ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

INTRODUCTION to DATA SCIENCE

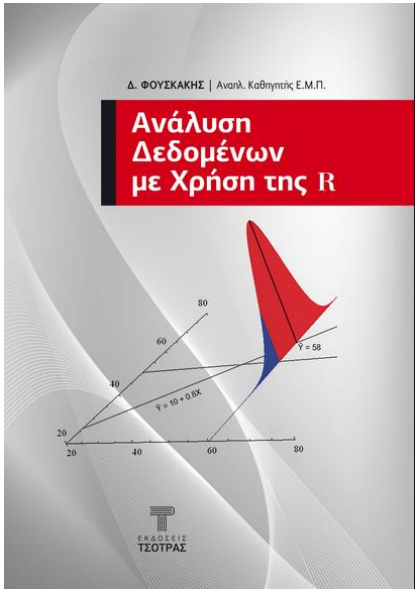
Using R Programming Language

Xanthi Pedeli

Assistant Professor, xpedeli@aueb.gr
Department of Statistics, AUEB

Notes by Ioannis Ntzoufras, *Professor*
Department of Statistics, AUEB

The following material is based on previous slides by D. Fouskakis, Associate professor @ NTUA

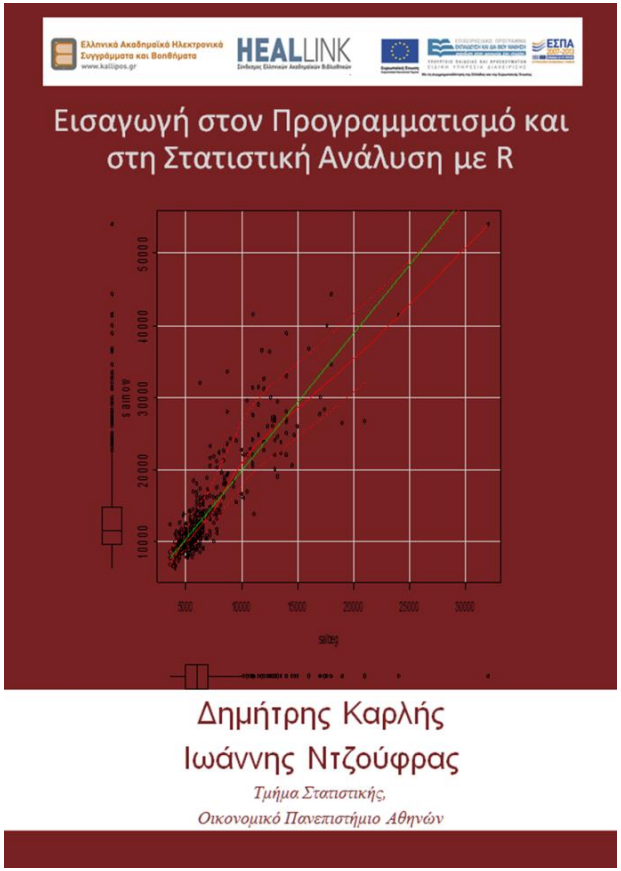


Reference:

Δ. Φουσκάκης (2013). *Ανάλυση Δεδομένων με Χρήση της R*. Εκδόσεις Τσότρας. Αθήνα**.**

Also updated by the e-book

Ντζούφρας , Ι & Καρλής Δ. (2015).
Εισαγωγή στον Προγραμματισμό και στη Στατιστική Ανάλυση με R.
[e-book] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών.
Διαθέσιμο στο: http://hdl.handle.net/11419/2601



What is R?

- R is a free implementation of a dialect of the S language.

What is S?

- Programming language developed by John Chambers and others at Bell Labs.
- Initiated in **1976** as an internal statistical analysis environment
- Originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In **1988** the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language).
- Version 4 of the S language was released in **1998** and is the version we use today.

Back to R

1991: Created in New Zealand by Ross Ihaka and Robert Gentleman.

Their experience developing R is documented in a 1996 JCGS paper.

The original R developers plotting world domination
at the *Black Crow Cafe* on Kitchener Street.

Back to R

1991: Created in New Zealand by Ross Ihaka and Robert Gentleman.

Their experience developing R is documented in a 1996 JCGS paper.

1993: First announcement of R to the public.

1995: Martin Machler convinces Ross and Robert to use the GNU General Public License to make R free software.

1996: A public mailing list is created (R-help and R-devel).

1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.

2000: R version 1.0.0 is released.

Features of R

- Syntax similar to S, making it easy for S-PLUS users to switch over.
- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugx releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow.
- It’s free!!!!!

Design of the R System

- The R system is divided into 2 conceptual parts:
  - The "base" R system that you download from CRAN.
  - Everything else.

- CRAN is the "Comprehensive R Archive Network".

  It is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.

Design of the R System

- R functionality is divided into a number of packages.
- The "base" R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.

Design of the R System

- And there are many other packages available.
- There are currently 18,088 packages (!!!) on CRAN that have been developed by users and programmers around the world.
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Some R Resources

Available from CRAN

http://cran.r-project.org

- An Introduction to R
- Writing R Extensions
- R Data Import/Export
- R Installation and Administration (mostly for building R from sources)
- R Internals (not for the faint of heart)

Books

Standard Texts:

- Adler (2010). R in a Nutshell, O'Reilly.
- Albert (2007). Bayesian Computation with R, Springer.
- Albert & Rizzo (2011). R by Example, Springer.
- Chambers (2008). Software for Data Analysis: Programming with R, Springer.
- Crawley (2007). The R book, Wiley.
- Dalgaard (2002). Introductory Statistics with R, Springer – Verlag.
- Everitt & Hothorn (2006). A Handbook of Statistical Analyses using R, Chapman & Hall/CRC Press.
- Venables & Ripley (2002). Modern Applied Statistics with S, Springer.
- Murrell (2005). R Graphics, Chapman & Hall/CRC Press.

Books

Textbooks and Notes in Greek

- Φουσκάκης (2013). *Ανάλυση Δεδομένων με Χρήση της R*. Εκδόσεις Τσότρας.
- Καρλής & Ντζούφρας (2006). *Εισαγωγή στον Προγραμματισμό με R/Splus*. Πανεπιστημιακές σημειώσεις. Τμήμα Στατιστικής. Οικονομικό Πανεπιστήμιο Αθηνών.
- Φωκιανός & Χαραλάμπους (2010). *Εισαγωγή στην R: Πρόχειρες Σημειώσεις*. Τμήμα Μαθηματικών & Στατιστικής, Πανεπιστήμιο Κύπρου. Διαθέσιμο στην ιστοσελίδα: http://cran.r-project.org/doc/contrib/mainfokianoscharalambous.pdf .

Other resources:

- Springer has a series of books called Use R!.
- A longer list of books is at http://www.r-project.org/doc/bib/R-books.html.

Install R

R-project home: http://r-project.org and CRAN.

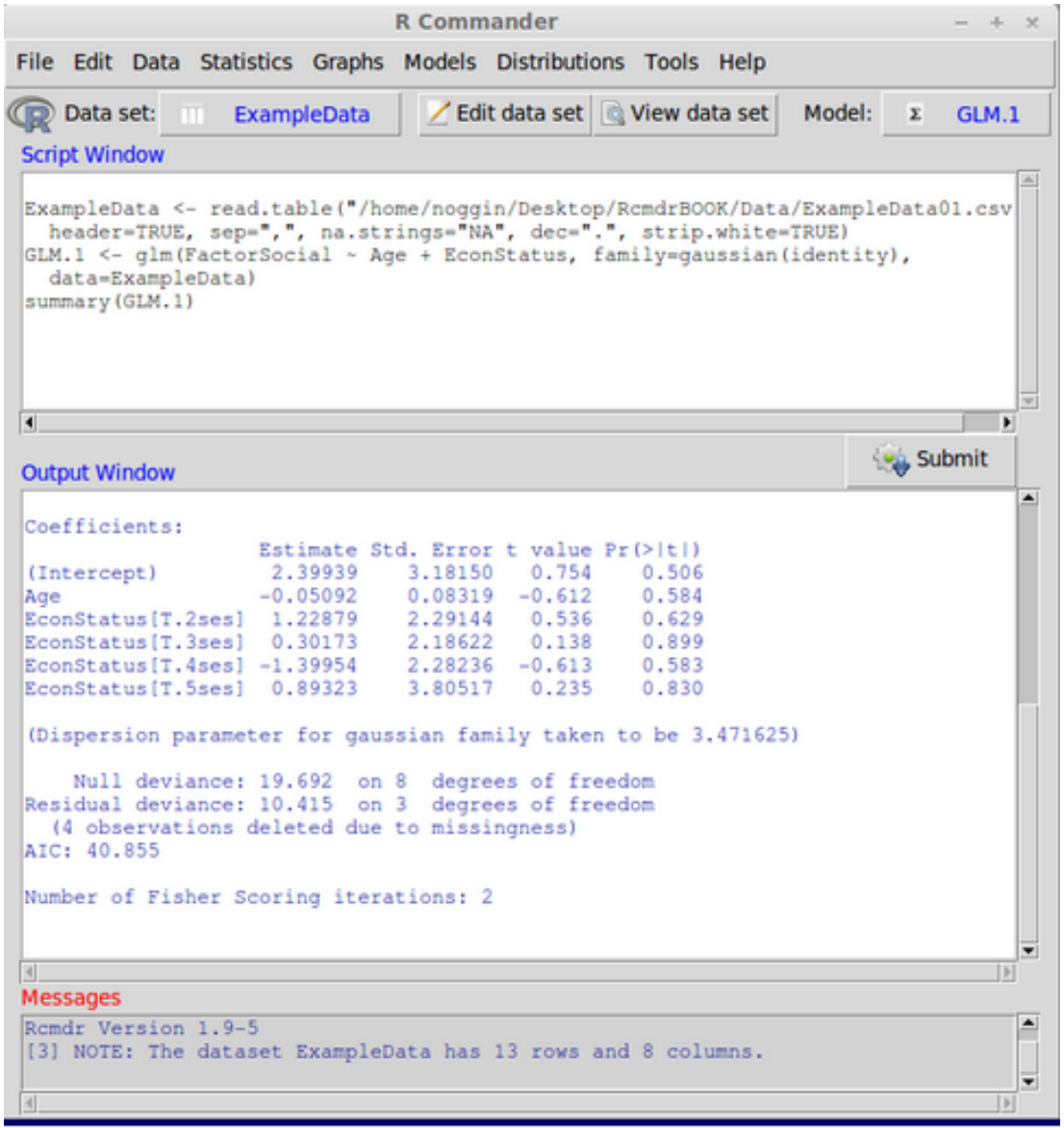
It includes:

- R binaries for windows and other OS for installation
- Add-on packages
- Documentation (manuals, e-books, frequently asked questions, and the R newsletter)
- Mirrored CRAN sites with identical copies of these files exist all around the world.

R Commander

- A user-friendly graphical interface that works in concert with R
- Created by John Fox.
- Web-page: http://www.rcommander.com/
- Rcmdr package
- Users familiar with SPSS and other drop-down menu type programs will initially feel more comfortable using R Commander than R.

