Forecast and Simulation Methods
A short introduction to R

Claudio Agostinelli
Dipartimento di Scienze Ambientali, Informatica e Statistica
Università Ca' Foscari, Venezia

Ver 1.0 – 4 February 2013

This document summarized the basic aspects of the S language as implemented in the R environment with emphasis on time series analysis. The document is a copy-and-paste, modified version of several material available at `http://cran.r-project.org`cran.r-project.org or in the internet. In particular we used the material in Kuhnert and Venables [2005], White [2009] plus material from the R websites (`http://www.r-project.org`www.r-project.org and `http://cran.r-project.org`cran.r-project.org).

1 Introduction to R

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R .

R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

One of R 's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

How to start

R can be download from the Comprehensive R Archive Network (CRAN) repository. The CRAN also includes detailed installation instructions and gentle introductions to the language.

- Download the software at: `cran.r-project.org`
- Installation instructions at: `http://cran.r-project.org/doc/FAQ/R-FAQ.html#How-can-R-be-installed_003f`
- An short introduction to the language: `http://cran.r-project.org/doc/contrib/Lemon-kickstart/index.html`

Main features

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

R is designed around a true computer language, and it allows users to add additional functionality by defining new functions. Much of the system is itself written in R , which makes it easy for users to follow the algorithmic choices made. For computationally-intensive tasks, C, C++ and Fortran code can be linked and called at run time. Advanced users can write C code to manipulate R objects directly.

R can be extended (easily) via packages. About 3450 packages are available through the CRAN family of Internet sites covering a very wide range of modern statistics.

1.1 Basic Syntax

It is important to learn some basic syntax of the R programming language before launching in to more sophisticated functions, graphics and modelling. Below is a compilation of some of the basic features of R that will get you going and help you to understand the R language.

R prompt

The default R prompt is the greater-than sign (>)

```
> 2 * 4
```

```
[1] 8
```

Continuation prompt

If a line is not syntactically complete, a continuation prompt (`+`) appears

```
> 2 *
+ 4
```

```
[1] 8
```

Assignment Operator

The assignment operator is the left arrow (`< -`) and assigns the value of the object on the right to the object on the left

```
> value <- 2 * 4
```

The contents of the object value can be viewed by typing value at the R prompt

```
> value
```

```
[1] 8
```

Last Expression

If you have forgotten to save your last expression, this can be retrieved through an internal object `.Last.value`

```
> 2 * 4
```

```
[1] 8
```

```
> value <- .Last.value
```

Removing Objects

The functions `rm()` or `remove()` are used to remove objects from the working directory

```
> rm(value)
> value
```

```
Error
```

Legal R Names

Names for R objects can be any combination of letters, numbers and periods (`.`) but they must not start with a number. R is also **case sensitive** so

```
> value <- 8
> value
```

```
[1] 8
```

is different from

```
> Value
```

```
Error : object 'Value' not found
```

Finding Objects

R looks for objects in a sequence of places known as the **search path**. The search path is a sequence of **environments** beginning with the **Global Environment**. You can inspect it at any time (and you should) by the `search()` function (or from the Misc menu). The `attach()` function allows copies of objects to be placed on the search path as individual components. The `detach()` function removes items from the search path.

Looking at the Search Path: An Example

```
> data(CO2)
> attach(CO2)
> search()

 [1] ".GlobalEnv"        "CO2"               "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"

> objects(1)

[1] "CO2"   "value"

> objects(2)

[1] "conc"      "Plant"     "Treatment" "Type"      "uptake"

> names(CO2)

[1] "Plant"     "Type"      "Treatment" "conc"      "uptake"

> find("CO2")

[1] ".GlobalEnv"       "package:datasets"
```

Assignments to Objects

Avoid using the names of built-in functions as object names. If you mistakenly assign an object or value to a built-in function and it is passed to another function you may get a warning but not always ... things may go wrong. R has a number of built-in functions. Some examples include `c`, `TRUE`, `FALSE`, `t`. An easy way to avoid assigning values/objects to built-in functions is to check the contents of the object you wish to use. This also stops you from overwriting the contents of a previously saved object.

```
> TRUE # Object with a built in R Value

[1] TRUE

> t # Built in R function

function (x)
UseMethod("t")
<bytecode: 0xad621b8>
<environment: namespace:base>
```

Space

s R will ignore extra spaces between object names and operators

```
> value <-  2   *   4
```

Spaces cannot be placed between the < and - in the assignment operator

```
> value < -2 * 4
```

```
[1] FALSE
```

Be careful when placing spaces in character strings

```
> value1 <- "Hello World"
```

is different to

```
> value2 <- "Hello  World"
> value1 == value2
```

```
[1] FALSE
```

Data Types

There are four atomic data types in R .

- Numeric

  ```
  > value <- 605
  > value
  ```

  ```
  [1] 605
  ```

- Character string <- ”Hello World” string
- Logical

  ```
  > 2 < 4
  ```

  ```
  [1] TRUE
  ```

- Complex number

  ```
  > cn <- 2 + 3i
  > cn
  ```

  ```
  [1] 2+3i
  ```

The attribute of an object becomes important when manipulating objects. All objects have two attributes, the mode and their length. The R function `mode` can be used to determine the mode of each object, while the function `length` will help to determine each object’s length.

```
> mode(10)
```

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

```
> length(10)

[1] 1

> mode("Hello world")

[1] "character"

> length("Hello world")

[1] 1

> mode(2 < 4)

[1] "logical"

> length(2 < 4)

[1] 1

> mode(3+2i)

[1] "complex"

> length(3+2i)

[1] 1

> mode(sin)

[1] "function"

> length(sin)

[1] 1
```

`NULL` objects are empty objects with no assigned mode. They have a length of zero.

```
> mode(NULL)

[1] "NULL"

> length(NULL)

[1] 0
```

1.1.1 Missing, Indefinite and Infinite Values

In many practical examples, some of the data elements will not be known and will therefore be assigned a missing value. The (default) code for missing values in R is `NA`. This indicates that the value or element of the object is unknown. Any operation on an `NA` results in an `NA`. The `is.na()` function can be used to check for missing values in an object.

```
> value <- c(3,6,23,NA)
> is.na(value)

[1] FALSE FALSE FALSE  TRUE

> any(is.na(value))

[1] TRUE

> na.omit(value)

[1]  3  6 23
attr(,"na.action")
[1] 4
attr(,"class")
[1] "omit"
```

Indefinite and Infinite values (`Inf`, `-Inf` and `NaN`) can also be tested using the `is.finite`, `is.infinite`, `is.nan` and `is.number` functions in a similar way as shown above.

These values come about usually from a division by zero or taking the log of zero.

```
> 5/0

[1] Inf

> log(0)

[1] -Inf

> 0/0

[1] NaN
```

Arithmetic and Logical Operators

The last few sections used a variety of arithmetic and logical operators to evaluate expressions. A list of arithmetic and logical operators are shown in Tablesbelow.

