Introduction to R

Vince Melfi

CSTAT workshop, October 28, 2011
Michigan State University

1 Introduction

According to its developers, "R is a language and environment for statistical computing and graphics.[1]" As such, R is a very powerful tool for implementing new statistical and graphical methods, as well as for making non-standard uses of existing methods. For this reason, cutting-edge research is often first implemented in R, before (sometimes slowly) making its way to other statistical packages. This is particularly evident in the area of statistical biology, with the Bioconductor project taking the lead (see `http://www.bioconductor.org`). But R is also becoming a commonly-used tool in finance (`http://www.rmetrics.org`) and other areas. A recent article in the New York Times, entitled *Data Analysts Captivated by R's Power*, discusses the adoption of R by leading firms such as Google, Pfizer, and others. Of course R is widely used by academic statisticians and other academics who develop new statistical tools.

R started in the early 1990s as a modest project of two statisticians in New Zealand, Ross Ihaka and Robert Gentleman. It has similarities with the statistical environment S, developed earlier at Bell Labs, and the commercial software S-Plus, which is built on the S base. Because R is freely available under the GNU General Public License, many other statisticians and programmers began contributing to R in the late 1990s, and it quickly developed into a powerful and robust tool.

1.1 Installing R

It's easy to install R on a variety of computers. We'll concentrate on installing R on computers running Microsoft Windows. For other operating systems, the instructions below will need a few minor modifications.

1. Go to `http://www.r-project.org`. This is the home page for the R project.

2. On the left side of the page will be a link called **CRAN** right under the words "Download, Packages." Click on this link.

3. You'll be faced with a list of many web servers from which you can download R. Choose one of these, preferably one in the US.

4. Near the top of the page will be a section titled "Download and install R." Click on the **Windows** link in this section. (This is where users of other operating systems would make a different choice.)

5. In the subsequent window, click on the **base** link, and then click on the link **Download R 2.13.2 for Windows** to download the setup program to your computer. There are also installation instructions in case they're needed. The setup program is currently called **R-2.13.2-win32.exe**. The name changes as new versions of R appear.

6. Double click the setup program on your computer to install R.

[1] See `http://www.r-project.org/about.html`

1.2 Working directory and starting R

As with any software, it's useful to keep information from a single session, or several related sessions, in a single directory. There are several ways to do this with R, but the following is probably the best. First create the directory. Then create a shortcut to R in this directory. One way to do this is to find R on the **Start** menu, but instead of left clicking on it, right click and choose **Copy**. Then go to the prepared directory and paste the copy there, creating a shortcut. There's one more step: Right click on the shortcut and choose **Properties**. Under **Start in:** type or paste the full path to the directory you're in. Then when you start R by double clicking the shortcut, it will automatically use the prepared directory as the working directory.

If you're running R and aren't sure what the working directory is set to, just type `getwd()` in your R session. When R starts you will see a large window titled **RGui** with some menus and clickable icons visible. Inside the RGui window will be a window titled **R Console**. This is where commands are typed and where text output is written.

1.3 Getting Help

There is a help function in R, and also a web-based help interface that allows easier searching and other functionality. For example, if help for the `median()` function is desired, typing `help(median)` within R. The resulting help page is shown in Figure 1.

The function `help.start()` opens a web-based interface that is often more informative than the `help()` function alone. The web interface has a search functionality that is useful if the user knows what he wants R to do, but doesn't know the function name. For example, suppose a user wants to know how to compute standard deviation in R, but doesn't know the appropriate function name. The user can search for "standard deviation" in the web interface and quickly learn that `sd()` is the appropriate function.

1.4 Packages

Functions and datasets in R are often provided as part of a *package*. For example, the `stats` package contains functions for commonly used statistical procedures; the `cluster` package contains functions for cluster analysis; and the `affy` package contains functions for analysis of Affymetrix microarrays. Some packages are included in the default R distribution and are loaded into any R session by default (for example, the `stats` package), while others must be installed and loaded by the user.

The `library()` command (with no arguments) lists the packages that are installed. Of these, some may be loaded for use, and others may not. The `search()` command lists those packages that are loaded. Output of these commands will vary depending on the installation.

Packages can be installed either at the command line or via a menu under Windows. To install packages via the menu option, use **Packages** > **Install Packages**. You may first be prompted to choose a package repository. Choose one close to home. After

Median Value

Description

Compute the sample median.

Usage

```
median(x, na.rm = FALSE)
```

Arguments

`x`	an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.

Details

This is a generic function for which methods can be written. However, the default method makes use of `sort` and `mean`, both of which are generic, and so the default method will work for most classes (e.g. `"Date"`) for which a median is a reasonable concept.

Value

The default method returns a length-one object of the same type as `x`, except when `x` is integer of even length, when the result will be double.
If there are no values or if `na.rm = FALSE` and there are `NA` values the result is `NA` of the same type as `x` (or more generally the result of `x[FALSE][NA]`).

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`quantile` for general quantiles.

Examples

```
median(1:4)# = 2.5 [even number]
median(c(1:3,100,1000))# = 3 [odd, robust]
```


Figure 1: Output of `help(median)`

this, a list of available packages is shown. Choose one and click OK, and the package will be installed on your system.[2]

At this point the package still is not available in your R session. To use the package, use the **Packages** > **Load Package** menu. [Or use the `library()` function.] There are ways to automatically load desired packages when the R session starts. Consult R documentation for details.

1.5 Further resources

Working through the examples in this document will give a basic understanding of R, but (as with learning any complex piece of software) there will be much more to learn to be proficient in using R. There are many introductory R sessions and R demonstrations available. In addition, there are many books (both free online and for purchase) that help users to learn R, both in general and for specific (e.g., biological applications) purposes. You may find it worthwhile to go through one or more of these at some point, to gain more familiarity with R and its capabilities. Ideally you'll do this with some specific applications in mind. Here are a few possibilities (all free):

- A brief (76 pages) but quite informative introduction to R written by Emmanuel Paradis is at `http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf`.
- The introductory session in Appendix A of *An Introduction to R* by W.N. Venables, D.M. Smith, and the R Development Core Team, avaliable at `http://cran.r-project.org/doc/manuals/R-intro.pdf`.
- A demonstration of some of R's graphics capabilities, available within any R session by typing the command `demo(graphics)`.
- Several manuals (some covering rather arcane aspects of R) are at `http://cran.r-project.org/manuals.html`.
- Contributed documentation (including the Paradis document referred to above), most in English but some in other languages, is at `http://cran.r-project.org/other-docs.html`.

For issues related to data, there are two important sources. One is the R Data Import/Export manual, available at `http://cran.r-project.org/doc/manuals/R-data.pdf`. The other is the book *Data manipulation with R* by Phil Spector. For graphics, the book *R Graphics* by Paul Murrell is fundamental. In addition, the *R for beginners* manual by Emmanuel Paradis referred to above has interesting information on both data and graphics. All four of these are listed in the references at the end of this document.

[2]This usually works. For Bioconductor packages, there is a different method. Consult the Bioconductor home page at `http://www.bioconductor.org` for instructions.

There are also a lot of other resources on the internet, but it's not always easy to find these resources using a search engine like google, since if a search string such as "linear regression R" is given, google doesn't know that the "R" refers to the statistical software R, but will also find pages where "R" refers to the correlation coefficient. For this reason there are some search engines devoted solely to indexing information about R. Two of these that are worth a look are `http://www.rseek.org` and `http://search.r-project.org`.

2 Data structures in R

Throughout the document some sections are starred. These sections may be omitted the first time through if necessary.

As with any statistical software, facility with bringing data into the program, exporting data to other programs, and manipulating data, are basic skills that usually take more time and effort than the actual data analysis. R has several structures for data, including vectors, matrices, data frames, and lists. (In addition, there are specialized data structures for particular types of data, e.g., gene expression microarray data or time series data. But understading the three basic data structures vectors, matrices, and data frames, is a good starting point.)

2.1 Data vectors

A vector can be thought of as a way to store the values of one variable. There are a variety of ways to enter data into R at the keyboard, including spreadsheet-like interfaces. We'll focus on methods for entering data through the console window interface, which are the most useful to know. Throughout much of our work with data we will use a small economic data set for illustration. The data are displayed in Table 1, and give the per-capita Gross National Product and the percent of the population employed in agriculture in 1994 for 12 countries in the European Union. We'll first enter the GNP

Country	GNP	AGRIC
Belgium	16.8	2.7
Denmark	21.3	5.7
Germany	18.7	3.5
Greece	5.9	22.2
Spain	11.4	10.9
France	17.8	6.0
Ireland	10.9	14.0
Italy	16.6	8.5
Luxembourg	21.0	3.5
Netherlands	16.4	4.3
Portugal	7.8	17.4
United Kingdom	14.0	2.3

Table 1: Data on per capita GNP and percent of population employed in agriculture for members of the European Union in 1994

values:

```
> GNP = c(16.8, 21.3, 18.7, 5.9,
+     11.4, 17.8, 10.9, 16.6, 21,
+     16.4, 7.8, 14)
> GNP
```

```
 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9
 [8] 16.6 21.0 16.4  7.8 14.0
```

```
> AGRIC = c(2.7, 5.7, 3.5, 22.2,
+     10.9, 6, 14, 8.5, 3.5, 4.3,
+     17.4, 2.3)
> AGRIC
```

```
 [1]  2.7  5.7  3.5 22.2 10.9  6.0 14.0
 [8]  8.5  3.5  4.3 17.4  2.3
```

Notice a few things. First, we chose the name `GNP` for the data values. Second, the R function `c()` is used to *concatenate* several data values into one vector. Third, to see what is in an object like `GNP`, just type its name. Fourth, when R displays the values in an object like `GNP`, it indicates in square brackets the position of the first data value displayed on a line. For example the `[8]` next to the data value `16.6` indicates that this is the eighth data point in the GNP data vector. Also, names of objects in R are case-sensitive, as the following shows:

```
> gnp
Error: object "gnp" not found
> Gnp
Error: object "Gnp" not found
> gNP
Error: object "gNP" not found
```

```
> GNP
```

```
 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9
 [8] 16.6 21.0 16.4  7.8 14.0
```

The `GNP` object contains numbers. Data vectors also can contain character data or logical data. We'll discuss logical data later. For character data, for example, we can enter the countries associated with the data in GNP. We'll use the official abbreviations for the countries to save some typing.

```
> countries = c("BE", "DK", "DE", "GR", "ES", "FR", "IE",
+     "IT", "LU", "NL", "PT", "UK")
> countries
```

```
 [1] "BE" "DK" "DE" "GR" "ES" "FR" "IE" "IT" "LU" "NL" "PT" "UK"
```

Note that the values are specified in quotes, to tell R that these are character values.

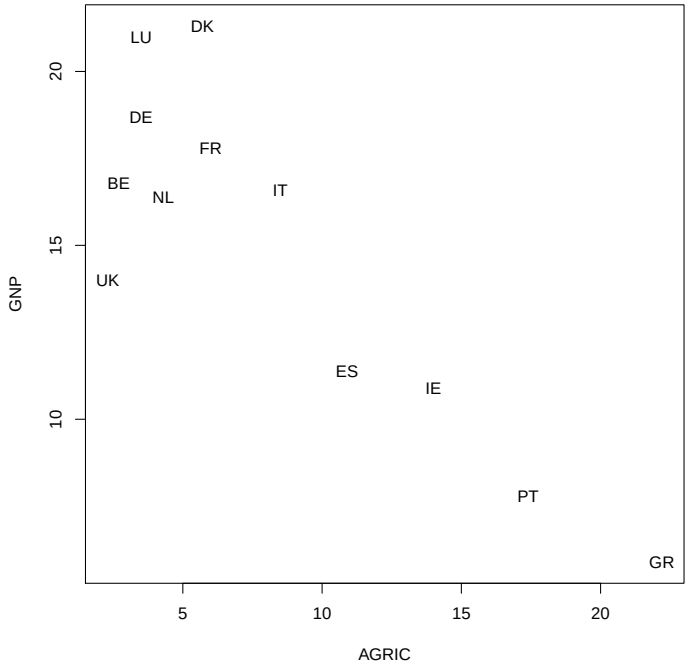

Figure 2: A scatter plot of the percent of the population employed in agriculture versus the per capita gross national product for twelve countries

2.1.1 Factors*

A plot of `AGRIC` verus `GNP` is shown in Figure 2. The four countries in the lower right seem to be separate from the other eight, having both relatively high percents of the population in agriculture and relatively small per-capita GNPs. It might be useful to have a variable that is set to one for the four countries in the lower right, and set to two for the other eight countries. Here is a first attempt:

```
> twogroups = c(2, 2, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2)
> twogroups

 [1] 2 2 2 1 1 2 1 2 2 2 1 2
```

This is OK, but a careless user might think that the values 1 and 2 have some quantitative significance, rather than just specifying category membership. Making `twogroups` a `factor` variable is what's needed. We'll change the name to help emphasize the differences.

```
> fac.twogroups = factor(twogroups)
> fac.twogroups

 [1] 2 2 2 1 1 2 1 2 2 2 1 2
Levels: 1 2
```

```
> mean(twogroups)

[1] 1.666667

> mean(fac.twogroups)

[1] NA
```

2.2 Matrices

We have created three vector objects, `GNP`, `AGRIC`, and `countries`, related to the data from Table 1. It would be nice to have a data structure that can contain more then one variable. Matrices are one such data structure. The R functions `matrix()`, `rbind()`, and `cbind()`, are all useful for creating matrices. First we'll use the `matrix()` function to create a matrix with four rows and three columns containing the integers from 1 through 12. Rather than typing the twelve numbers in explicitly, we'll use `1:12`, which returns all twelve numbers.

```
> mat1 = matrix(data = 1:12, nrow = 4, ncol = 3)
> mat1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Note that we specified the values to be entered in the matrix via the `data` argument, the number of rows via the `nrow` argument, and the number of columns via the `ncol` argument. (Of course since there are 12 data values, once we specified the number of rows we didn't need to specify the number of columns.)

In `mat1` the matrix was filled "by column," i.e., the first column was filled with the first four numbers, the second column was filled with the next four numbers, etc. We can change this behavior by using the `byrow` argument:

```
> mat2 = matrix(data = 1:12, nrow = 4, ncol = 3, byrow = T)
> mat2

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

Now let's create a matrix with the GNP data in the first column and the AGRIC data in the second column:

```
> euromat = matrix(data = c(GNP, AGRIC), nrow = 12)
> euromat

      [,1] [,2]
 [1,] 16.8  2.7
 [2,] 21.3  5.7
 [3,] 18.7  3.5
 [4,]  5.9 22.2
 [5,] 11.4 10.9
 [6,] 17.8  6.0
 [7,] 10.9 14.0
 [8,] 16.6  8.5
 [9,] 21.0  3.5
[10,] 16.4  4.3
[11,]  7.8 17.4
[12,] 14.0  2.3
```

The `rbind()` and `cbind()` functions provide a somewhat easier and more intuitive way to join together several equal length data vectors into a matrix:

```
> euromat2 = cbind(GNP, AGRIC)
> euromat2

       GNP AGRIC
 [1,] 16.8   2.7
 [2,] 21.3   5.7
 [3,] 18.7   3.5
 [4,]  5.9  22.2
 [5,] 11.4  10.9
 [6,] 17.8   6.0
 [7,] 10.9  14.0
 [8,] 16.6   8.5
 [9,] 21.0   3.5
[10,] 16.4   4.3
[11,]  7.8  17.4
[12,] 14.0   2.3

> euromat3 = rbind(GNP, AGRIC)
> euromat3

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
GNP   16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0  16.4   7.8  14.0
AGRIC  2.7  5.7  3.5 22.2 10.9  6.0 14.0  8.5  3.5   4.3  17.4   2.3
```

2.2.1 All elements of matrices are the same type

Consider the following example:

```
> euromat4 = rbind(GNP, countries)
> euromat4

          [,1]   [,2]   [,3]   [,4]  [,5]   [,6]   [,7]   [,8]   [,9]
GNP       "16.8" "21.3" "18.7" "5.9" "11.4" "17.8" "10.9" "16.6" "21"
countries "BE"   "DK"   "DE"   "GR"  "ES"   "FR"   "IE"   "IT"   "LU"
          [,10]  [,11] [,12]
GNP       "16.4" "7.8" "14"
countries "NL"   "PT"  "UK"
```

Note that the elements of GNP were converted to character values. In R, all elements of a matrix have to be of the same type, such as character, integer, etc. If vectors of different types are combined as in the example above, R will change them all to one type. There are specific rules for deciding which type takes precedence. (See `help(rbind)`, for example, for the rules.) A good rule of thumb is to avoid what we did above, combining vectors of different types in a matrix.

But this presents a problem. For example, our data set has several types of data. The country names are character. The GNP and AGRIC values are numerical. And there is a factor variable indicating membership in the two groups. For representing a data set, a `data frame` is often a better choice than a matrix.

2.3 Data Frames

Data frames are the most common way to organize several variables in a data set. Unlike matrices, the variables don't all have to be of the same type. Here we create a data frame containing the `GNP`, `AGRIC`, `countries`, and `fac.twogroups` variables.

```
> euro.data = data.frame(countries = countries, percapGNP = GNP,
+     percAGRIC = AGRIC, grouping = fac.twogroups)
> euro.data

   countries percapGNP percAGRIC grouping
1         BE      16.8       2.7        2
2         DK      21.3       5.7        2
3         DE      18.7       3.5        2
4         GR       5.9      22.2        1
5         ES      11.4      10.9        1
6         FR      17.8       6.0        2
7         IE      10.9      14.0        1
8         IT      16.6       8.5        2
9         LU      21.0       3.5        2
10        NL      16.4       4.3        2
```

```
11        PT       7.8      17.4         1
12        UK      14.0       2.3         2
```

The function `data.frame()` creates a data frame out of its arguments. Note that the name before the `=` sign is the name to be given to the varible in the data frame, and the name after the `=` sign is the object containing the data for the variable. So `countries = countries` tells R to call a variable `countries` in the data frame and to fill this with the values in the variable `countries` that was created earlier. And `percapGNP = GNP` tells R to create a variable called `percapGNP` in the data frame and to fill this with the values in the variable `GNP` that was created earlier.

At this point you may wonder why R has matrices, since a data frame seems a lot like a matrix, but allows different types of variables. But R has powerful matrix functions that operate on matrices with numerical entries, that are of great use in statistical analysis. Examples include matrix algebra functions such as matrix multiplication, transpose, etc., computing eigenvalues, and much more.

2.4 Naming objects and cleaning up the R workspace

Names of objects in R should be chosen with some care. Of course it is good to choose names that are informative. It is probably better to use `AGRIC` than `x` for the percentage of the population employed in agriculture. In addition, names of objects that are already defined in R should be avoided. As an example, `c` is the name of the concatenate function that we have used several times to create vectors. So naming a vector `c` would be unwise. Usually if a name is informative it won't already be chosen, but it's a good idea to check first. Here's an example: Suppose a user is thinking of using the name `consumption` for a variable, and is also thinking of using the word `solve` for another variable.

```
> consumption
Error: object "consumption" not found
> solve
function (a, b, ...)
UseMethod("solve")
<environment: namespace:base>
```

The message `Error: object "consumption" not found` tells the user that there isn't an object named `consumption`, so this is a safe name to use. But the fact that this isn't indicated about `solve` tells the user that another name should be found. (Don't worry about understanding what the message means.)

Objects can build up quickly in the R workspace. The function `ls()` lists the objects in the workspace, and the function `rm()` removes objects. For example, here are the objects in an R session:

```
> ls()
 [1] "AGRIC"         "countries"     "euro.data"     "euromat"
```

```
 [5] "euromat2"      "euromat3"      "euromat4"      "fac.twogroups"
 [9] "GNP"           "heightframe"   "heights"       "heights1"
[13] "heights2"      "heights3"      "mat1"          "mat2"
[17] "newlist"       "newlist2"      "twogroups"
```

Now we remove `twogroups` and `mat1`:

```
> rm(twogroups, mat1)
> ls()
 [1] "AGRIC"         "countries"     "euro.data"     "euromat"
 [5] "euromat2"      "euromat3"      "euromat4"      "fac.twogroups"
 [9] "GNP"           "heightframe"   "heights"       "heights1"
[13] "heights2"      "heights3"      "mat2"          "newlist"
[17] "newlist2"
```

Of course it is possible to ask R to list only certain objects, for example, those whose names start with the letter `m`, and to remove only certain objects. To do this effectively requires some understanding of so-called *regular expressions*, which would take us too far afield, but here is one simple example, listing all objects whose names begin with the letter `e`:

```
> ls(pattern = "^e")

[1] "euro.data" "euro.lm"   "euromat"   "euromat2"  "euromat3"
[6] "euromat4"
```

2.5 Saving R objects

During the R session the objects created are not saved to disk unless this is explicitly requested. The R workspace can be saved to disk by selecting **File** > **Save Workspace** when the R console window is active. By default this will save the objects into a file called `.Rdata` in the working directory. In addition, when the R session is terminated either by typing `q()` in the R console window or by selecting **File** > **Exit**, the user is asked whether he or she wants to save the workspace. If this is not done, the objects created will not be available when R is restarted.

2.6 Lists*

Data frames are rectangular, i.e, all columns in a data frame must have the same number of elements. This makes sense for representing certain kinds of data, but is sometimes not flexible enough for more complex data. `Lists` provide a data structure that does not have this limitation. (In fact, a data frame is just a special type of list.)

Consider the following:

```
> newframe = data.frame(x = c(1, 2, 3), y = c(2, 3, 5, 7), z = "dog")
Error in data.frame(x = c(1, 2, 3), y = c(2, 3, 5, 7), z = "dog") :
  arguments imply differing number of rows: 3, 4, 1
```

Instead of using a data frame, we need to use a list:

```
> newlist = list(x = c(1, 2, 3), y = c(2, 3, 5, 7), z = "dog")
> newlist

$x
[1] 1 2 3

$y
[1] 2 3 5 7

$z
[1] "dog"
```

Lists are quite flexible. Components of a list can even be other lists:

```
> newlist2 = list(onelist = newlist, gg = function(x, y) {
+     x^2 * y
+ }, hh = c(1, 2, 2))
> newlist2

$onelist
$onelist$x
[1] 1 2 3

$onelist$y
[1] 2 3 5 7

$onelist$z
[1] "dog"


$gg
function (x, y)
{
    x^2 * y
}

$hh
[1] 1 2 2
```

3 Subscripting R objects

Data analysis often requires selecting and possibly altering specific variables, or specific cases. For example, it may be necessary to correct an error in the height measurement

of the third subject. Or it may be desired to select the first and third variables for all subjects who are male. Or data may need conversion from degrees Celsius to degrees Fahrenheit. R has a powerful subscripting capability for this purpose.

3.1 Simple subscripting

Selecting specific elements of vectors and matrices is simple. Here are some examples:

```
> GNP

 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0

> GNP[4]

[1] 5.9

> length(GNP)

[1] 12

> GNP[length(GNP)]

[1] 14

> GNP[length(GNP) - 4]

[1] 16.6

> GNP[-4]

 [1] 16.8 21.3 18.7 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0

> GNP[1:5]

[1] 16.8 21.3 18.7  5.9 11.4

> GNP[-c(1, 3, 5, 7)]

[1] 21.3  5.9 17.8 16.6 21.0 16.4  7.8 14.0

> GNP[]

 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0
```

Note that the index or indices of the element(s) to be selected is placed inside square brackets. So `GNP[4]` selects the fourth element of `GNP`, while `GNP[1:5]` selects the first through fifth elements of `GNP`. The expression inside the brackets can be simple or complex. If a negative sign is put in front of the expression, R removes the specified elements. So `GNP[-4]` removes the fourth element, and returns the others. Note also that if there is nothing inside the square brackets, then R returns all the elements of the object.

Next is an example using a matrix.

```
> euromat

      [,1] [,2]
 [1,] 16.8  2.7
 [2,] 21.3  5.7
 [3,] 18.7  3.5
 [4,]  5.9 22.2
 [5,] 11.4 10.9
 [6,] 17.8  6.0
 [7,] 10.9 14.0
 [8,] 16.6  8.5
 [9,] 21.0  3.5
[10,] 16.4  4.3
[11,]  7.8 17.4
[12,] 14.0  2.3

> euromat[1, 2]

[1] 2.7

> euromat[1:5, 2]

[1]  2.7  5.7  3.5 22.2 10.9

> euromat[1, ]

[1] 16.8  2.7

> euromat[1, -2]

[1] 16.8

> euromat[, 1]

 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0
```

Note that in all the cases above, the object returned is just a vector.

Next, an example using a data frame:

```
> euro.data

   countries percapGNP percAGRIC grouping
1         BE      16.8       2.7        2
2         DK      21.3       5.7        2
3         DE      18.7       3.5        2
4         GR       5.9      22.2        1
5         ES      11.4      10.9        1
6         FR      17.8       6.0        2
7         IE      10.9      14.0        1
8         IT      16.6       8.5        2
9         LU      21.0       3.5        2
10        NL      16.4       4.3        2
11        PT       7.8      17.4        1
12        UK      14.0       2.3        2

> euro.data[1, 3]

[1] 2.7

> euro.data[1:4, 1:2]

  countries percapGNP
1        BE      16.8
2        DK      21.3
3        DE      18.7
4        GR       5.9

> euro.data[, 1:2]

   countries percapGNP
1         BE      16.8
2         DK      21.3
3         DE      18.7
4         GR       5.9
5         ES      11.4
6         FR      17.8
7         IE      10.9
8         IT      16.6
9         LU      21.0
10        NL      16.4
11        PT       7.8
12        UK      14.0

> euro.data[-c(1, 3, 5, 7), ]
```

```
   countries percapGNP percAGRIC grouping
2         DK      21.3       5.7        2
4         GR       5.9      22.2        1
6         FR      17.8       6.0        2
8         IT      16.6       8.5        2
9         LU      21.0       3.5        2
10        NL      16.4       4.3        2
11        PT       7.8      17.4        1
12        UK      14.0       2.3        2
```

Of course, especially for a data set with many variables, we shouldn't have to remember the column number of each variable. R also allows referring to the columns by name.

```
> euro.data["percapGNP"]

   percapGNP
1       16.8
2       21.3
3       18.7
4        5.9
5       11.4
6       17.8
7       10.9
8       16.6
9       21.0
10      16.4
11       7.8
12      14.0

> euro.data[c("percapGNP", "countries")]

   percapGNP countries
1       16.8        BE
2       21.3        DK
3       18.7        DE
4        5.9        GR
5       11.4        ES
6       17.8        FR
7       10.9        IE
8       16.6        IT
9       21.0        LU
10      16.4        NL
11       7.8        PT
12      14.0        UK
```

```
> euro.data[c("percapGNP", "countries")][1:3, ]

  percapGNP countries
1      16.8        BE
2      21.3        DK
3      18.7        DE
```

In addition, the "dollar sign operator" extracts a vector from a data frame:

```
> euro.data$percapGNP

 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0

> euro.data$countries

 [1] BE DK DE GR ES FR IE IT LU NL PT UK
Levels: BE DE DK ES FR GR IE IT LU NL PT UK

> cbind(euro.data$percapGNP, euro.data$countries)

      [,1] [,2]
 [1,] 16.8    1
 [2,] 21.3    3
 [3,] 18.7    2
 [4,]  5.9    6
 [5,] 11.4    4
 [6,] 17.8    5
 [7,] 10.9    7
 [8,] 16.6    8
 [9,] 21.0    9
[10,] 16.4   10
[11,]  7.8   11
[12,] 14.0   12
```

Note that there are subtle differences. For example, using `euro.data[c("percapGNP", "countries")]` returns a data frame with the two specified columns, while `cbind(euro.data$percapGNP, euro.data$countries)` returns a matrix.