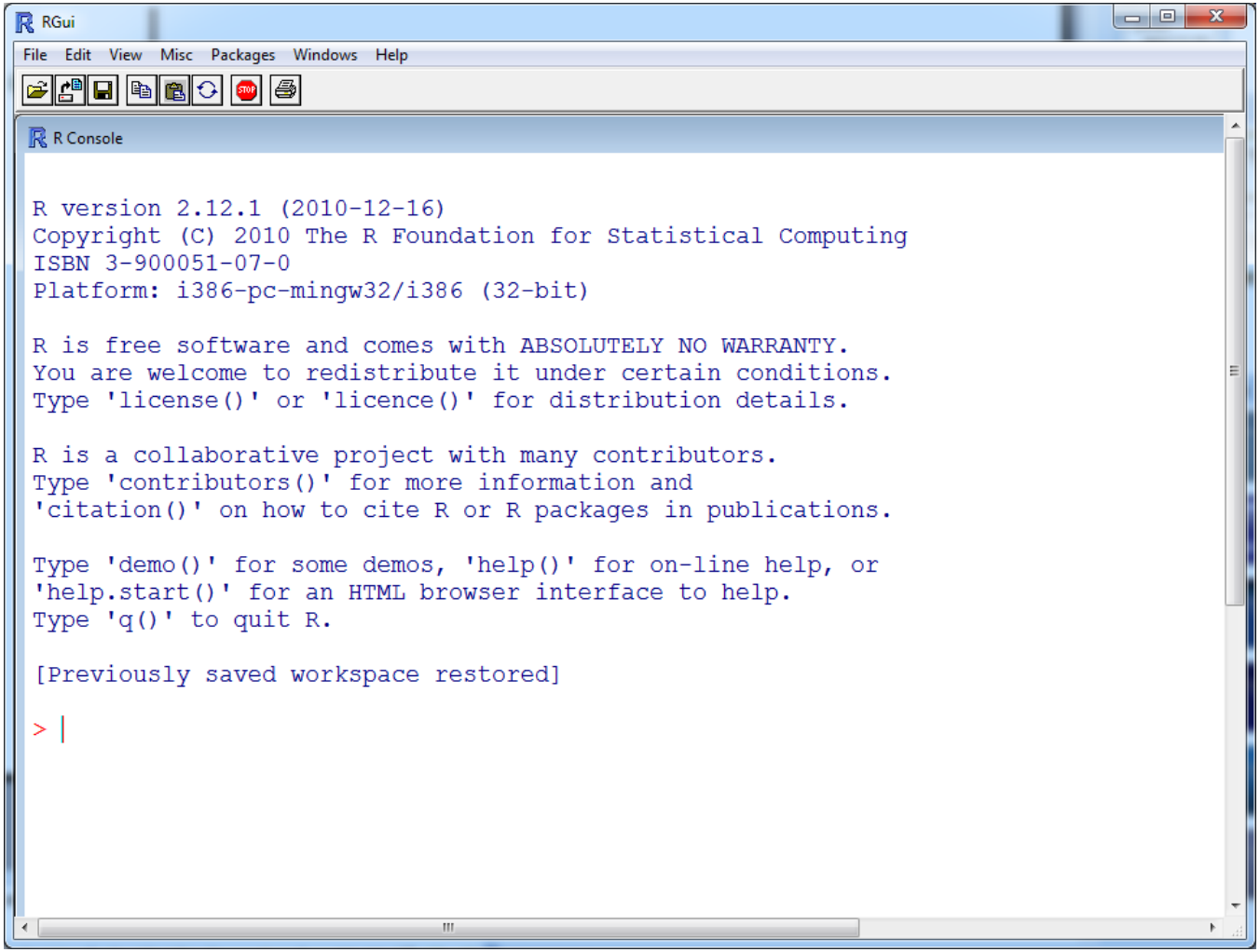
R Commander


R Commander
File Edit Data Statistics Graphs Models Distributions Tools Help
Data set: ExampleData
Edit data set
View data set
Model: GLM.1
Script Window
ExampleData <- read.table("/home/noggin/Desktop/RcmdrBOOK/Data/ExampleData01.csv
header=TRUE, sep=",", na.strings="NA", dec=".", strip.white=TRUE)
GLM.1 <- glm(FactorSocial ~ Age + EconStatus, family=gaussian(identity),
data=ExampleData)
summary(GLM.1)
Submit
Output Window
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.39939 3.18150 0.754 0.506
Age -0.05092 0.08319 -0.612 0.584
EconStatus[T.2ses] 1.22879 2.29144 0.536 0.629
EconStatus[T.3ses] 0.30173 2.18622 0.138 0.899
EconStatus[T.4ses] -1.39954 2.28236 -0.613 0.583
EconStatus[T.5ses] 0.89323 3.80517 0.235 0.830
(Dispersion parameter for gaussian family taken to be 3.471625)
Null deviance: 19.692 on 8 degrees of freedom
Residual deviance: 10.415 on 3 degrees of freedom
(4 observations deleted due to missingness)
AIC: 40.855
Number of Fisher Scoring iterations: 2
Messages
Rcmdr Version 1.9-5
[3] NOTE: The dataset ExampleData has 13 rows and 8 columns.

The R default environment

- The R environment is run with scripts
- In the long run it is more advantageous and flexible
- You can easily bring back up as you perform multiple calculations
- Here we will use the default graphical interface

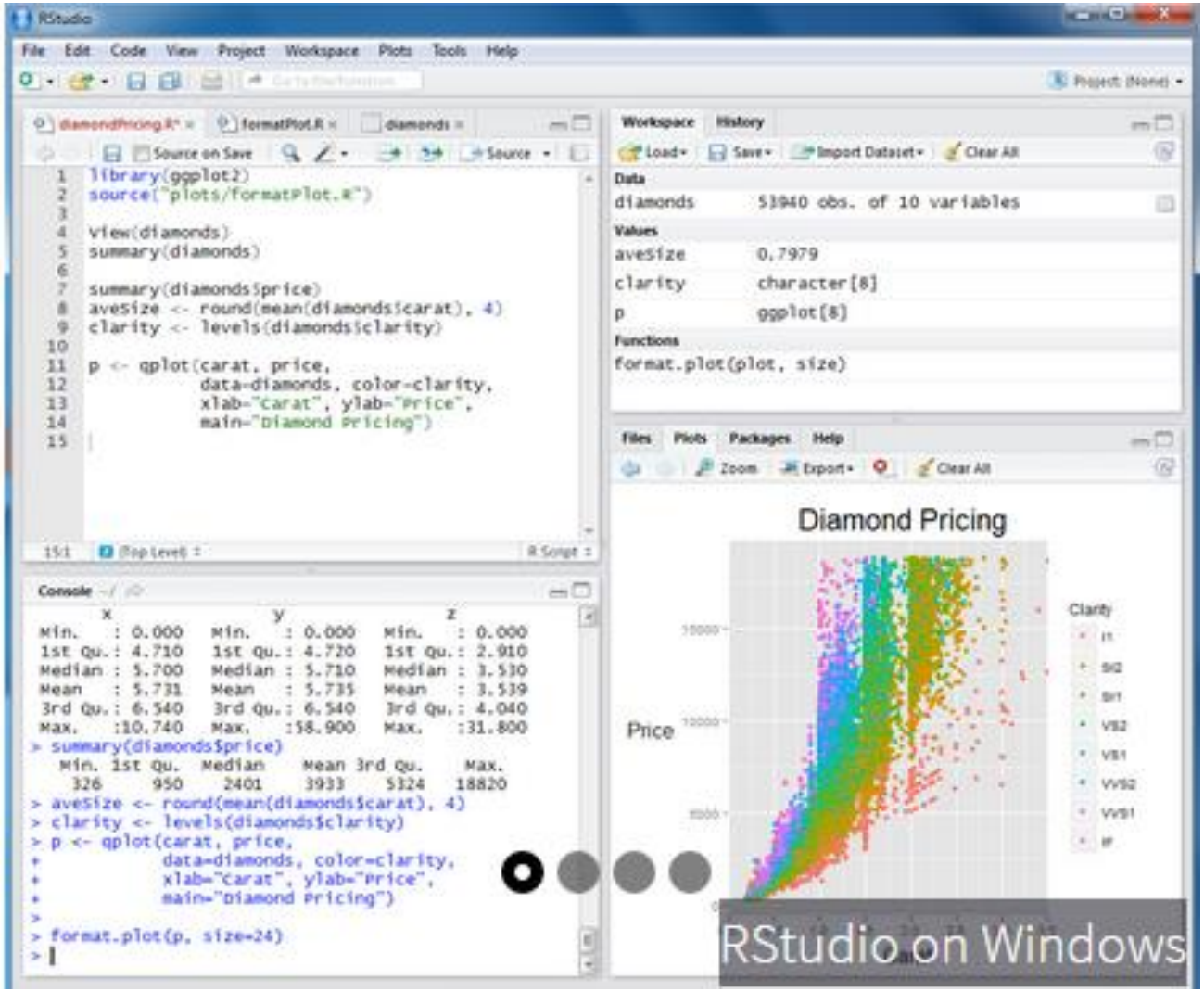
The R default environment



An alternative environment: R-studio

- The advanced R environment
- Monitor all windows simultaneously
- Better for presentations

An alternative environment: R-studio



Running R

- Once installation is complete, the recommended next step for a new user would be to start R by double click the R program.
- Then the R console shows up.
- The `>' character is the prompt, and commands are executed once the user presses the RETURN (Enter) key.
- To terminate the program either press q(), or close the R main window (not the R console), or from the menu file choose exit. In all cases you will be asked if you wish to save the workspace image and all the objects you have created in your session.

