
## Model 1: Life.Exp ~ Income + HS.Grad
## Model 2: Life.Exp ~ Income + Frost + HS.Grad
##   Res.Df  RSS Df Sum of Sq    F Pr(>F)
## 1      47 58.3
## 2      46 58.1  1        0.24 0.19  0.66
```

After creation of the initial model object, `summary` is called upon to get the output. The model is then updated and the code is telling the `update` function to keep all previous explanatory variables (`~.`) and add `Population` and `HS.Grad`. The third model includes an update of the second, again telling it to initially keep all other previous predictors but to drop `Population`. The `anova` function can be used for model

comparison, and works beyond lm objects.

One of the powerful aspects of modeling in R is that the model objects have many things of use within them that allow for immediate further analysis and exploration. For example, one can use the function `coef` to extract coefficients directly, one can also extract residuals and fitted values. In addition, the basic plot and other functions are often applicable to both base R and analysis objects from other packages, though various package will offer unique versions of output and graphs.

```
mod1$coef

## (Intercept)      Income
##  6.758e+01    7.433e-04

coef(mod1)

## (Intercept)      Income
##  6.758e+01    7.433e-04

# mod1$res #not run
confint(mod1)

##              2.5 %   97.5 %
## (Intercept) 6.491e+01 70.25058
## Income      1.472e-04  0.00134

plot(mod1) #not shown
```

As your needs vary you will likely move to packages beyond the base offerings, and this will require you to get used to the specifics of each. For example, some may not require a summary function to produce the basic output, others require matrix rather than formula input, plots will produce different graphs or perhaps no default plot is available etc. Depending on the analysis of choice there may be quite a few options, and as you use a new package you should investigate the help file thoroughly before getting too carried away.

```
# Example poisson, additive and mixed model.
glm(Y ~ X, data = d, family = poisson)
gam(Y ~ s(X))
lmer(Y ~ X + (1 | group), data = d)
```

Visualization of Information

Examples

To start with visualization in R, you might peruse some of the graphs in the margin to see what's possible, sometimes remarkably easily even.

Creating graphs in R allows you fine detail control over all aspects of the graph and you have the ability to create extremely impressive images that are publication ready.

The typical statistical plots are available, but most have enhanced versions in other packages. One can use generic functions such as `plot` followed by `lines`, `points`, `legend`, `text()`, and others to add further information or changes. The generic plot function is also able to be used after many analyses, such that using it on a *model object* will bring up plot(s) specific to the analysis. In other cases, there are functions for specific types of graphs such as the ones we did earlier. To get an idea of the control you have over your plots look at the help file for `plot` and `par`.

Enhancing Standard Graphs

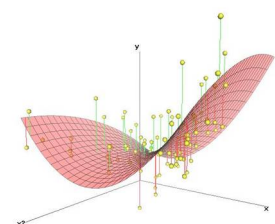
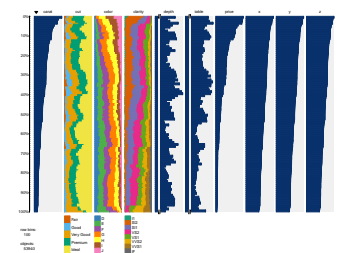
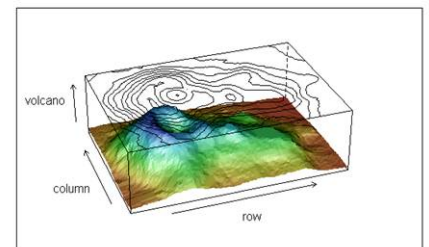
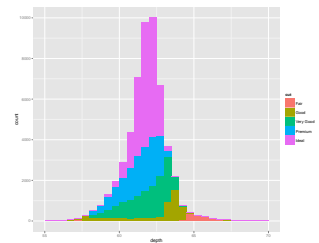
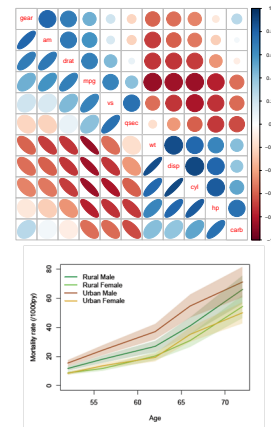
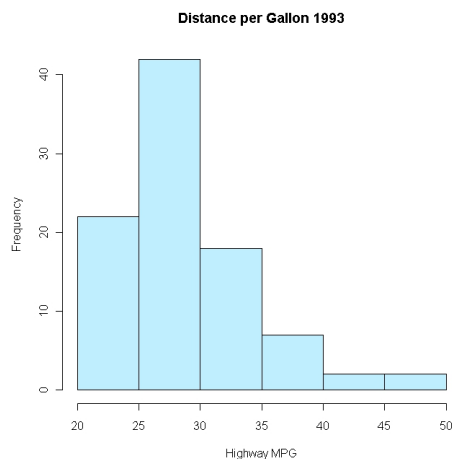
Earlier we created some standard graphs, but they typically aren't what we want, especially if we want to present them to others either formally or informally. However it's best not to rush things so what we'll do here is start with the usual and tweak as we go along. To begin with we will take some random data and plot the histogram and a boxplot (the latter is not shown).