3.2 Logical vectors and subsetting

Vectors can have "logical" data taking the values `FALSE` and `TRUE`:

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

[1]  TRUE FALSE FALSE  TRUE FALSE
```

```
> c(1, 2, 3, 4, 5) == c(2, 2, 3, 5, 4)

[1] FALSE  TRUE  TRUE FALSE FALSE

> bigAGRIC = AGRIC > 3.5
> bigAGRIC

 [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
[12] FALSE
```

Note that the comparison operators such as `<` compare each element of the first vector to the corresponding element of the second vector. Note that to test for equality, the operator `==` (two equal signs in a row) is used. In the statement `bigAGRIC = AGRIC > 3.5` the object to the right of the greater than sign is a vector of length one, while the object to the left is a vector of length 12. In this case R "recycles" the object of smaller length, comparing each element of `AGRIC` to `3.5`. In the next example a vector of length 10 is compared to a vector of length 2. The two elements of the smaller vector are recycled 5 times, leading to the same result as comparing the vector on the left to `c(1,2,1,2,1,2,1,2,1,2)`

```
> c(1, 2, 1, 4, 2, 3, 4, 2, 1, 3) > c(1, 2)

 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
```

Using logical vectors for comparisons can be very powerful. For example, consider the following data on 54 types of hot dogs, and their calorie and sodium contents. The data are stored on a web server, and the first command below reads the data into R. (We'll learn more about the `read.table()` function below.)

```
> hotdogs = read.table("http://www.stt.msu.edu/~melfi/cstat/hotdogs.txt",
+     header = T)
> hotdogs[1:5, ]

  Type Calories Sodium
1 Beef      186    495
2 Beef      181    477
3 Beef      176    425
4 Beef      149    322
5 Beef      184    482
```

It may be of interest to compare the calorie content of high-sodium hot dogs to the calorie content of lower sodium hot dogs.

```
> hotdogs$Calories[hotdogs$Sodium < 400]

 [1] 149 158 139 148 152 111 141 131 149 135 132 147 146 139 136 153
[17] 107 138 132 102 106  94  87  99  86
```

```
> hotdogs$Calories[hotdogs$Sodium >= 400]

 [1] 186 181 176 184 190 175 153 190 157 173 191 182 190 172 175 179
[17] 195 135 140 129 102 107 113 135 142 143 152 146 144

> median(hotdogs$Calories[hotdogs$Sodium < 400])

[1] 136

> median(hotdogs$Calories[hotdogs$Sodium >= 400])

[1] 172
```

Related to this, the `cut()` function can create a factor variable that classifies another variable based on its magnitude. For example, we can classify the hot dogs by sodium content into low (0-350 mg), medium (350-500 mg), and high (500 and above mg) sodium content as follows:

```
> cut(hotdogs$Sodium, breaks = c(0, 350, 500, 700), labels = c("low",
+     "medium", "high"))

 [1] medium medium medium low    medium high   medium low    medium
[10] medium low    low    medium medium high   medium low    low
[19] low    low    medium high   medium high   medium medium medium
[28] medium high   medium medium medium low    high   medium medium
[37] low    medium medium medium medium medium high   medium medium
[46] high   high   medium high   medium high   high   high   high
Levels: low medium high
```

By default the `cut()` function does not include the left endpoint of the interval, so for example "medium" hot dogs have more than 350 and at most 500 mg of sodium. It would be more useful for the factor variable to be part of the data frame, as follows:

```
> hotdogs$sodlevel = cut(hotdogs$Sodium, breaks = c(0,
+     350, 500, 700), labels = c("low", "medium", "high"))
> hotdogs[1:10, ]

   Type Calories Sodium sodlevel
1  Beef      186    495   medium
2  Beef      181    477   medium
3  Beef      176    425   medium
4  Beef      149    322      low
5  Beef      184    482   medium
6  Beef      190    587     high
7  Beef      158    370   medium
8  Beef      139    322      low
9  Beef      175    479   medium
10 Beef      148    375   medium
```

3.3 Sorting data*

It is also quite common to want to sort data. For a vector this is trivial, but it is a bit more complex for a data frame. Consider a (fictitious) data set on heights and weights of 10 men.

```
> htwt = read.table("http://www.stt.msu.edu/~melfi/cstat/htwt.txt",
+     header = T, stringsAsFactors = F)
> htwt

   subj.names hts wts
1        John  71 162
2       James  72 156
3       Allen  66 145
4       Leroy  70 148
5     Zebulon  72 161
6      Egbert  72 165
7        Eric  67 155
8      Anders  68 154
9       Wayne  72 149
10      Mario  65 145
```

To order the data in increasing order of heights, we would want the 10th case (Mario) first, then the third case (Allen), then the eigth case (Anders), and so on. We could do this "by hand." Note that for those whose heights are tied at 72, we left them in the original order.

```
> htwt.ordered = htwt[c(10, 3, 7, 8, 4, 1, 2, 5, 6, 9),
+     ]
> htwt.ordered

   subj.names hts wts
10      Mario  65 145
3       Allen  66 145
7        Eric  67 155
8      Anders  68 154
4       Leroy  70 148
1        John  71 162
2       James  72 156
5     Zebulon  72 161
6      Egbert  72 165
9       Wayne  72 149
```

Clearly we don't want to do this by hand in general. The `order()` function does the hard work for us:

```
> order(htwt$hts)
```

```
 [1] 10  3  7  8  4  1  2  5  6  9
```

This tells us that the 10th case has the smallest height, followed by the third, then the seventh, then the eighth, and so on. We can use this to reorder the whole dataset:

```
> htwt.ordered = htwt[order(htwt$hts), ]
> htwt.ordered

   subj.names hts wts
10      Mario  65 145
3       Allen  66 145
7        Eric  67 155
8      Anders  68 154
4       Leroy  70 148
1        John  71 162
2       James  72 156
5     Zebulon  72 161
6      Egbert  72 165
9       Wayne  72 149
```

In this case there are ties in the heights, so it makes sense to specify a second variable to be used to break the ties. For the ties, it might be better to order these based on the weights. We can do this by specifying a second argument to `order()`:

```
> order(htwt$hts, htwt$wts)

 [1] 10  3  7  8  4  1  9  2  5  6

> htwt.ordered = htwt[order(htwt$hts, htwt$wts), ]
> htwt.ordered

   subj.names hts wts
10      Mario  65 145
3       Allen  66 145
7        Eric  67 155
8      Anders  68 154
4       Leroy  70 148
1        John  71 162
9       Wayne  72 149
2       James  72 156
5     Zebulon  72 161
6      Egbert  72 165
```

3.4 Adding to a data frame*

Quite commonly data frames must be augmented either with more cases (i.e, more rows) or more variables (i.e., more columns). Either is easy to accomplish, but some care must be taken. Consider the `htwt` data frame again, and suppose that another subject, Roger, has entered the study, and that his height is 70 inches and his weight is 179 pounds. His data can be added to the data frame either via `rbind()` or via indexing. We'll use indexing. We might think that `htwt[11,] = c("Roger", 70, 179)` would be a good idea. But remember that elements of vectors all have to be of the same type, so in this case the numbers would be converted to be characters, and the whole data frame would become character. So we need to do the following:

```
> htwt[11, 1] = "Roger"
> htwt[11, 2:3] = c(70, 179)
> htwt
```

```
   subj.names hts wts
1        John  71 162
2       James  72 156
3       Allen  66 145
4       Leroy  70 148
5     Zebulon  72 161
6      Egbert  72 165
7        Eric  67 155
8      Anders  68 154
9       Wayne  72 149
10      Mario  65 145
11      Roger  70 179
```

If we wanted to use `rbind()` we would just write `htwt = rbind(htwt, data.frame(subj.names = "Roger", hts = 70, wts = 179))`.

Several methods, including use of `cbind()`, can add variables to a data frame. We'll use what seems to be the most natural method here. Suppose we want to add a column giving the body mass index (BMI) for each person in the `htwt` data frame. The BMI is defined as

$$BMI = (wt \times 703)/(ht^2),$$

where weight is measured in pounds and height in inches.

```
> htwt$BMI = (htwt$wts * 703)/(htwt$hts * htwt$hts)
> htwt
```

```
   subj.names hts wts      BMI
1        John  71 162 22.59195
2       James  72 156 21.15509
3       Allen  66 145 23.40106
```

```
4       Leroy  70 148 21.23347
5     Zebulon  72 161 21.83314
6      Egbert  72 165 22.37558
7        Eric  67 155 24.27378
8      Anders  68 154 23.41306
9       Wayne  72 149 20.20583
10      Mario  65 145 24.12663
11      Roger  70 179 25.68102
```

3.5 Mathematical functions*

Of course R has a full range of mathematical functions including basics like addition, subtraction, logarithms, trigonometric functions, as well as functions for computing minimums, maximums, sums, cumulative sums, and the like. It would take many pages to list all the mathematical functions built in to R.

Many R functions expect vectors as arguments. Consider the following examples:

```
> GNP + AGRIC

 [1] 19.5 27.0 22.2 28.1 22.3 23.8 24.9 25.1 24.5 20.7 25.2 16.3

> GNP * AGRIC

 [1]  45.36 121.41  65.45 130.98 124.26 106.80 152.60 141.10  73.50
[10]  70.52 135.72  32.20

> GNP/AGRIC

 [1] 6.2222222 3.7368421 5.3428571 0.2657658 1.0458716 2.9666667
 [7] 0.7785714 1.9529412 6.0000000 3.8139535 0.4482759 6.0869565

> log(GNP)

 [1] 2.821379 3.058707 2.928524 1.774952 2.433613 2.879198 2.388763
 [8] 2.809403 3.044522 2.797281 2.054124 2.639057

> sum(GNP)

[1] 178.6

> min(GNP)

[1] 5.9

> cumsum(GNP)

 [1]  16.8  38.1  56.8  62.7  74.1  91.9 102.8 119.4 140.4 156.8 164.6
[12] 178.6

> cumprod(GNP)

 [1] 1.680000e+01 3.578400e+02 6.691608e+03 3.948049e+04 4.500776e+05
 [6] 8.011380e+06 8.732405e+07 1.449579e+09 3.044116e+10 4.992351e+11
[11] 3.894034e+12 5.451647e+13
```

4 Data import and export*

If pressed for time, all of Section 4 may be omitted at first reading.

Importing data into R and exporting data from R are covered in Chapter 2 of Spector [1] and in R development core team [4]. We will focus on `read.table()` for getting data into R and `write.table()` for exporting data. Of course there are other methods that are discussed in the two references, and there are also specialized methods developed for particular types of data. As an example, the bioconductor project includes a large number of functions for reading in biological data such as data from gene expression microarray studies.

4.1 Reading data into R using read.table*

The `read.table()` function is the workhorse for reading data into R. We have used it above to read some data sets into R, and now will look more carefully at how it works. At its simplest, the user need only specify the name of the "file" containing the data. Note that the file may be specified, for example, via a URL. The function `read.table()` always creates a data frame. Consider the hot dog data set. It is contained in a file called `hotdogs.txt`, and the first few lines of the file look like this:

```
Type Calories Sodium
Beef 186 495
Beef 181 477
Beef 176 425
Beef 149 322
Beef 184 482
```

The top line of the file contains the names of the variables, and each subsequent line coontains the value of each variable for one observation. Values are separated by spaces. We used the following command to read the data into a data frame called `hotdogs`.

```
> hotdogs = read.table("http://www.stt.msu.edu/~melfi/cstat/hotdogs.txt",
+     header = T)
> hotdogs[1:3, ]

  Type Calories Sodium
1 Beef      186    495
2 Beef      181    477
3 Beef      176    425
```

The second argument, `header=T`, tells R that the first line of the file contains variable names. The first argument specifies the location of the file, in this case a url pointing to a file on a web server.

Consider now the file `census1.txt` which is on the same web server. Here are the first few lines of that file:

```
stateno,state,region,pop,poplt5,pop5_17,pop18p,pop65p
2,Alaska,West,401851,38949,91796,271106,11547
50,Wyoming,West,469557,44845,100708,324004,37175
45,Vermont,NE,511456,35998,109320,366138,58166
8,Delaware,South,594338,41151,125444,427743,59179
```

Again the first line contains variable names and the subsequent lines contain variable values, but this time the values are separated by commas rather than spaces. So we add the `sep=","` argument to the command.

```
> census1 = read.table("http://www.stt.msu.edu/~melfi/cstat/census1.txt",
+     header = T, sep = ",")
> census1[1:3, ]
```

```
  stateno   state region    pop poplt5 pop5_17 pop18p pop65p
1       2  Alaska   West 401851  38949   91796 271106  11547
2      50 Wyoming   West 469557  44845  100708 324004  37175
3      45 Vermont     NE 511456  35998  109320 366138  58166
```

Sometimes files have other information at the top before the variable names and data begin. This can be handled by the `skip` argument, which is set to the number of lines to be skipped. (The default is `skip=0`.) There are many other arguments to allow the reading of a variety of file formats. Consult the help for `read.table()` for details. By default `read.table()` converts character variables into factors. Usually this doesn't present problems, but sometimes users do not want this behavior, so there is an argument to override this default. Here is an example:

```
> htwt = read.table("http://www.stt.msu.edu/~melfi/cstat/htwt.txt",
+     header = T, stringsAsFactors = F)
> htwt[1:3, ]
```

```
  subj.names hts wts
1       John  71 162
2      James  72 156
3      Allen  66 145
```

The `stringsAsFactors=F` argument tells R not to convert all character variables into factors.

4.2 Data export via write.table()*

The function `write.table()` is the export counterpart to `read.table()`. It can be used to export data in a format that other statistical packages can read. Let's first use it to write the `euro.data` data frame to a file.

```
> euro.data[1:3, ]
```

```
  countries percapGNP percAGRIC grouping
1        BE      16.8       2.7        2
2        DK      21.3       5.7        2
3        DE      18.7       3.5        2
```

```
> write.table(euro.data, "euro.data.r", sep = " ")
```

This will write the data from the data frame into a file called `euro.data.r` in the working directory of the R session. Of course other locations can be specified. Here are the first few lines of the resulting file:

```
"countries" "percapGNP" "percAGRIC" "grouping"
"1" "BE" 16.8 2.7 "2"
"2" "DK" 21.3 5.7 "2"
"3" "DE" 18.7 3.5 "2"
```

There is a surprise. The first line has the four variable names. But each subsequent row has a number in quotes before the first variable value. By default `write.table()` writes out row names. In this case, the row names are just the row numbers. Often we don't want this, so we need to specify `row.names=F`.

The other commonly used argument is `sep`. For example, it's reasonably easy to read comma-delimited data into Microsoft Excel. To export a version of `euro.data` for input to Excel, the following would work:

```
> write.table(euro.data, "euro.data.csv", sep = ",", row.names = F)
```

Here are the first few lines of the resulting file.

