R in a few hours – a brief introduction

Søren Højsgaard and Ulrich Halekoh
Statistics and Decision Theory Research Unit,
Danish Institute of Agricultural Sciences,
Research Center Foulum, DK–8830 Tjele, Denmark

October 25, 2006

Contents

1 Introduction

On the Comprehensive R Archive Network (CRAN)[1] several introductions to R are available. See section 16 for some references to additional material on R.

Most of these are very good but some people find that they often become too technical at a too early stage. These notes attempt to introduce R without getting very technical.

The explanations are are kept very brief. You are assumed to make extensive use of the help functions to obtain more information.

While reading this document, we suggest that you execute the code fragments. Remember that you can copy text from a pdf file if you read it using Acrobat reader, so you do not have to type in the code fragments yourself.

Some sections are indicated with an asterisk ("*") indicating that they can be omitted at first reading.

Graphics in R is not described in these notes.

Input to R is displayed as

[1] http://www.r-project.org

```
....
```

and output from R is displayed as

```
....
```

2 Getting help

The most important step in learning a new program is to know where to get help. In R there are several options.

- It is recommended to always keep a web-browser with the R help pages open with

  ```
  help.start()
  ```

- Use the manual pages to get detailed information. For example, the function rnorm generates a random sample from a normal distribution. To get help on rnorm:

  ```
  help(rnorm)
  ```

  This opens the manual page for rnorm. Note: at the bottom of the manual pages there are usually informative examples.

  To get information about documentation dealing with a specific issue, do e.g.

  ```
  help.search("linear model")
  ```

- To get a list of functions with "norm" in its name, use

  ```
  apropos("norm")
  ```

- An overview of possible arguments for a given function is obtained by:

  ```
  args(rnorm)
  ```

- Search for key words or phrases in the R-help mailing list archives, or R manuals and help pages:

  ```
  RSiteSearch("quantile normal distribution")
  ```

- The R–help list: You may consider subscribing to the R-help mailing list, which is very active. Please do read the posting guide before sending a question to the list. Also, please remember that nobody is paid for answering questions on the list.

3 Functions in R

R consists essentially of a collection of functions and most functions take some arguments as input and return a value as output.

For example, the function `rnorm` generates a random sample from a normal distribution. To see the input arguments:

```
args(rnorm)
```

```
function (n, mean = 0, sd = 1)
NULL
```

The second and third arguments have been assigned default values, so these need not to be specified. The first argument has no default value, so it must be specified:

```
rnorm(n = 4)
```

```
[1] -0.6041289 -0.4092270  0.9078563 -1.5089080
```

To sample from a normal with mean 5 and standard deviation 1, we can do

```
rnorm(n = 4, mean = 5)
```

```
[1] 5.076935 4.541182 4.940224 5.218318
```

It is also allowed simply to write

```
rnorm(4, 5)
```

because R will match the given arguments with the order in which the function expects its arguments. If we want to sample with mean 0 and standard deviation 3, we can do one of the following:

```
rnorm(n = 4, sd = 3)
rnorm(4, 0, 3)
```

It is generally recommended to be explicit about naming the arguments to functions, i.e. to write `rnorm(n=4,sd=3)` rather than `rnorm(4,0,3)`. Readability will be greatly enhanced – especially if returning to your work after a few months!

4 Vectors – Simple R objects

4.1 Numbers

In R you can make simple calculations like

```
(10 + 2) * 5
```

```
[1] 60
```

Often one wants to store the result in a variable (or an object). This is done with the the "assign operator" `<-` (observe that R is case–sensitive):

```
n <- (10 + 2) * 5
```

```
[1] 60
```

Calculations can be made on objects, e.g:

```
n^2
```

```
[1] 3600
```

```
n + 10
```

```
[1] 70
```

```
n <- n + n
```

```
[1] 120
```

Observe: If the object already exists, its previous value is overwritten.

