



University of Torino, Italy  
Earth Science Department



# R Summer School

**COST Action ES0601**

**HOME: Advances in HOMogenisation MEmods for climate series: an integrated approach**

September 7<sup>th</sup>-10<sup>th</sup>, 2009

Physics Department, University of Patras, Greece

*Silvia Terzago*

---

# Overview

- R: introduction and basics
- Reading and writing files
- Programming
- Introduction to graphics

---

# Introduction and Basics

---

---

# Introduction

Statistics:

A collection of methods to transform *data* in *information*.

Steps for a good statistical analysis:

1. Accurate collection and “preparation” of data (long time is required!)
2. Development and/or application of statistical methods.

Many statistical software can help in the second step.

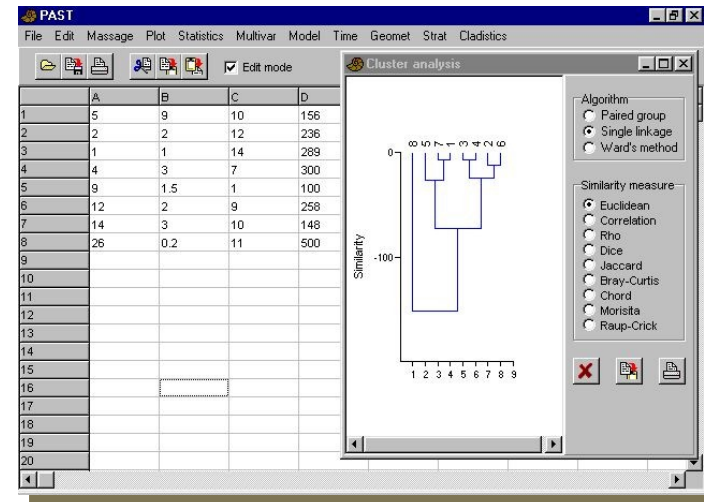
---

# Statistical Software

- “Spreadsheet”  
Excell, OpenCalc,...
- Commercial statistical software  
SPSS, Statistica, Matlab,...
- Free statistical software  
R, PAST,...
- Software developed *ad hoc* by the user

# Data management

- “Spreadsheet”  
(Excell, OpenOffice, PAST,...)



- Console with command line  
(R, Matlab,...)

```

#####
o          Dataplot          o
o  Interactive Graphics & Data Analysis Language  o
o      James J. Filliben & Alan Heckert          o
o  Computing & Applied Mathematics Laboratory    o
o  National Institute of Standards & Technology  o
o      301-975-2855 & 301-975-2899              o
#####

Number of Observations per Variable = 20000
Number of Variables                  = 10
Total Internal Data Space Size      = 200000
Substitution/Replacement Character = ^

1. Welcome to DATAPLOT (version 8/1999). For assistance at
   any time during a run, enter HELP .

2. For a list of new commands/capabilities,
   enter NEWS (updated 8/1999)

3. The alternate Graphics output files (creatable
   via the DEVICE 2 and 3 commands) are
   dgm11f.tex and dgm12f.tex respectively.

Paused | Input pending in Graphic1
```

---

# Spreadsheet

## Advantages:

- World-wide diffusion
- Usually very easy to use
- Free software (as OpenOffice Calc) are available

## Disadvantages:

- Limited statistical functions

---

# Commercial software

## Advantages:

- Many statistical functions
- Excellent manuals, help

## Disadvantages:

- Very expensive!
  - Big-size program
  - Quite complicated to use
-

---

# Open Source Statistical Software

## Advantages:

- FREE
- Oriented for specific fields of research
- Frequently updated
- Community of users's support

## Disadvantages:

- Not so easy to use

---

# R Software

---

# What is R?



R is a “*language* and *environment*” for statistical computing”

implementation of the **S** *programming language* which was developed at Bell Laboratories by Rick Becker, John Chambers and Allan Wilks

R as an “environment” within which statistical techniques can be implemented.

All R functions are stored in “*packages*”

---

## Base packages

They contain the basic functions that allow R to work (e.g. functions for data manipulation, basic statistical analysis and graphics).

They are considered part of the R source code and they are automatically available anytime you run R.

## Contributed packages

These packages allow to implement specialized statistical methods. Some of them (the *recommended* packages) are distributed with any binary distribution of R, the others can be downloaded from the net.

About 2000 packages are available at the moment!