Arithmetic Operators

Operator	Description	Example
`+`	Addition	`2+5`
`-`	Subtraction	`2-5`
`*`	Multiplication	`2*5`
`/`	Division	`2/5`
ˆ	Exponentiation	2 ˆ 5
%/%	Integer Divide	5%/%2
%%	Modulo	5%%2

Logical Operators

Operator	Description	Example
==	Equals	`3==5`
!=	Not Equals	`3!=5`
<	Less Than	`3 < 5`
>	Greater Than	`3 > 5`
<=	Less Than or Equal To	`3 <= 5`
>=	Gretaer Than or Equal to	`3 >= 5`
&	Elementwise And	`3==5 & 4!= 5`
|	Elementwise Or	`3==5 | 4!= 5`
&&	Control And	`is.na(value) && value==1`
||	Control Or	`is.na(value) || value==1`
xor	Elementwise Exclusive Or	`xor(is.na(value1), value2 == 2)`
!	Logical Negation	`!is.na(value)`

The build-in constants `TRUE` and `FALSE` are logical values and `T` and `F` are variables with those values. This is the opposite for S-PLUS. Although they have a different nature they can be used interchangeable. However we suggest to always use the long form.

1.2 Documentation and help

- **R is fully documented**. From within R use the command `help.start()` to start an HTML browser interface to the help system.
- **Manuals**: Official and non–official manuals are available at the CRAN website. Manuals are available at `http://cran.r-project.org/manuals.html`, while non official manuals are available at `http://cran.r-project.org/other-docs.html`.
- **Task view**: Task view: An annotate discussion of what is available (`http://cran.r-project.org/web/views/`).
- Several **Mailing lists** (`http://www.r-project.org/mail.html`) might be used for simple user questions (not covered by the main documentation) or to get information about latest developments. Also, there are mailing lists devoted to specific topics, the so call R-SIG (Special Interest Group), e.g. `R-sig-ecology`.

Several functions can be used to get help from within the R console. To have help on a specific function, e.g. on `lm`, we can type

```
> help(lm)
```

or, simply

```
> ?"lm"
```

To get information on a specific package

```
> help(package="splines")
```

or to look for a specific function in all the available packages

```
> help("bs", try.all.packages=TRUE)
```

To search for the documentation

```
> help.search("linear models")
> ??"linear models"
> keyword??"data"
> title??"linear models"
```

1.3 Data Objects

The four most frequently used types of data objects in R are **vectors**, **matrices**, **data frames** and **lists**. A vector represents a set of elements of the same mode whether they are logical, numeric (integer or double), complex, character or lists. A matrix is a set of elements appearing in rows and columns where the elements are of the same mode whether they are logical, numeric (integer or double), complex or character. A data frame is similar to a matrix object but the columns can be of different modes. A list is a generalisation of a vector and represents a collection of data objects.

Creating Vectors

`c` function
The simplest way to create a vector is through the concatenation function, `c`. This function binds elements together, whether they are of character form, numeric or logical. Some examples of the use of the concatenation operator are shown in the following

```
> value.num <- c(3,4,2,6,20)
> value.char <- c("koala","kangaroo","echidna")
> value.logical <- c(FALSE,FALSE,TRUE,TRUE)
```

`rep` and `seq` Functions
The `rep` function replicates elements of vectors. For example,

```
> rep(5,6)

[1] 5 5 5 5 5 5
```

replicates the number 5, six times to create a vector. The `seq` function creates a regular sequence of values to form a vector. The following script shows some simple examples of creating vectors using this function.

```
> seq(from=2,to=10,by=2)

[1]  2  4  6  8 10

> seq(from=2,to=10,length=5)

[1]  2  4  6  8 10

> seq(along=c(1,6,11))

[1] 1 2 3
```

`c`, `rep` and `seq` functions As well as using each of these functions individually to create a vector, the functions can be used in combination. For example,

```
> c(1,3,4,rep(3,4),seq(from=1,to=6,by=2))
```

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

uses the `rep` and `seq` functions inside the concatenation function to create a vector. It is important to remember that elements of a vector are expected to be of the same **mode**. So an expression

```
> c(1:3,"a","b","c")
```

will produce an error message.

1.3.1 Creating Matrices

`dim` and `matrix` functions
The `dim` function can be used to convert a vector to a matrix value <- rnorm(6) dim(value) <- c(2,3) value attributes(value) This piece of script will fill the columns of the matrix. To convert back to a vector we simply use the `dim` function again.

```
> dim(value) <- NULL
> attributes(value)
```

```
NULL
```

Alternatively we can use the `matrix` function to convert a vector to a matrix

```
> matrix(data=value,nrow=2,ncol=3)
```

```
     [,1] [,2] [,3]
[1,]    3   23    3
[2,]    6   NA    6
```

If we want to fill by rows instead then we can use the following script

```
> matrix(data=value,nrow=2,ncol=3,byrow=TRUE)
```

```
     [,1] [,2] [,3]
[1,]    3    6   23
[2,]   NA    3    6
```

`rbind` and `cbind` functions To bind a row onto an already existing matrix, the `rbind` function can be used

```
> value <- matrix(data=rnorm(6),nrow=2,ncol=3,byrow=TRUE)
> rbind(value,c(1,1,2))
```

```
          [,1]       [,2]       [,3]
[1,] -1.516261 -1.2968599  3.2164940
[2,]  1.369237 -0.3718412 -0.7930436
[3,]  1.000000  1.0000000  2.0000000
```

To bind a column onto an already existing matrix, the cbind function can be used

```
> cbind(value,c(1,1))
```

```
          [,1]       [,2]       [,3] [,4]
[1,] -1.516261 -1.2968599  3.2164940    1
[2,]  1.369237 -0.3718412 -0.7930436    1
```

Data Frame

`data.frame` function
The function `data.frame` converts a matrix or collection of vectors into a data frame

```
> value.data.frame <- data.frame(matrix(data=value, nrow=2, ncol=3))
```

Another example joins two columns of data together.

```
> data.frame(rnorm(3),runif(3))
```

```
   rnorm.3.  runif.3.
1 0.2801852 0.8121004
2 0.3856315 0.7387293
3 0.1893322 0.6034560
```

Row and column names are already assigned to a data frame but they may be changed using the `names` and `row.names` functions. To view the row and column names of a data frame:

```
> names(value.data.frame)
```

```
[1] "X1" "X2" "X3"
```

```
> row.names(value.data.frame)
```

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

Alternative labels can be assigned by doing the following

```
> names(value.data.frame) <- c("C1","C2","C3")
> row.names(value.data.frame) <- c("R1","R2")
```

Names can also be specified within the `data.frame` function itself.

```
> data.frame(C1=rnorm(3),C2=runif(3),row.names=c("R1","R2","R3"))
```

```
           C1        C2
R1  1.0447429 0.3472907
R2 -1.0807418 0.8465924
R3  0.6349684 0.5429353
```

See also `dimnames`, `rownames` and `colnames` to manipulate row and column names for a matrix.

Accessing Elements of a Vector or Matrix

Accessing elements is achieved through a process called indexing. Indexing may be done by

- a vector of positive integers: to indicate inclusion
- a vector of negative integers: to indicate exclusion
- a vector of logical values: to indicate which are in and which are out
- a vector of names: if the object has a names attribute

For the latter, if a zero index occurs on the right, no element is selected. If a zero index occurs on the left, no assignment is made. An empty index position stands for the lot!