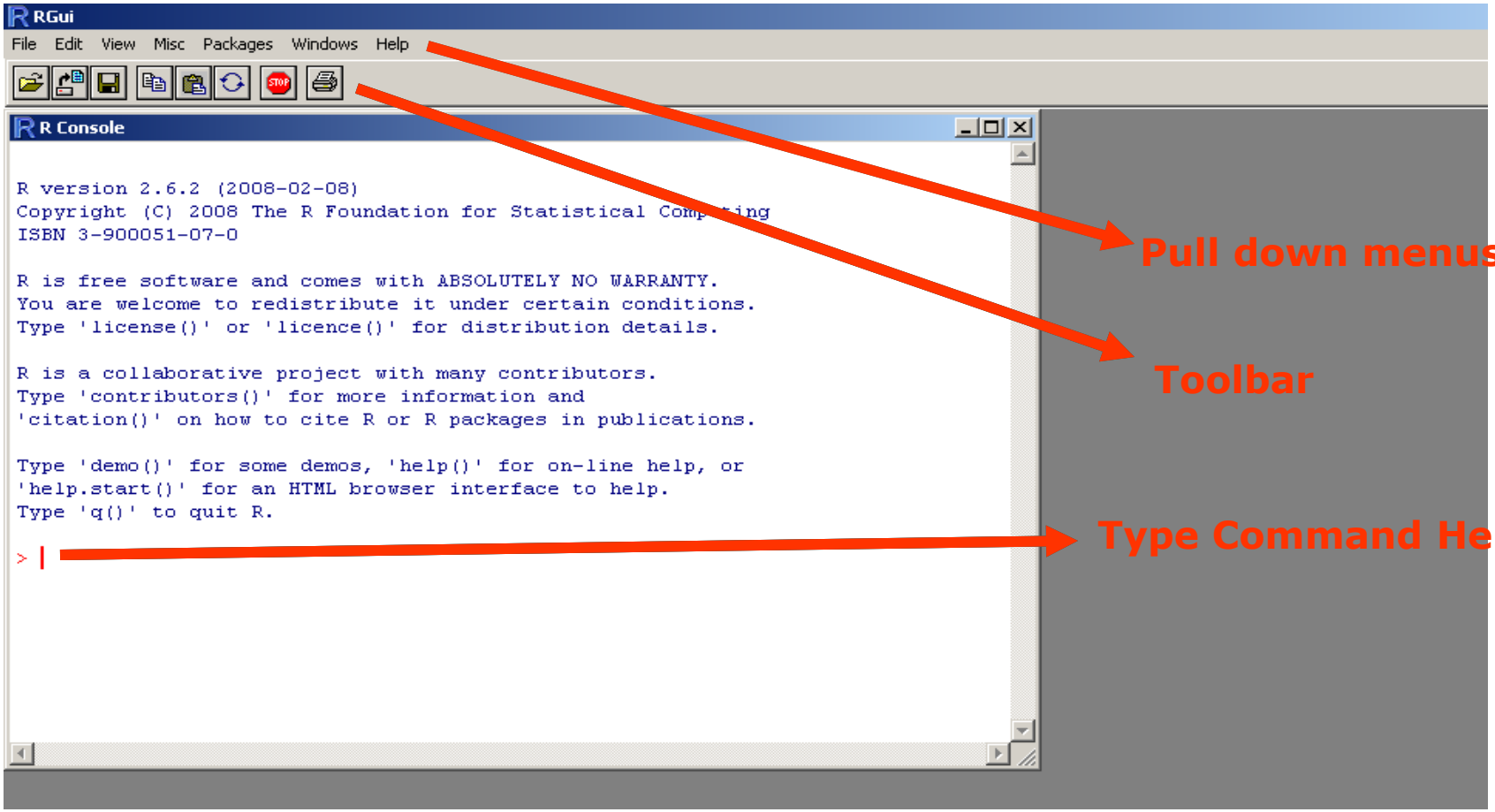
Windows available in R

1. R console
2. The script editor
3. The graphics windows
4. R data editor

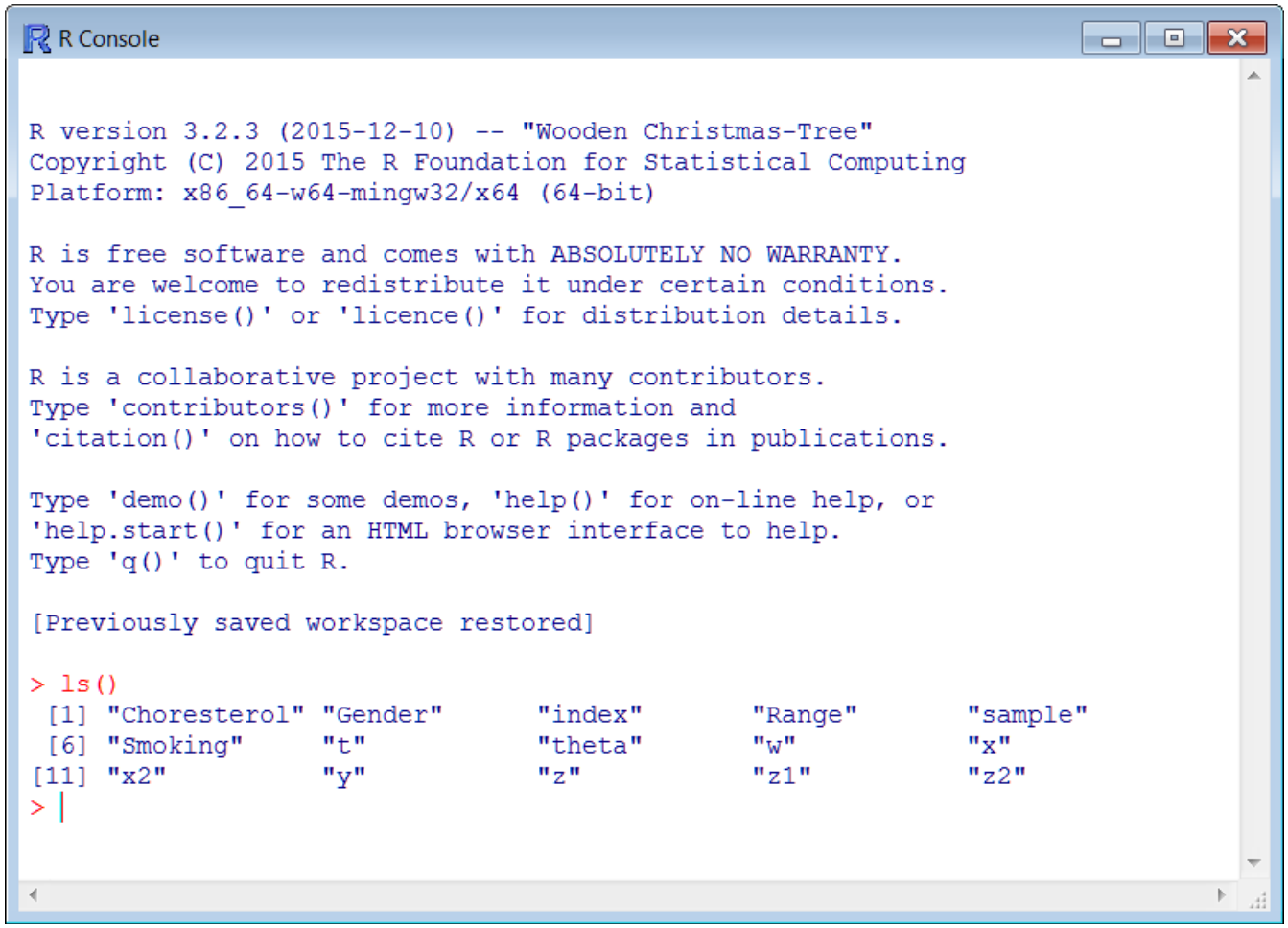
R – Console

- You may write R commands there.
- This window also displays all the commands R has run, the results, and error report.
- This window appears when you launch R.
- You can type and run commands in this window line by line.

The R console



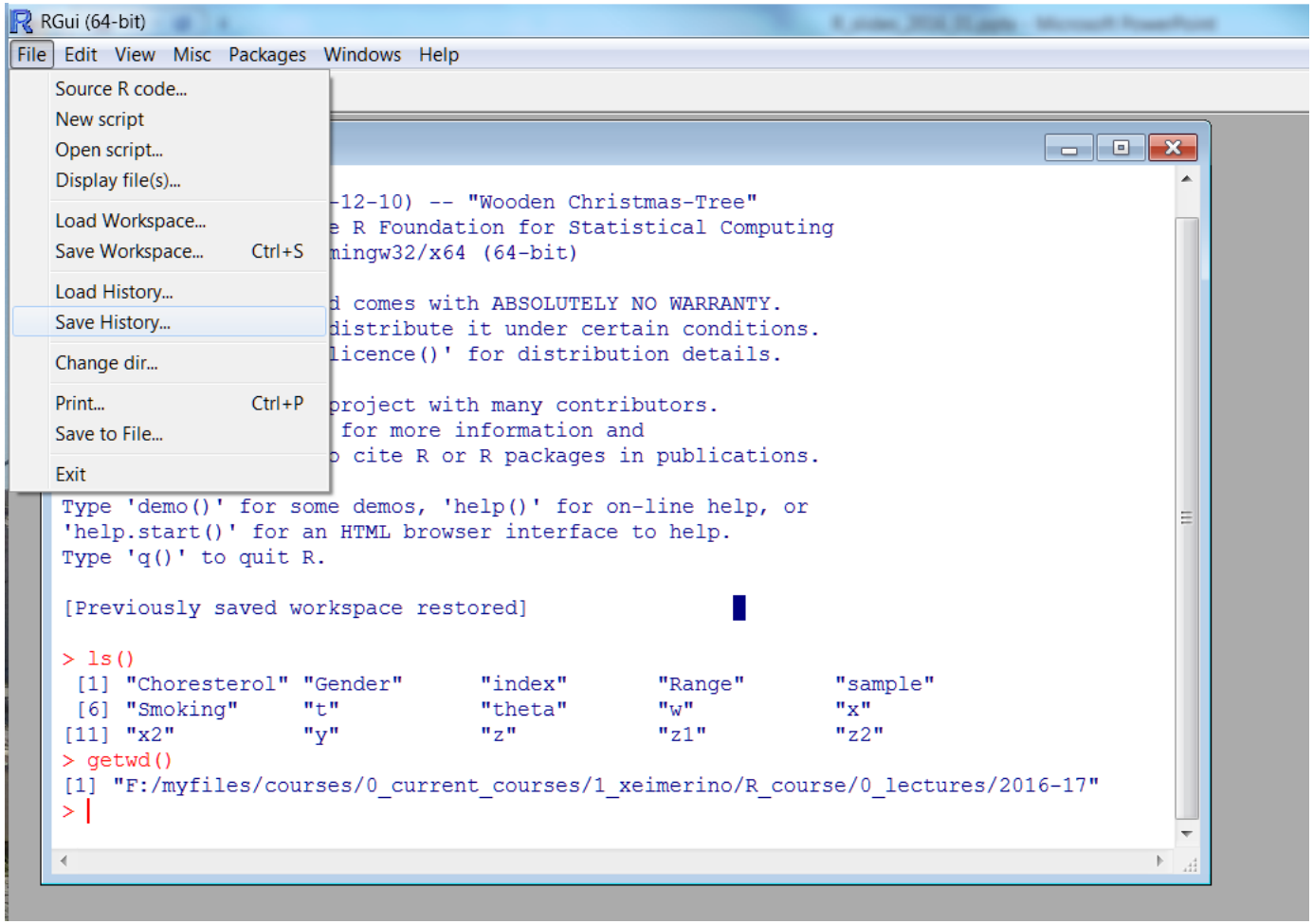
The R default environment



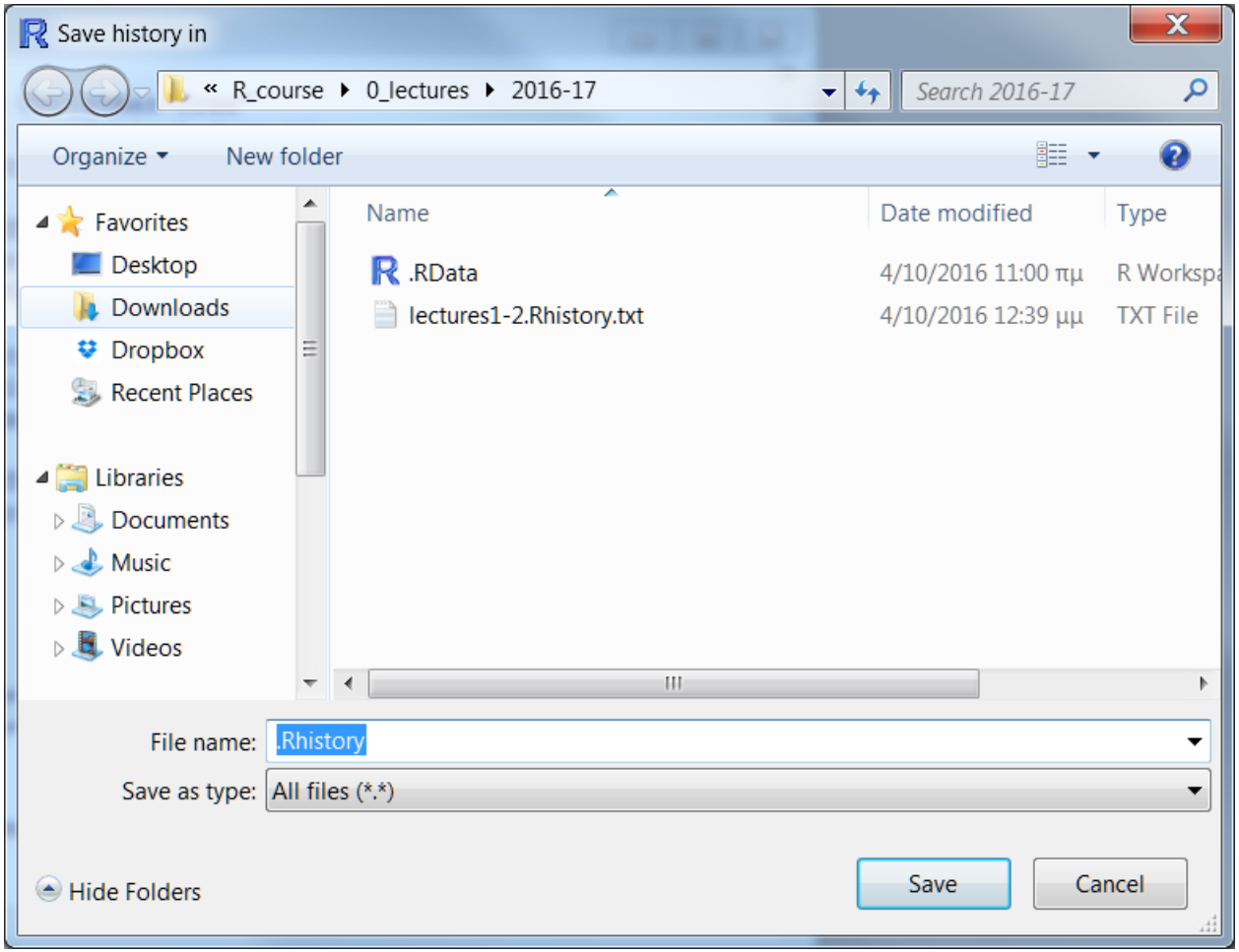
The R work space

```
> getwd()
[1] "F:/myfiles/courses/0_current_courses/1_xeimerino/R_course/0_lectures/2020-21"
```