---

# Why R?

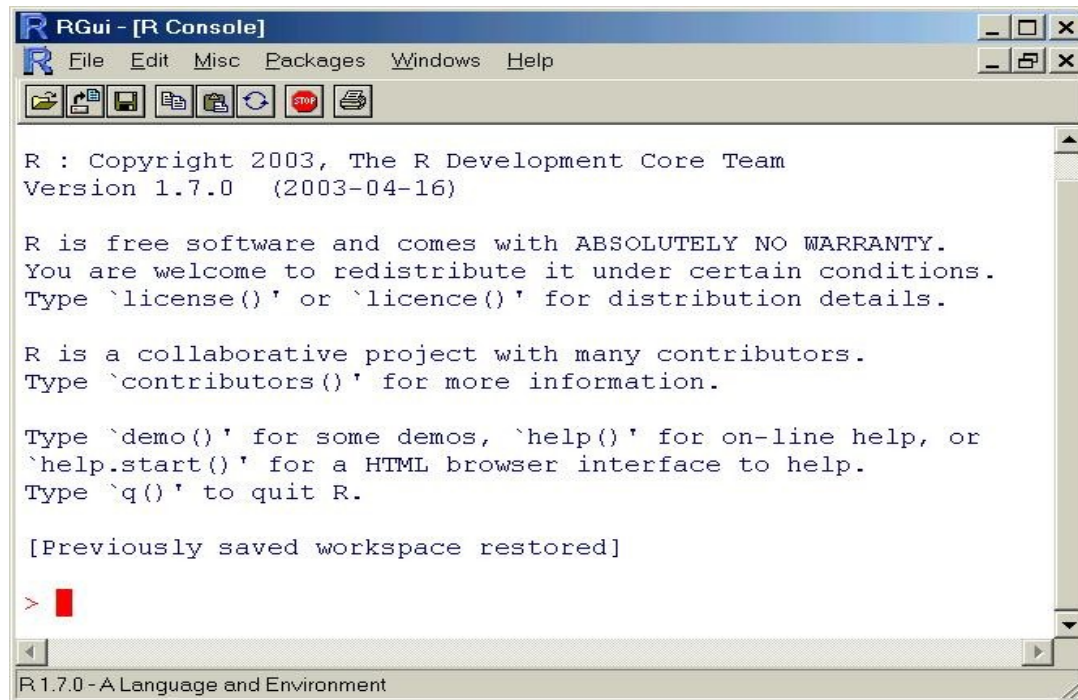
- Free Software
- Available for Windows, Linux, MacOS
- Easy data manipulation and storage
- Graphical facilities for data analysis and display
- Unlimited possibilities for statistical analysis (flexible)
- Developement of user-defined functions
- Excellent help

... but

some time is required to get started to the R language

R interface is a command line: when you start R it will appear the prompt:

>\_



The screenshot shows the RGui - [R Console] window. The title bar reads "RGui - [R Console]". The menu bar includes "File", "Edit", "Misc", "Packages", "Windows", and "Help". The toolbar contains icons for file operations and a stop button. The main text area displays the following output:

```
R : Copyright 2003, The R Development Core Team
Version 1.7.0 (2003-04-16)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type `license()` or `licence()` for distribution details.

R is a collaborative project with many contributors.
Type `contributors()` for more information.

Type `demo()` for some demos, `help()` for on-line help, or
`help.start()` for a HTML browser interface to help.
Type `q()` to quit R.

[Previously saved workspace restored]

> █
```

The status bar at the bottom reads "R 1.7.0 - A Language and Environment".

---

# Installing and running R

---

---

# R on the web

The R website:

<http://www.r-project.org/>

here you can find information on the software,  
download the current version  
“R-2.9.2” (released on 2009-08-24),  
packages, tutorials and manuals.

---

# Installing and running R

To download R first choose a Comprehensive R Archive Network (**CRAN**) mirror:

A CRAN is a network of ftp and web servers around the world that store identical, up-to-date versions of code and documentation for R) -> Use the CRAN mirror nearest to you to minimize network load

Download the R version suitable to your **operative system**, following the installation instructions.

## Windows:

the executable file is auto-installing, just follow the instructions. Create an icon on the Desktop!  
Double click on it to run R



## Linux:

Download the R-2.9.1.tar.gz file and unpack it with

```
tar xvfz R-x.y.z.tar.gz
```

```
(or gzip -dc R-x.y.z.tar.gz | tar xvf - )
```

```
./configure
```

```
make
```

```
make install
```

Once installed, to run the program just type R on shell.

It will appear a prompt “>”, meaning that R is waiting for you to input a command.

---

# Basic Commands

---

---

# R commands

R commands consists of either *expressions* or *assignments*.

An *expression* is evaluated, printed on the terminal and its value is “*lost*”. No memory of the output is kept.

Example:

```
> objects()
character(0)
> (5+3/2)*0.1
[1] 0.65
> objects()
character(0)
>
```

---

An *assignment* also evaluate an expression but passes the value to a variable and the result is not automatically printed.

To assign a value to a variable it is used the simbol “<-” pointing to the variable which receives the value:

Example

```
> objects()
character(0)
> a <- (5+3/2)*0.1
> objects()
[1] "a"
>
```

---

Commands are separated by a semi-colon(“;”) or by a new line.

If a command is not complete at the end of the line, R will give a different prompt, “+” by default, at any subsequent lines until the command is syntactically complete

*Comments* have to be preceded by a hashmark (“#”): everything following “#” to the end of the line is ignored.

---

# Recall and correction of previous commands

R provides a mechanism for recalling and re-executing previous commands.

Vertical arrow keys on the keyboard of the computer can be used to scroll forward and backward the list of commands (history) you input.

Once you find the command you are looking for you can modify it using the keys.

---

# Executing commands from a file

Commands can be typed on the R console or they can be stored in an external file, say “commands.r”

The list of commands in “commands.r” can be executed at any time in a R session typing:

```
source(“commands.r”)
```

The output is printed on the console

---

# Printing output to a file

Sometimes it is useful to divert the output from the console to an external file.