Indexing Vectors

The first example involves producing a random sample of values between one and five, twenty times and determining which elements are equal to 1.

```
> x <- sample(1:5, 20, rep=TRUE)
> x

 [1] 1 4 3 1 1 5 3 5 1 5 4 4 1 2 4 1 3 5 2 5

> x == 1

 [1]  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[13]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE

> ones <- (x == 1) # parentheses unnecessary
```

We now want to replace the ones appearing in the sample with zeros and store the values greater than 1 into an object called `y`.

```
> x[ones] <- 0
> x

 [1] 0 4 3 0 0 5 3 5 0 5 4 4 0 2 4 0 3 5 2 5

> others <- (x > 1) # parentheses unnecessary
> y <- x[others]
> y

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

The following command queries the `x` vector and reports the position of each element that is greater than 1.

The following command queries the `x` vector and reports the position of each element that is greater than 1.

```
> which(x > 1)

 [1]  2  3  6  7  8 10 11 12 14 15 17 18 19 20
```

Indexing Data Frames

Data frames can be indexed by either row or column using a specific name (that corresponds to either the row or column) or a number. Some examples of indexing are shown below. Indexing by column:

```
> value.data.frame[, "C1"] <- 0
> value.data.frame
```

```
   C1         C2         C3
R1  0 -1.2968599  3.2164940
R2  0 -0.3718412 -0.7930436
```

Indexing by row:

```
> value.data.frame["R1", ] <- 0
> value.data.frame
```

```
   C1         C2         C3
R1  0  0.0000000  0.0000000
R2  0 -0.3718412 -0.7930436
```

```
> value.data.frame[] <- 1:6
> value.data.frame
```

```
   C1 C2 C3
R1  1  3  5
R2  2  4  6
```

To access the first two rows of the matrix/data frame

```
> value.data.frame[1:2,]
```

```
   C1 C2 C3
R1  1  3  5
R2  2  4  6
```

To access the first two columns of the matrix/data frame:

```
> value.data.frame[,1:2]
```

```
   C1 C2
R1  1  3
R2  2  4
```

To access elements with a value greater than five we can use some subsetting commands and logical operators to produce the desired result.

```
> as.vector(value.data.frame[value.data.frame > 5])
```

```
[1] 6
```

Manipulating Data: An Example

The iris dataset (`iris3`) is a three dimensional dataset. One dimension is represented for each species: Setosa, Versicolor and Virginica. Each species has the sepal lengths and widths, and petal lengths and widths recorded. To make this dataset more manageable, we can convert the three-dimensional array into a d-dimensional data frame. To begin with, we examine the names of the three-dimensional array.

```
> data(iris3)
> dim(iris3)

[1] 50  4  3

> str(iris3)

 num [1:50, 1:4, 1:3] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 - attr(*, "dimnames")=List of 3
  ..$ : NULL
  ..$ : chr [1:4] "Sepal L." "Sepal W." "Petal L." "Petal W."
  ..$ : chr [1:3] "Setosa" "Versicolor" "Virginica"

> dimnames(iris3)

[[1]]
NULL

[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

[[3]]
[1] "Setosa"     "Versicolor" "Virginica"
```

We see that the first dimension has not been given any names. This dimension corresponds to the row names of the dataset for each species. The second dimension corresponds to the explanatory variables collected for each species. The third dimension corresponds to the species. Before coercing this three dimensional array into a two dimensional data frame, we first store the species name into a vector.

```
> Snames <- dimnames(iris3)[[3]]
```

We now convert the three dimensional array into a 150×3 matrix and coerce the matrix into a data frame.

```
> iris.df <- rbind(iris3[,,1],iris3[,,2],iris3[,,3])
> iris.df <- as.data.frame(iris.df)
```

Now we check the column names of the data frame.

```
> dim(iris.df)

[1] 150   4

> str(iris.df)

'data.frame':        150 obs. of  4 variables:
 $ Sepal L.: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal W.: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal L.: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal W.: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...

> names(iris.df)
```

```
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
```

Using the `Snames` vector, we create a species factor and bind it to the columns of `iris.df`.

```
> iris.df$Species <- factor(rep(Snames,rep(50,3)))
```

To check that we have created the data frame correctly, we print out the first five rows of the data frame.

```
> str(iris.df)

'data.frame':        150 obs. of  5 variables:
 $ Sepal L.: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal W.: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal L.: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal W.: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species : Factor w/ 3 levels "Setosa","Versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

> iris.df[1:5,]

  Sepal L. Sepal W. Petal L. Petal W. Species
1      5.1      3.5      1.4      0.2  Setosa
2      4.9      3.0      1.4      0.2  Setosa
3      4.7      3.2      1.3      0.2  Setosa
4      4.6      3.1      1.5      0.2  Setosa
5      5.0      3.6      1.4      0.2  Setosa
```

Creating Lists

Lists can be created using the `list` function. Like data frames, they can incorporate a mixture of modes into one list and each component can be of a different length or size. For example, the following is an example of how we might create a list from scratch.

```
> L1 <- list(x = sample(1:5, 20, rep=TRUE), y = rep(letters[1:5], 4),
+            z = rpois(20, 1))
> L1

$x
 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4

$y
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d"
[20] "e"

$z
 [1] 2 0 0 3 2 0 1 3 1 0 2 1 2 1 3 0 1 1 0 0
```

There are a number of ways of accessing the first component of a list. We can either access it through the name of that component (if names are assigned) or by using a number corresponding to the position the component corresponds to. The former approach can be performed using subsetting (`[[]]`) or alternatively, by the extraction operator (`$`). Here are a few examples:

```
> L1[["x"]]

 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4

> L1$x

 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4

> L1[[1]]

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

To extract a sublist, we use single brackets. The following example extracts the first component only.

```
> L1[1]

$x
 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4
```

Working with Lists
The length of a list is equal to the number of components in that list. So in the previous example, the number of components in `L1` equals 3. We confirm this result using the following line of code:

```
> length(L1)

[1] 3
```

To determine the names assigned to a list, the names function can be used. Names of lists can also be altered in a similar way to that shown for data frames.

```
> names(L1) <- c("Item1","Item2","Item3")
```

Indexing lists can be achieved in a similar way to how data frames are indexed:

```
> L1$Item1[L1$Item1 > 2]

[1] 4 4 4 4 4 4 3 4 4
```

Joining two lists can be achieved either using the concatenation function or the append function. The following two scripts show how to join two lists together using both functions. Concatenation function:

```
> L2 <- list(x=c(1,5,6,7), y=c("apple","orange","melon","grapes"))
> c(L1,L2)

$Item1
 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4

$Item2
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d"
[20] "e"
```

```
$Item3
 [1] 2 0 0 3 2 0 1 3 1 0 2 1 2 1 3 0 1 1 0 0

$x
[1] 1 5 6 7

$y
[1] "apple"  "orange" "melon"  "grapes"
```

Append Function:

```
> append(L1,L2,after=2)
```

```
$Item1
 [1] 4 2 2 2 1 1 2 4 4 1 2 4 2 2 4 4 3 1 4 4

$Item2
 [1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b" "c" "d"
[20] "e"

$x
[1] 1 5 6 7

$y
[1] "apple"  "orange" "melon"  "grapes"

$Item3
 [1] 2 0 0 3 2 0 1 3 1 0 2 1 2 1 3 0 1 1 0 0
```

Adding elements to a list can be achieved by

- adding a new component name:

  ```
  > L1$Item4 <- c("apple","orange","melon","grapes")
  > # alternative way
  > L1[["Item4"]] <- c("apple","orange","melon","grapes")
  ```

- adding a new component element, whose index is greater than the length of the list

  ```
  > L1[[4]] <- c("apple","orange","melon","grapes")
  > names(L1)[4] <- c("Item4")
  ```

  There are also many functions within R that produce a list as output. Examples of these functions include `spline()`, `density()` and `locator()`.