The R work space



The R work space



The R work space

We can change the working directory (and the workspace) with the function setwd()

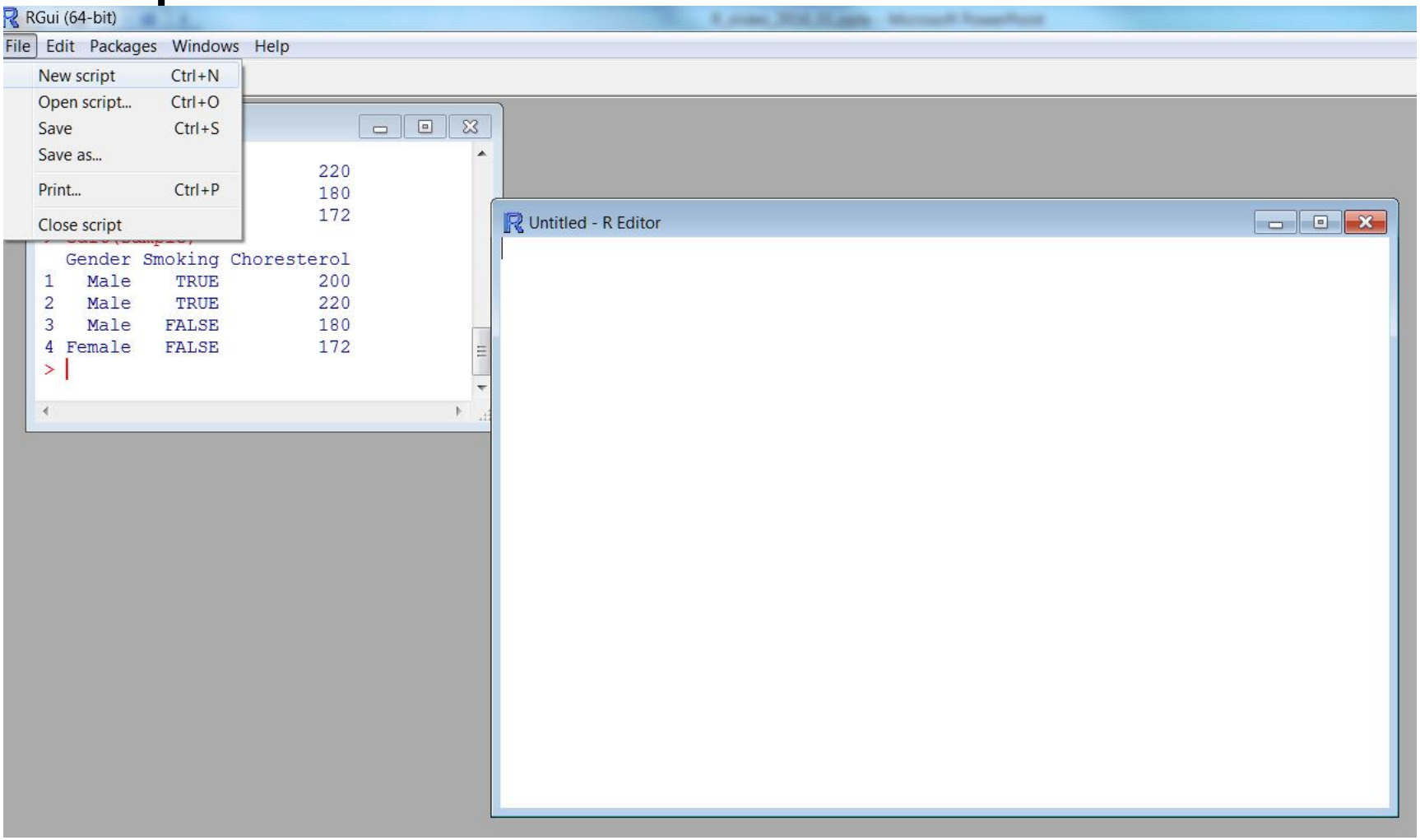
We recommend to

1. Save one workspace (.Rdata file) for each project/assignment/analysis in a separate directory
2. Empty the workspace from all elements before you save it for first time
3. Open R using double click on the desired directory and workspace

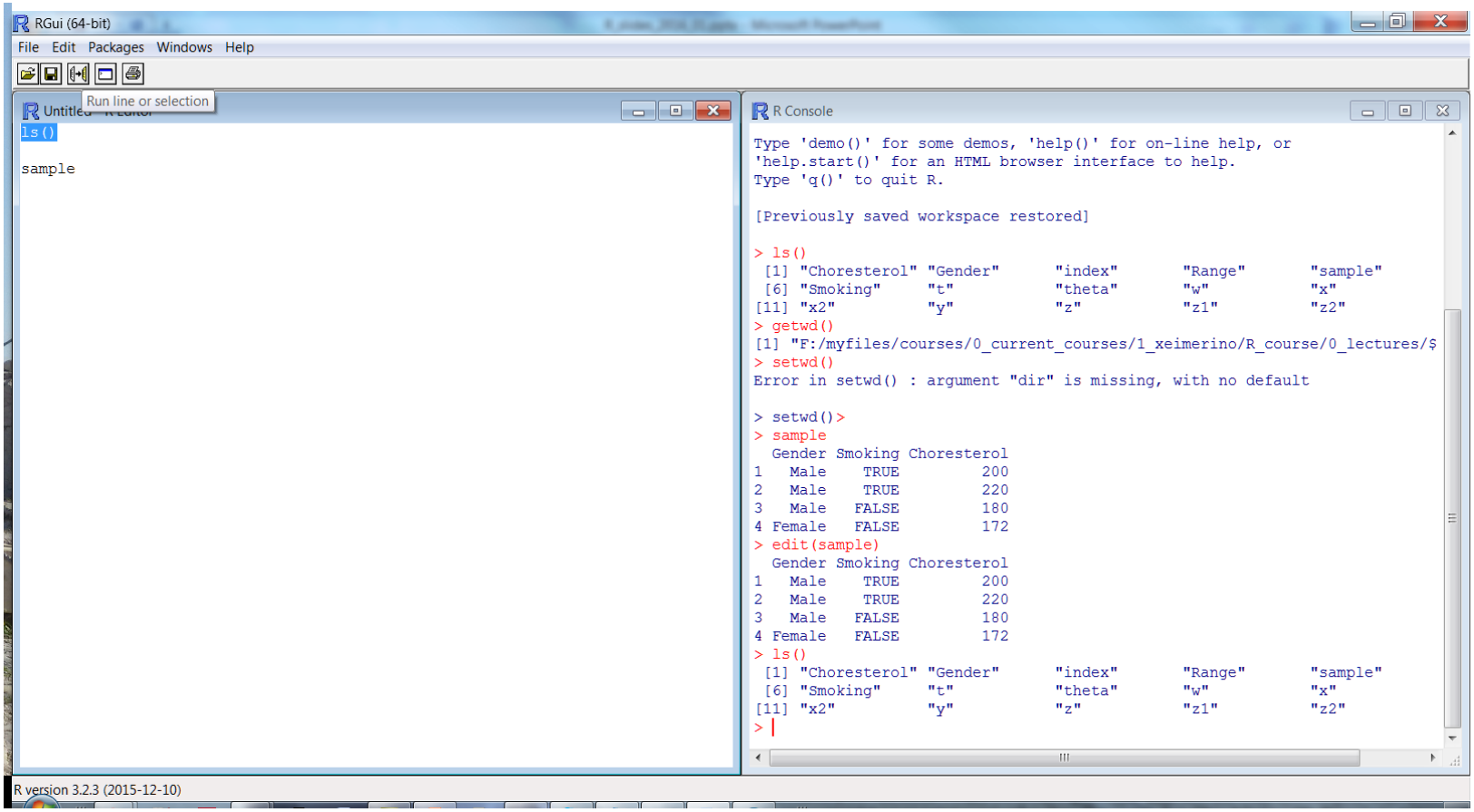
The Script Editor in R

- You can write all your commands in the R-editor
- Run part or all of the commands at once.
- To open => New Script option in the File menu.
- If you want to repeat your analysis at a later time, or send your code in a file to someone else, you can save the scripts.
- You can also print the scripts (from the Print option from the File menu).