Using the command:

```
sink("output.txt")
```

all the subsequent output will be printed to the external file "output.txt".

The command:

```
sink()
```

will restore the output to the console again.

---

# R objects

The entities that R creates and manipulates are known as *objects*. They may be:

- variables
- vectors
- arrays of numbers
- character strings
- functions

or more general structures built from these components.

---

Once objects are created, the R commands:

`objects()`

or alternatively:

`ls()`

can be used to display the names of the objects which are currently stored within R.

The collection of objects currently stored is called the *workspace*.

To remove objects, use the function `rm()`:

`rm(x)`            to remove x

`rm(list=ls())` to remove all the objects of the workspace

---

# Saving Data

When you quit an R Session with the command

**q()**

you will be asked if you want to save the workspace: typing “y” two files will be created in the working directory:

- **“.RData”** containing all the objects created during the session
- **“.Rhistory”** containing the list of commands you typed during the session

The same files are created if you type

**save.image(); savehistory()**

before quitting the session

---

It is possible to save the workspace in an other directory in a specific file:

```
save.image(file="mydirectory/myfile.RData")
```

or

```
save(file="mydirectory/myfile.RData", list=ls())
```

To access the objects in "myfile.RData" in the following R sessions you can type on the console:

```
load("mydirectory/myfile.RData")
```

and the objects in myfile.RData will be added in the current workspace.

---

The same for saving the list of commands in an other directory in a specific file:

**savehistory(file="mydirectory/myfile.Rhistory")**

To load the commands saved in myfile.Rhistory type:

**loadhistory(file="mydirectory/myfile.Rhistory")**

As well you can access the history typing:

**history()** to display the last 25 commands or

**history(n)** to display the last n commands

*NB: .Rhistory files can be opened by text editors*

---

---

# Managing directories

During a R session it can be necessary to get information on the working directory or to change the current directory.

Use:

- **getwd()** to get the working directory
- **list.files()** or **dir()** to see the files in the working directory
- **setwd("path/mydir")** to set the working directory

---

# R help

R has an excellent help.

- The inbuilt help facility (like “man” in Linux)

It can be accessed from the command line using:

```
> ?function_name
```

*or*

```
> help(function_name)
```

The help window will appear and you will find information on the function, its arguments, examples, correlated topics and further details.

- The HTML format help

The command:

```
> help.start()
```

will launch a Web browser that allows the help pages to be browsed with hyperlinks.

# Further help...

The command **example**(*function\_name*) runs the examples contained in the help pages.

**Example:**

```
>example(array)
```

```
array> dim(as.array(letters))
```

```
[1] 26
```

```
array> array(1:3, c(2,4)) # recycle 1:3 "2 2/3 times"
```

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

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

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

---

# Arithmetics, Logical Operators & Elementary Functions

---

---

# Arithmetical Operators

In R the elementary arithmetic is the usual:

- + addition
- – subtraction
- \* multiplication
- / division
- ^ raising to a power

and operators are used as in a common calculator.

Example:

> (4+3)\*2

> 5^2

---

# Logical operators

To compare objects (numbers, vectors, arrays,...) the following operators are used:

- `<` less than
- `>` more than
- `<=` less than or equal to
- `>=` more than or equal to
- `==` equal to
- `!=` inequal to

Example:

`4 > 2`

`A == B`

The output is “True” or “False”.

---

To compose logical expressions the following symbols are used:

- & “and”
- | “or”
- ! “not”

If  $c_1$  and  $c_2$  are logical expressions:

- $c_1 \& c_2$  is their intersection
- $c_1 | c_2$  is their union
- $!c_1$  is the negation of  $c_1$

# Elementary functions available in R:

- **sqrt()** square root
- **abs()** modulus
- **exp()** exponential function
- **log()** natural logarithm
- **log10()** logarithm base10
- **ceiling()** rounds to the smallest integer greater than the argument
- **floor()** rounds to the greatest integer smaller than the argument
- **trunc()** truncates decimals
- **round(x,n)** round x to the specified n decimal places
- **signif(x,n)** round to the n significant places
- **sin() cos() tan()** trigonometric functions
- **asin(), acos(), atan()** inverse trigonometric functions
- **sinh(), cosh() tanh()** hyperbolic functions
- **asinh(), acosh(), atanh()** inverse hyperbolic functions

---

# Variables and Assignments

---

# Assign a value to a variable

It has been said that if a command is an *expression*, R evaluate it and the output is printed on the terminal and “lost”.

In order to keep memory of the output it is necessary to pass its value to a variable, making an *assignment*.

To assign a value to a variable it is used the simbol “<-” pointing to the variable which receives the value:

```
x <- 3*4 (the command 3*4 -> x is equivalent)
```

```
y <- 2
```

```
xy <- x*y
```

or the function **assign()**:

```
assign("x", 3*4)
```

```
assign("y", 2)
```

```
assign("xy", x*y)
```

---

To display the value of a variable just type the name of the variable itself.