1.4 Reading and Writing Datasets

Small to moderate size data sets can be easily handled using tools presented so far. However, quite often we have a garden variety of data sources from data handling programs. By far the easiest way to import and export the data in R is using text files. Save the data in plain text format which may be imported to a different software. That way, you can easily view the data using any of the capable text editor even when the original software that produced the data is no longer available.

Main functions

There are a few principal functions reading data into R .

- `scan` read data into a vector or list from the console or file
- `read.table`, `read.csv`, for reading tabular data
- `read.fwf` can be used to read in data files that have a fixed width format
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (inverse of `dump`)
- `dget`, for reading in R code files (inverse of `dput`)
- `load`, for reading in saved workspaces (inverse of `save` and `save.image`)

When reading from excel files, the simplest method is to save each worksheet separately as a csv file and use `read.csv()` on each. A better way is to open a data connection to the excel file directly and use the odbc facilities.

There are analogous functions for writing data to files

- `write.table`
- `writeLines`
- `dump`
- `dput`
- `save`, `save.image`

`write.table()` outputs the specified data frame to a file. A blank space is used to separate columns when `sep=" "` is specified within its argument. Other popular choices include comma (`sep=","`), and tab (`sep=""`).

```
> data(CO2)
> write.table(CO2, file="./data/CO2.txt", sep=";")
```

On the other hand, `read.table()` reads in an external text file and creates a data frame. For example, since the first line of the text data file CO2.txt consists of variable names, the following command will do the job:

```
> dati <-read.table("./data/CO2.txt", header=TRUE, sep=";")
```

`getwd()` returns the current working directory and `setwd()` changes it. For instance, my actual setting is

```
> percorso <- getwd()
> percorso
```

```
[1] "/home/claudio/didattica/lezioni/forecastsimulationmethods/Rnw"
```

```
> setwd(paste(percorso, "/data/", sep=""))
> getwd()
```

```
[1] "/home/claudio/didattica/lezioni/forecastsimulationmethods/Rnw/data"
```

```
> CO2again <- read.table(file="CO2.txt", header=TRUE, sep=";")
> setwd(percorso)
```

For small to moderately sized datasets, you can usually call `read.table` without specifying any other arguments. R will automatically skip lines that begin with a #, figure out how many rows there are (and how much memory needs to be allocated), figure what type of variable is in each column of the table. Telling R all these things directly makes R run faster and more efficiently. `read.csv` is identical to `read.table` except that the default separator is a comma.

With much larger datasets, doing the following things will make your life easier and will prevent R from choking. Read the help page for `read.table`, which contains many hints. Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here. Set comment.char = "" if there are no commented lines in your file. For writing larger datasets, `write.matrix` (in package "MASS") may be more efficient in terms of how memory is handled with respect to `write.table`. Data can be output in blocks using the `blocksize` argument as shown in the following script

```
> require(MASS)
> write.matrix(myData, file="myData.csv", sep=";", blocksize=1000)
```

Interfaces to the Outside World

Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

Connections

```
> str(file)

function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),
    raw = FALSE)
```

`description` is the name of the file, `open` is a code indicating "r" read only, "w" writing (and initializing a new file), "a" appending "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
> con <- file("foo.txt", "r")
> data <- read.csv(con)
> close(con)
```

is the same as

```
> data <- read.csv("foo.txt")
```

Importing Binary Files

Binary data written from another statistical package can be read into R (but really should be avoided). The R package `foreign` provides import facilities for: EpiInfo, Minitab, S-Plus, SAS, SPSS, Stat and Systat binary files. Here is a list of them.

- `read.epiinfo` reads in EpiInfo text files
- `read.mtp` imports Minitab worksheets
- `read.xport` reads in SAS files in TRANSPORT format
- `read.S` reads in binary objects produced by S-PLUS 3.x, 4.x or 2000 on (32-bit) Unix or Windows Data dumps from S-PLUS 5.x and 6.x using `dump(..., oldStyle=T)` can be read using the `data.restore` function
- `read.spss` reads in files from SPSS created by the save and export commands
- `read.dta` reads in binary Stata files
- `read.systat` reads in rectangular files saved in Systat

Saving Data in Non-tabular Forms

For temporary storage or for transport, it is more efficient to save data in (compressed) binary form using `save` or `save.image`.

```
> x <- 1
> y <- data.frame(a = 1, b = "a")
> save(x, y, file = "./data/example.RData")
> load("./data/example.RData") ## overwrites existing x and y!
```

Binary formats are not great for long-term storage because if they are corrupted, recovery is usually not possible.

Deparsing R Objects

Another way to pass data around is by deparsing the R object with `dput` and reading it back in using `dget`.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)

structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",
"b"), row.names = c(NA, -1L), class = "data.frame")

> structure(list(a = 1,
+ b = structure(1L, .Label = "a",
+ class = "factor")),
+ .Names = c("a", "b"), row.names = c(NA, -1L),
+ class = "data.frame")

  a b
1 1 a
```

```
> dput(y, file = "./data/example.Rput")
> new.y <- dget("./data/example.Rput")
> new.y

  a b
1 1 a
```

Dumping R Objects
Multiple objects can be deparsed using the dump function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "./data/example.Rdump")
> rm(x, y)
> source("./data/example.Rdump")
> y

  a b
1 1 a

> x

[1] "foo"
```

Textual Formats
Dumping and dputing are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable. Unlike writing out a table or csv file, dump and dput preserve the metadata (sacrificing some readability), so that another user does not have to specify it all over again.

Textual formats work much better with programs like **git** which can only track changes meaningfully in text files. Textual formats adhere to the "Unix philosophy".

Reading Lines of a Text File
The `readLines` function can be used to simply read lines of a text file and store them in a character vector.