```
"countries","percapGNP","percAGRIC","grouping"
"BE",16.8,2.7,"2"
"DK",21.3,5.7,"2"
"DE",18.7,3.5,"2"
```

4.3 Direct import and export*

There is an R library called `foreign` which contains functions for importing and exporting directly from and to other statistical packages. For example, it contains a function `read.mtp` that reads Minitab portable worksheets. In most cases it is better to use text-based formats such as comma-delimited files for import and export. Indeed the R Data Import/Export manual in several places gives advice to avoid trying to read and write directly from and to other statistical packages.

5 Statistical Analyses

Of course it is possible to perform a wide variety of statistical analyses in R, either using the default packages, or additional packages installed by the user. These range

from the very simple and common, such as computing summary statistics like means, medians, standard deviations, and the like, to more specialized and computationally intensive analyses such as clustering.

5.1 Linear Regression Models

We'll focus on linear regression models. There is a data set built in to R that has information on population, per capita income, illiteracy rate, life expectancy, murder rate, percent of high school graduates, mean number of days with minimum temperature below freezing, and land area for the 50 states in the United States. The data are mainly from the early 1970s. First read in the data and convert it to a data frame (it's a matrix when read in). The command `data(state)` reads in several data sets. The one we'll use is called `state.x77`.

```
> data(state)
> state.x77 = data.frame(state.x77)
> state.x77[1, ]

        Population Income Illiteracy Life.Exp Murder HS.Grad Frost
Alabama       3615   3624        2.1    69.05   15.1    41.3    20
         Area
Alabama 50708
```

The `lm()` function is used to fit linear regression models. Here's how to fit a simple linear regression model with `Life.Exp` as the response variable and `HS.Grad` as the predictor.

```
> state.lm.1 = lm(Life.Exp ~ HS.Grad, data = state.x77)
> state.lm.1

Call:
lm(formula = Life.Exp ~ HS.Grad, data = state.x77)

Coefficients:
(Intercept)      HS.Grad
   65.73965      0.09676
```

A wide variety of models can be specified. The simplest, with one predictor and one response, is specified as `response ~ predictor`. Some other possibilities include

- `response ~ predictor - 1` for a simple linear regression model through the origin.
- `response ~ predictor1 + predictor2 + predictor3` for the usual multiple regression model with 3 predictors.
- `log(response) ~ predictor` for a model with log-transformed response.

See `help(formula)` for a general description of formulas for specifying models. The same syntax is used whether specifying linear regression models, generalized linear models, analysis of variance models, etc.

The linear model object contains a wealth of information. Here are a few ways to extract some of this information. See `help(lm)` for more details on the quantities computed when an `lm` object is created.

```
> summary(state.lm.1)

Call:
lm(formula = Life.Exp ~ HS.Grad, data = state.x77)

Residuals:
     Min       1Q   Median       3Q      Max
-3.01867 -0.67517 -0.07538  0.64483  2.17311

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 65.73965    1.04748  62.760  < 2e-16 ***
HS.Grad      0.09676    0.01950   4.961  9.2e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.103 on 48 degrees of freedom
Multiple R-squared: 0.339,        Adjusted R-squared: 0.3252
F-statistic: 24.61 on 1 and 48 DF,  p-value: 9.196e-06

> anova(state.lm.1)

Analysis of Variance Table

Response: Life.Exp
          Df Sum Sq Mean Sq F value    Pr(>F)
HS.Grad    1 29.931  29.931  24.615 9.196e-06 ***
Residuals 48 58.368   1.216
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> resid(state.lm.1)

       Alabama         Alaska        Arizona       Arkansas
   -0.68600972    -2.88381740    -0.81164630     1.05945999
    California       Colorado    Connecticut       Delaware
   -0.08708466     0.13712203     1.32155828    -0.96297201
       Florida        Georgia         Hawaii          Idaho
```

```
   -0.16944385    -1.12827487     1.87065019     0.37288399
      Illinois        Indiana           Iowa         Kansas
   -0.68944385     0.02152693     1.11126603     1.04417836
      Kentucky      Louisiana          Maine       Maryland
    0.63492971    -1.06309740    -0.64264842    -0.58041462
 Massachusetts       Michigan      Minnesota    Mississippi
    0.42964807    -0.21879666     1.64673574    -1.61698050
      Missouri        Montana       Nebraska        Nevada
    0.22825966    -0.90808679     1.12223681    -3.01867128
 New Hampshire     New Jersey     New Mexico       New York
   -0.08326426     0.11023256    -0.76103046    -0.28912025
North Carolina   North Dakota           Ohio       Oklahoma
   -0.25507029     2.17311354    -0.06750230     0.68732024
       Oregon   Pennsylvania   Rhode Island South Carolina
    0.58450195    -0.16721005     1.67049346    -1.43733544
  South Dakota      Tennessee          Texas          Utah
    1.18282130     0.32560824     0.57372938     0.64812415
       Vermont       Virginia     Washington  West Virginia
    0.37511779    -0.28497625    -0.16417234    -0.28503895
     Wisconsin        Wyoming
    1.46670440    -1.53611389
```

The fitted line is also easy to plot. The resulting plot is given in Figure 3.

```
> plot(state.x77$HS.Grad, state.x77$Life.Exp, xlab = "high school grad. rate",
+     ylab = "life expectancy")
> abline(state.lm.1)
```

5.2 Inference in regression

Of course it is reasonably easy to obtain confidence intervals and perform tests of hypotheses in the context of regression in R. We'll focus on hypothesis tests here. This will be done in the context of the `state.x77` dataset.

5.2.1 Model comparison two ways

Consider two models. The first includes `HS.Grad` and `Murder` as predictors, while the second includes `Income`, `Frost`, `HS.Grad`, and `Murder` as predictors. Note that the smaller model is a "submodel" of the larger, that is, it is obtained from the larger by eliminating some predictors. This is important for the F testing methodology. The F statistic for testing the null hypothesis that the smaller model is correct versus the alternative hypothesis that the larger model is correct is given by

$$F = \frac{(RSS_{small} - RSS_{large})/(df_{small} - df_{large})}{RSS_{large}/df_{large}}.$$

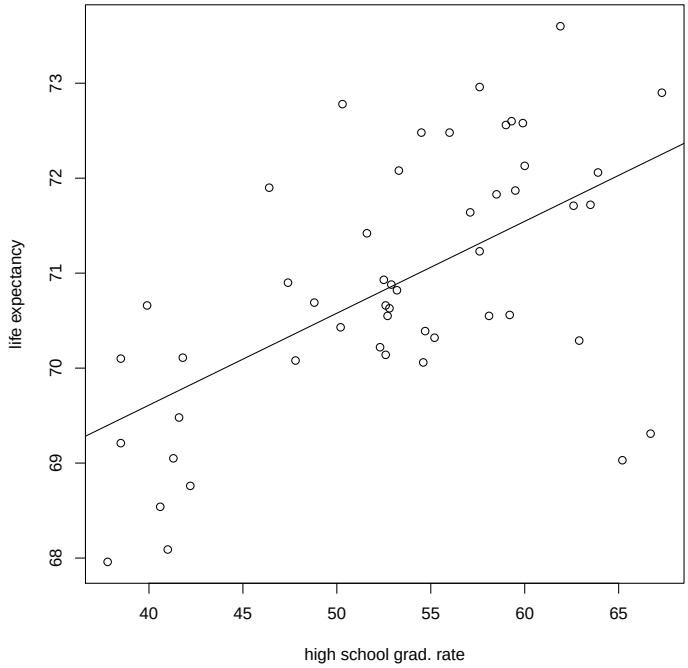

Figure 3: A plot of the `state.x77` data along with the fitted least squares regression line

This has a null $F(df_{small} - df_{large}, df_{large})$ distribution. First we fit the two models:

```
> state.lm.small = lm(Life.Exp ~ HS.Grad + Murder, data = state.x77)
> state.lm.large = lm(Life.Exp ~ HS.Grad + Murder + Income +
+     Frost, data = state.x77)
```

Now we compute the residual sums of squares for the two models, and the F statistic, and compute the p-value. The `resid()` function returns the residuals from the model.

```
> RSS.small = sum(resid(state.lm.small) * resid(state.lm.small))
> RSS.large = sum(resid(state.lm.large) * resid(state.lm.large))
> RSS.small

[1] 29.77036

> RSS.large

[1] 25.1893

> Fstat = ((RSS.small - RSS.large)/(47 - 45))/(RSS.large/45)
> Fstat

[1] 4.091965
```

```
> 1 - pf(Fstat, 2, 45)

[1] 0.02329281
```

The F statistic of 4.092 and the resulting p-value of 0.0233 provide reasonably strong evidence against the null hypothesis.

We've computed the various quantities "by hand" above, but the `anova()` function does a lot of the work for us.

```
> anova(state.lm.small, state.lm.large)

Analysis of Variance Table

Model 1: Life.Exp ~ HS.Grad + Murder
Model 2: Life.Exp ~ HS.Grad + Murder + Income + Frost
  Res.Df    RSS Df Sum of Sq     F  Pr(>F)
1     47 29.770
2     45 25.189  2    4.5811 4.092 0.02329 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

5.3 More on model formulas and their use in R*

Specifying models using the `response ~ predictor` notation is useful not only for linear regression models, but more generally for a wide variety of models in R, including generalized linear models. So next we'll gain a bit more comfort with model specification. This section takes much from the presentation the "Stem and Tendril" project at the City University of New York. See `http://wiener.math.csi.cuny.edu/st/` and `http://wiener.math.csi.cuny.edu/st/stRmanual/ModelFormula.html`.

As we have seen, generally a model is specified in R as `response ~ predictors`. For example

```
y ~ x1 + x2 + x3
```

specifies a model where the response is `y` and there are three predictors, `x1, x2,` and `x3`. Here are some more details on adding `predictors` to the model formula:

- `+` adds a predictor to a model. For example, `y ~ x1 + x2` adds a second predictor `x2` to a model with one predictor `x1`.
- `-` removes a predictor from a model.
- `:` is used for interactions between terms.
- `*` adds terms and their interactions. For example, `y ~ x1 + x2 + x1:x2` and `y ~ x1*x2` specify the same model.

- `I()` is used to "insulate" the arguments from the formula. For example, suppose a model is desired with `y` as the response and `x1` and `x2 + x3` as the two predictors. The model specification `y ~ x1 + x2 + x3` isn't correct, since this includes `x1, x2,` and `x3` as predictors. The correct model specification is `y ~ x1 + I(x2 + x3)`.
- The `data =` argument can be used to specify the data frame in which the variables are stored.
- The `subset =` argument can restrict the model to a subset of the cases (rows) in the data frame.

Some examples come next. These use the `mtcars` data set, available within R, which has data on miles per gallon (`mpg`), number of cylinders (`cyl`), horsepower (`hp`), weight (`wt`), automatic or manual transmission (`am`), number of forward gears (`gear`), among others. The data come from *Motor Trend* magazine for 1973-1974 model cars.

```
> data(mtcars)
> names(mtcars)

 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
[10] "gear" "carb"

> mtcars[1, ]

          mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

First, here's how to fit a linear model with `mpg` as response and `wt` as predictor. There's nothing new here.

```
> mpg.wt.lm = lm(mpg ~ wt, data = mtcars)
> anova(mpg.wt.lm)

Analysis of Variance Table

Response: mpg
          Df Sum Sq Mean Sq F value    Pr(>F)
wt         1 847.73  847.73  91.375 1.294e-10 ***
Residuals 30 278.32    9.28
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Next, we can use the formula notation for scatterplots. The plot is shown in Figure 4.

```
> plot(mpg ~ wt, data = mtcars)
```

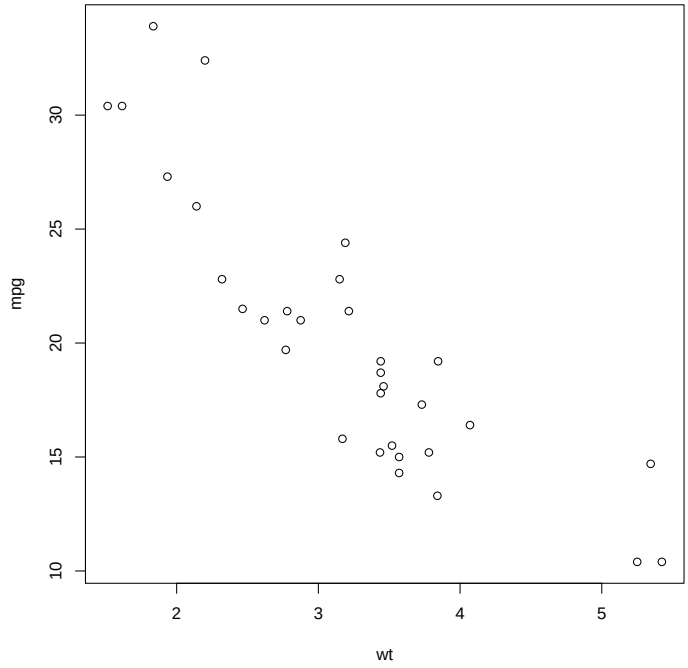

Figure 4: Plotting `mpg` versus `wt` for the `mtcars` data.

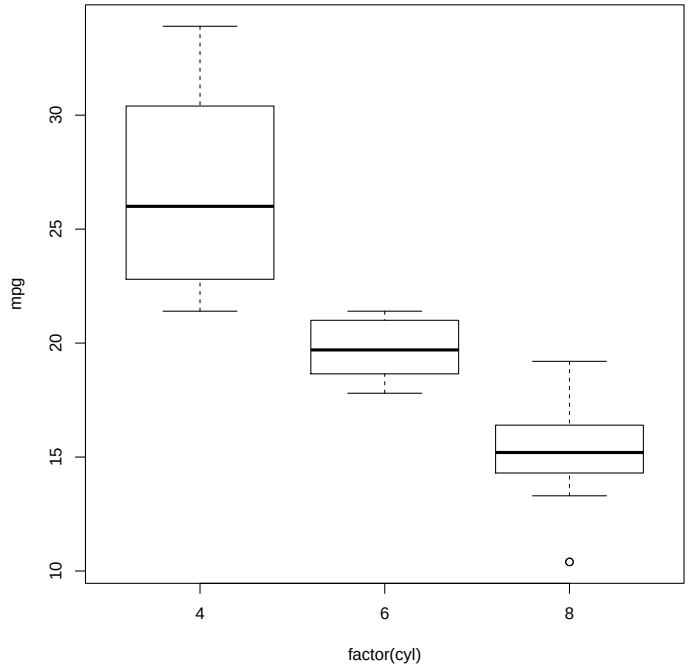

Figure 5: Boxplots of `mpg` for the various values of `cyl` for the `mtcars` data.