4.2 Numerical vectors

A numerical vector is created by the function `c` which combines all its arguments:

```
a <- c(1, 5.8, -77)
b <- c(44, 678)
c(a, b, -800)
```

```
[1]    1.0    5.8  -77.0   44.0  678.0 -800.0
```

Entries of a vector can be named as:

```
a <- c(x = 1, y = 5.8, z = -77)
a
```

```
   x    y    z
 1.0  5.8 -77.0
```

An alternative method is:

```
a <- c(1, 5.8, -77)
names(a) <- c("x", "y", "z")
```

The names of elements in a vector can be retrieved as

```
names(a)
```

```
[1] "x" "y" "z"
```

4.2.1 Arithmetic on numerical vectors – vectorized arithmetic

Most mathematical operations in R are vectorized, i.e. they are applied to each element in a vector:

1. Add a number to a vector

   ```
   5 + c(4, 7, 17)
   ```

   ```
   [1]  9 12 22
   ```

2. Multiply a number with a vector

   ```
   5 * c(4, 7, 17)
   ```

   ```
   [1] 20 35 85
   ```

3. Add two vectors of the same length

   ```
   c(-1, 3, -17) + c(4, 7, 17)
   ```

   ```
   [1]  3 10  0
   ```

4. Apply a function to each element of a vector, e.g.

   ```
   c(2, 4, 5)^2
   ```

   ```
   [1]  4 16 25
   ```

   ```
   sqrt(c(2, 4, 25))
   ```

   ```
   [1] 1.414214 2.000000 5.000000
   ```

4.2.2 Recycling

Two vectors of same length are added/subtracted/divided/multiplied elementwise as

```
c(1, 2, 3) + c(2, 4, 8)
```

```
[1]  3  6 11
```

If one vector is shorter than the other, the shortest is recycled to make the lengths match:

```
c(1, 2, 3) + c(2, 4, 8, 12, 14, 18, 22)
```

```
[1]  3  6 11 13 16 21 23
```

4.3 Vector of characters

A vector can also be a collection of characters

```
c("green", "blue sky", "-77")
```

```
[1] "green"    "blue sky" "-77"
```

Note that "-77" is a character, not a number. You cannot multiply the vector by a number (try!).

4.4 Vector as factor

A factor is a special character vector. It is of importance in modelling explanatory variables with discrete levels in regression models

The vector has an attribute `levels`. The first level is often the reference level.

1. Turn a character vector into a factor:

```
a <- c("green", "blue", "green", "yellow")
```

```
[1] "green"  "blue"   "green"  "yellow"
```

```
factor(a)
```

```
[1] green  blue   green  yellow
Levels: blue green yellow
```

2. Turn a numerical vector `b` into factor:

```
b <- c(2, 1, 3, 1)
```

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

```
b <- factor(b)
```

```
[1] 2 1 3 1
Levels: 1 2 3
```

3. You can give the levels of factor `b` informative names

```
levels(b) <- c("low", "middle", "high")
```

```
[1] "low"    "middle" "high"
```

```
b
```

```
[1] middle low    high   low
Levels: low middle high
```

4.5 Logical expressions

A logical expression is an expression which is either TRUE or FALSE, which in R can be abbreviated as T and F.

1. Determine if two numbers are equal (==) or not equal (!=):

```
7 == 6
```

```
[1] FALSE
```

```
7 != 6
```

```
[1] TRUE
```

2. Comparison of numbers:

```
7 > 6
```

```
[1] TRUE
```

```
6 <= 7
```

```
[1] TRUE
```

3. ! indicates logical negation (NOT): It changes TRUE to FALSE and FALSE to TRUE:

```
!TRUE
```

```
[1] FALSE
```

```
!FALSE
```

```
[1] TRUE
```

```
!(7 == 6)
```

```
[1] TRUE
```

4. Logical operations can be combined with OR (|) and AND (&). For example; is `(7 == 9) OR (7>0)`?

```
(7 == 9) | (7 > 0)
```

```
[1] TRUE
```

Is `7 == 9 AND 7 > 0`

```
(7 == 9) & (7 > 0)
```

```
[1] FALSE
```

5. The logical operations are also vectorized, e.g.

```
c(13, 4, 9, -7, 18) > 7
```

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

```
c(T, F, T) == F
```

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

6. The functions `all` and `any` are useful on vectors of logical values:

```
a1 <- c(TRUE, TRUE, FALSE)
a2 <- c(TRUE, TRUE, TRUE)
a3 <- c(FALSE, FALSE, FALSE)
any(a1)
```

```
[1] TRUE
```

```
all(a1)
```

```
[1] FALSE
```

```
any(a2)
```

```
[1] TRUE
```

```
all(a2)
```

```
[1] TRUE
```

```
any(a3)
```

```
[1] FALSE
```

```
all(a3)
```

```
[1] FALSE
```

5 Indexing of vectors

5.1 Indexing a vector

To select specific elements of a vector use the square brackets `[ ]`:

1. Select elements 3, 4 and 1 of a vector:

```
a <- c(13, 4, 9, -7, 18)
a[c(3, 4, 1)]
```

```
[1]  9 -7 13
```

2. Take all elements out except elements number 2 and 5:

```
a[-c(2, 5)]
```

```
[1] 13  9 -7
```

5.2 Conditional subsetting using logical values

1. Select the elements of vector `a` which are larger than 4:

```
a <- c(13, 4, 9, -7, 18)
a[a > 4]
```

```
[1] 13  9 18
```

Note: The vector `a>4` is actually a vector of logical values

```
a > 4
```

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

Therefore, `a[a>4]` selects all elements of `a` where the logical values are `TRUE`.

2. You can also have more complicated expressions.

```
a <- c(13, 4, 9, -7, 18)
b <- c("yellow", "green", "blue", "yellow", "brown")
a[a > 4 & b == "brown"]
```

```
[1] 18
```

6 Generating vectors

6.1 Regular sequences

1. All numbers between two integers: The numbers from 4 to 8

```
v <- 4:8
```

```
[1] 4 5 6 7 8
```

2. A sequence of equally spaced numbers: 5 numbers between 4.7 and 6.1

```
seq(from = 4.7, to = 6.1, length.out = 5)
```

```
[1] 4.70 5.05 5.40 5.75 6.10
```

3. A sequence of numbers between 4.7 and 6.1 in steps of 0.4

```
seq(from = 4.7, to = 6.1, by = 0.4)
```

```
[1] 4.7 5.1 5.5 5.9
```

4. The sequence 1,2,3 repeated four times

```
rep(1:3, times = 4)
```

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

5. The sequence 1,1,2,2,3,3 repeated four times

```
rep(1:3, times = 4, each = 2)
```

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

6.2 Random sequences

R can generate random data for a large number of probability density functions. Some examples

```
runif(n = 5)
```

```
[1] 0.1210134 0.6123321 0.9912573 0.6740875 0.3353667
```

```
rnorm(n = 5, mean = 5, sd = 0.1)
```

```
[1] 4.973377 5.001092 4.991781 4.970077 4.962391
```

7 Missing values

Missing values are indicated by `NA`:

```
a <- c(3, 1, NA, 5, NA, 8, 4)
```

```
[1]  3  1 NA  5 NA  8  4
```

The index of entries which are missing can be found using `is.na`:

```
is.na(a)
```

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

To take out the non–missing values do:

```
a[!is.na(a)]
```

```
[1] 3 1 5 8 4
```

The entries of non–missing values are found using `which`:

```
which(!is.na(a))
```

```
[1] 1 2 4 6 7
```

8 Various functions on vectors

Apply a summary function to a vector

```
a <- c(3, 1, 7, 5, 4, 8, 4)
```

```
[1] 3 1 7 5 4 8 4
```

```
mean(a)
```

```
[1] 4.571429
```

```
var(a)
```

```
[1] 5.619048
```

```
range(a)
```

```
[1] 1 8
```

Sorting and finding the ordering of elements in a vector (note the difference):

```
sort(a)
```

```
[1] 1 3 4 4 5 7 8
```

```
order(a)
```

```
[1] 2 1 5 7 4 3 6
```

A numerical vector can be turned into a factor using the `cut` function:

```
cut(a, breaks = c(0, 5, 10))
```

```
[1] (0,5]  (0,5]  (5,10] (0,5]  (0,5]  (5,10] (0,5]
Levels: (0,5] (5,10]
```

Applying a function to vectors with missing data generally requires special attention:

```
a <- c(3, 1, NA, 5, NA, 8, 4)
mean(a)
```

```
[1] NA
```

```
mean(a, na.rm = T)
```

```
[1] 4.2
```

9 Important data structures

9.1 Data frame

A dataframe is the spreadsheet of R. To the user, it is a rectangular array but in contrast to a matrix, it columns can be integers, numericals, characters or factors.