```
x <- rnorm(100)
hist(x)
boxplot(x)
```

Such graphs are as basic as they come and not too helpful except as an initial pass. For the following we'll use the Cars93 data from the MASS package. Load up the MASS library if you haven't already.

```
hist(Cars93$MPG.highway, col = "lightblue1", main = "Distance per Gallon 1993",
     xlab = "Highway MPG")
```

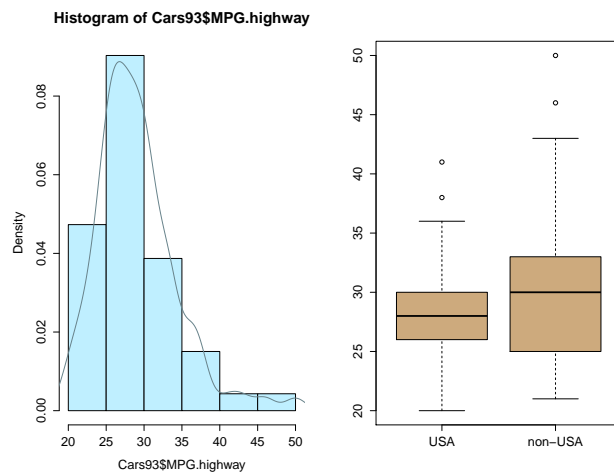
Which will produce the following:



However we can tweak it a little bit, as well as add an additional plot to the overall graph. The `par` function is used here to set the number of rows and columns within the actual graphics device so that more than one plot at a time may be displayed, at which point it is reset for displaying one plot at a time in full view.

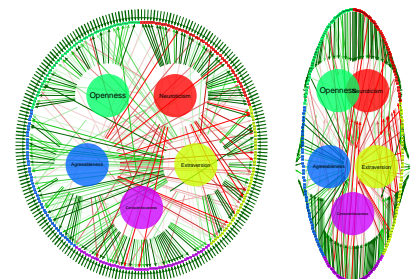
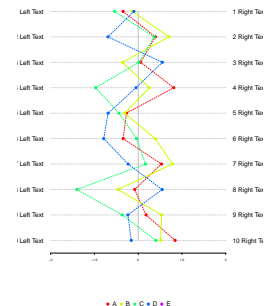
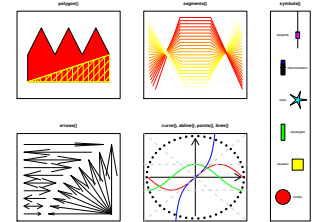
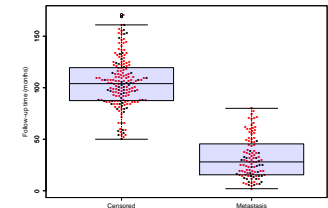
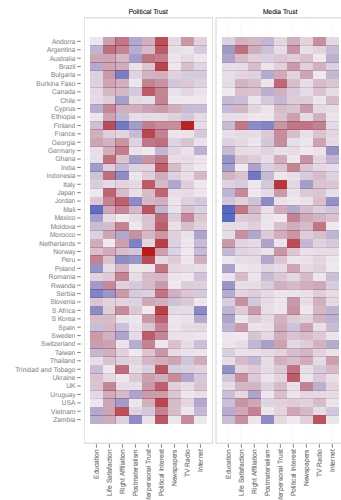
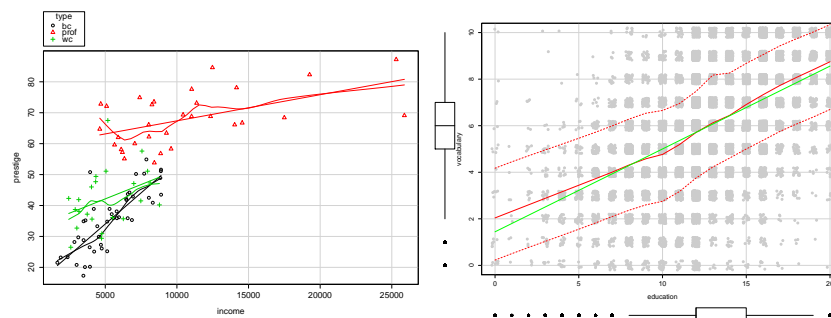
```
par(mfrow = c(1, 2))
hist(Cars93$MPG.highway, col = "lightblue1", prob = T)
lines(density(Cars93$MPG.highway), col = "lightblue4")
boxplot(MPG.highway ~ Origin, col = "burlywood3", data = Cars93)
par(mfrow = c(1, 1))
```

Which will produce the following graphic:



It is worth noting that some typically used plots come with extras by default. For example, the scatterplot function in the car library automatically provides a smoothed loess line and marginal boxplots, providing very useful information with ease.