Next, separate boxplots of `mpg` for each value of `cyl`. We use the `factor()` function

to be sure that `cyl` is treated as a factor. The plot is shown in Figure 5.

```
> plot(mpg ~ factor(cyl), data = mtcars)
```

Next an analysis of variance model to look at the effect of the number of cylinders on gas mileage is performed. Note that the `lm()` function can also be used to fit an analysis of variance model. The output of summary will be different. Try it yourself.

```
> mpg.cyl.aov = aov(mpg ~ factor(cyl), data = mtcars)
> summary(mpg.cyl.aov)

             Df Sum Sq Mean Sq F value    Pr(>F)
factor(cyl)   2 824.78  412.39  39.697 4.979e-09 ***
Residuals    29 301.26   10.39
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Look at `help(formula)` for details on specifying model formulas, and try some more examples with the `mtcars` data.

5.4 Another linear regression model*

As another example, consider the data set provided by the statistician Frank Anscombe, that illustrates the importance of visualizing data and not exclusively relying on summary statistics in the context of regression.

First we get access to the data, list it, and then plot the variable `y1` versus the variable `x1`. The plot produced is in Figure 6.

```
> data(anscombe)
> anscombe

   x1 x2 x3 x4    y1   y2    y3    y4
1  10 10 10  8  8.04 9.14  7.46  6.58
2   8  8  8  8  6.95 8.14  6.77  5.76
3  13 13 13  8  7.58 8.74 12.74  7.71
4   9  9  9  8  8.81 8.77  7.11  8.84
5  11 11 11  8  8.33 9.26  7.81  8.47
6  14 14 14  8  9.96 8.10  8.84  7.04
7   6  6  6  8  7.24 6.13  6.08  5.25
8   4  4  4 19  4.26 3.10  5.39 12.50
9  12 12 12  8 10.84 9.13  8.15  5.56
10  7  7  7  8  4.82 7.26  6.42  7.91
11  5  5  5  8  5.68 4.74  5.73  6.89

> plot(anscombe$x1, anscombe$y1, xlab = "x1", ylab = "y1",
+     main = "First Anscombe Data")
```

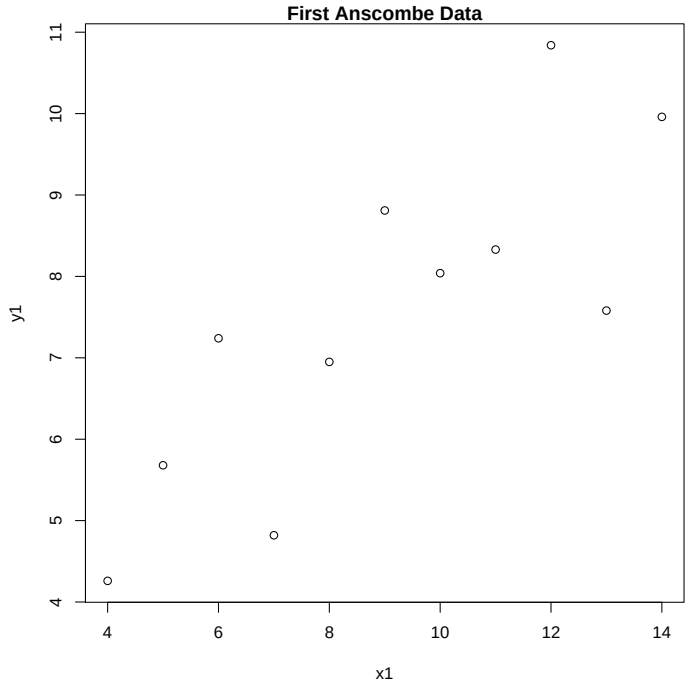

Figure 6: A plot of $y1$ versus $x1$ from the Anscombe data, with user-specified axis labels and main title

Now we'll fit linear regression models to `y1` versus `x1`, then `y2` versus `x2`, etc., and look at their fitted intercepts and slopes:

```
> lm1 = lm(y1 ~ x1, data = anscombe)
> lm2 = lm(y2 ~ x2, data = anscombe)
> lm3 = lm(y3 ~ x3, data = anscombe)
> lm4 = lm(y4 ~ x4, data = anscombe)
> coefficients(lm1)

(Intercept)          x1
  3.0000909   0.5000909

> coefficients(lm2)

(Intercept)          x2
   3.000909    0.500000

> coefficients(lm3)

(Intercept)          x3
  3.0024545   0.4997273

> coefficients(lm4)
```

```
(Intercept)          x4
  3.0017273   0.4999091
```

Note that the fitted regression lines for the four data sets have essentially the same slopes and intercepts. As plots will soon show, however, the data sets are rather different. The command `par(mfrow=c(2,2))` below tells R that we want four plots per page, in a 2 by 2 arrangement. The `abline` function adds the line from a fitted regression model to the plot. The plots are shown in Figure 7.

```
> par(mfrow = c(2, 2))
> plot(anscombe$x1, anscombe$y1)
> abline(lm1)
> plot(anscombe$x2, anscombe$y2)
> abline(lm2)
> plot(anscombe$x3, anscombe$y3)
> abline(lm3)
> plot(anscombe$x4, anscombe$y4)
> abline(lm4)
> par(mfrow = c(1, 1))
```

6 Basics of graphics in R

The graphics capabilities of R are quite powerful; indeed the graphics capabilities are a big reason for R's popularity. Most types of plots and charts that we can think of are already available in R, and if not, can probably be implemented. In addition the graphics are customizable.

6.1 Initial examples

R has many functions for creating plots, including `plot()`, `barplot()`, `stripchart()`, `boxplot()`, etc. A more complete list is given on pp. 40-41 of Paradis [2]. The `plot()` function is the workhorse of the bunch, and can produce a variety of plots depending on the input. Consider the European economic data again.

```
> euro.data
```

```
   countries percapGNP percAGRIC grouping
1         BE      16.8       2.7        2
2         DK      21.3       5.7        2
3         DE      18.7       3.5        2
4         GR       5.9      22.2        1
5         ES      11.4      10.9        1
6         FR      17.8       6.0        2
7         IE      10.9      14.0        1
```

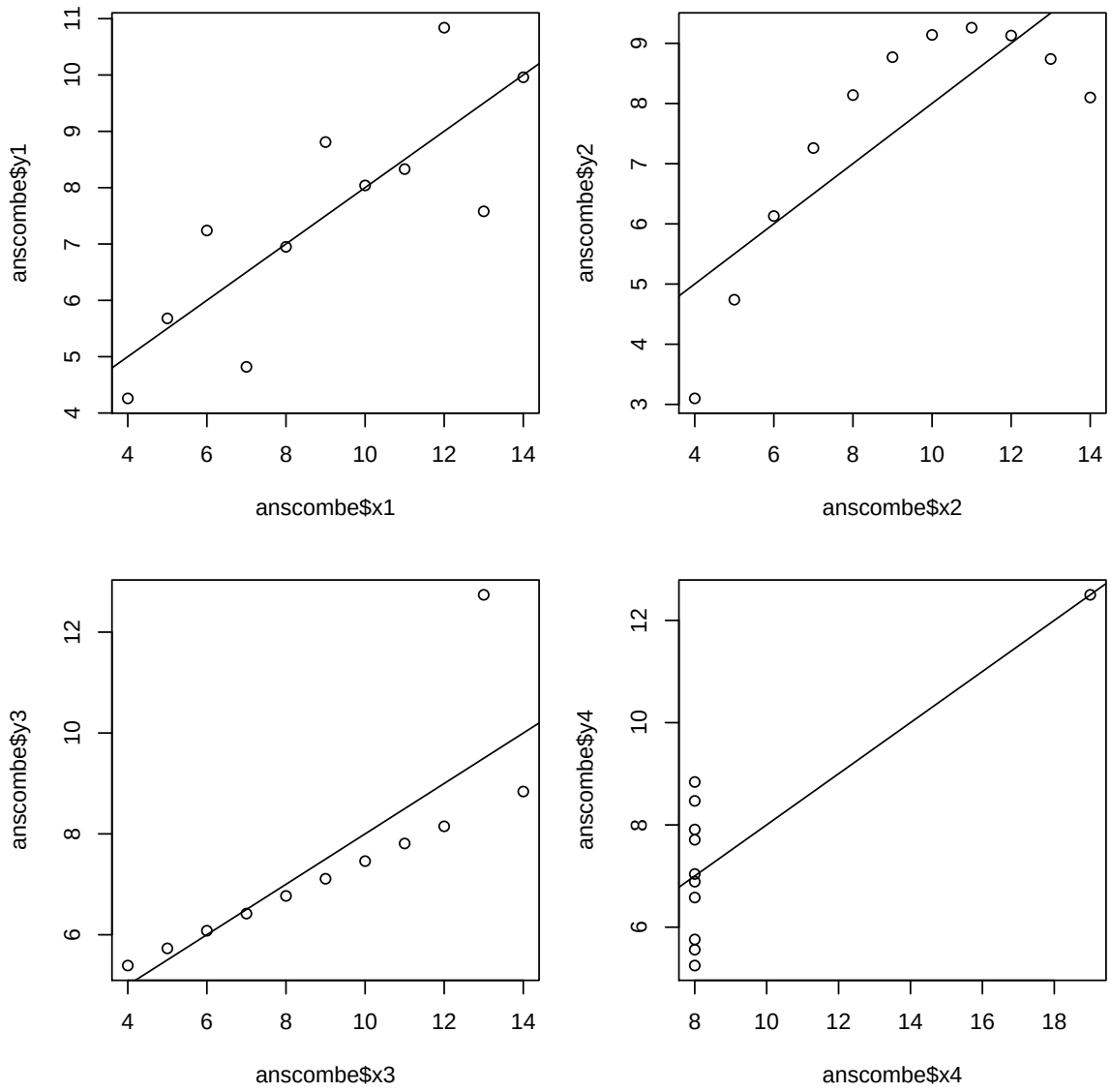

Figure 7: Plots of the four Anscombe data sets along with their fitted regression lines

```
8          IT      16.6       8.5        2
9          LU      21.0       3.5        2
10         NL      16.4       4.3        2
11         PT       7.8      17.4        1
12         UK      14.0       2.3        2
```

A simple scatter plot is produced by

```
> plot(euro.data$percAGRIC, euro.data$percapGNP)
```

See Figure 8. Now consider

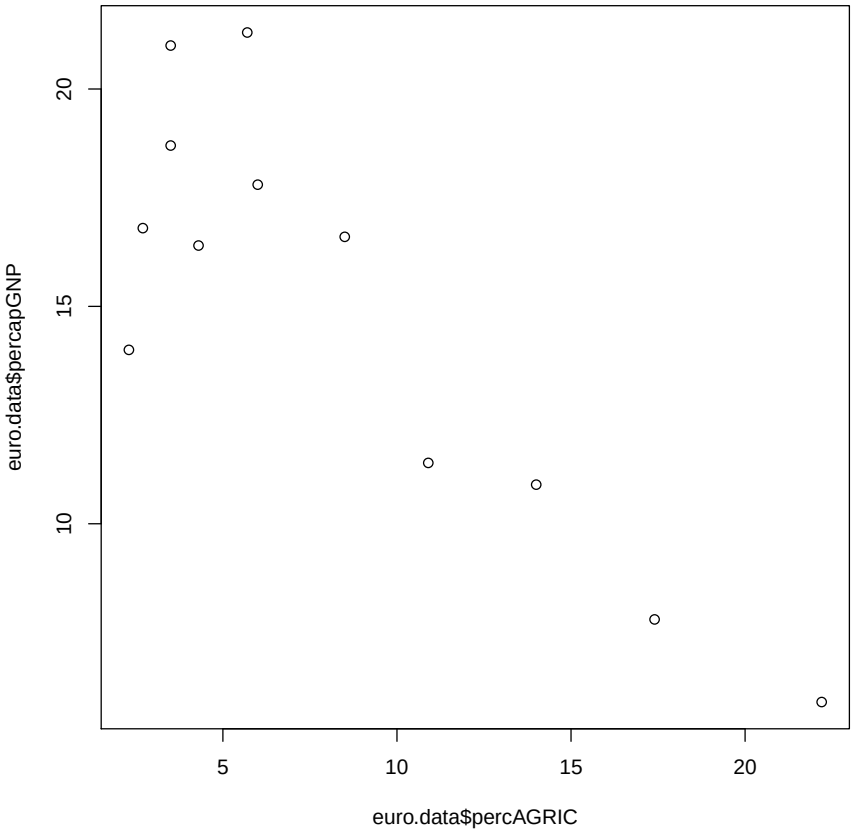

Figure 8: A scatter plot produced by the plot command