9.1.1 Puromycin – a built–in data set

With the R distribution comes a collection of data sets. One of these is the Puromycin data which is about the velocity of an enzymatic reaction. To "load" this data set into R:

```
data(Puromycin)
```

9.1.2 Indexing a dataframe

A dataframe is two–dimensional so to acces a specific element, row and column must be given, e.g.

```
Puromycin[1, 1]
```

```
[1] 0.02
```

To extract the first five rows of data, the column specifictions is omitted

```
Puromycin[1:5, ]
```

```
  conc rate   state
1 0.02   76 treated
2 0.02   47 treated
3 0.06   97 treated
4 0.06  107 treated
5 0.11  123 treated
```

To acces specific columns, these columns can be specified the same way as rows. Often it is more convenient to address the columns by their names

```
Puromycin[c(1, 3, 5), c("conc", "state")]
```

```
  conc   state
1 0.02 treated
3 0.06 treated
5 0.11 treated
```

Specific variables (columns) can be extracted using `$` as

```
Puromycin$conc
```

```
 [1] 0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 1.10 1.10 0.02
[14] 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 1.10
```

or as

```
Puromycin[, "state"]
```

```
 [1] treated   treated   treated   treated   treated   treated
 [7] treated   treated   treated   treated   treated   treated
[13] untreated untreated untreated untreated untreated untreated
[19] untreated untreated untreated untreated untreated
Levels: treated untreated
```

9.1.3 Adding new variables to a data frame

We want to add a new variable 'iconc' containing 1/conc to the data frame. There are several ways of doing this:

- Basic method: Take the column of the dataframe, transform it and assign it to the dataframe.

  ```
  Puromycin$iconc <- 1/Puromycin$conc
  ```

  We cannot just write `1/conc` because R does not immediately know where to find 'conc'.

- `with` function: Using the function `with` we can tell where to find conc:

  ```
  Puromycin$iconc <- with(Puromycin, 1/conc)
  ```

- `transform` function: You can specify several transformations in one call, e.g.

  ```
  Puromycin <- transform(Puromycin, iconc = 1/conc, sqrtconc = sqrt(conc))
  head(Puromycin)
  ```

  ```
    conc rate   state    iconc  sqrtconc
  1 0.02   76 treated 50.00000 0.1414214
  2 0.02   47 treated 50.00000 0.1414214
  3 0.06   97 treated 16.66667 0.2449490
  4 0.06  107 treated 16.66667 0.2449490
  5 0.11  123 treated  9.09091 0.3316625
  6 0.11  139 treated  9.09091 0.3316625
  ```

9.1.4 Conditional subsetting

Often one wants to select rows of a dataframe for certain values of some column variables. The `subset` function does this.

E.g. take out those rows where `state` is `treated` and `rate>160`:

```
subset(Puromycin, state == "treated" & rate > 160)
```

```
   conc rate   state     iconc  sqrtconc
9  0.56  191 treated 1.7857143 0.7483315
10 0.56  201 treated 1.7857143 0.7483315
11 1.10  207 treated 0.9090909 1.0488088
12 1.10  200 treated 0.9090909 1.0488088
```

Take out those rows where conc is larger than the mean of all conc values:

```
subset(Puromycin, conc > mean(conc))
```

```
   conc rate     state     iconc  sqrtconc
9  0.56  191   treated 1.7857143 0.7483315
10 0.56  201   treated 1.7857143 0.7483315
11 1.10  207   treated 0.9090909 1.0488088
12 1.10  200   treated 0.9090909 1.0488088
21 0.56  144 untreated 1.7857143 0.7483315
22 0.56  158 untreated 1.7857143 0.7483315
23 1.10  160 untreated 0.9090909 1.0488088
```

9.1.5 Creating a dataframe from vectors

We generate a data frame from two vectors, the numerical vector weight and the character vector `x`. Note that we may give `x` the explicit name `age` in the `data.frame` function.

```
weight <- c(70.6, 56.4, 80, 59.5)
x <- (c("adult", "teen", "adult", "teen"))
wag <- data.frame(weight, age = x)
```

```
  weight   age
1   70.6 adult
2   56.4  teen
3   80.0 adult
4   59.5  teen
```

9.1.6 Creating a dataframe from external data set

Most commonly a dataframe is created by reading a data file into `R`. See Section 11 about this.

9.2 Matrix

A matrix is a two-dimensional array consisting of numbers.

9.2.1 Creating a matrix

A matrix can be created with the function `matrix`.

```
v <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
m <- matrix(v, nrow = 3, ncol = 4)
m
```

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

Note that numbers are entered columnwise into the matrix. If we want to read the numbers into the array rowwise we can do

```
m <- matrix(v, nrow = 3, ncol = 4, byrow = TRUE)
```

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

Matrices have a dimension:

```
dim(m)
```

```
[1] 3 4
```

9.2.2 Indexing a matrix

For a matrix or a data frame `m`, the value of the ith line and jth column is accessed with `m[i, j]`.