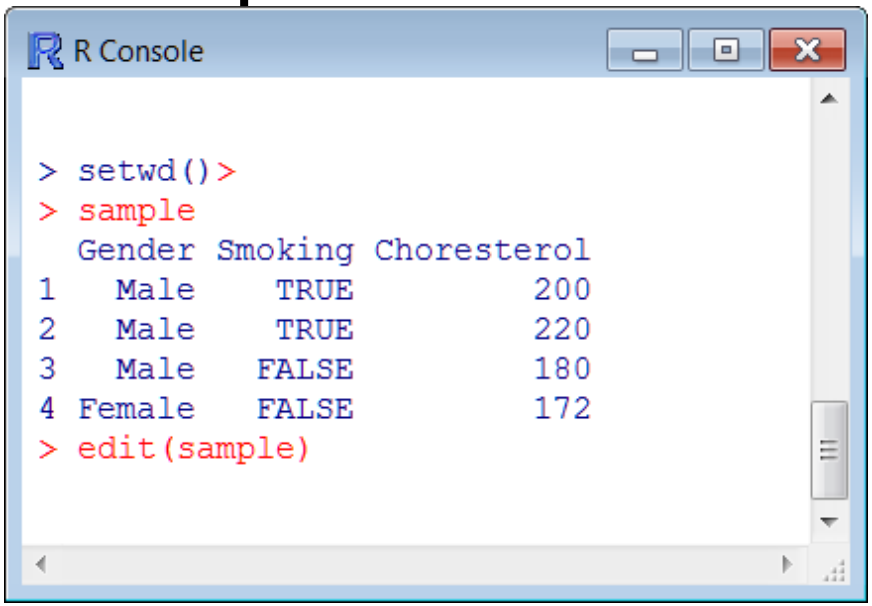
The Script Editor in R

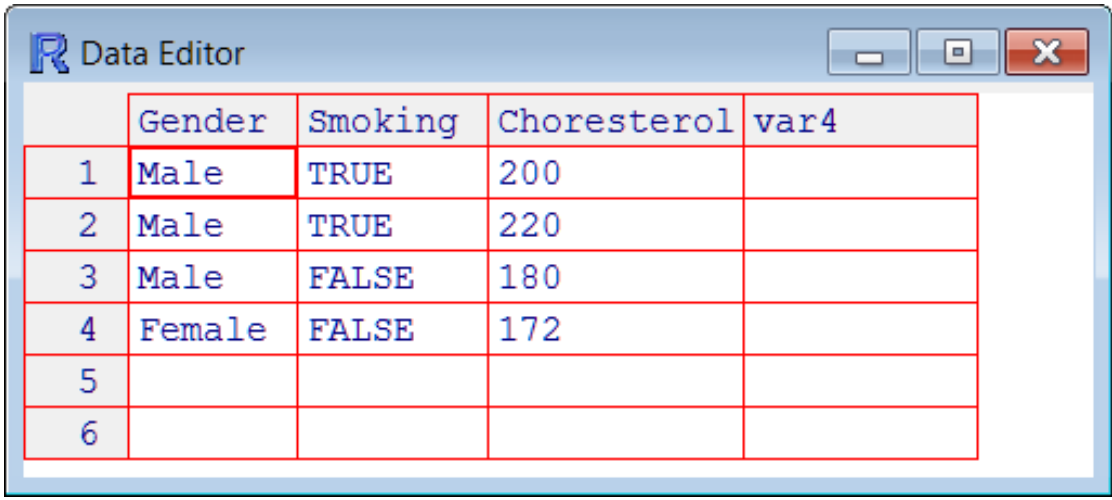


The Script Editor in R



The R Data Editor



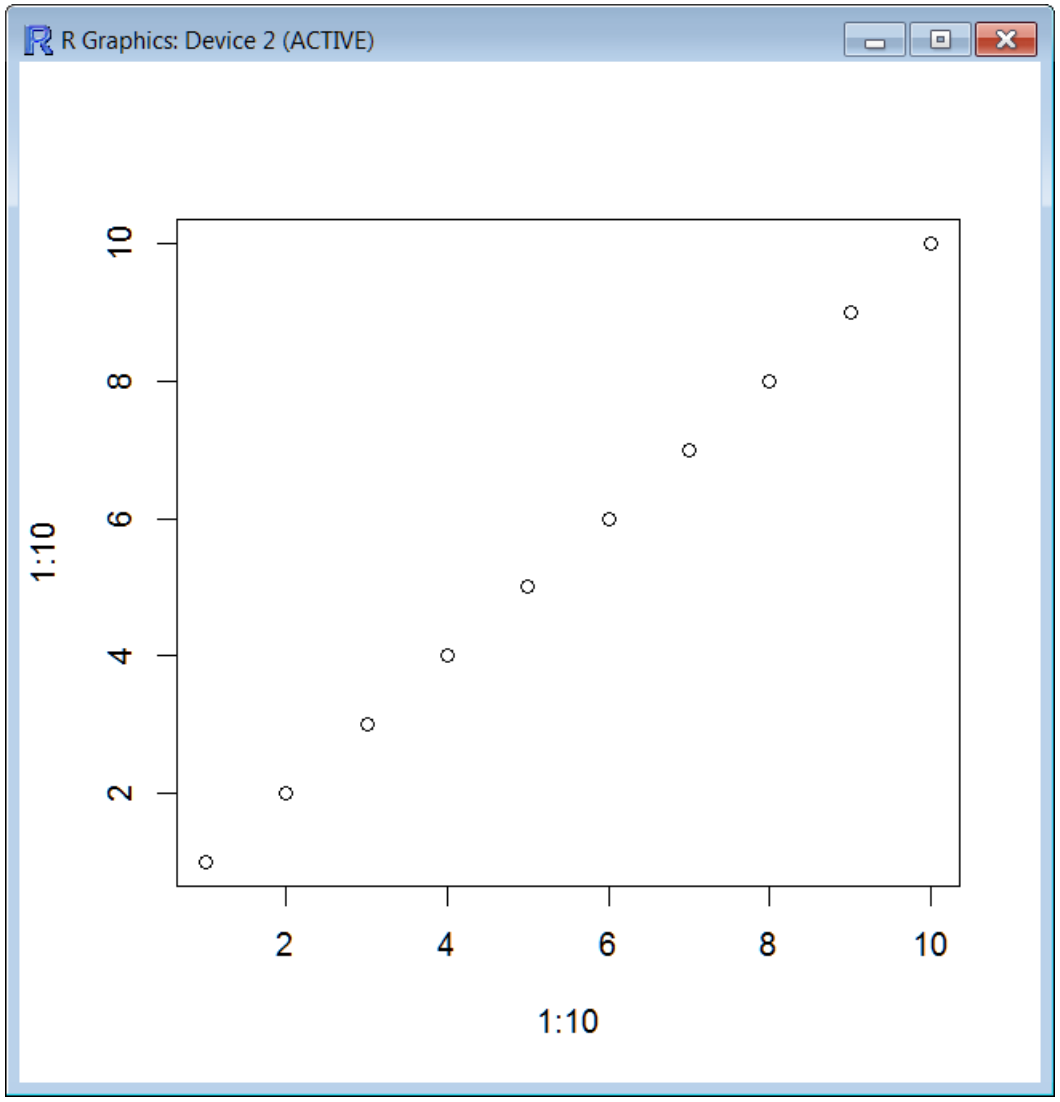


R graphics windows

- Graphics window are activated automatically when you create a graph.
- To bring a graph to the foreground: => Windows pull-down menu and choose the R Graphics window.
- Graphs can be saved in various formats (e.g. jpg, png, bmp, ps, pdf).

R graphics windows

```
> # New plot
> plot(1:10,1:10)
>#  New graphics window
> win.graph()
>#  close all graphic devices
> graphics.off()
>
```



Toolbar & Menu Bar

Like most window-based programs, R has a toolbar and a menu bar with pull-down menus that you can use to access many of the features of the program.

Toolbar & Menu Bar

Menu	*Functions*
File	Open source R code, create, open and save script, load and save workspace, load and save history, display files, change working directory, print files, and exit R.
Edit	Copy, paste, select all, clear console, data editor, R configuration window editor.
Misc	Stop current computation, buffer output, list objects in the memory, remove all objects, and list search path.
Packages / Packages & Data	Load, install, update packages, set CRAN mirror, select repositories, install packages, install packages from local zip files.
Window(s)	Cascade and tile R console windows. Arrange icons and switch among windows.
Help	Get help on R procedures, commands, and connect to the R website for more help information.

Getting Help

R provides help files for all its functions. To access a help file, use the Help pull-down menu,

Or type ? followed by a function name in the R console.

For example, to get help on the scan function, type:

```
> ?scan
```

Getting Help

```
> ?mean
> help(mean)
```

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

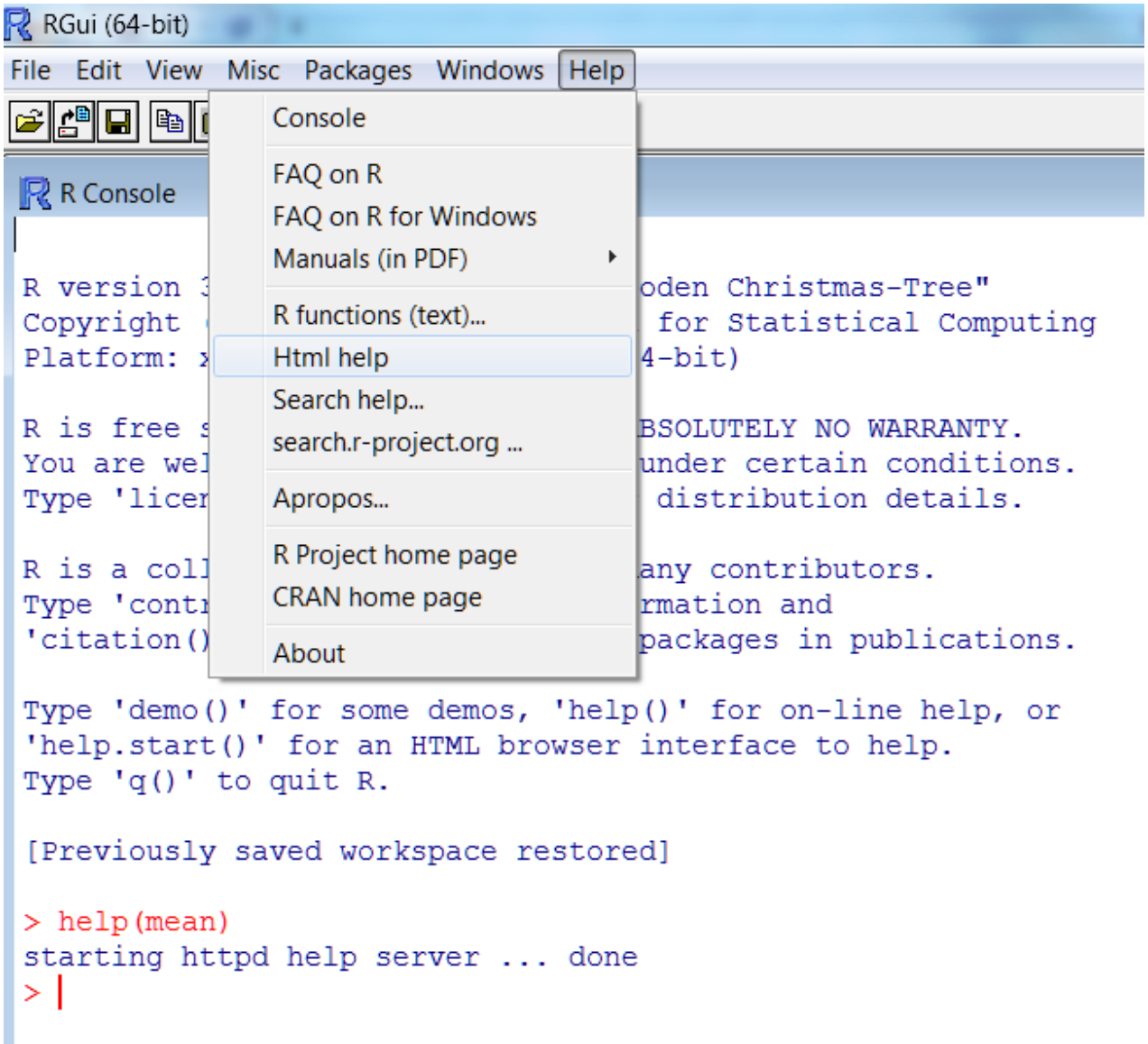
`x`

An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim`

the fraction (0 to 0.5) of observations to be trimmed from each end

Getting Help



Getting Help

Statistical Data Analysis 

Manuals

An Introduction to R
Writing R Extensions
R Data Import/Export
The R Language Definition
R Installation and Administration
R Internals

Reference

Packages
Search Engine & Keywords

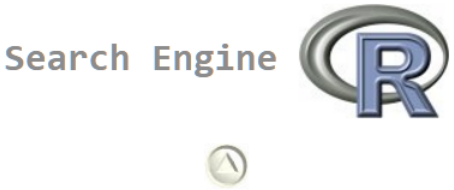
Miscellaneous Material

About R
License
NEWS
Authors
Frequently Asked Questions
User Manuals
Resources
Thanks
Technical papers

Material specific to the Windows port

CHANGES up to R 2.15.0
Windows FAQ

Getting Help



Search

You can search for keywords, function and data names, concepts and within help page titles. The first search of a session will be slower.

Usage: Enter a string in the text field below and click the Search button or hit RETURN.

mean

Fields: Topics Titles Concepts Keywords

Options: Ignore case Fuzzy match

Types: Help pages Vignettes Demos

Search Reset

Concepts

Browse concepts available for searching the help system

Keywords

Getting Help

Search Results

The search string was **"mean"**

Help pages:

actuar::aggregateDist	Aggregate Claim Amount Distribution
actuar::mean.grouped.data	Arithmetic Mean
agricolae::bar.err	Plotting the standard error or standard deviance of a multiple comparison of means
agricolae::bar.group	Plotting the multiple comparison of means
agricolae::diffograph	Plotting the multiple comparison of means
base::colSums	Form Row and Column Sums and Means
base::Date	Date Class
base::DateTimeClasses	Date-Time Classes
base::difftime	Time Intervals
base::mean	Arithmetic Mean
bayesm::condMom	Computes Conditional Mean/Var of One Element of MVN given All Others
BiasedUrn::BiasedUrn-Univariate	Biased urn models: Univariate distributions
BiasedUrn::BiasedUrn-Multivariate	Biased urn models: Multivariate distributions
boa::boa.batchMeans	Batch Means
boot::sunspot	Annual Mean Sunspot Numbers

Notation & Common R Operators

> Indicates the prompt at the start of each new line in the R console.

The comment operator.