In the previous example:

```
> x
```

```
[1] 12
```

```
> y
```

```
[1] 2
```

```
> xy
```

```
[1] 24
```

To display the list of the names of all the variables in the workspace type:

```
ls()
```

or

```
objects()
```

---

---

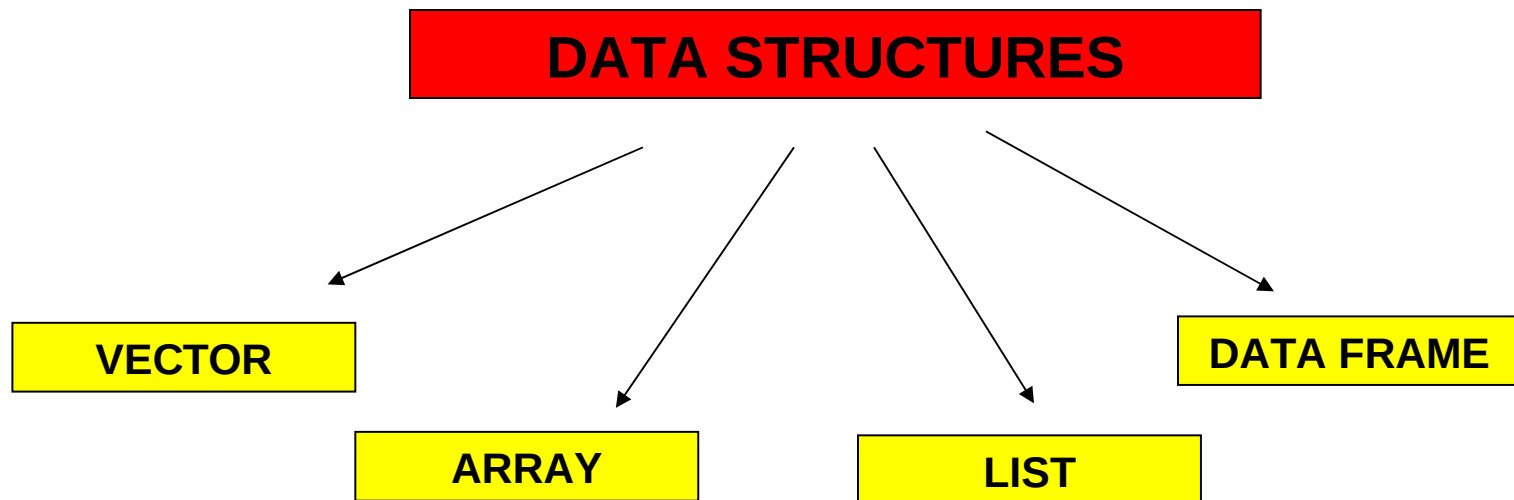
# Data structures

---

- Vectors
- Matrices and arrays
- Lists
- Data frames

# Data structures

Each object created in R can be referred to one of the following *data structures*.



The simplest data structure is the vector

# Vectors

A vector is an entity consisting of an ordered collection of elements all of the same type or “*mode*”.

The “mode” of an object is the basic type of its fundamental constituents.

After the “mode” of its elements, a vector can be:

- Numeric
- Logical
- Character
- Complex

# Numeric Vectors

The numeric vector is “a single entity consisting of an ordered collection of *numbers*”

To input in the R workspace a vector named “x” whose elements are (0.5, 3.1, 2.2, 4) type:

```
x <- c (0.5, 3.1, 2.2, 4)
```

where “<-” is the assignment operator

and **c()** is the concatenation function whose output is a vector obtained by concatenating its arguments end to end.

A numeric variable is considered itself a vector of length one.

# Vector Arithmetic

It is possible to calculate arithmetical expression using vectors, for example:

```
a<- c(1,2,3)
b<- c(0.1, 0.2, 0.3)
c<- a+b; d<- a*b; e <- b^a
```

Vectors occurring in the same expression may have different lengths:

- The output will have the same length as the longest vector
- Shorter vectors in the expression will be **recycled** until they match the length of the longest vector

Example:

```
> x <- c(1,2,3,4,5); y <- c(1,2); z<- x + y -2
```

Warning message:

```
In x + y : longer object length is not a multiple of shorter object length
```

```
> z
```

```
[1] 0 2 2 4 4
```

# Generating regular sequences

R provides some functions to generate regular sequences of numbers:

**seq(from=,to=,by=, length=)**

- Either “by” or “length” need to be provided
- If neither is provided, the default is “by”=1

Example:

```
> seq(1,5)
```

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

N.B.: the same results is obtained using the colon “:”

```
> x<-1:5
```

while `x <- 5:1` generates the sequence backwards.

---

Another function to replicate objects in many different ways is:

**rep(arg=, times=, each=, len=, ....)**

Example:

```
> rep(1:4, times=2)
```

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

```
> rep(1:4, each = 2)
```

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

```
> rep(1:4, each = 2, len = 4) # first 4 only.
```

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

```
> rep(1:4, each = 2, len = 10) # 8 integers plus two recycled 1's.
```

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

# Logical vectors

A logical vector is generated by logical expressions:

```
> x<- 1:10
```

```
> y<- x>3
```

```
> y
```

```
FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Its elements can assume the value TRUE or FALSE (or NA if the value is Not Available).

Logical vectors can be used in arithmetic expressions, where they are coerced into numeric vectors, **FALSE** becoming **0** and **TRUE** becoming **1**.

---

# Character vectors

- Characters vectors are frequently used in the customization of the plots;
- They are created with the concatenation function `c()`;
- Their elements are entered using either double (“”) or single (‘’) quotes.

Example:

```
a<- c(“Hello”, “everybody!”)
```

# Complex vectors

Complex vectors can be created with the function **complex()**.

The vector can be specified either by giving its length, its real and imaginary parts, or modulus and argument.