```
library(car)
scatterplot(prestige ~ income | type, data = Prestige)
scatterplot(vocabulary ~ education, jitter = list(x = 1, y = 1), data = Vocab,
  col = c("green", "red", "gray80"), pch = 19)
```



Beyond R

R allows one to take things beyond its statistical environment relatively easily also. For example, packages allow one to make use of the Google visualization API, writing out to other formats viable for network visualization in other packages, animation and so forth²³. As far as pure statistical packages go, most are only getting to some of the capabilities R has long had, while R continues to advance along with what modern analysis requires to meet its full potential.

One more thought regarding graphics and presentation in general—the goal is to provide as much information as possible in as clear a manner as possible. If your graph is only showing single variable outcomes or is filled with chart junk, you’re just wasting space and likely the audience’s time. For guidance on graphs, data visualization etc., check out websites such as [NY Times](#) (they have stats folk in their graphics department)²⁴, [Flowing Data](#), and [Infosthetics](#), folks like Edward Tufte and William Cleveland for some traditional sources, and within R definitely familiarize yourself with [lattice](#) and [ggplot2](#) packages.

²³ Note that R has similar sorts of capabilities as is, such as interactive plots ([shiny](#)), animation ([animate](#)) etc.

²⁴ See this [link](#) to see how R was integrated with other approaches to produce one of their interactive graphics.

Getting Help

Basics

Now that you have some idea of what R can do for you and are ready to try it on your own, it is imperative that you find out how you can continue to work with it. It likely won’t take but a minute or two before you start hitting snags, and truth be told, you’ll probably spend a lot of time looking up how to do things even after you get very familiar with it. However that’s not a sign of R’s difficulty so much as its capability. As you get used to the sorts of things you already are familiar with, you’ll come across more and more alternatives and enhancements, and you will actually be getting a lot more done in the same amount of time in the end.

As a start in becoming self-sufficient, get used to using the help files. As mentioned previously, typing `?` or `??` followed by the function name, e.g. `?hist` `??scatterplot`, will bring up the help file in the former case, or do a search for whatever term you’ve inserted²⁵. Typing `help.start()` at the console will bring up manuals etc. Note that while there are help files for every function, not all have the same amount of information. Some functions/packages come with vignettes, links to reference articles etc. Others give only the bare minimum of information needed to

²⁵ Same as `help` and `findit` in Stata

use the function. Some examples are exactly what you need, some will spend 3/4 of many lines of code simply constructing the data set to be used for the example that's actually of interest, others might be a single line or two that do little to illuminate the possibilities. Such is the way of things when you have thousands of people contributing their efforts, but do note that *all* help files contain the same type of information as they are required to.

A very useful thing to help you find the information you need is via the Rseek search engine browser addon; example results are pictured at right. It will return a specialized Google search (just fyi, there happen to be lots of webpages with the letter R in them, and given Google's search tendencies of late any help is useful), introductions, associated Task Views, R-help list results etc. You might spend some time getting acquainted with the R-help list or even subscribing to it, but with this search engine it's not necessary²⁶.

Increasingly of late, many of those who are well-known on the R-help lists, along with many other folks from all walks of life, are now seen more on [StackExchange/Cross-Validated](#). You'll also find fewer folks complaining about reading the posting guide (or lack thereof).

A Plan of Attack for Diving In

The following are some guidelines that should help someone make for a quicker transition toward using R regularly.

Use the built-in script editor or better, one with syntax highlighting, rather than the console²⁷.

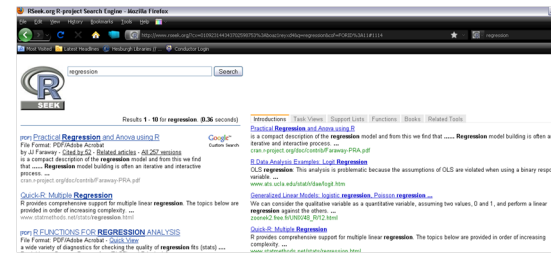
Get used to installing packages and importing data into R, and use Rcmdr menus to help with this initially if need be.

Start with `?functionname` to see how to implement any particular function first. This will get you more acquainted with the help files and in the habit of trying the examples. Saves time in the long run.²⁸

There are many users and places for support on the web. Use *Rseek* and regular web searching to find the names of useful functions, peruse the [R bloggers](#) as you start out, and even Rcmdr menus for initial analyses and plots, and follow the code that it spits out.²⁹

Take simple functions or commands you use in other packages such as for mean, sd, summary, and simple graphs and reproduce them in R with no frills. Spend some time at the [Quick-R](#) website to help with this.