```
> plot(euro.data$grouping, euro.data$percapGNP)
```

Since `grouping` is a factor, R provides side by side boxplots of `percapGNP`, one for each level of `grouping`. See Figure 9. Next, fit a linear model to the data, and plot the

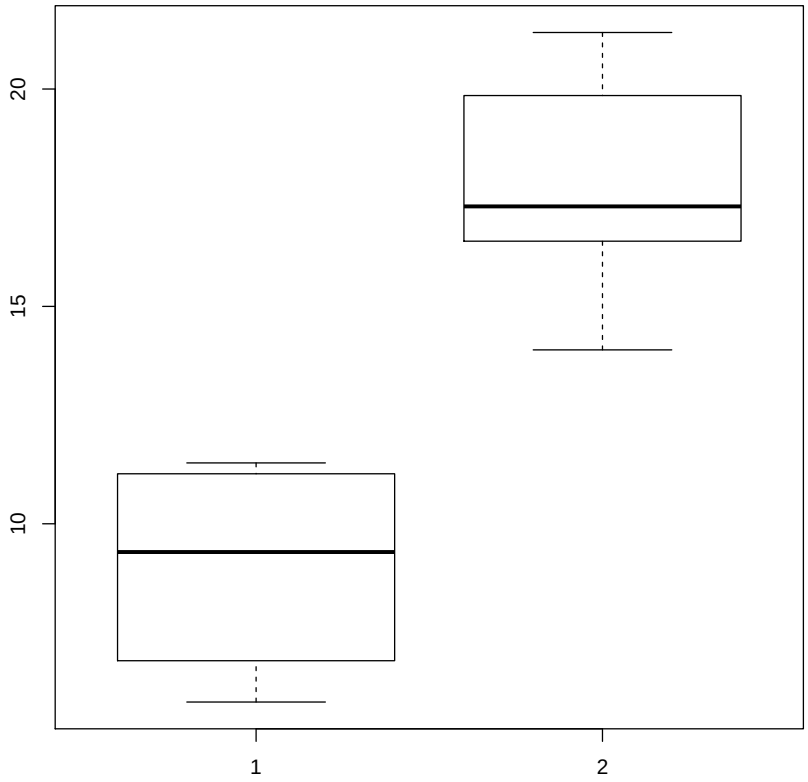

Figure 9: Side by side box plots produced by the plot command

resulting linear model:

```
> euro.lm = lm(percapGNP ~ percAGRIC, data = euro.data)
> plot(euro.lm)
```

This produces four plots relevant to linear models, shown in Figure 10.

7 High level graphics functions

It is useful to divide R graphics functions into *high-level* functions, which produce a complete plot, and *low-level* functions, which add features to an already existing plot. In this section we investigate customization of `plot()`, the fundamental high-level plotting function.[3] It is most instructive to do this via examples. We'll use the

[3]Much of what we learn about `plot()` applies to other high-level plotting functions.

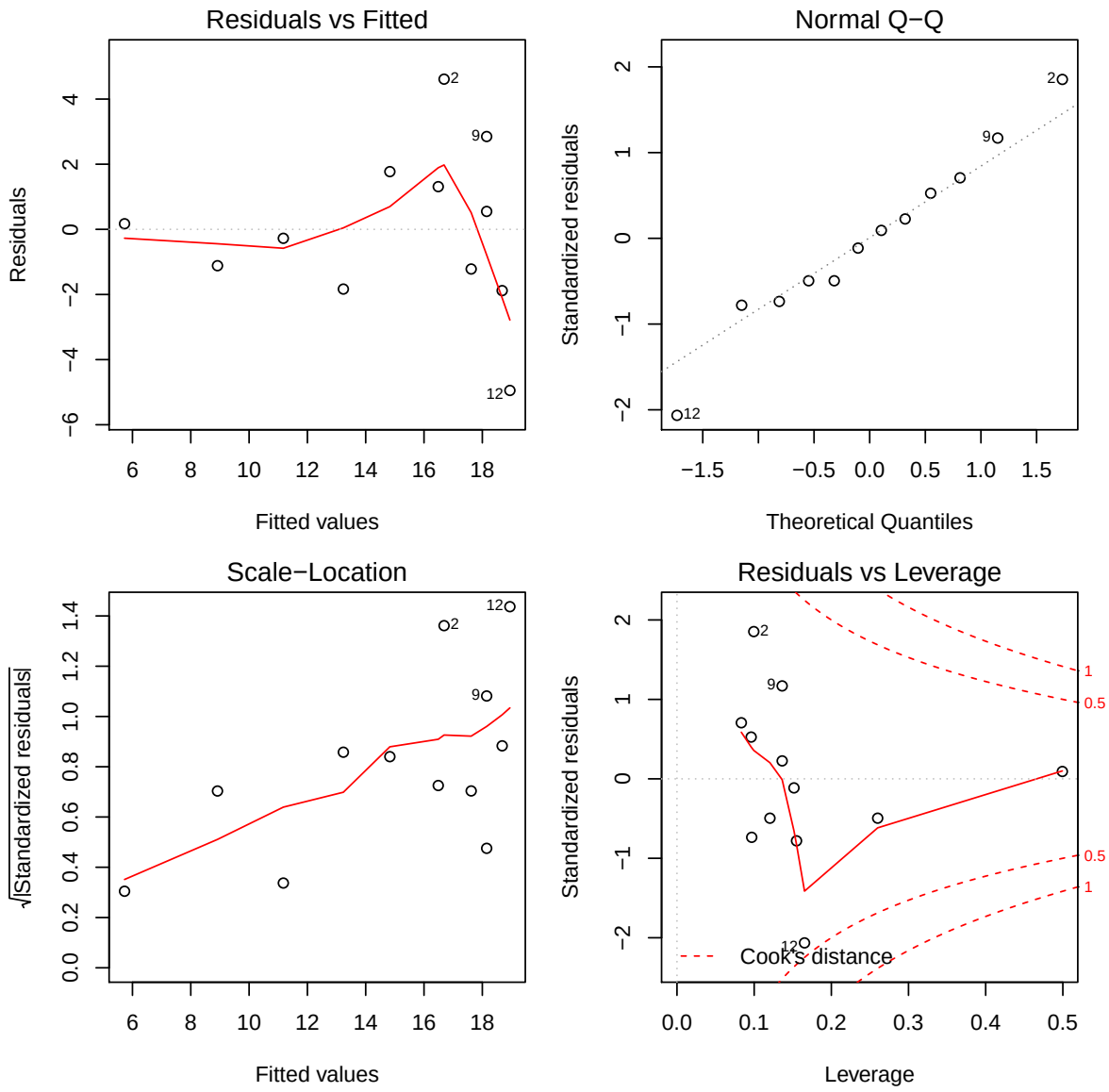

Figure 10: Plot of a linear model object produced by the plot command

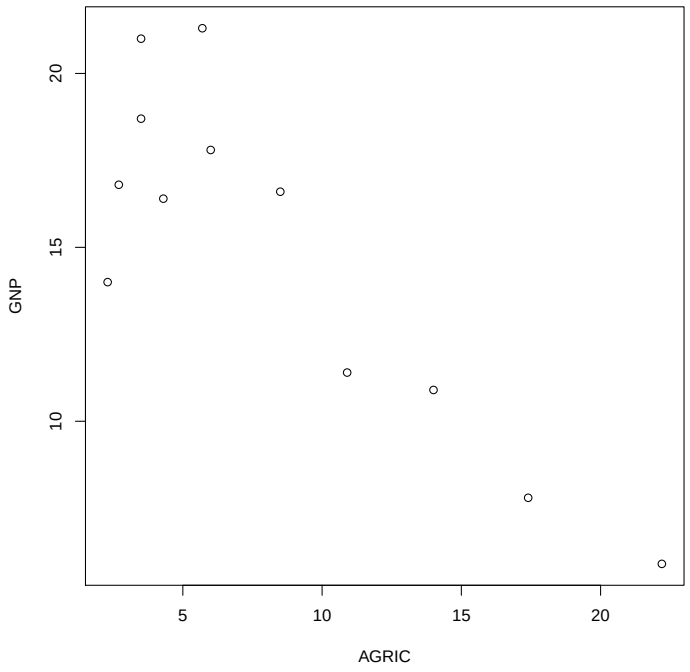

Figure 11: A scatter plot of `GNP` versus `AGRIC`, using the default settings

gross national product and percent of population working in agriculture variables from the European economic data set. The following code creates variables `AGRIC` and `GNP` (just to make them easier to refer to) and then creates the basic scatter plot, using the default settings, shown in Figure 11.

```
> AGRIC = euro.data$percAGRIC
> GNP = euro.data$percapGNP
> plot(AGRIC, GNP)
```

7.1 Overriding the defaults

What if we want the plot characters to be filled squares rather than open circles, if we want to change the limits on the horizontal and vertical axes, etc.? Next we'll add some arguments to the `plot` command to change the appearance of the plot. The plot is in Figure 12. Comments follow the R code.

```
> plot(AGRIC, GNP, xlab = "percent employed in agriculture",
+     ylab = "per capita GNP", xlim = c(0, 30), ylim = c(0,
+         30), pch = 15, col = "blue", bty = "L", cex = 1.5)
```

The arguments `xlab` and `ylab` set the horizontal and vertical axis labels. The arguments `xlim` and `ylim` set the limits on the axes. The argument `pch` sets the character used for plotting the points. Some of the possible plotting symbols are shown in Figure 13. The argument `col` sets the color of the plotted points. The argument `bty`

sets the type of box (in this case L-shaped). The argument `cex` expands the plotting character by a certain amount.[4]

Of course there are many other arguments to the plot function. The following two commands will show the arguments for the standard `plot()` function and the (many) graphical parameters that are available. The resulting output isn't included in this document.

```
> help(plot.default)
> help(par)
```

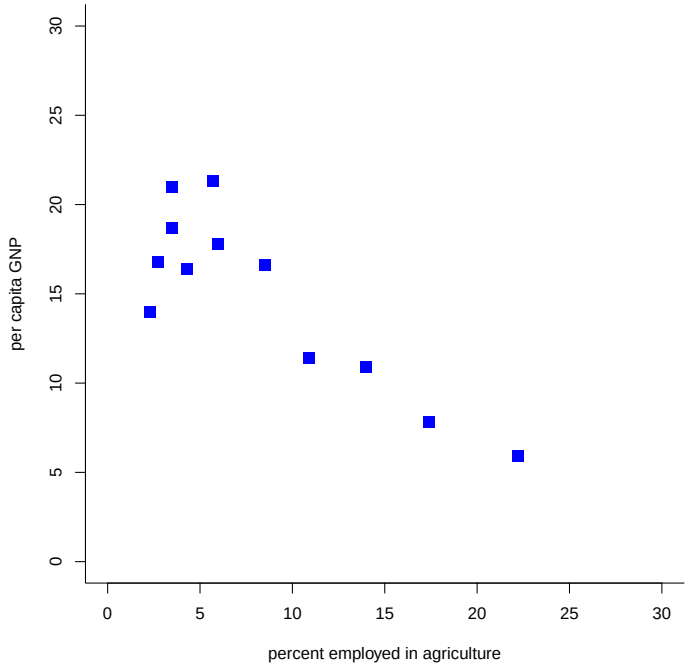

Figure 12: A scatter plot of `GNP` versus `AGRIC`, with some customized settings

As another example consider data on the brain and body weight of terrestrial animals. A plot of the data is given in Figure 14.

```
> brainbody = read.table("http://www.stt.msu.edu/~melfi/cstat/brainbody.txt",
+     header = T)
> brainbody[1:7, ]
```

```
           Animal BodyWeight BrainWeight
1 Mountain Beaver       1.35         8.1
2             Cow     465.00       423.0
3       Grey Wolf      36.33       119.5
```

[4]Technically, some of these, such as `xlim`, are arguments to the `plot` function, while others, such as `pch`, are graphical parameters. Often it won't hurt to ignore this distinction.

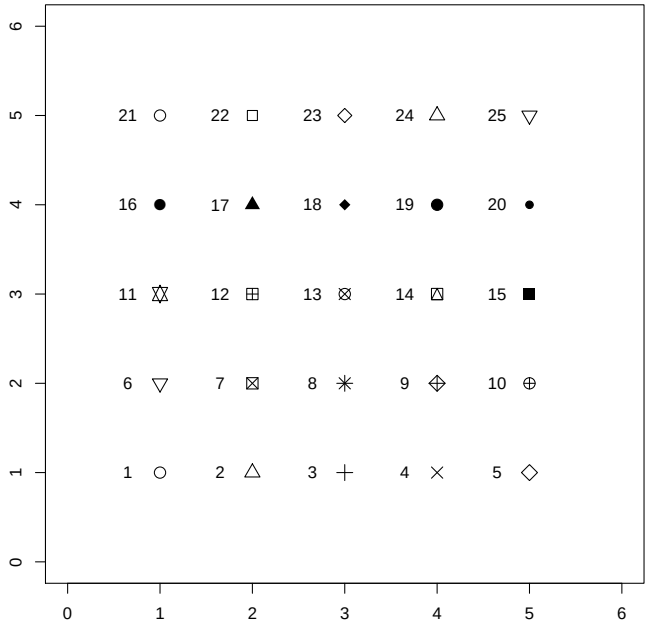

Figure 13: The standard plotting symbols in R

```
4            Goat      27.66       115.0
5      Guinea Pig       1.04         5.5
6      Diplodocus   11700.00        50.0
7  Asian Elephant    2547.00      4603.0
```

```
> plot(brainbody$BodyWeight, brainbody$BrainWeight)
```

Based on the plot, shown in Figure 14 (or a close look at the data), it may make sense to take logarithms of brain and body weights. In Figure 15 a plot of the log-transformed data is given, with some customizations. The R code is below.