<- The assignment operator. Also by "="

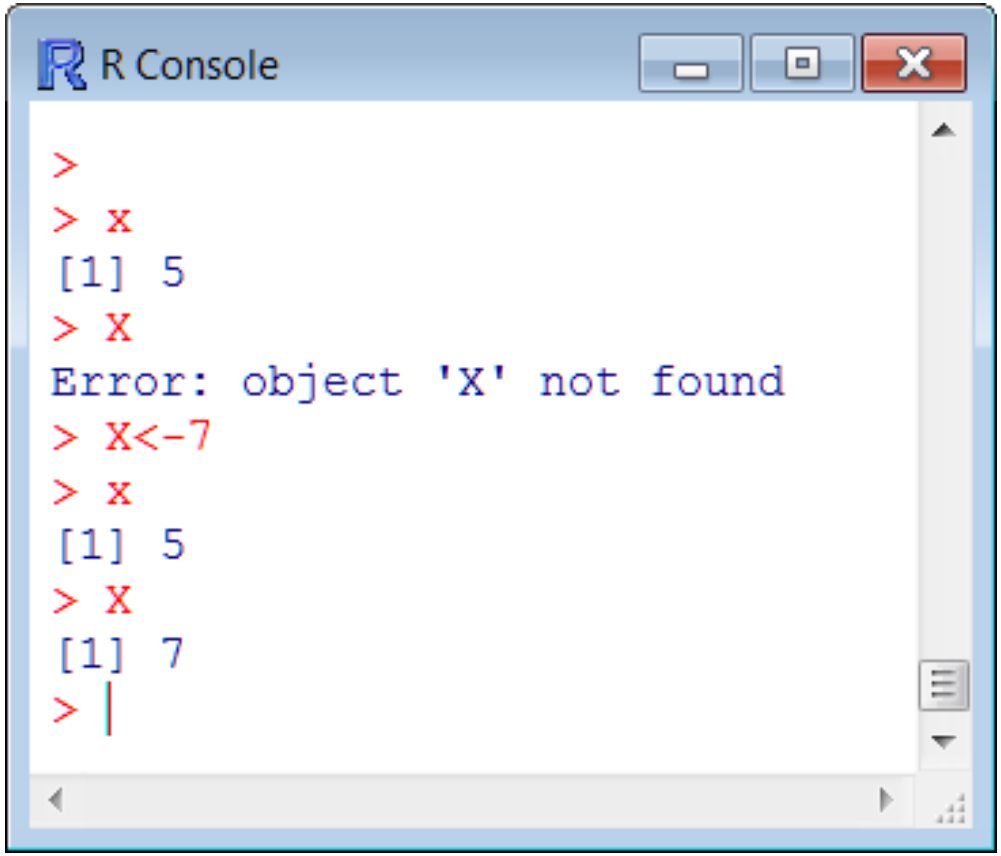
== Boolean equality operator. It is used in logical statements, assessing whether two quantities are equal.

Examples

```
x <- 5   # assigns the value 5 to x
x == 5   # returns TRUE
# assigns the value 6 to x, overwriting the previous statement x <- 5.
x = 6
x == 5 # returns FALSE
```

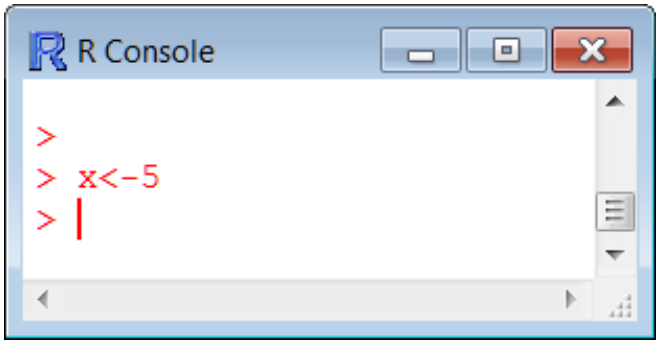
R is case sensitive

- R is case sensitive, i.e. x and X are different objects.

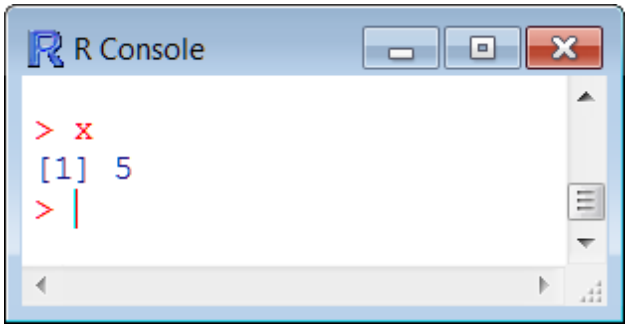


Small details

- When you use the assignment operator the result is not visible on screen.



- To see the value that your object took just type its name.



- Everything is assigned on objects (vectors, matrices, data frames, or lists) (more later).

Assignment, list and removal

```
> ls()
character(0)
> 5+6
[1] 11
> ls()
character(0)
> y
Error: object 'y' not found
```

```
> y<-5+6
> y
[1] 11
> ls()
[1] "y"
> rm(y)
> y
Error: object 'y' not found
> ls()
character(0)
>
```

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
% / %	Integer Division
% %	Modulus

Examples

```
> 3+3  # this is my first command
[1] 6
> 9-2
[1] 7
> 17/3
[1] 5.666667
> 4*2
[1] 8
> 2^3
[1] 8
> 17%/%3
[1] 5
> 17%%3
[1] 2
> 7/0
[1] Inf
```

Use of parentheses

```
# Calculations and parentheses
> 4+3*6
[1] 22
> (4+3)*6
[1] 42
> (4+3)
[1] 7
> 4+3*6^2
[1] 112
> (4+3*6)^2
[1] 484
> 6^2
[1] 36
> 6^2*3
[1] 108
```

Use of parentheses

```
# Calculations and parentheses
> 3*6^2
[1] 108
> ((3+4)*2+10)^4
[1] 331776
> 3+4*2+10^4
[1] 10011
> ((1+2)*2+3)^2
[1] 81
> 1+2*2+3^2
[1] 14
```

Positive and Negative Numbers

```
> +4
[1] 4
> -4
[1] -4
> 4+
+ 6
[1] 10
> 4+
+ 4+
+ 4+*5
Error: unexpected '*' in:
"4+
4+*"
> 4+*5
Error: unexpected '*' in "4+*"
```

```
> +4*5
[1] 20
> -4*5
[1] -20
> -4*-5
[1] 20
> (-4)*(-5)
[1] 20
> -4*+5
[1] -20
> sign(5)
[1] 1
> sign(-3)
[1] -1
```

Incomplete computations and expressions

When a computation (or more generally a syntax) is incomplete, then it is continued in the next line.

Continuation is highlighted with "+" instead of ">"

```
> 4+5/
+ 3
[1] 5.666667
```

Using objects in computations

We can save values to objects and use then in mathematical expressions or computations.

```
> r<-10
> 4+5/r
[1] 4.5
```

Using objects in computations

```
> # Illustration of %% and %/% using
objects a and b.
> a<-10
> b<-3
> a%%b
[1] 1
> a-b*(a%/%b)
[1] 1
> a%%b/b
[1] 0.3333333
> a<-140534
> b<-323
> a%%b
[1] 29
```

```
> a-b*(a%/%b)
[1] 29
> a%%b/b
[1] 0.08978328
```

Using objects in computations

Estimate the quantity

$$-\frac{\log\left[1-\left(1-e^{-\theta}\right)e^{-t}\right]}{\theta}$$

for θ=3 and t=5.

```
> theta <- 3
> t <- 5
> -log(1-(1-exp(-theta))*exp(-t))/theta
[1] 0.002141023
```

! Caution with the order of operations

Special Numeric values

R supports 4 special numeric values:

Inf:	Infinity
-Inf:	Minus infinity
NA:	Not available
NaN:	Not a Number
NULL:	Null object in R (with no elements)

There are special functions available to check for these special values (see examples in the next slides).

Special Numbers Example

```
> exp(709)
[1] 8.218407e+307
> exp(710)
[1] Inf
> -exp(-709)
[1] -1.216781e-308
> -exp(710)
[1] -Inf
> 7/0
[1] Inf
> -1/0
[1] -Inf
> -10^1000
[1] -Inf
> exp(-Inf)
[1] 0
> log(-2)
[1] NaN
> 0/0
[1] NaN
> x<-1
> names(x)
[1] NULL
> 7/0-7/0
[1] NaN
> 7/0-1
[1] Inf
```

Checking for special values

Special logical functions that check for the existence of values:

is.infinity:	Infinity
is.na:	Not available
is.nan:	Not a Number
is.null:	Null object in R (with no elements)

Checking for special values

```
> is.null(x)
[1] FALSE
> is.null(names(x))
[1] TRUE

> is.infinite(exp(700))
[1] FALSE
> is.infinite(exp(710))
[1] TRUE
> is.na(exp(710))
[1] FALSE
> is.nan(exp(710))
[1] FALSE
```

```
> is.na(7/0)
[1] FALSE
> x<-0/0
> x
[1] NaN
> is.na(x)
[1] TRUE
> is.nan(x)
[1] TRUE
> x<-NA
> is.na(x)
[1] TRUE
> is.nan(x)
[1] FALSE
>
```

Checking for special values

```
> x<-c(0,Inf,-Inf, NaN, NA)
> is.finite(x)
[1] TRUE FALSE FALSE FALSE FALSE
> is.infinite(x)
[1] FALSE TRUE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE TRUE FALSE
> is.na(x)
[1] FALSE FALSE FALSE TRUE TRUE
```

Logical Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x|y	x OR y
x&y	x AND y

Examples

```
> x<-56
> x
[1] 56
> 5*2->y
> y
[1] 10
> x>y
[1] TRUE
> y<4
[1] FALSE
```

```
> x>=56
[1] TRUE
> y<=9
[1] FALSE
> x==56
[1] TRUE
> y!=1
[1] TRUE
> !(5>3)
[1] FALSE
```

Examples

```
> 16.0000000000001 == 16
[1] FALSE
> round(16.0000000000001,3) == 16
[1] TRUE
> abs(16.0000000000001-16) < 0.000001
[1] TRUE
```

More on Logical Operators

We usually use the symbols & (AND), | (OR) to combine two or more logical operators.

```
> (5>3) & (8>10)
[1] FALSE
> (5>3) | (8>10)
[1] TRUE
```

Logical operators are frequently used in loops, e.g. if, for, etc. (more later).

Examples

```
> x<-5
> y<-10
> x>y
[1] FALSE
> x<y
[1] TRUE
> !(x<y)
[1] FALSE
> x<y
[1] TRUE
> (x<y) & (5!=4)
[1] TRUE
> (x<y) & (5==4)
[1] FALSE
> (x<y) | (5==4)
[1] TRUE
```

```
> x
[1] 5
> (x>8)
[1] FALSE
> (x<5)
[1] FALSE
> (x>8) | (x<5)
[1] FALSE
```

Basic Mathematical Functions in R

Function	Description	Function	Description
sqrt()	Square root	asin()	Inverse sine
abs()	Absolute value	atan()	Inverse tangent
log()	Natural logarithm (ln)	gamma()	Gamma function
log2()	Logarithm base 2	lgamma()	Natural logarithm of gamma function
log10()	Logarithm base 10	beta()	Beta function
exp()	Exponential function	floor()	Previous integer
cos()	Cosine	ceiling()	Next integer
sin()	Sine	factorial()	Factorial
tan()	Tangent	choose()	Combinations
acos()	Inverse cosine	lchoose()	Natural logarithm of combinations

Examples

```
> sqrt(16)
[1] 4
> abs(-2)
[1] 2
> log(10)
[1] 2.302585
> log2(10)
[1] 3.321928
> log10(10)
[1] 1
> exp(3)
[1] 20.08554
> cos(pi)
[1] -1
> sin(2*pi)
[1] -2.449213e-16
> tan(pi/2)
[1] 1.633178e+16
```

```
> sin(pi/2)
[1] 1
> tan(0)
[1] 0
> acos(0.2)
[1] 1.369438
> atan(2)
[1] 1.107149
> asin(0)
[1] 0
> gamma(2)
[1] 1
> beta(2,3)
[1] 0.08333333
```

```
> lgamma(4)
[1] 1.791759
> floor(4.9)
[1] 4
> ceiling(4.1)
[1] 5
> factorial(5)
[1] 120
> choose(5,2)
[1] 10
> lchoose(5,2)
[1] 2.302585
```

Logarithm

log {base} R Documentation

Logarithms and Exponentials

Description

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes $log(1+x)$ accurately also for $|x| << 1$.