Selecting the single entry $(2, 3)$ at row 2 and column 3

```
m[2, 3]
```

```
[1] 7
```

Selecting a submatrix, row 1 and 2 and columns 2 and 4

```
m[1:2, c(2, 4)]
```

```
     [,1] [,2]
[1,]    2    4
[2,]    6    8
```

Select the whole row 2 (entry after the comma is empty)

```
m[2, ]
```

```
[1] 5 6 7 8
```

Select the whole column 3 (entry before the comma is empty)

```
m[, 3]
```

```
[1]  3  7 11
```

To change all values of the third column, we can type:

```
m[, 3] <- 10.2
m
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2 10.2    4
[2,]    5    6 10.2    8
[3,]    9   10 10.2   12
```

Indexing can be used to suppress one or several lines or columns.

```
m[, -c(1, 3)]
```

```
     [,1] [,2]
[1,]    2    4
[2,]    6    8
[3,]   10   12
```

9.3 Lists

Lists are probably the most flexible data structure in R. A list is created with the function list:

```
l1 <- list(188.45, 83, c("peter", "hansen"))
```

```
[[1]]
[1] 188.45

[[2]]
[1] 83

[[3]]
[1] "peter"  "hansen"
```

The elements of a list can be of different types and of different lengths.

Take out a sub-list by:

```
l1[2:3]
```

```
[[1]]
[1] 83

[[2]]
[1] "peter"  "hansen"
```

To access a single element of the list:

```
l1[[3]]
```

```
[1] "peter"  "hansen"
```

Hence we can obtain "peter" with

```
l1[[3]][1]
```

```
[1] "peter"
```

It is usually a good idea to name the elements of a list:

```
l2 <- list(height = 188, weight = 83, name = c("peter",
     "hansen"))
```

```
$height
[1] 188

$weight
[1] 83

$name
[1] "peter"  "hansen"
```

We can then do

```
l2$name
```

```
[1] "peter"  "hansen"
```

```
l2$name[1]
```

```
[1] "peter"
```

10 Type of an object and conversion of type

10.1 class

All objects in R has a class attribute describing the type of the object. These can be retrieved using the `class` function:

```
class(1)
```

```
[1] "numeric"
```

```
class(c(1, 2, 3))
```

```
[1] "numeric"
```

```
class(c("e", "g", "h"))
```

```
[1] "character"
```

```
class(factor(c("e", "g", "h")))
```

```
[1] "factor"
```

```
class(TRUE)
```

```
[1] "logical"
```

```
class(list(1, 2, 3))
```

```
[1] "list"
```

```
class(Puromycin)
```

```
[1] "data.frame"
```

```
class(matrix(1:4, ncol = 2))
```

```
[1] "matrix"
```

10.2 as.something

Sometimes one wish to convert an object from one class to another. The general way of doing so is to use functions with names like `as.something`:

```
an <- 1:3
```

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

```
ac <- as.character(an)
```

```
[1] "1" "2" "3"
```

```
af <- as.factor(an)
```

```
[1] 1 2 3
Levels: 1 2 3
```

```
as.list(an)
```

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

[[2]]
[1] 2

[[3]]
[1] 3
```

```
as.numeric(ac)
```

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

```
as.numeric(af)
```

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

A matrix can always be converted to a dataframe and vice versa:

```
m <- matrix(1:4, nrow = 2)
```

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

```
md <- as.data.frame(m)
```

```
  V1 V2
1  1  3
2  2  4
```

```
as.matrix(md)
```

```
  V1 V2
1  1  3
2  2  4
```

However, the entries in a matrix must all be of the same type and R does the conversion

```
md <- data.frame(trt = c("a", "b"), y = 1:2)
```

```
  trt y
1   a 1
2   b 2
```

```
as.matrix(md)
```

```
  trt y
1 "a" "1"
2 "b" "2"
```

11 Getting data in and out of R

11.1 Directories – Slash and backslash

In R you must separate directiories with a slash (/) or with a double backslash (\\). A single backslash (\) (as is used on Windows platforms) will not work.

11.2 Writing data

11.2.1 Write as text file

1. To write the dataframe

```
d <- data.frame(obs = c(1, 2, 3), treat = c("A", "B",
     "A"), weight = c(2.3, NA, 9))
```

   as a simple text file use