```
> con <- gzfile("./data/CO2.txt.gz")
> x <- readLines(con, 10)
> x

 [1] "\"Plant\";\"Type\";\"Treatment\";\"conc\";\"uptake\""
 [2] "\"1\";\"Qn1\";\"Quebec\";\"nonchilled\";95;16"
 [3] "\"2\";\"Qn1\";\"Quebec\";\"nonchilled\";175;30.4"
 [4] "\"3\";\"Qn1\";\"Quebec\";\"nonchilled\";250;34.8"
 [5] "\"4\";\"Qn1\";\"Quebec\";\"nonchilled\";350;37.2"
 [6] "\"5\";\"Qn1\";\"Quebec\";\"nonchilled\";500;35.3"
 [7] "\"6\";\"Qn1\";\"Quebec\";\"nonchilled\";675;39.2"
 [8] "\"7\";\"Qn1\";\"Quebec\";\"nonchilled\";1000;39.7"
 [9] "\"8\";\"Qn2\";\"Quebec\";\"nonchilled\";95;13.6"
[10] "\"9\";\"Qn2\";\"Quebec\";\"nonchilled\";175;27.3"
```

```
> conx <- textConnection(x)
> read.table(conx, sep=";", header=TRUE)

  Plant   Type  Treatment conc uptake
1   Qn1 Quebec nonchilled   95   16.0
2   Qn1 Quebec nonchilled  175   30.4
3   Qn1 Quebec nonchilled  250   34.8
4   Qn1 Quebec nonchilled  350   37.2
5   Qn1 Quebec nonchilled  500   35.3
6   Qn1 Quebec nonchilled  675   39.2
7   Qn1 Quebec nonchilled 1000   39.7
8   Qn2 Quebec nonchilled   95   13.6
9   Qn2 Quebec nonchilled  175   27.3
```

`writeLines` takes a character vector and writes each element one line at a time to a text file. `readLines` can be useful for reading in lines of webpages

```
> con <- url("http://www.unive.it", "r")
> x <- readLines(con)
> head(x)

[1] "   "
[2] "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" \"http://www.w3.org/
[3] "\t<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"it\" lang=\"it\">"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\" />"
[6] "\t<link href=\"http://www.unive.it/css3/stile.css\" rel=\"stylesheet\" type=\"text/cs
```

Editing Data

The functions edit and fix allow you to make changes to data files using a default editor. This is useful for small datasets and for obtaining a snapshot of your data without writing to file or printing to the screen. The fix function allows you to make edits and then assigns the new edited version to the workspace

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

The edit function invokes a text editor on the object, the result of which is a copy that can be assigned to a new object.

```
> y <- edit(x)
```

1.5 A tour on Graphics

Overview of Graphics Functions

R has a variety of graphics functions. These are generally classed into

- High-level plotting functions that start a new plot
- Low-level plotting functions that add elements to an existing plot

Each function has its own set of arguments. The most common ones are

- `xlim`, `ylim`: range of variable plotted on the x and y axis respectively
- `pch`, `col`, `lty`: plotting character, colour and line type
- `xlab`, `ylab`: labels of x and y axis respectively
- `main`, `sub`: main title and sub-title of graph

General graphing parameters can be set using the `par()` function. For example, to view the setting for line type