```
Z <- complex (length.out = , real = numeric(), imaginary =  
numeric(), modulus = , argument = )
```

Length.out: numeric, desired length of the output vector, inputs being recycled as needed.

Real: numeric vector

Imaginary: numeric vector

Modulus: numeric vector

Argument: numeric vector

# Index vectors

Subsets of the elements of a vector may be selected by appending to the name of the vector an *index vector* in square brackets.

Here will be presented three types of index vectors:

- **Vector of positive integer quantities:**

the values in the index vector have to be in the interval [1, length(vector)]

Example:

**x[n]**

will select the element of place n

**x[c(2,5,7)]**

will select the elements of place 2,5 and 7

**x[1:10]**

will select the first 10 elements of x  
(it is assumed that length(x) >= 10)

## 1. **Vector of negative integer quantities:**

it identifies the values to be excluded from the selection

Example:

```
y <- x[-(1:5)]
```

will give all but the first five elements of x

```
y <- x[-c(2,5,7)]
```

will give all but the 2nd, 5th and 7th elements of x

## 7. **Logical vector:**

the elements of the vector which satisfy the index vector logical expression are selected, the others are excluded

Example:

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

y is shorter than x, its elements are the x-elements greater than zero

# Arrays and Matrices

An array is a multidimensional object whose elements are all of the same mode (*numeric, logic or character*).

An array is entered with the command **array()** specifying:

- the vector containing the array elements
- a *dimension vector*

A dimension vector is a vector of non-negative integers: if its length is  $k$ , the array is  $k$ -dimensional ( $k=2$ , *2-dimensional array or matrix*).

*Example:*

```
A <- array(1:12, dim=c(3,2,2))
```

*or*

```
A <- 1:12
```

```
dim(A) <- c(3,2,2)
```

The array is filled with the elements of the vector according to the “**column major order**”, with the first subscript moving faster and the last subscript slowest.

Example:

```
A <- array(1:12, dim=c(3,2,2))
```

**A[, , 1]**

**A[, , 2]**

**[,1] [,2]**

**[1,] 1 4**

**[2,] 2 5**

**[3,] 3 6**

**[,1] [,2]**

**[1,] 7 10**

**[2,] 8 11**

**[3,] 9 12**

# Array indexing

- Individual elements of an array can be referenced by giving the name of the array followed by the subscripts in square brackets, separated by commas.

Example:

```
A[2,1,1]
```

- Subsections of arrays can be created by giving a sequence of index vectors

Example: `select A[2, , ]`

```
sel<-c(A[2,1,1], A[2,2,1], A[2,3,1], A[2,1,2], A[2,2,2], A[2,3,2])
```

# Matrices

A matrix is a 2-dimensional array. In R handling matrices is easy and computing time is short.

## Arithmetics:

- Element by element operation:  
A+B, A-B, A\*B, A/B
- Matrix product: A %\*% B

## Functions:

- nrow() number of rows
- ncol() number of columns
- t() transpose
- det() determinant
- diag() diagonalize
- eigen() eigenvalues, -vectors

An easy way to build up matrices from vectors or other matrices) is through the functions:

- **rbind()** forms matrices concatenating the arguments by row
- **cbind()** forms matrices concatenating the arguments by column

The arguments can either be vector or matrices with:

- the same number of columns (rbind) or
- the same number of rows (cbind).

Example:

```
> x <- array(1:12,dim=c(3,4))  
> y <- c(0,0,0,0)  
> xy <- rbind(x,y)
```



```
> xy  
  [,1] [,2] [,3] [,4]  
  1   4   7  10  
  2   5   8  11  
  3   6   9  12  
y  0   0   0   0
```

---

# More on R packages

---

---

# Packages

R has a set of packages that enlarge its potential.  
They can be:

- Installed by default and loaded on startup
- Installed by default but not loaded
- Packages to be installed by the user

## PACKAGES INSTALLED AND LOADED BY DEFAULT

When you start R, default loaded packages are:

```
> getOption("defaultPackages")
```

```
[1] "datasets" "utils" "grDevices" "graphics" "stats"  
     "methods" (plus, of course, base)
```

---

## PACKAGES INSTALLED BUT NOT LOADED BY DEFAULT

You can get the list of available packages that can be loaded typing:

➤ `library()`

To load packages:

**Windows:**

Packages -> Load Packages

**Linux:**

> `library("Package1")`

---

## PACKAGES TO BE INSTALLED BY THE USER

The R website provide links of the CRAN where it is possible to download the “contributed extension packages”

To install one of them:

### Windows:

Packages -> Install packages (chose CRAN or local zip file)

Packages -> Load packages

### Linux:

- Download the package .tar.gz
- R CMD INSTALL “\$path/package1.tar.gz”

---

# Lists and Data Frames

---

# Lists

R *lists* are objects containing ordered collections of objects (*components*).

- The modes of the components of a list can be different (numeric, logic, character, complex)
- The data structure of components can be different (vectors, arrays, data frame and list as well).

# List generation

To generate a list the function `list()` is used:

```
> x <- list(station="Vinadio", elev=1200, month=c("N","D","J"),  
            snowdepth=c(6,21,44))
```

```
> x
```

```
$station
```

```
[1] "Vinadio"
```

```
$elev
```

```
[1] 1200
```

```
$month
```

```
[1] "N" "D" "J"
```

```
$snowdepth
```

```
[1] 6 21 44
```

---

An object can be added to a list by passing it as a new component of the list

```
> snowyday=c(4,5,6)
> x[5] <- list(snowyday)
```

```
>
```

```
> x
```

```
$station
```

```
[1] "Vinadio"
```

```
$elev
```

```
[1] 1200
```

```
$month
```

```
[1] "N" "D" "J"
```

```
$snowdepth
```

```
[1] 6 21 44
```

```
[[5]]
```

```
[1] 4 5 6
```

To join together different lists, say x, y and z, and create a new list xyz, the concatenation function c() can be used.

Example:

```
> y<-0
> z <-1
> y<-as.list(y)
> z<-as.list(z)
> xyz<-c(x,X=y,Z=z)
```

```
> xyz
$station
[1] "Vinadio"
$elev
[1] 1200
$month
[1] "N" "D" "J"
$snowdepth
[1] 6 21 44
$snowydays
[1] 4 5 6
$X
[1] 0
$Z
[1] 1
```

# List indexing

The single element can be accessed using either its position in the list or its name:

Example:

`x[[1]]`

will give "Vinadio"

`x$elev`

will give 1200

N.B.:

The double bracket `[[ ]]` is used to select components of the list

The single bracket `[ ]` is used to select elements of the `i` component

Example:

`x[[4]] [2,3]`

will give: 21,44

# Data frames

Data frames are objects sharing many of the properties of matrices and of lists:

- they are collections of objects (vectors, matrices, lists or other data frames) of differing types (numeric, logical, character, complex);
- they are matrix-like structure (matrices, lists, data frames provide as many variables to the data frame as they have columns, elements or variables respectively;
- Vector structures must all have the *same length* (if not they are recycled), matrix structures must all have the *same row size*.

---

# Generating a data frame

Objects which satisfies the conditions required can be arranged in a data frame by column using the function **data.frame()**:

```
stationX<- data.frame(date, T, P, p, wind, RH)
```

N.B.: arguments must have all the same length/number of rows!

A list or a matrix which satisfies the restriction of a data frame can be coerced to a data frame using the functions **as.data.frame()**

# Accessing variables of a data frame

The components of a data frame can be accessed using:

- the \$ notation:

*name\_dataframe\$name\_component*

- the double brackets `[[ ]]`

*name\_dataframe [[ 1 ]]*

- the functions:

**`attach(name_dataframe)`**

...

**`detach():`**

Through the function **attach()** it is possible to access the components of a data frame (list) *without quoting the name of the data frame* (list) each time.

**stationX\$wind**            **wind**

The components of the data frame become temporarily available as variables until the **detach()** function is called.

Once **detach()** is called, the components are no longer visible with their names only.

Again they can be accessed only with the list notation:

**stationX\$wind**

---

## *ATTENTION*

After the function `attach()` is called, all the operations on the components of the data frame do not affect the original data frame, but a *copy* of its components.

In order to modify the data frame it is necessary to use the list notation (“\$” or `[[ ]]`)

```
dataframe [[1]] <- dataframe [[1]] *(100)
```

## Example:

```
>x<-1:3; y<-3:1; z<-0
```

```
>df <- data.frame(x,y,z)
```

```
> df
```

```
  x y z
```

```
1 1 3 0
```

```
2 2 2 0
```

```
3 3 1 0
```

```
> rm(x,y,z)
```

```
> attach(df)
```

```
> z<-x+y
```

```
> detach(df)
```

```
> df
```

```
  x y z
```

```
1 1 3 0
```

```
2 2 2 0
```

```
3 3 1 0
```

```
df$z <-df$x+df$y
```

```
> df
```

```
  x y z
```

```
1 1 3 4
```

```
2 2 2 4
```

```
3 3 1 4
```

# Testing and coercing

If you need to verify the *mode* or the *structure* of an object you can use the function **is.()**:

**is.mode(object)**

**is.structure(object)**

- is.integer(x)
- is.double(x)
- is.numeric(x)
- is.character(x)
- is.complex(x)
- is.logic(x)
- is.vector(x)
- is.matrix(x)
- is.array(x)
- is.list(x)
- is.data.frame(x)

They return the value TRUE or FALSE

---

To coerce an object to have of a specific structure you can use the function

- **as.()** to coerce an object to have a specific structure

- `as.integer(x)`

- `as.double(x)`

- `as.numeric(x)`

- `as.character(x)`

- `as.complex(x)`

- `as.logic(x)`

- `as.vector(x)`

- `as.matrix(x)`

- `as.array(x)`

- `as.list(x)`

- `as.data.frame(x)`

# Example