```
write.table(d, file = "d:/foo.txt", row.names = F, quote = F)
```

2. To write the dataframe as a comma-separated file use

```
write.csv(d, file = "d:/foo.csv", row.names = F, quote = F)
```

   Note: On Windows platforms, comma separated files can be read by Excel by simply double clicking on the file icon.

   Note: `write.csv` uses period as decimal point and comma as separator. In some countries, like Denmark, the convention is that the decimal point is comma and in this case the semicolon is used as separator. The function `write.csv2` accomodates this convention.

11.2.2 Save in R format

R uses an internal format for saving data which can afterwards be read by R again. To save data in this format use:

```
save(d, file = "d:/foo.Rdata")
```

11.3 Reading data

11.3.1 Reading text files

1. Suppose your file `d:/foo.txt` is a simple text file as

```
obs treat weight
1    A     3.4
2    B      NA
3    A     5.8
```

   Your read it by

```
foo <- read.table(file = "d:/foo.txt", header = T)
```

2. If the entries of `d:/foo.csv` are comma separated as

```
obs, treat, weight
1,    A,     3.4
2,    B,     NA
3,    A,     5.8
```

use

```
foo <- read.csv(file = "d:/foo.csv", header = T)
```

In both cases, `foo` will now be a dataframe – the spreadsheet of R – containing the data.

Note: `read.csv` uses period as decimal point and comma as separator. In some countries, like Denmark, the convention is that the decimal point is comma and in this case the semicolon is used as separator. The function `read.csv2` accommodates this convention.

11.3.2 EXCEL files

1. A simple way is to open the spreadsheet, mark the desired area and copy it to the clipboard using ctrl-c. Then in R type

```
foo <- read.delim("clipboard")
```

2. Alternatively, use the RODBC package. The sheet "Sheet1" in the file `d:/foo.xls` is read as

```
library(RODBC)
z <- odbcConnectExcel("d:/foo.xls")
foo <- sqlFetch(z, "Sheet1")
close(z)
```

11.3.3 Reading data from other statistical packages

The `foreign` package contain facilities for reading data from various other statistical packages. Unfortunately, one can not read a SAS dataset directly into R. Some of the simple options in connection with SAS are: 1) Export the SAS dataset as an Excel file/comma separated file. Read this file into R. 2) View the SAS dataset as HTML file. Use copy–and–paste to get data into R.

11.3.4 Load in R format

A data frame (or any other R object) saved in R format can be loaded with

```
load("d:/foo.Rdata")
```

11.4 Data shipped with R packages

To see the data sets available in R (including those packages which are loaded) type

```
data()
```

12 Installing and using R packages

Much of Rs functionality is provided by the add–on packages. These are not shipped with the R distribution but must be installed separately.

There are two ways of installing packages: 1) In the R console (on Windows), go to menu item "packages" and select install packages. Then you can select the packages you want to install. You will be prompted to select a CRAN server. Pick one close to where you are. 2) The other way is as follows: To install e.g. the doBy package you can do

```
install.packages("doBy")
```

Once the package is installed on your computer it must be "loaded" into R before it can be used:

```
library(doBy)
```

A package must be loaded once per session with R, because the package is unloaded when Ris shut down. It is not necessary to install the package in each R session.

R packages are frquently updated on CRAN, and it is generally a good idea to ensure that you have the most recenter versions on your computer. This can be ensured by

```
update.packages()
```

To see what is in the doBy package do

```
help(package = doBy)
```

13 Manipulation of dataframes

Consider two data frames:

```
d1 <- data.frame(trt = rep(c("A", "B"), 2), y = c(1,
     2, 3, 4))
d2 <- data.frame(trt = rep(c("C", "D"), 2), y = c(5,
     6, 7, 8))
d1
```

```
  trt y
1   A 1
2   B 2
3   A 3
4   B 4
```

```
d2
```

```
  trt y
1   C 5
2   D 6
3   C 7
4   D 8
```

To put them together, one on top of the other into a new dataframe, do:

```
rbind(d1, d2)
```

```
  trt y
1   A 1
2   B 2
3   A 3
4   B 4
11  C 5
21  D 6
31  C 7
41  D 8
```

The `cbind` function puts its arguments together, one next to the other:

```
d3 <- data.frame(trt2 = rep(c("C", "D"), 2), y2 = c(5,
     6, 7, 8))
cbind(d1, d3)
```

```
  trt y trt2 y2
1   A 1    C  5
2   B 2    D  6
3   A 3    C  7
4   B 4    D  8
```

Dataframes can be merged as follows:

```
d4 <- data.frame(trt = c("A", "B"), place = c("here",
     "there"))
d4
```

```
  trt place
1   A  here
2   B there
```

```
merge(d4, d1)
```

```
  trt place y
1   A  here 1
2   A  here 3
3   B there 2
4   B there 4
```

14 Towards statistics

14.1 Functions for data frames

A few very basic functions for looking into a data frame are:

- `help`: Gives detailed information about the R dataset.