```
> par()$lty
```

```
[1] "solid"
```

To set the line type using the par function

```
> par(lty=2)
```

Multiple Plots

There are two main ways of placing several plots on the one surface. The graphics parameter fig allows you to place several plots, possibly irregularly, on the one figure region. It is also possible, and more common to have more than one figure to a page as a regular $n \times m$ array of figures. This behaviour is controlled by the `mfrow` or `mfcol` graphics parameter. For example

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

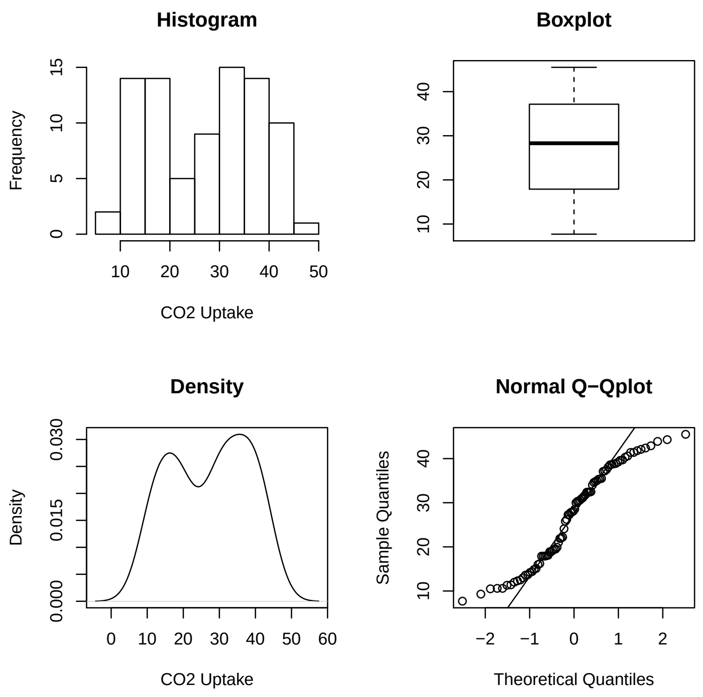
will produce a plotting region with three rows and two columns. Each high-level plotting command starts plotting on a new figure. When all figures are exhausted, a new page is generated. The `mfg` graphics parameter keeps track of the row and column of the current figure and the dimensions of the figure array. By setting this parameter unusual figure arrangements can be achieved.

Displaying Univariate Data

Graphics for univariate data are often useful for exploring the location and distribution of observations in a vector. Comparisons can be made across vectors to determine changes in location or distribution. Furthermore, if the data correspond to times we can use time series methods for displaying and exploring the data. Graphical methods for exploring the distributional properties of a vector include

- `hist` (histogram)
- `boxplot`
- `density`
- `qqnorm` (Normal quantile plot) and `qqline`

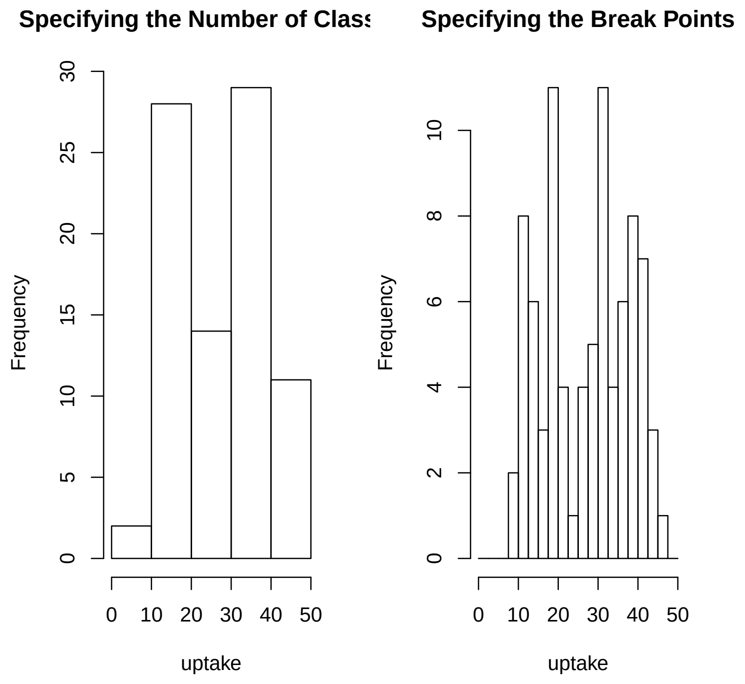
```
> data(CO2)
> uptake <- CO2$uptake
> par(mfrow=c(2,2))
> # Histogram
> hist(uptake, xlab="CO2 Uptake", main="Histogram")
> # Boxplot
> boxplot(uptake, main="Boxplot")
> # Density
> plot(density(uptake),type="l", xlab="CO2 Uptake", main="Density")
> # Q-Q Plot
> qqnorm(uptake, main="Normal Q-Qplot")
> qqline(uptake)
```



Histograms

Histograms are a useful graphic for displaying univariate data. They break up data into cells and display each cell as a bar or rectangle, where the height is proportional to the number of points falling within each cell. The number of breaks/classes can be defined if required. The following shows example code for producing histograms.

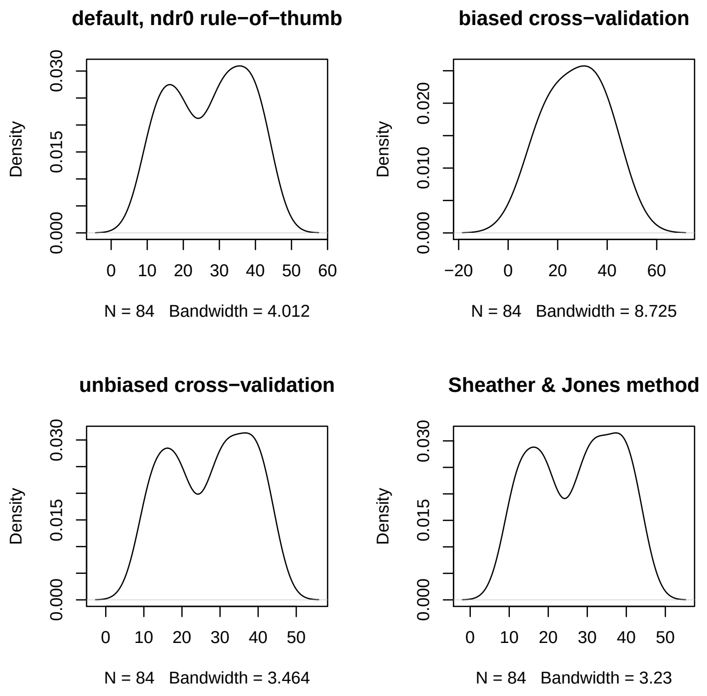
```
> par(mfrow=c(1,2))
> hist(uptake, nclass=4, main="Specifying the Number of Classes")
> hist(uptake, breaks=seq(from=0,to=50,by=2.5), main="Specifying the Break Points")
```



Densities

Densities can be used to compute smoothed representations of the observed data. The function density produces kernel density estimates for a given kernel and bandwidth. By default, the Gaussian kernel is used but there is an array of other kernels available in R . Look up the R help on density and see what options are available. The bandwidth controls the level of smoothing. By default, this represents the standard deviation of the smoothing kernel but this too, can be changed depending on your requirements. The following script produces a range of smoothed densities for the `uptake` variable in the `CO2` dataset, using different methods to determined the bandwidth.

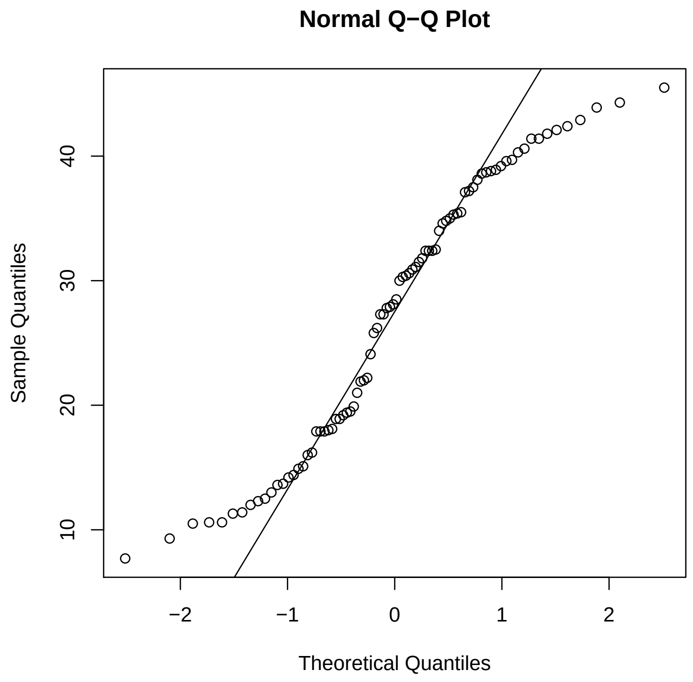
```
> par(mfrow=c(2,2))
> plot(density(uptake), type="l", main="default, ndr0 rule-of-thumb")
> plot(density(uptake, bw=bw.bcv(uptake, upper=10)), type="l", main="biased cross-validati
> plot(density(uptake, bw=bw.ucv(uptake, upper=10) ), type="l", main="unbiased cross-valid
> plot(density(uptake, bw="SJ"),type="l", main="Sheather & Jones method")
```



Quantile-Quantile Plots

Quantile-quantile plots are useful graphical displays when the aim is to check the distributional assumptions of your data. These plots produce a plot of the quantiles of one sample versus the quantiles of another sample and overlays the points with a line that corresponds to the theoretical quantiles from the distribution of interest. If the distributions are of the same shape then the points will fall roughly on a straight line. Extreme points tend to be more variable than points in the centre. Therefore you can expect to see slight departures towards the lower and upper ends of the plot. The function `qqnorm` compares the quantiles of the observed data against the quantiles from a Normal distribution. The function `qqline` will overlay the plot of quantiles with a line based on quantiles from a theoretical Normal distribution.