```
> plot(log(brainbody$BodyWeight), log(brainbody$BrainWeight),
+     xlab = "Body Weight", ylab = "Brain Weight", bty = "n",
+     pch = 2, col = "red", main = "Log transformed brain and body weights")
```

Here `bty="n"` eliminates the box that is usually put around the plot.

7.1.1 Plot types

By default the `plot()` command plots points. Other options include connecting the points by line segments, vertical line segments, etc. Four possibilities in addition to the default are shown in Figure 16, generated by the R code below. (The `mfrow` parameter sets the number of plots per page. This will be explained later.)

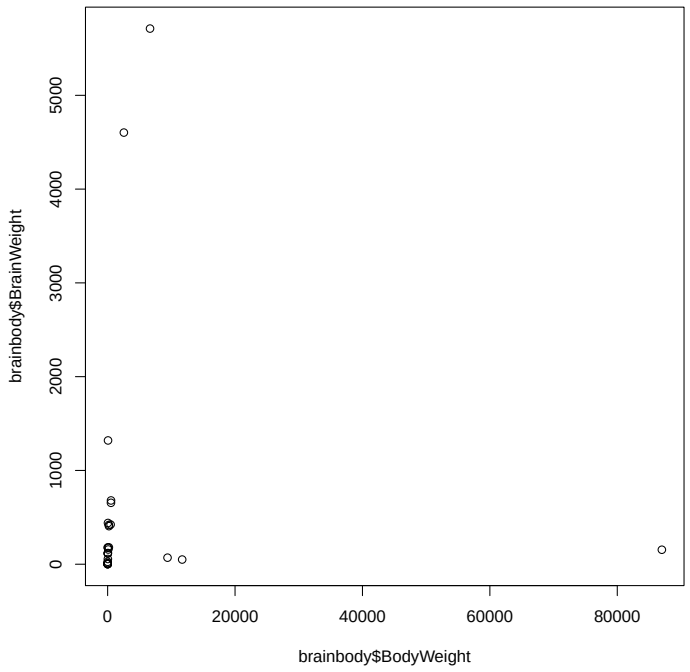

Figure 14: A scatter plot of brain weight versus body weight for terrestrial animals

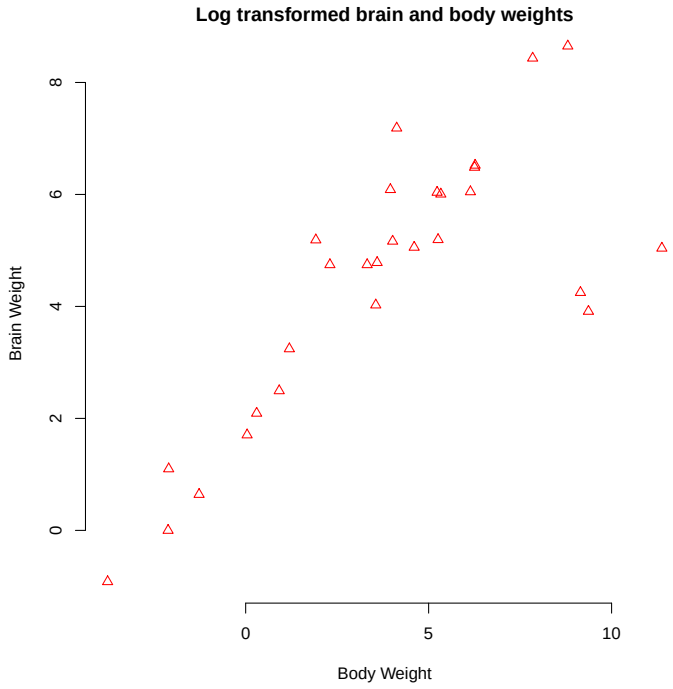

Figure 15: A scatter plot of brain weight versus body weight, log transformed, for terrestrial animals

```
> par(mfrow = c(2, 2))
> tmpx = seq(-3, 3, 0.01)
> plot(tmpx, dnorm(tmpx, 0, 1), type = "l", xlab = "x",
+     ylab = "density at x", main = "standard normal density")
> rm(tmpx)
> tmpx = 0:20
> plot(tmpx, dbinom(tmpx, 20, 0.4), type = "h", xlab = "x",
+     ylab = "prob of x", main = "binomial probabilities")
> plot(tmpx, pbinom(tmpx, 20, 0.4), type = "s", xlab = "x",
+     ylab = "cdf at x", main = "binomial cdf")
> rm(tmpx)
> tmpx = 1:15
> tmpy = rnorm(15, 5, 1)
> plot(tmpx, tmpy, type = "b", main = "connect the dots")
> rm(tmpx)
> rm(tmpy)
> par(mfrow = c(1, 1))
```

Some explanations are in order. The `dnorm()` function computes values of the normal density function. The `dbinom()` function computes binomial probabilities, while the `pbinom()` function computes binomial cumulative probabilities.

7.2 Other high-level graphics functions*

Our focus has been on `plot()`, but of course R has other high-level graphics functions. For several examples see either `demo(graphics)`, `demo(persp)`, and other demos in R, or Chapters 1 and 2 of Murrell [3]. Here an example of a 3-d perspective plot and a pie chart are given. The examples are taken from the R help pages.

First, the perspective plot, shown in Figure 17.

```
> x <- seq(-10, 10, length = 30)
> y <- x
> f <- function(x, y) {
+     r <- sqrt(x^2 + y^2)
+     10 * sin(r)/r
+ }
> z <- outer(x, y, f)
> z[is.na(z)] <- 1
> persp(x, y, z, theta = 30, phi = 30, expand = 0.5)
```

Second, a pie chart, shown in Figure 18.

```
> pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
> names(pie.sales) <- c("Blueberry", "Cherry", "Apple",
+     "Boston Cream", "Other", "Vanilla Cream")
> pie(pie.sales)
```

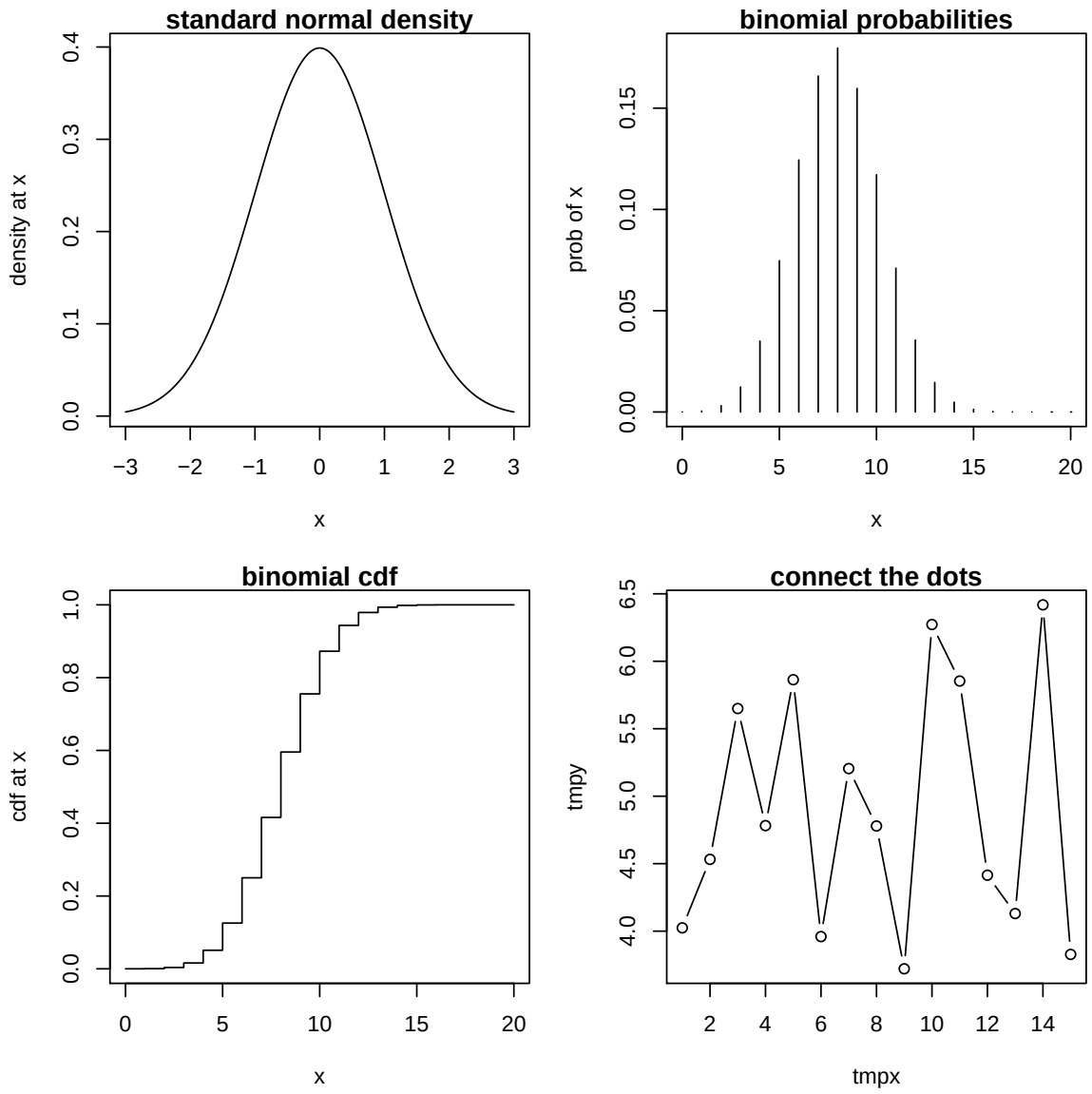

Figure 16: Illustration of different plot types

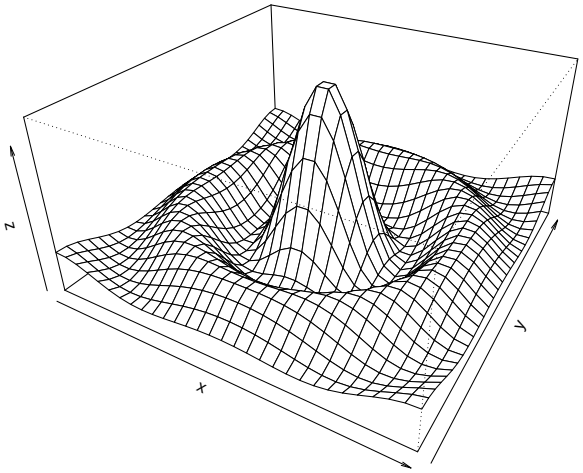

Figure 17: A perspective plot

.

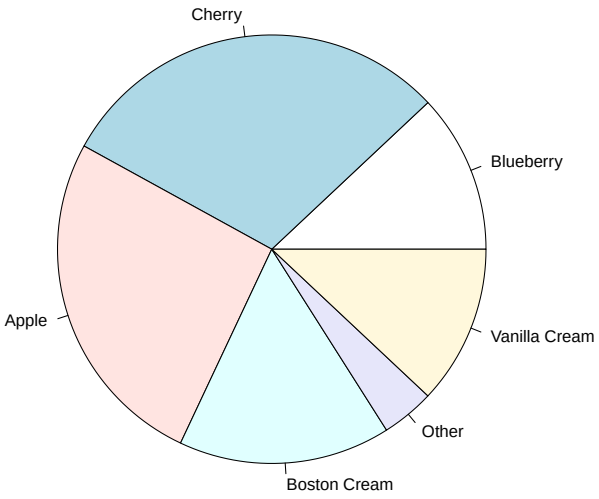

Figure 18: A pie chart

8 Low level graphics functions

Despite their humble-sounding name, low level graphics functions provide a lot of the power for R graphics. They allow the user to modify and enhance graphics already created by a high-level graphics function.

8.1 A first example

Consider a relatively simple task first: Plotting the power functions for two different decision rules, on the same set of axes, to help a reader understand the trade-offs. Specifically, suppose that we are performing an experiment to assess whether a subject has ESP. We repeat the following experiment n times:

- Shuffle a standard deck of cards.
- Choose a card at random, and do not show it to the subject.
- Ask the subject to identify the suit (hearts, clubs, spades, or diamonds) of the chosen card.

Let X represent the number of correct identifications. It is reasonable to model X with a binomial distribution with parameters n and p, where p represents the probability that the subject will correctly identify a card's suit. We want to test

$$H_0 \colon p \leq 0.25 \quad \text{versus} \quad H_a \colon p > 0.25.$$

If our decision rule is to reject H_0 if $X \geq k$, then the power of the test at parameter value p is $P(X > k)$ when X has a binomial distribution with parameters n and p. This is equal to $1 - P(X \leq k)$. This is easily computed in R, since the function `pbinom()` returns cumulative probabilities for binomial distributions. For example, if $n = 20, p = 0.25$, and $k = 10$, then the power is given by

```
> 1 - pbinom(10, 20, 0.25)

[1] 0.003942142
```

A plot of the power function in this case, produced by the R code below, is given in Figure 19.

```
> tmpp = seq(0, 1, 0.01)
> tmppower = 1 - pbinom(10, 20, tmpp)
> plot(tmpp, tmppower, type = "l", xlab = "p", ylab = "power")
> rm(tmpp)
> rm(tmppower)
```

The `points()` command adds points (or lines) to an existing plot. We'll use this to add the power function for the decision rule which rejects H_0 if $X > 8$ to the plot of the power function we've already drawn. The resulting plot is shown in Figure 20.

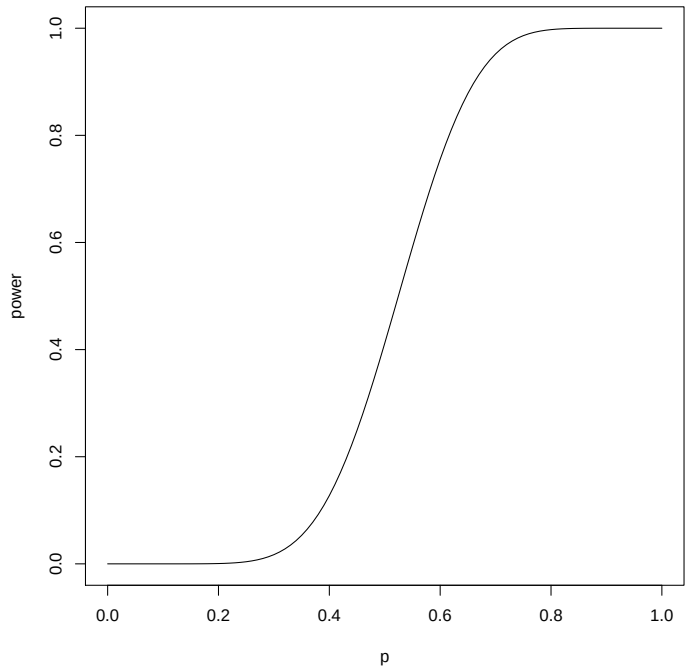

Figure 19: The power function when $n = 20$ and $k = 10$.

```
> tmpp = seq(0, 1, 0.01)
> tmppower = 1 - pbinom(10, 20, tmpp)
> plot(tmpp, tmppower, type = "l", xlab = "p", ylab = "power")
> tmppower = 1 - pbinom(8, 20, tmpp)
> points(tmpp, tmppower, type = "l", lty = "dashed")
> rm(tmpp)
> rm(tmppower)
```

The specification `lty="dashed"` tells R to draw a dashed line rather than the default solid line.

Next, maybe we want to draw points representing the powers at $p = 0.25$, add a legend indicating that the dashed line corresponds to $k = 8$ and the solid line corresponds to $k = 10$, and add a title to the graphic. The resulting graphic is shown in Figure 21.