Redo the things you've learned thus far, adding options (extra arguments) along the way.



²⁶ Furthermore, the help list has gotten out of hand for every day perusal, often going over 100 messages for a single daily digest with most of them quoting previous messages which were quoting previous messages and so on. Basically the thing as a whole is a mess to look at, so you're better off searching it via Rseek.

²⁷ At this point there's no real reason not to use Rstudio.

²⁸ Type `example(functionname)` at the console for any function you want to see in action.

²⁹ Just note that some of the functions will be specific to the Rcmdr package.

Use the manuals, websites etc. specifically listed in the References section, there is a ton of information on R available.

Alternatively, you can find some things R does that you like and/or does better than your main statistical package, and simply pull out R as you need it. That is how I initially did things myself, though I wish I hadn't. Some R is better than none to be sure, but it didn't make it any easier to work within the environment in general, nor does it give a good sense of the capabilities of the program, as you may just come to think that it's merely a more complicated way of doing things slightly better, instead of a much better way to do statistical analysis in general. But again, for some that may be the most practical approach to start.

Some Issues

Be prepared to have to spend some time, perhaps a lot of time, within R before getting used to it. For most it is more difficult to learn compared to other statistical programs, though this depends in part on one's prior experiences with other programs or programming languages. As mentioned previously, the actual help in help files can vary quite a bit, often assume the knowledge one might in fact be looking for and it may simply provide syntax. In the actual implementation and as with other stat programs, cryptic error messages can leave one guessing whether there is a data, code or estimation problem. Some may find it to be slower than other programming languages like C, but for typical use in applied social science research this will typically not be an issue. R can at times be relatively slow with huge data sets (hundreds of thousands) or even smaller with inefficient code, but not for typical 'large' sizes. Furthermore there are packages and specialized versions (Revolution Analytics) that can help optimize processing of large amounts data and packages such as [parallel](#), [snow](#) and others, one can utilize the cpus on their own machine or over a cluster for increased processing speeds.³⁰

So yes, R can be a pain in the rear at times, but despite occasional issues one may come across there are many avenues for assistance that are extremely helpful, and various solutions available when you do hit a snag. The reference section provides a list of places to find additional help on your own.

³⁰ With big data problems your desk or laptop can't handle, see what the [CRC](#) here on campus can do for you. Also, I have used the multiprocessor packages in R for both single machines and clusters, so can provide direct assistance.

Summary

Why R?

R is the premier statistical package going right now. *R* is free, and being open-source, rapidly updated, debugged, and advanced. *You* are free to use it in any way you wish to accomplish what you want to statistically. What more could you ask for really?

Who is R for?

R is for anyone that wants to engage in thoughtful statistical analysis of scientific problems. It will take effort, possibly quite a bit, but you wouldn't be interested in the first place if you were short on that.

R as a solution

R can do what you want, and you are not forced into limitations by design in the way you are with other packages which have no flexibility. The question is not *if* R can do this or that analysis, but instead, *how*.

References

General Web

- Main R website* The place to start.
- CRAN Task Views* Packages grouped by discipline/subject.
- R forge development site* Most recent versions and unreleased packages.
- Cross-Validated* General stats help site where people post lots of R questions.
- Quick-R website* Particularly geared toward SAS and SPSS users.

Manuals

- List at the R website* These are accessible via the help menu.
- An Introduction to R* The main R manual.
- Simple R* Dated but still useful to someone starting out.
- Reference Card* Print out and keep handy when starting out with R.

Graphics

- ggplot2 library website* Hadley Wickham's helpful website and package.
- Paul Murrel's website* Another core developer.
- Addicted to R* See the possibilities, as well as tough judges.

Commercial Versions

- Revolution Analytics* ³¹
- S-Plus*

³¹ I guess it says something when the creator of SPSS jumps ship for R. I don't know what it says though.

Miscellaneous

- UCLA Statistical Computing group* A great resource for all general packages.
- R blogs* For those that like their R with too many links and stories about cats³²
- Crantastic* Note new packages and updates.
- R For SAS AND SPSS USERS* Actually a good way to learn all three (plus S).

³² Just kidding, they are much more useful than standard blogs, but who doesn't need more lolcatz?.