```
> a<- 1:5
> a
[1] 1 2 3 4 5
> is.vector(a)
[1] TRUE
> a<-as.matrix(a)
> a
  [,1]
[1,]  1
[2,]  2
[3,]  3
[4,]  4
[5,]  5
> is.matrix(a)
[1] TRUE
> is.integer(a)
[1] TRUE
```

---

# Reading and writing files

# Importing data in R

Data contained in external text files can be imported in R using one of the following functions:

- `scan()`
- `read.table()`
- `read.csv()`
- `read.csv2()`
- `read.delim()`
- `read.delim2()`

The requirements on the form of data set are sometimes quite strict, so it is better to modify input files to satisfy this requirements.

---

# The function `scan()`

This function is the most flexible: it can be used to read logical, integer, numeric, complex, character, raw data and lists

```
scan(file = " ", what = double(0), n = -1, sep = "", dec =  
".", skip = 0, na.strings = "NA")
```

file: the name of the file which the data are to be read from;

what: the type of data to be read (logical, integer, numeric, complex, character, raw and **list**);

n: the maximum number of data values to be read, defaulting to no limit;

```
scan(file = "", what = double(0), n = -1,  
sep = "", dec = ".", skip = 0, na.strings =  
"NA")
```

sep: the field separator character. If sep = "", the separator is 'white space', that is one or more spaces, tabs;

dec: the character used in the file for decimal points;

skip: the number of lines of the input file to skip before beginning to read data values;

na.strings: character vector. Elements of this vector are to be interpreted as missing (NA) values. Blank fields are also considered to be missing values in logical, integer, numeric and complex fields

# The function `read.table()`

This function reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

```
read.table(file, header = FALSE, sep = "", dec = ".",  
           row.names, col.names)
```

`header`: a logical value indicating whether the file contains the names of the variables as its first line

`sep`: the field separator character. If `sep = ""` (the default for `read.table`) the separator is 'white space', that is one or more spaces, tabs;

`dec`: the character used in the file for decimal points;

`row.names`: a vector of row names;

`col.names`: a vector of column names.

# Input file format

	N.Amer	Europe	Asia	S.Amer	Oceania	Africa	Mid.Amer
1951	45939	21574	2876	1815	1646	89	555
1956	60423	29990	4708	2568	2366	1411	733
1957	64721	32510	5230	2695	2526	1546	773
1958	68484	35218	6662	2845	2691	1663	836
1959	71799	37598	6856	3000	2868	1769	911
1960	76036	40341	8220	3145	3054	1905	1008
1961	79831	43173	9053	3338	3224	2005	1076

Column names

Row names

```
a <-read.table("Worldtelephones.txt")  
>str(a)
```

```
'data.frame': 7 obs. of 7 variables:  
 $ N.Amer : int 45939 60423 64721 68484 71799 76036 79831  
 $ Europe : int 21574 29990 32510 35218 37598 40341 43173  
 $ Asia : int 2876 4708 5230 6662 6856 8220 9053  
 $ S.Amer : int 1815 2568 2695 2845 3000 3145 3338  
 $ Oceania : int 1646 2366 2526 2691 2868 3054 3224  
 $ Africa : int 89 1411 1546 1663 1769 1905 2005  
 $ Mid.Amer: int 555 733 773 836 911 1008 1076
```

If only column labels are present add the option "header=T"

# The function `read.csv()` and `read.csv2()`

- `read.csv()` is intended for reading 'comma separated value' (CSV) files, where the decimal point is "."
- `read.csv2()` is the variant used in countries that use a comma (",") as decimal point and a semicolon (";") as field separator.

```
read.csv(file, header = TRUE, sep = ",", dec=".")
```

```
read.csv2(file, header = TRUE, sep = ";", dec=",")
```

# The functions `read.delim()` and `read.delim2()`

- They are intended to read TAB separated files

```
read.delim(file, header = TRUE, sep = "\t", dec=".", fill = TRUE, ...)
```

```
read.delim2(file, header = TRUE, sep = "\t", dec=",", fill = TRUE,...)
```

`sep`: the field separator character. “\t” (default for `read.delim`) stands for TAB separator;

`fill`: if TRUE then in case the rows have unequal length, blank fields are implicitly added

---

# Exporting Data

R objects can be exported to a text file using the **cat()** function:

```
cat (x , file = "", sep = " ", fill = FALSE, labels = NULL,  
    append = FALSE)
```

x: **R** object

file: character string naming the file to print to. If "" (the default), cat prints to the standard output connection, the console unless redirected by **sink**.

sep: a character vector of strings to append after each element.

---

fill: controls how the output is broken into successive lines.

append:logical. If TRUE output will be appended to file; otherwise, it will overwrite the contents of file.

---

# Writing data frames

Often it is useful to write a matrix or a data frame to a file as a grid of elements.

- `write()` writes out a matrix or vector in a specified number of columns.
- `write.table()` writes out a data frame (or an object that can be coerced to a data frame) with row and column labels

---

# The function write()

```
write (x, file = "data", ncolumns =, append = FALSE,  
      sep = " ")
```

x	the data to be written out.
file	the file to write to
ncolumns	the number of columns to write the data in.
append	if TRUE the data x are appended to the file.
sep	a string used to separate columns. Using sep = "\t" gives tab delimited output; default is white space " ".

---

# The function `write.table()`