```
> qqnorm(uptake)
> qqline(uptake)
```



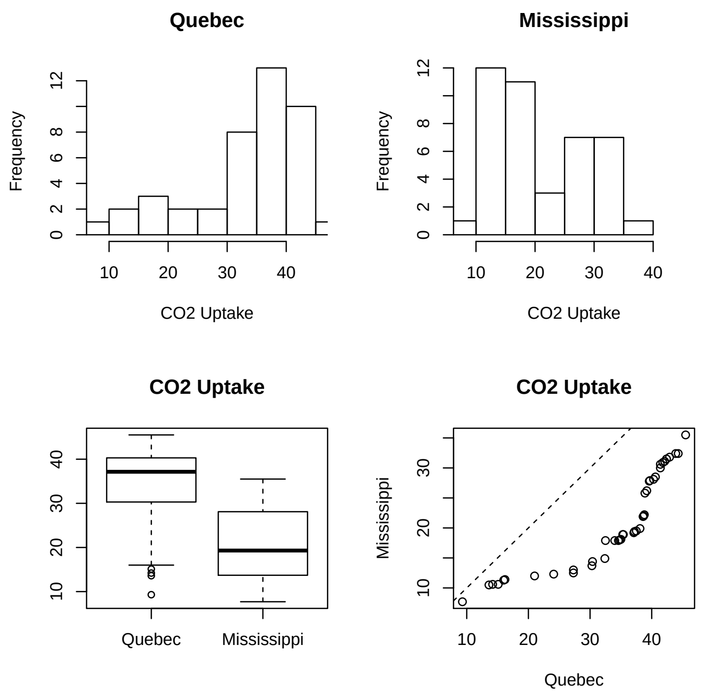
Comparing Groups

There may be instances where you want to compare different groupings to investigate differences between location and scale and other distributional properties. There are a couple of graphical displays to help with these types of comparisons.

- Multiple histograms plotted with the same scale
- Boxplots split by groups
- Quantile-Quantile plots

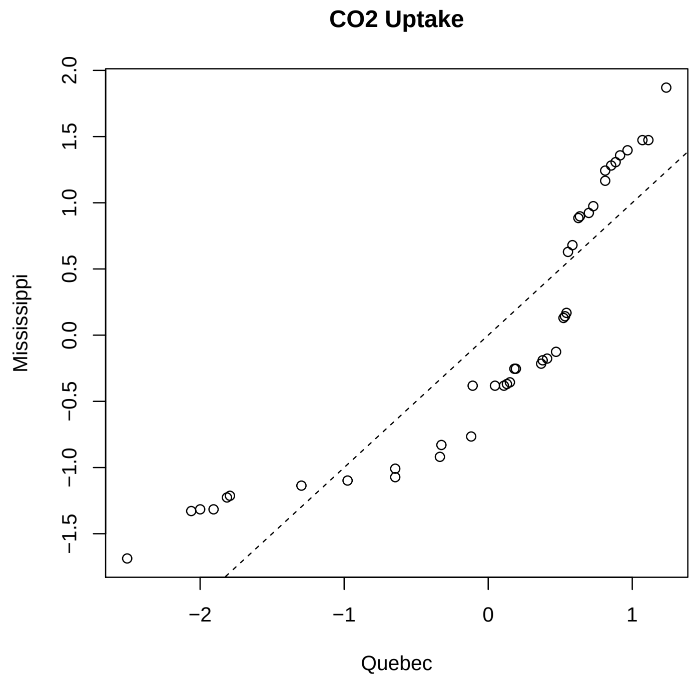
These plots enable the comparison of quantiles between two samples to determine if they are from similar distributions. If the distributions are similar, then the points should lie in a straight line (roughly). Gaps between tick mark labels indicate differences in location and scale for the two datasets. The following script produces histograms, boxplots and quantile-quantile plots to enable comparison between observations taken in the two regions.

```
> uptakeXregion <- split(uptake,CO2$Type)
> # Set up plotting region
> par(mfrow=c(2,2))
> # Produce histograms to compare each dataset
> hist(uptakeXregion[[1]], xlim=range(uptake), main="Quebec", xlab="CO2 Uptake")
> hist(uptakeXregion[[2]], xlim=range(uptake), main="Mississippi", xlab="CO2 Uptake")
> # Produce boxplot split by type of driving
> boxplot(uptakeXregion, names=c("Quebec","Mississippi"), main="CO2 Uptake")
> # Q-Q plot to check distribution shape and scale
> qqplot(uptakeXregion[[1]], uptakeXregion[[2]], main="CO2 Uptake", xlab="Quebec", ylab="M
> abline(0,1, lty=2)
```



To look for difference in the shape of the two distributions regardless location and scale we can use `scale` to standardize the two set of observations

```
> qqplot(scale(uptakeXregion[[1]]), scale(uptakeXregion[[2]]), main="CO2 Uptake", xlab="Qu
> abline(0,1, lty=2)
```



which shows a difference in kurtosis between the two distributions.

Getting Out Graphics

There are four ways to output graphics from R . Some are more straight forward than others and it will depend on how you want to use and display the graphics. The following lists the choices available:

- The postscript function will start a graphics device and produce an encapsulated postscript file containing the graphic;
  - This is a flexible function that allows you to specify the size and orientation of the graphic.
  - These can be incorporated directly into LATEXusing the includegraphics command

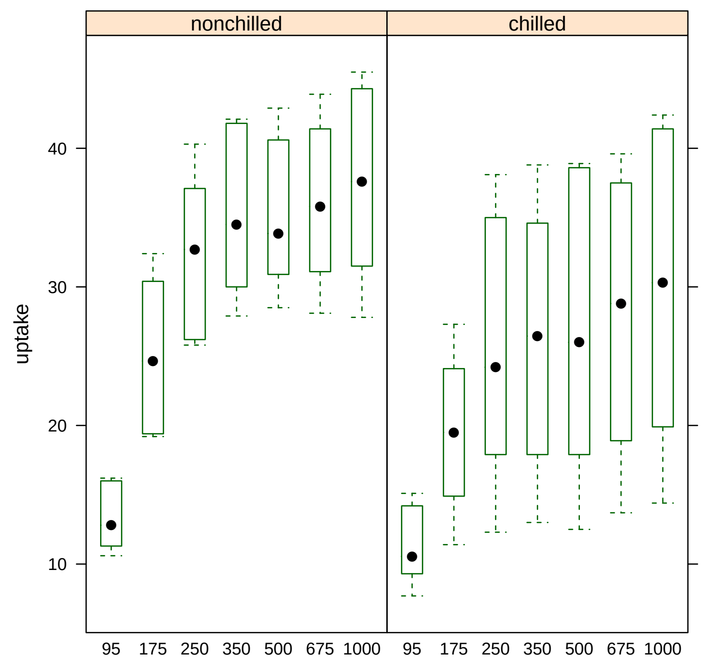
```
> postscript("./data/graph.ps", paper="a4")
>   hist(rnorm(100))
> dev.off()
```

```
pdf
  2
```

- Other functions are available within R and can be used in a similar way: `windows`, `pdf`, `pictex`, `png`, `jpeg`, `bmp` and `xfig`;
- Under Windows, a simpler, less flexible version is achieved via the R Graphics window. A graphic can be saved by going "File -> Save As". Many options are listed: metafile, postscript, pdf, png, bitmap or jpeg.

- Under Windows, graphics can be copied to the clipboard and pasted into Windows applications such as Word, Excel or Powerpoint "File -> Copy to the clipboard -> as Metafile".
- Printing directly from R. This can be achieved using the print feature on the R Graphics Window.

```
> require(lattice) # trellis graphics
> trellis.par.set(col.whitebg())
> data(CO2)
> print(bwplot(uptake~I(factor(conc)) | Treatment, data=CO2))
```



1.6 Object Oriented Programming, a small example on S3

The first thing you should know is that the S3 object-oriented system feels like a hack that was put on top of the original S language. If you are familiar with Perl language, you will feel right at home with the basic ideas behind this approach; you will probably also appreciate the conceptual simplicity of the results.

With that in mind, you should be aware that any normal piece of R code is filled with objects, because the base packages and most well-designed user packages use lots of custom classes. You can see this for yourself by starting to use `class()` to perform introspection on some of the items in your programs:

```
> class(1)

[1] "numeric"
```

```
> class('a')

[1] "character"

> x <- 1:10
> y <- 5 * x + rnorm(length(x), 0, 1)
> fit <- lm(y ~ x)
> class(fit)

[1] "lm"
```

More interesting than looking at the classes of existing objects is defining new classes of your own. Thankfully, `class()`, like `names()` and many other functions in R , can be assigned to directly, like so:

```
> a <- 1
> class(a) <- 'octonion'
> class(a)