```
> tmpp = seq(0, 1, 0.01)
> tmppower = 1 - pbinom(10, 20, tmpp)
> plot(tmpp, tmppower, type = "l", xlab = "p", ylab = "power")
> tmppower = 1 - pbinom(8, 20, tmpp)
> points(tmpp, tmppower, type = "l", lty = "dashed")
> points(0.25, 1 - pbinom(10, 20, 0.25), pch = 20)
> points(0.25, 1 - pbinom(8, 20, 0.25), pch = 20)
> legend(0.6, 0.4, legend = c("k=10", "k=8"), lty = c("solid",
```

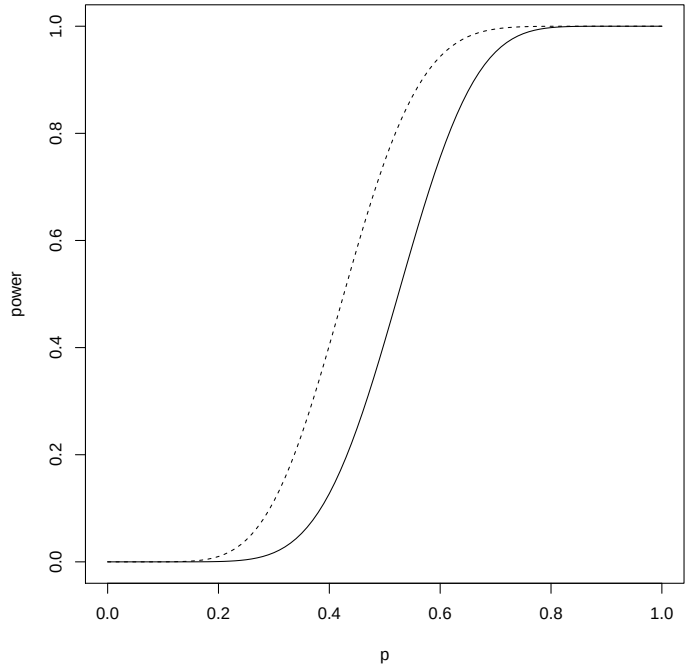

Figure 20: The power functions when $n = 20$ and $k = 10$ and $k = 8$.

```
+     "dashed"))
> title("Power function for two values of k")
> rm(tmpp)
> rm(tmppower)
```

Before leaving this example, we'll use it to illustrate that the width of lines can be changed. Here we draw the same graphic as in Figure 19, but with a much thicker line. The resulting plot of the power function is shown in Figure 22. Although not illustrated here, there are even ways to control how the ends of lines are drawn (squared off, rounded, etc.) and how lines are joined together.

```
> tmpp = seq(0, 1, 0.01)
> tmppower = 1 - pbinom(10, 20, tmpp)
> plot(tmpp, tmppower, type = "l", lwd = 15, xlab = "p",
+     ylab = "power")
> rm(tmpp)
> rm(tmppower)
```

8.2 Text in graphics

There are many reasons to add text to graphics. We may want titles, axis labels, text identifying points, etc. R has sophisticated facilities for these purposes. We'll illustrate with a few examples.

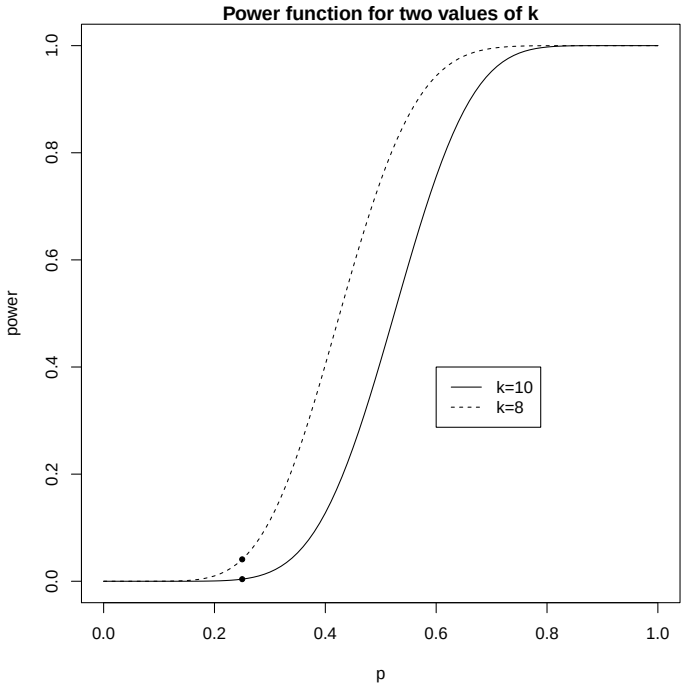

Figure 21: The power functions when $n = 20$ and $k = 10$ and $k = 8$.

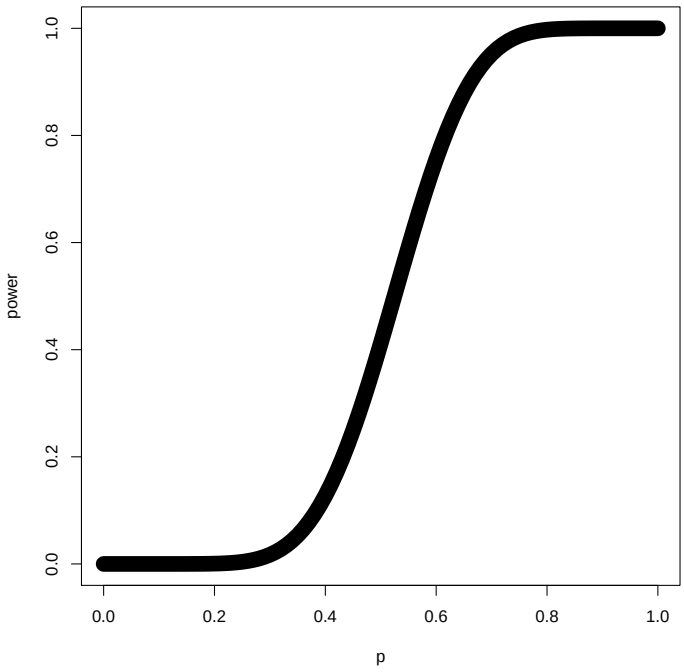

Figure 22: The original power function, drawn with a very thick line.

First, recall the plot of the European economic data given in Figure 2. There rather than plotting symbols like circles or squares, the (abbreviated) country names were used as plot symbols. The `text()` command was used. In a very simple use `text()` takes as input an `x` and a `y` coordinate, and a character string, and places the character string at the location specified by the coordinates. For example, `text(1, 2, labels = "a")` would place the letter `a` at the point $(1, 2)$. This was the idea used in producing Figure 2. Here is the code. (The relevant vectors are displayed first for reference.)

```
> AGRIC

 [1]  2.7  5.7  3.5 22.2 10.9  6.0 14.0  8.5  3.5  4.3 17.4  2.3

> GNP

 [1] 16.8 21.3 18.7  5.9 11.4 17.8 10.9 16.6 21.0 16.4  7.8 14.0

> countries

 [1] "BE" "DK" "DE" "GR" "ES" "FR" "IE" "IT" "LU" "NL" "PT" "UK"

> plot(AGRIC, GNP, type = "n")
> text(AGRIC, GNP, labels = countries)
```

The first `plot()` command with `type="n"` sets up the axes, etc., without plotting any points. Then the `text()` command adds the points.

Now suppose that instead of using the country names as plotting symbols, we want to plot the data as usual, but put the country names below the plotted points. We again use `text()`, but this time specify the `pos` argument to tell R that we want the text below the specified location. (This can be fine-tuned in a number of ways.) The graphic is in Figure 23.

```
> plot(AGRIC, GNP)
> text(AGRIC, GNP, labels = countries, pos = 1, cex = 0.75)
```

8.3 Other plot annotations

There are many other ways to annotate plots, in addition to using `text()`. For example, line segments and arrows can be drawn, rectangles and other polygons can be added, etc. As usual, a picture is worth a thousand words.

The functions `abline()` and `arrows()` can be used to add lines and arrows to a plot. Four examples are given next, with pictures in Figure 24.

```
> par(mfrow = c(2, 2))
> plot(AGRIC, GNP)
> euro.lm = lm(GNP ~ AGRIC)
> abline(euro.lm)
```

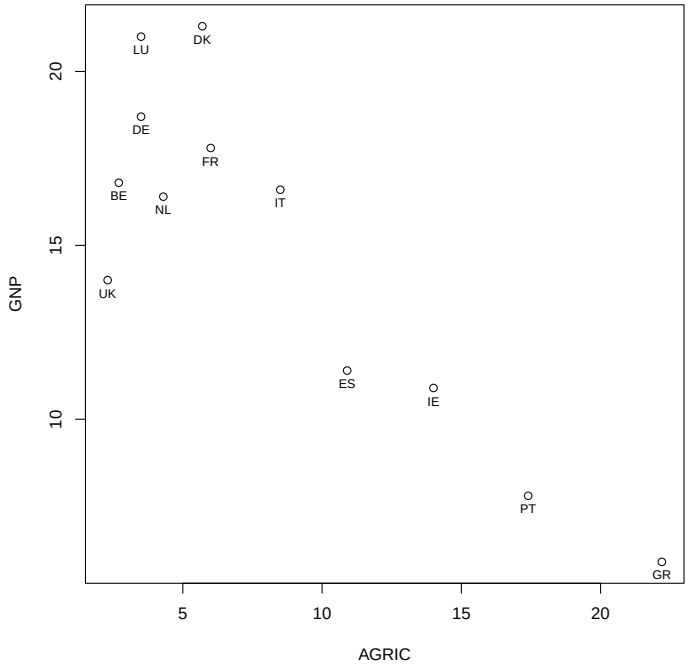

Figure 23: Data plotted with labels below the plotted points

```
> plot(AGRIC, GNP)
> abline(h = mean(GNP))
> abline(v = mean(AGRIC), lty = "dashed")
> plot(AGRIC, GNP)
> arrows(AGRIC[5], GNP[5], AGRIC[1], GNP[1], code = 1)
> plot(AGRIC, GNP)
> arrows(AGRIC[5], GNP[5], AGRIC[1], GNP[1], code = 2)
> par(mfrow = c(1, 1))
```

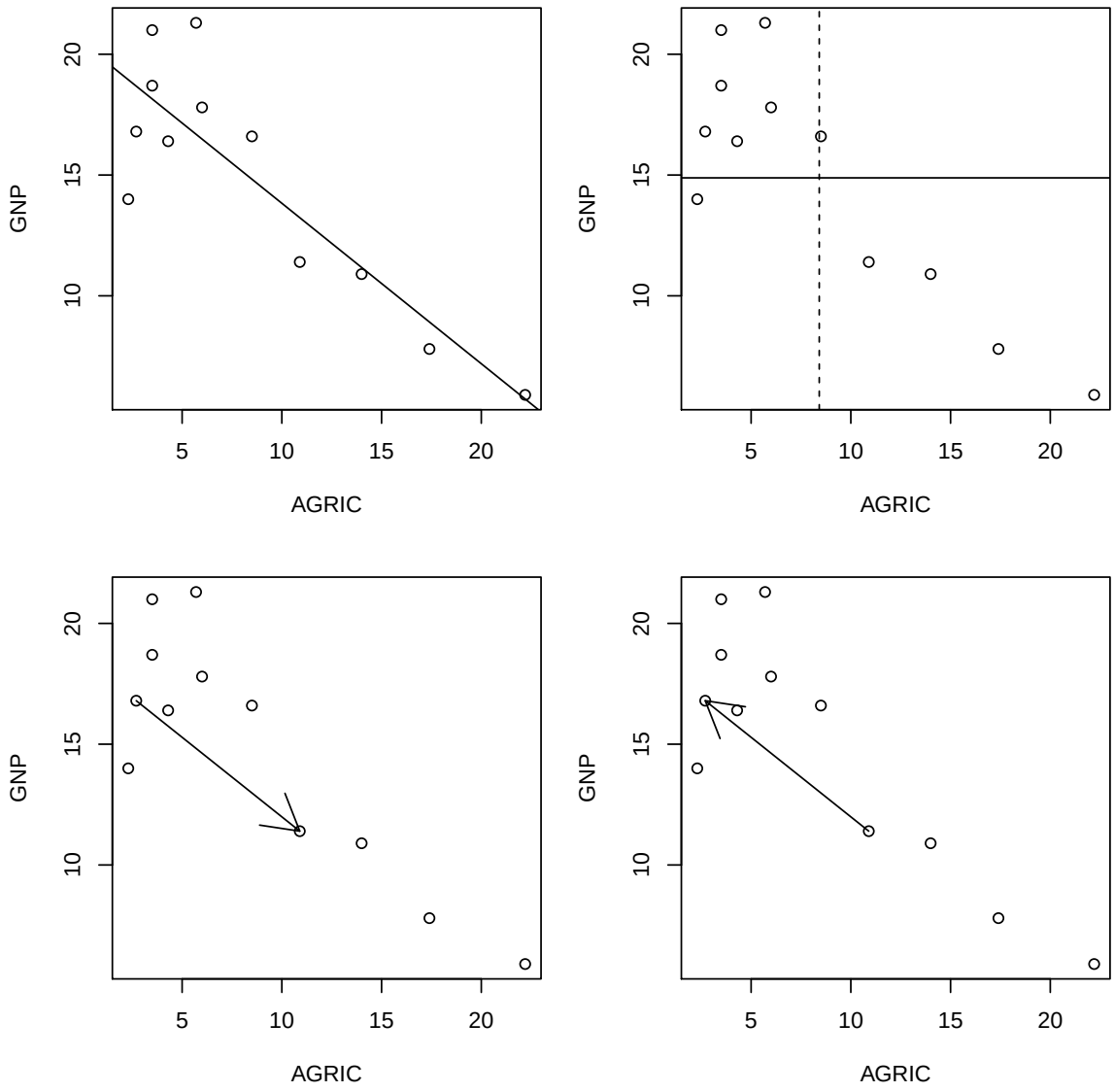

Figure 24: Using `abline()` and `arrows()` to annotate graphics

References

[1] Phil Spector (2008) *Data Manipulation with R*, Springer-Verlag, New York, NY.

[2] Emmanuel Paradis *R for Beginners*, available at `http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf`.

[3] Paul Murrell (2006), *R Graphics*, Chapman & Hall/CRC, Boca Raton.

[4] R development core team (2009) *R Data Import/Export*, available at `http://cran.r-project.org/doc/manuals/R-data.pdf`.