```
write.table(x, file = "", append = FALSE, sep = " ", na =  
  "NA", dec = ".", row.names = TRUE, col.names = TRUE)
```

`na`:

the string to use for missing values in the data (default is NA)

`row.names` (`col.names`):

either a logical value indicating whether the row (column) names of `x` are to be written along with `x`, or a character vector of row (column) names to be written.

---

# The functions `write.csv()` and `write.csv2()`

`write.csv(x, file=" ")`

write to a file using the comma (“,”) as the field separator

`write.csv2(x, file=" ")`

write to a file using semicolon (“;”) as the field separator and the comma as the decimal point

# Reading Excel files

An excel file can contain many sheets, and the sheets can contain formulae, macros and so on: not all readers can manage this.

Instead of importing the data contained in .xls files in R, it would be easier if:

- you export data of the .xls file in a .txt file, in tab-delimited or comma-separated form
- you use R functions *read.delim()* or *read.csv()* to import the .txt file into the R workspace

---

# Programming

- 
- ❑ Conditional executions
  - ❑ Repetitive executions

---

# Conditional executions: if statement

*if* (*expr\_1*) *expr\_2* **else** *expr\_3*

*where:*

*expr\_1* is a conditional statement which can assume value TRUE or FALSE:

- if *expr\_1* is TRUE, *expr\_2* will be evaluated
- if *expr\_1* is FALSE, *expr\_3* will be evaluated

- 
- Relation operators `&&` and `||` are often used in the if statement:

Say `c1` and `c2` are the conditions:

if (`c1 && c2`) `expr_2` **else** `expr_3`

if (`c1 || c2`) `expr_2` **else** `expr_3`

*Note:*

- *`&&` and `||` apply to vectors of length one*
- *`&` and `|` apply to each element of the vector*

# Example

```
> a<- seq(5,50,by=5)
> a
[1] 5 10 15 20 25 30 35 40 45 50
> b <- c(1:10)
> b
[1] 1 2 3 4 5 6 7 8 9 10
>
>
> if (length(a)==length(b)) c=a*b else cat ("Warning, the vectors have different length ")
> c
[1] 5 20 45 80 125 180 245 320 405 500
>
> b <- c(2*(0:10))
>
> b
[1] 0 2 4 6 8 10 12 14 16 18 20
> length(b)
[1] 11
> if (length(a)==length(b)) c=a*b else cat ("Warning, the vectors have different length ")
Warning, the vectors have different length
```

---

# Repetitive executions: for loop

```
for (var in range )  
{  
    command_1  
        command_2  
            ...  
                command_n  
}
```

- “var” is the loop variable varying in “range”
- command\_1 ... command\_n is repeatedly evaluated till “var” is in the range

# Example

```
> for (i in 1:5)  
+ {cat ("Natural logarithm of ", i , "is ", log(i), "\n")}
```

Natural logarithm of 1 is 0

Natural logarithm of 2 is 0.6931472

Natural logarithm of 3 is 1.098612

Natural logarithm of 4 is 1.386294

Natural logarithm of 5 is 1.609438

---

# Base graphics

- 
- High level plotting commands

---

# Introduction to graphics

Excellent graphics can be created using R:

- ❑ wide variety of graphics
- ❑ lots of facilities to customize plots

We here describe “base” graphics:

Other functions to create more specific graphics are available in contributed packages (fields, grid,...).

---

# Plotting commands

Three basic groups:

- **High-level plotting functions:**

create a new plot on the graphics device (with axes, labels, titles, ...)

- **Low-level plotting functions:**

add more information to an existing plot (extra points, lines and labels)

- **Interactive graphics functions:**

allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

# High level plotting commands

- They generate a complete plot of the data passed as arguments
- A graphic windows appear
- If you evaluate again HL plotting command, a new graphic will overwrite the previous one.
- To generate a graphic in a new window, use the command:
  - `windows()` in Windows
  - `X11()` in Linux

---

# The function plot( )

If  $x, y$  are vectors with the same length

- `plot(x,y)`

will produce a scatter plot of  $x$  against  $y$ :

- `plot(x)`

will produce a plot of the values in the vector  $x$  against their index in the vector.

---

# Argument to the high level plotting commands

The following arguments can be passed to the high level graphic functions:

- `plot(x,y, axis=F,.....)`

## `axes=FALSE`

Suppresses generation of axes. -> add your own custom axes with the `axis()` function. The default, `axes=TRUE`, means include axes.

---

`log="x"`

`log="y"`

`log="xy"`

Causes the x, y or both axes to be logarithmic

`xlab=string, ylab=string`

Axis labels for the x and y axes.

`main=string`

figure title, placed at the top of the plot in a large font.

`sub=string`

Sub-title, placed just below the x-axis in a smaller font.

---

---

## type=

Its value controls the type of plot produced, as follows:

"p" Plot individual points (the default)

"l" Plot lines

"b" Plot points connected by lines (both)

"o" Plot points overlaid by lines

"h" Plot vertical lines from points to the zero axis

"n" No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

# References

- <http://cran.r-project.org/>
  - The R Development Core Team, 2009: **R, A Language and Environment for Statistical Computing - Reference Index.**
  - W. N. Venables, D. M. Smith and the R Development Core Team, 2009: **An introduction to R.**
  - The R Development Core Team, 2009: **R Data Import/Export**
  
- R help