`exp` computes the exponential function.

`expm1(x)` computes $exp(x) - 1$ accurately also for $|x| << 1$.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
```

Examples for log

```
> log(10)
[1] 2.302585
> log(10,2)
[1] 3.321928
> log(10,10)
[1] 1
> exp(2.302585)
[1] 9.999999
> exp(log(10))
[1] 10
> log2(10)
[1] 3.321928
> log10(10)
[1] 1
```

General Functions in R

- **builtins():** built-in functions in R.
- **cat()** or **print():** Print on screen.
- **ls():** list all objects
- **rm():** remove objects.
- **getwd():** returns working directory.
- **setwd():** changes working directory.
- **list.files():** returns a list with all files in working directory.
- **date()** or **Sys.time():** Returns current day & time.

Type of object elements in R

R has the following type of elements:

- Numeric (real number)
- Complex
- Logical
- Character

We can use the function mode() check the type of the elements of an objects.

Type of object elements in R

- Numeric (real number)

```
> x<- 3
```

- Complex

```
> x<-complex(real=4, imaginary=3)
> x
[1] 4 +3i
```

- Logical

```
> x <-3
> y <- (x > 4)
> y
[1] FALSE
```

- Character

```
> x <- "DIMITRIS"
> x
[1] "DIMITRIS"
```

Type of object elements in R

```
x1<-3
mode(x1)
[1]  "numeric"
x2<-3.004
x3<-1.9e-5
```

```
x1<-3
x2<-  x1>4
x2
[1] FALSE
mode(x2)
[1]  "logical"
x3<-  x1<4
x3
[1] TRUE
```

```
x1<-'Ioannis'
x1
[1] "Ioannis"
mode(x1)
[1] "character"
```

```
x1<-complex(real=4,imaginary=3)
x1
[1] 4+3i
mode(x1)
[1] "complex"
```

Types of Objects in R

- Vectors.
- Matrices.
- Arrays.
- Data frames.
- Lists.

In the 3 first cases all elements should be of the same type.

Data frames => each column can be a vector of different class with the same length

A list allows you to gather a variety of (possibly unrelated) objects under one name.

Use class to obtain the type of object

Class vs. mode

Function mode will give

- Numeric (real number)
- Complex
- Logical
- Character

If the object has elements of the same type (i.e. for vectors, matrices and arrays).

For lists/data.frames the mode is list

Class vs. mode

Function class is more detailed and it gives

- Usually the same as mode for vectors (except for factors)
- The names of the object type for matrices, arrays, lists and data.frames

Class vs. mode

```
> x<-1:10
> mode(x); class(x)
[1] "numeric"
[1] "integer"
>
> x<- 1:10<5
> mode(x); class(x)
[1] "logical"
[1] "logical"
>
```

```
> x<- c('Yiannis','Grigoris')
> mode(x); class(x)
[1] "character"
[1] "character"
>
> x<- complex(1,5)
> mode(x); class(x)
[1] "complex"
[1] "complex"
>
```

Class vs. mode

```
> x <- matrix(1:10,2)
> mode(x); class(x)
[1] "numeric"
[1] "matrix"
>
> x <- array(1:8, c(2,2,2))
> mode(x); class(x)
[1] "numeric"
[1] "array"
```

```
> x <- list(x=1:10)
> mode(x); class(x)
[1] "list"
[1] "list"


> x <- data.frame(x=1:10)
> mode(x); class(x)
[1] "list"
[1] "data.frame"
```

Vectors

A vector in R is a sequence of data elements of the same class

- **Numerical Vectors**
- **Character Vectors**
- **Logical Vectors**
- **Factors**

Numerical Vectors

- To create a numerical vector use the function **c()** which ***combines*** elements in a vector

```
> x<-c(1,2,3,4,5)
> x
[1] 1 2 3 4 5
```

- The same function can be used to concatenate already defined vectors

```
> x<-c(1,2,3,4,5)
> y<-c(6,7)
> z<-c(10,x,y)
> z
[1] 10  1  2  3  4  5  6  7
```

Functions for vectors

- length(x): length of the vector (number of elements of the vector)
- names(x) : extract or give a name to each element of the vector
- mode(x) : type of vector (i.e. numeric, logical, etc.)
- Vector functions (or statistical): Input is a vector. The function uses all values of the vector and the result is (usually) one value

  mean, median, sd, min, max, sum, prod

Functions for vectors

- length(x): length of the vector

```
> x<-c(1,2,3,4,5)
> length(x)
[1] 5
```

Functions for vectors

- `names(x)` : extract or give a name to each element of the vector

```
> height<-c(1.75,1.84,1.81,1.63)
> names(height)
NULL
> names(height)<-c("Jim","George","John","Mary")
> names(height)
[1] "Jim" "George" "John" "Mary"
> height
Jim George John Mary
1.75   1.84   1.81   1.63
> names(height) <- NULL # removes the names
```

Functions for numerical Vectors

- **Min & Max**

```
> min(x)
[1] 1
> max(x)
[1] 5
```

- **Sum & Product**

```
> sum(x)
[1] 15
> prod(x)
[1] 120
```

- **Other examples**

```
mean(x)
     [1] 3
var(x)
     [1] 2.5
median(x)
    [1] 3
quantile(x)
    0%  25%  50%  75% 100%
     1    2    3    4    5
```

Extracting elements in vectors

Squared brackets "[ ]" can be used to extract specific elements of a vector.

x[i] => (i is an integer) extracts the i-th element of x

x[a] => if a is a vector with integers < length(x) – results in a vector with a[1]-th, a[2] -th, a[3] -th … elements of x

x[-a] => if a is a vector with integers < length(x) – results in a vector without the a[1]-th, a[2] -th, a[3] -th … elements of x

x[L] => if L is logical vector of length(x) – results in a vector containing only the elements for which L is true

Extracting elements in vectors

```
> x<-seq(from=1,to=9,by=2)
> x
[1] 1 3 5 7 9
> x[2]
[1] 3

> x[2:4]
[1] 3 5 7

> x[c(1,3)]
[1] 1 5
```

```
> x[-c(1,3)]
[1] 3 7 9

> y<-c(F,T,T,F,T)
> x[y]
[1] 3 5 9
```

Missing Values

```
x3<-c(1,2,3,NA,9)
> x3
[1]  1  2  3 NA  9
> is.na(x3)
[1] FALSE FALSE FALSE  TRUE FALSE
```

Missing Values

```
> x3<-c(x3,NA, NA,x3)
> x3
 [1]  1  2  3 NA  9 NA NA  1  2  3 NA  9 NA
> is.na(x3)
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
     FALSE FALSE  TRUE FALSE  TRUE
> x3[is.na(x3)]
[1] NA NA NA NA NA
> x3[!is.na(x3)]
[1] 1 2 3 9 1 2 3 9
> w<-x3[!is.na(x3)]
> w
[1] 1 2 3 9 1 2 3 9
```

Missing Values

```
> which(is.na(x3))
[1]  4  6  7 11 13
> index<-1:length(x3)
> index
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
> index[is.na(x3)]
[1]  4  6  7 11 13
> is.na(x3)
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
    FALSE FALSE  TRUE FALSE  TRUE
```

Creating sequences of numbers

The colon Operator

Syntax a:b => generates a sequence of numbers from a to b with step 1 (or -1 if a>b)

```
> x<-1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10
> x<- -4:10
> x
[1] -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
> x<-6:1
> x
[1] 6 5 4 3 2 1
> x<-3.3:10.3
> x
[1]  3.3  4.3  5.3  6.3  7.3  8.3  9.3 10.3
```

Creating sequences of numbers

The colon Operator

If (b-a) is not an integer, then R will stop before b

```
> x<-3.3:6.9
> x
[1] 3.3 4.3 5.3 6.3
```

Creating sequences of numbers

The sequence command – seq()

More general than the colon operator since you can define the step

by – controls the step

```
> seq(from=1,to=9, by=2)
[1] 1 3 5 7 9
```

length.out – controls the length of the vector. The step is calculated accordingly

```
> seq(from=1,to=9, length.out=3)
[1] 1 5 9
> seq(to=9, length.out=3)
[1] 7 8 9
> seq(from=1,by=2, length.out= 10)
[1]  1  3  5  7  9 11 13 15 17 19
```

Creating sequences of numbers

The sequence command – seq()

length – controls the length of the vector. The step is calculated accordingly

```
> seq(from=1,to=9, length.out=3)
[1] 1 5 9
> seq(to=9, length.out=3)
[1] 7 8 9
> seq(from=1,by=2, length.out=10)
[1]  1  3  5  7  9 11 13 15 17 19
```

along – sets the length of the vector to be the same with another vector

```
> y<-1:10
> seq(from=1,by=2,along=y)
[1]  1  3  5  7  9 11 13 15 17 19
```

Creating sequences of numbers

The sequence command – seq()

From all the arguments from, to, by, length, along you need to specify only 3.

if the user specifies 2 then by default R values for the rest of the arguments is one (1).

The seq() last element is always lower or equal to the value of the argument "to".

e.g.

```
> seq(from=1,to=10,by=2)
[1] 1 3 5 7 9
```

Replicating values

The command rep().

- x: a vector to replicate
- times:
  1. Single value=> how many times to replicate x
  2. Vector of the same length as x: how many times to repeat each element of x
- each: (non-negative integer) each element of x is repeated each times. Default value = 1

```
> rep(2,5)
[1] 2 2 2 2 2
> x<-c(1,2,3)
> rep(x,5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> rep(x,each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Sorting numerical Vectors

Sort

```
> x<-c(3,5,2,1,6)
> sort(x)
[1] 1 2 3 5 6
> sort(x,decreasing=T)
[1] 6 5 3 2 1
```

Rank

```
> x<-c(3,5,2,1,6)
> rank(x)
[1] 3 4 2 1 5
```

Order

```
> x<-c(3,5,2,1,6)
> order(x)
[1] 4 3 1 2 5
> x[order(x)]
[1] 1 2 3 5 6
```

the smallest value is on the 4th position, the next one on the 3rd, etc

Handling ties in rank

Average: assigns each tied element the "average" rank

```
> x2<-c(3,5,2,1,6,3)
> rank(x2, ties.method="average")
[1] 3.5 5.0 2.0 1.0 6.0 3.5
```

First: lets the "earlier" entry "win", so the ranks are in numerical order

```
> rank(x2, ties.method="first")
[1] 3 5 2 1 6 4
```

Random: breaks ties randomly

```
> rank(x, ties.method="random")
[1] 4 5 2 1 6 3
> rank(x, ties.method="random")
[1] 3 5 2 1 6 4
```

Min/Max: assigns every tied element to the lowest/highest rank

```
> rank(x, ties.method="min")
[1] 3 5 2 1 6 3
> rank(x, ties.method="max")
[1] 4 5 2 1 6 4
```

Operations with vectors

- All arithmetic operations can be used to numerical vectors.
- Operations will be done element by element.
- It works for the case of different length because R recycles the sorter vector.