  ```
  help(Puromycin)
  ```

To see the entire data set type `Puromycin` at the prompt. To see the first few lines of data:

```
head(Puromycin)
```

```
  conc rate   state    iconc  sqrtconc
1 0.02   76 treated 50.00000 0.1414214
2 0.02   47 treated 50.00000 0.1414214
3 0.06   97 treated 16.66667 0.2449490
4 0.06  107 treated 16.66667 0.2449490
5 0.11  123 treated  9.09091 0.3316625
6 0.11  139 treated  9.09091 0.3316625
```

- `summary`: This function gives a quick overview of the data:

```
summary(Puromycin)
```

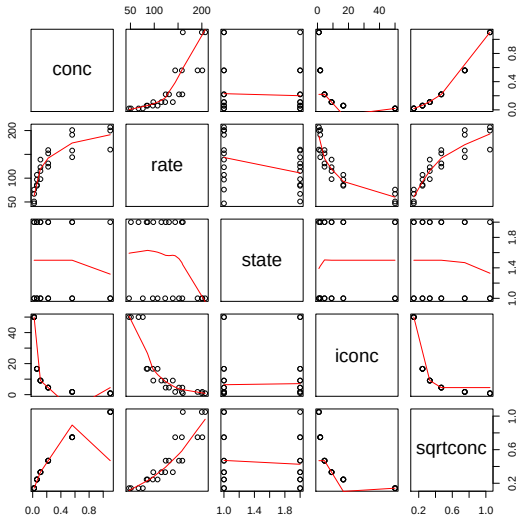
```
      conc             rate             state        iconc        
 Min.   :0.0200   Min.   : 47.0   treated  :12   Min.   : 0.9091  
 1st Qu.:0.0600   1st Qu.: 91.5   untreated:11   1st Qu.: 1.7857  
 Median :0.1100   Median :124.0                  Median : 9.0909  
 Mean   :0.3122   Mean   :126.8                  Mean   :14.3949  
 3rd Qu.:0.5600   3rd Qu.:158.5                  3rd Qu.:16.6667  
 Max.   :1.1000   Max.   :207.0                  Max.   :50.0000  
    sqrtconc      
 Min.   :0.1414  
 1st Qu.:0.2449  
 Median :0.3317  
 Mean   :0.4734  
 3rd Qu.:0.7483  
 Max.   :1.0488  
```

The summary shows that conc and rate are numerical variables while state is a factor.

- `pairs`

  A quick graphical overview by the scatterplot matrix. All variables are plotted against each other.

```
pairs(Puromycin, panel = panel.smooth)
```



- `xtabs`: Cross-classifies variables counting how often a combination of their levels occur:

```
xtabs(~state + conc, data = Puromycin)
```

```
           conc
state       0.02 0.06 0.11 0.22 0.56 1.1
  treated      2    2    2    2    2   2
  untreated    2    2    2    2    2   1
```

14.2 Functions applied to a matrix

1. In some cases one is interested in row wise (column wise) computations on a matrix, e.g. the row wise means

```
m <- matrix(rnorm(n = 12), nrow = 3)
m
```

```
          [,1]       [,2]       [,3]       [,4]
[1,] 1.8697297  1.2072133 -0.4506998  1.0698196
[2,] 0.5025639 -0.5239133 -1.2303481 -0.8847430
[3,] 0.3465198  1.3604751 -1.4901651 -0.2333460
```

   The mean for each row across columns is obtained with the `apply` function:

```
apply(m, MARGIN = 1, FUN = mean)
```

```
[1]  0.924015681 -0.534110136 -0.004129059
```

2. To center all columns in a matrix to have mean zero and to rescale the columns to have variance one we can to