[1] "octonion"
```

Recall that, the only thing that decides the class of an object is the value set for the class attribute. To convince yourself that the class of an object is just an attribute, you can use `attr`:

```
> attr(a, 'class')

[1] "octonion"
```

This simple metadata driven approach to building objects is the core of the S3 object system. Of course, setting class by itself is pretty useless: you want to be able to define class methods. That's where the other two main ideas come in. The first trick is to use **generic methods** to get polymorphism out of the language; the second trick is to define methods on your classes using a simple naming convention. To see how this works, let's use an example.

We will define a new class ”serie” to handle our time series data (R already have all the infrastructure necessary to handle time series, our example is just for illustrating the Object Oriented S3 capabilities of R). To define an object of class ”serie”, we can do the following:

```
> dati <- list(giorno =sample(1:10, size=10, replace=FALSE),
+              temperatura = rnorm(10, 20, 1),
+              pressione = rnorm(10, 950, 10))
> class(dati) <- "serie"
> str(dati)

List of 3
 $ giorno     : int [1:10] 5 10 6 2 4 8 3 7 1 9
 $ temperatura: num [1:10] 19.5 19.7 20.2 18.7 19.1 ...
 $ pressione  : num [1:10] 949 952 945 924 962 ...
 - attr(*, "class")= chr "serie"

> dati
```

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

$temperatura
 [1] 19.48504 19.73575 20.22332 18.74423 19.07785 19.12869 20.07046 20.36681
 [9] 18.07868 17.93839

$pressione
 [1] 949.4684 952.2366 945.1895 924.3262 962.2658 937.3968 947.1404 944.5386
 [9] 947.6587 941.0456

attr(,"class")
[1] "serie"
```

The first thing that comes to mind after building this object is that we should define getter and setter methods for accessing and modifying the contents of this ”serie” class object. For example, we want a way to get the `giorno` element for our object. We would like our approach to generalize to other objects with an `giorno` element. Ideally, we should be able to write something like this:

```
> giorno(object)
```

Now, it is obviously possible to define a `giorno()` function that operates only on ”serie” objects, but that's not the right approach if we also want to have another sort of object that would have a `giorno` method as well. This polymorphism concern is the problem that generic methods and specialized naming conventions solve. First, we are going to define a specialized `giorno` method that only operates on ”serie” objects. Because it will only work on ”serie” objects, we will call it giorno.serie:

```
> giorno.serie <- function(object) {
+   return(object[['giorno']])
+ }
> giorno.serie(dati)

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

Given this, we can then define a generic method that will operate on objects of many classes and reroute our general function calls to the correct class-level method:

```
> giorno <- function(object) {
+   UseMethod('giorno', object)
+ }
> giorno(dati)

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

Here `UseMethod` just searches for an `giorno.serie()` function, finds it and then calls it with `dati` as its argument. If we had a ”matrice” class object, `giorno` would search for an `giorno.matrice()` function and call that. You can see this by trying `giorno` on a variable whose class we set to be ”matrice” using class.

```
> matrice <- list(giorno = 1)
> class(matrice) <- "matrice"
```

```
> giorno(matrice)
> # Error in UseMethod("giorno", object) :
> #  no applicable method for 'giorno' applied to an object of class "matrice"
```

With these ideas in mind, it is easy to do something similar for the rest of the elements in our object:

```
> temperatura.serie <- function(object) {
+   return(object[['temperatura']])
+ }
> temperatura <- function(object) {
+   UseMethod('temperatura', object)
+ }
> pressione.serie <- function(object){
+   return(object[['pressione']])
+ }
> pressione <- function(object) {
+   UseMethod('pressione', object)
+ }
```

Defining setter methods is a little more tricky. For, instance

```
> "temperatura<-" <- function(x, value) {
+ UseMethod("temperatura<-")
+ }
> "temperatura<-.serie" <- function(x, value){
+   if (class(x) == "serie") {
+     if (length(x$giorno)==length(value))
+       x$temperatura <- value
+     else
+       stop("The length of 'value' is not appropriate")
+   }
+   return(x)
+ }
> temperatura(dati) <- rnorm(10, 9, 1)
> dati$temperatura
```

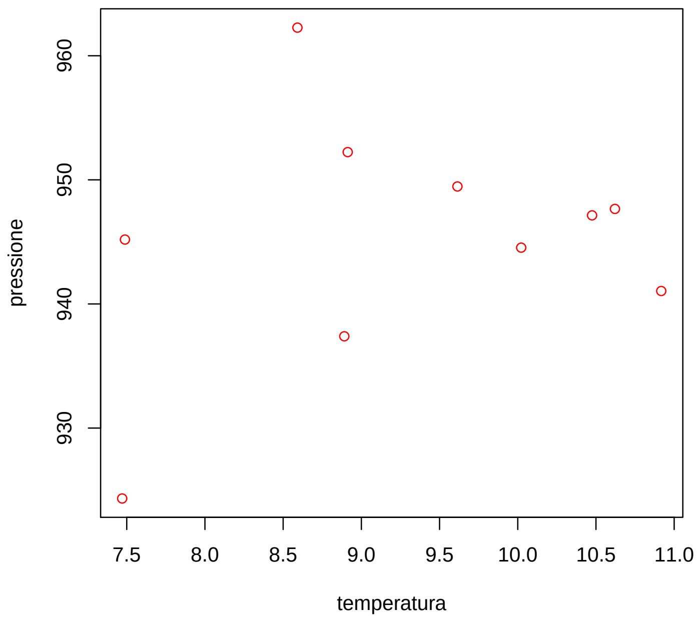
```
 [1]  9.614280  8.912722  7.488487  7.470829  8.591378  8.891064 10.475117
 [8] 10.022033 10.621362 10.917448
```

Or we can define methods for already existing generic functions

```
> print.serie <- function(x, ...) {
+   cat("\n Serie di dati con lunghezza ", length(x$giorno), "\n")
+   cat("Sintesi sulla temperatura\n")
+   print(summary(x$temperatura))
+   cat("Sintesi sulla pressione\n")
+   print(summary(x$pressione))
+   invisible(x)
+ }
> print(dati)
```

```
 Serie di dati con lunghezza  10
Sintesi sulla temperatura
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  7.471   8.666   9.264   9.300  10.360  10.920
Sintesi sulla pressione
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  924.3   941.9   946.2   945.1   949.0   962.3
```

```
> plot.serie <- function(x, xlab="temperatura", ylab="pressione", ...) {
+   plot.default(x=x$temperatura, y=x$pressione, xlab=xlab, ylab=ylab, ...)
+   invisible(x)
+ }
> plot(dati, col=2)
```



References

Petra Kuhnert and Bill Venables. *An Introduction to R: Software for Statistical Modelling & Computing, Course Material and Exercises.* CSIRO Mathematical and Information Sciences, Cleveland, Australia, 2005. URL `http://cran.r-project.org/other-docs.html`.

John Myles White. The most basic elements of object-oriented programming in R. Blog Post, 2009. URL `http://www.johnmyleswhite.com/notebook/2009/12/13/the-most-basic-elements-of-object-oriented-programming-in-r/`.