```
scale(m, center = T, scale = T)
```

```
           [,1]       [,2]       [,3]       [,4]
[1,]  1.1496851  0.5025758  1.1209111  1.0911161
[2,] -0.4817396 -1.1516007 -0.3203124 -0.8728182
[3,] -0.6679454  0.6490250 -0.8005987 -0.2182979
attr(,"scaled:center")
[1]  0.90627112  0.68125835 -1.05707100 -0.01608982
attr(,"scaled:scale")
[1] 0.8380195 1.0465187 0.5409628 0.9952281
```

3. If instead of subtracting the mean we want to subtract the median we can do

```
row.med <- apply(m, MARGIN = 1, FUN = median)
sweep(m, MARGIN = 1, STATS = row.med, FUN = "-")
```

```
          [,1]       [,2]       [,3]       [,4]
[1,] 0.7312133 0.06869686 -1.5892162 -0.06869686
[2,] 1.2068921 0.18041483 -0.5260199 -0.18041483
[3,] 0.2899329 1.30388822 -1.5467520 -0.28993291
```

14.3 Groupwise calculations

Suppose we want to calculate the mean, median and variance of `rate` for each state. A simple way of doing this is by the `summaryBy` function in the `doBy` package:

```
library(doBy)
summaryBy(rate + log(rate) ~ state, data = Puromycin,
      FUN = c(mean, median, var))
```

```
      state rate.mean log(rate).mean rate.median log(rate).median
1   treated  141.5833       4.871207       145.5         4.979177
2 untreated  110.7273       4.650422       115.0         4.744932
  rate.var log(rate).var
1 2805.356     0.2053777
2 1334.218     0.1346631
```

14.4 Sorting data

To sort the Puromycin data e.g. by conc we can use the `orderBy` function in the `doBy` package

```
PuromycinOrder <- orderBy(~conc, data = Puromycin)
head(PuromycinOrder)
```

```
   conc rate     state    iconc  sqrtconc
1  0.02   76   treated 50.00000 0.1414214
2  0.02   47   treated 50.00000 0.1414214
13 0.02   67 untreated 50.00000 0.1414214
14 0.02   51 untreated 50.00000 0.1414214
3  0.06   97   treated 16.66667 0.2449490
4  0.06  107   treated 16.66667 0.2449490
```

15 Getting R output into Word or OpenOffice

There are two types of output from R which one often wants to get into a word processing program for writing a report or a scientific paper: Graphs and tables. We discuss below how to get the output into Microsoft Word. Note that the same procedure applies if you use OpenOffice `www.OpenOffice.org` which is a free office package very very similar to Microsoft Office.

15.1 Graphs

The easiest way to get a graph into word is as follows: When the graph window in R is active, click `File -> Copy to clipboard` and then choose one of the possible formats. Generally the metafile format gives the best results. Then go to your Word document and paste the graph into the document using Ctrl-V.

Alternatively you can go to `File -> Save as`. You must then choose a format for the file to be saved in. Word can read files of the type `Metafile`, `Jpeg`, `Png` and `BMP`. It is not so important which format you choose, except that we suggest not to use `BMP` format because these files can be quite large. After saving the file, you can go to word and include the file as a picture.

15.2 Tables etc.

To get tables etc. into Word, we first save the tables as an HTML file and then read this into Word. The `R2HTML` package can direct R output into a HTML file.

```
library(R2HTML)
HTMLStart(".", filename = "my-report", Title = "My report")
summary(rnorm(100))
table(c("here", "comes", "the", "sun", "sun", "sun"))
HTMLStop()
```

All R instructions which appear between `HTMLStart(...)` and `HTMLStop()` will be written to the file `my-report.html`. The first argument `"."` specifies that `my-report.html` will be put into the working directory. The file `my-report.html` can now be read into Word.

16 First reading

After working through this note, it is suggested to have a look at the on–line manual "An introduction to R[2]". On Windows platforms, this manual is also available from the menu `Help -> Manuals`. Note: Appendix A contains a sample session with R. This is a good place to start. There are several good introductions[3] available from the R project website.

Books on `R` that may be helpful are

- Dalgaard: Introductory Statistics with R, Springer-Verlag.
- Julian Faraway: Extending the linear model in R, Chapman-Hall/CRC.
- Venables and Ripley: Modern Applied Statistics with S, Springer-Verlag. Probably most suited for people familiar with statistics.
- Pinheiro and Bates: Mixed–Effects Models in S and S–PLUS, Springer Verlag. Probably most suited for people familiar with statistics.

[2] http://cran.r-project.org/doc/manuals/R-intro.pdf
[3] http://cran.r-project.org/other-docs.html