Elementwise numerical operations for vectors

Arithmetic Operators & Vectors: +, - , *, /, %/%, %%

Can be used with vectors of equal length

Each operation is applied in each element of the same order

```
> x<-c(1,2,3)
> x*3
[1] 3 6 9
> x^2
[1] 1 4 9
> y<-c(4,5,6)
> y/x
[1] 4.0 2.5 2.0
```

Examples of vector operations

```
> x<-1:10
> y<-rep(c(2,3),each=5)
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y
[1] 2 2 2 2 2 3 3 3 3 3
> x+y
[1] 3 4 5 6 7 9 10 11 12 13
> x-y
[1] -1 0 1 2 3 3 4 5 6 7
> x^y
[1] 1 4 9 16 25 216 343 512 729 1000
```

Examples of vector operations

```
> x*y
[1] 2 4 6 8 10 18 21 24 27 30
> x/y
[1] 0.500000 1.000000 1.500000 2.000000 2.500000
[6] 2.000000 2.333333 2.666667 3.000000 3.333333
> (x^2)/(y-5)
[1] -0.3333333 -1.3333333 -3.0000000 -5.3333333
[5] -8.3333333 -18.0000000 -24.5000000 -32.0000000
[9] -40.5000000 -50.0000000
> x*y-2
[1] 0 2 4 6 8 16 19 22 25 28
> x*(y-2)
[1] 0 0 0 0 0 6 7 8 9 10
```

Examples of vector operations

```
> exp(x-y)
 [1] 0.3678794 1.0000000 2.7182818 7.3890561
 [5] 20.0855369 20.0855369 54.5981500 148.4131591
 [9] 403.4287935 1096.6331584
> x<-1:10
> y<-rep(1:2,3)
> x
 [1] 1 2 3 4 5 6 7 8 9 10
> y
 [1] 1 2 1 2 1 2
> x+y
 [1] 2 4 4 6 6 8 8 10 10 12
```

Character Vectors

Every vector with character elements is included in quotes ("*")

```
> x<-c("Statistics", "Mathematics")
> x
[1] "Statistics" "Mathematics"
> x<-c( 'Yiannis', 'Yiorgos', 'Grigoris' )
> x
 [1] "Yiannis"  "Yiorgos"  "Grigoris"
> x[2]
[1] "Yiorgos"
```

Arithmetic operators and characters

```
> x1<-1
> x2<-mode(x1)
> x1
[1] 1
> x2
[1] "numeric"
> # adding a character and a numeric vector
> x1+x2
Error in x1 + x2 : non-numeric argument to binary operator
> x1+5
[1] 6
> '1'+5
Error in "1" + 5 : non-numeric argument to binary operator
```

Arithmetic operators and characters

```
> # using as.numeric() in character vectors
> # converting a character to a number and adding it to another
    number
> as.numeric('1')+5
[1] 6
> as.numeric('16g')+5
[1] NA
Warning message:
NAs introduced by coercion
> x1<- "5"
> x1^2
Error in x1^2 : non-numeric argument to binary operator
> as.numeric(x1)^2
[1] 25
```

Arithmetic operators and characters

```
> x1<-T
> x1
[1] TRUE
> x2^2
Error in x2^2 : non-numeric argument to binary operator
> x1^2
[1] 1
```

Equalities with characters

```
> x1<-"5"
> x1=="5"
[1] TRUE
> x1=="5  ” # spaces are important
[1] FALSE
x<-'          5    '
> x=='5'
[1] FALSE
> x=='    5'
[1] FALSE
x<-          5 # white spaces in numeric are not important
> x2<-'g'
> x2=="G” # it is case sensitive
[1] FALSE
> x2=="g"
[1] TRUE
```

Equalities with characters

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u"
     "v" "w" "x" "y" "z"
> # checking if each element of character vector 'letters'  is equal to 'b'
>  letters=='b'
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> x<-letters=='b'
> index<-1:length(letters)
> index
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
> index[x]
[1] 2
> which(x)
[1] 2
```

Equalities with characters

```
> letters==c('b','c')
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> letters==c('g','b','c')
 [1] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
Warning message:
In letters == c("g", "b", "c") :
  longer object length is not a multiple of shorter object length
> as.numeric(letters==c('g', 'b','c'))
 [1] 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Warning message:
In letters == c("g", "b", "c") :
  longer object length is not a multiple of shorter object length
```

Using c() to combine characters and numbers

```
> A<- c("A","B","C","D")
> A
[1] "A" "B" "C" "D"
> B<- c("A","B","C","D",1,2,3) # Numbers are converted to characters.
> B
[1] "A" "B" "C" "D" "1" "2" "3“
> mode(c(1,2,3))
[1] "numeric"
> mode(A)
[1] "character“
> # combine two vectors
> x<-c(3,5,6,5,4,4,3,3,8)
> c(A,x)
 [1] "A" "B" "C" "D" "3" "5" "6" "5" "4" "4" "3" "3" "8"
> c(x,A)
 [1] "3" "5" "6" "5" "4" "4" "3" "3" "8" "A" "B" "C" "D"
```

Functions for character vectors

- **character(length = i)** : creates a character vector with length i
- **as.character(x, ...)** : turns a vector to a character vector
- **is.character(x)** : checks if a vector is a character vector
- **print**: prints all elements with quotes
- **noquote** : prints all elements without quotes
- **nchar** : number of element's characters

Functions for character vectors

- **character(length = 10)**: creates a character vector with length 10

```
> character(10)
 [1] "" "" "" "" "" "" "" "" "" ""
```

- **as.character(x, ...)** : turns a vector to a character vector

```
> as.character(1:10)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
> as.character(c(T,F))
[1] "TRUE"  "FALSE"
```

- **is.character(x)** : checks if a vector is a character vector

```
> x<-as.character(c(T,F))
> is.character(x)
[1] TRUE
> is.character(1:10)
[1] FALSE
```

Functions for character vectors

- **print**: prints all elements with quotes
- **noquote** : prints all elements without quotes

```
> x<-c('Statistics', 'Mathematics')
> x
[1] "Statistics"  "Mathematics"
> print(x)
[1] "Statistics"  "Mathematics"
> noquote(x)
[1] Statistics  Mathematics
> z<-noquote(x)
> z[1]
[1] Statistics
> z[2]
[1] Mathematics
```

Functions for character vectors

- **nchar** : number of element's characters

```
> xsurnames<-c('Papadopoulos','Kitsos','Ioannou', 'Tsapara', 'Grigoriadou')
> nchar(xsurnames)
[1] 12  6  7  7 11
```

More examples

is.character()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"  "6"
     "7"  "8"  "9"  "10"
> is.character(y)
[1] TRUE
> x<-c("-0.1","2.7","B")
> x
[1] "-0.1" "2.7"  "B"
> is.character(x)
[1] TRUE
> x<-as.numeric(x)
Warning message:
NAs introduced by coercion
> x
[1] -0.1  2.7   NA
> is.character(x)
[1] FALSE
```

noquote()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> noquote(y)
[1] 1  2  3  4  5  6  7  8  9
    10
```

nchar()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> nchar(y)
 [1] 1 1 1 1 1 1 1 1 1 2
```

Pasting character vectors

paste(vector1, ..., vectorn, sep = " ", collapse = NULL)

Combines different character vectors

sep is the separator between the vectors to be combined

collapse checks whether the new vector is going to be a sequence or not

Pasting character vectors Examples

```
> paste('Yiannis','Nicolaou')
[1] "Yiannis Nicolaou "

> x1<-'Yiannis'
> x2<-'Nicolaou'
> paste(x1,x2)
[1] "Yiannis Nicolaou "

> x1<-5
> x2<-13
> paste(x1,x2, x1+x2, sep=' + ')
[1] "5 + 13 + 18"
> paste(paste(x1,x2, sep=' + '), x1+x2, sep=' = ')
[1] "5 + 13 = 18"
```

Pasting character vectors Examples

```
> xnames<-c('Yiannis','Yiorgos','Barbara','Aleka', 'Eugenia')
>  xsurnames<-c('Papadopoulos','Kitsos','Ioannou', 'Tsapara',‘
    Grigoriadou')
>  paste(xnames, xsurnames)
[1] "Yiannis Papadopoulos" "Yiorgos Kitsos "  "Barbara Ioannou"
    "Aleka Tsapara" "Eugenia Grigoriadou"

>  paste(xsurnames,xnames, sep=', ')
[1] "Papadopoulos, Yiannis" "Kitsos, Yiorgos" "Ioannou, Barbara"
    "Tsapara, Aleka"  "Grigoriadou, Eugenia"
```

Pasting character vectors Examples

```
> paste(1:10, collapse=' ')
[1] "1 2 3 4 5 6 7 8 9 10"
> paste(1:10, collapse='')
[1] "12345678910"
> paste(1:10, collapse='+')
[1] "1+2+3+4+5+6+7+8+9+10"
> paste(xnames, collapse=', ')
[1] "Yiannis, Yiorgos, Barbara, Aleka, Eugenia "
> paste(xnames, xsurnames, collapse=',')
[1]  "Yiannis Papadopoulos,Yiorgos Kitsos,Barbara Ioannou,Aleka
     Tsapara,Eugenia Grigoriadou"
```

Pasting vectors - summary

paste()

```
> paste(`Mathematical',`Statistics')
[1] "Mathematical Statistics"
```

Argument "sep"

```
> paste(`3',`5',`8', sep="+")
[1] "3+5+8"
> paste(`Chapter',2, sep=" ")
[1] "Chapter 2"
> paste("Today is", date())
[1] "Today is Mon Jun 03 20:42:41 2013"
```

Nested pasted

```
> paste(paste(3,5, sep=` + '), 8, sep=` = ')
[1] "3 + 5 = 8"
```

Pasting elements of two vectors

```
> a<-c(`Kwstas', `Maria')
> b<-c(`Papadopoulos', `Kyriakou')
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"
```

Pasting vectors - summary

Pasting a character string with a vector

```
> paste("Chapter", 1:2, sep=" ")
[1] "Chapter 1" "Chapter 2"
```

Pasting two vectors of different length

```
> a<-c(`Kwstas', `Maria')
> b<-c(`Papadopoulos', `Kyriakou', `Anagnostou')
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"  "Kwstas Anagnostou"
```

The collapse argument

```
> a<-c(`Kwstas', `Maria')
> paste(a, collapse=",")
[1] "Kwstas,Maria"
> paste(1:10, collapse=`+')
[1] "1+2+3+4+5+6+7+8+9+10"
> b<-c(`Papadopoulos', `Kyriakou', `Anagnostou')
> paste(a, b, collapse=", ")
[1] "Kwstas Papadopoulos, Maria Kyriakou, Kwstas
Anagnostou"
```

Changing to capital or lower case letters

Changing to upper and lower case letters using the functions: toupper ( ) & tolower( )

```
> x
[1] "Statistics"  "Mathematics"
> tolower(x)
[1] "statistics"  "mathematics"
> toupper(x)
[1] "STATISTICS"  "MATHEMATICS"
```

Splitting characters

strsplit(x, split)

splits characters of a vector

x the vector to be split

split: a vector with the characters to which the separation will take place

```
> strsplit('Yiannis','i')
[[1]]
[1] "Y"   "ann" "s "
>  strsplit('Yiannis',NULL)
 [[1]]
 [1] "Y" "i" "a" "n" "n" "i" "s "
```

Splitting characters

```
x<-paste( xnames, xsurnames, sep=', ')
> x
[1] "Yiannis, Papadopoulos" "Yiorgos, Kitsos " " Barbara, Ioannou"
[4] "Aleka, Tsapara“  "Eugenia, Grigoriadou"
> strsplit(x,split=',')
[[1]]
[1] "Yiannis"       " Papadopoulos“

[[2]]
[1] "Yiorgos" " Kitsos“

[[3]]
[1] "Barbara"  " Ioannou“

[[4]]
[1] "Aleka"    " Tsapara“

[[5]]
[1] "Eugenia"      " Grigoriadou"
```

Splitting characters

Splitting characters function strsplit()

```
> x<-c("Statistics", "Mathematics")
> strsplit(x,split="a")
[[1]]
[1] "St"      "tistics"
[[2]]
[1] "M"    "them" "tics"
> strsplit(x, split='')
[[1]]
 [1] "S" "t" "a" "t" "i" "s" "t" "i" "c" "s"
[[2]]
 [1] "M" "a" "t" "h" "e" "m" "a" "t" "i" "c" "s"
> strsplit(x, split="th")
[[1]]
[1] "Statistics"
[[2]]
[1] "Ma"      "ematics"
```

Substrings

Extract substrings from a character using the function substr()

```
> substr("abcdef",2,4)
[1] "bcd"
> x<-c("Statistics", "Mathematics")
> substr(x,2,4)
[1] "tat" "ath"
```

Identifying substrings in characters using grep()

```
> countries<-c("Greece", "United States", "United Kingdom", "Italy",
    "France", "United Arab Emirates")
> grep("United", countries)
[1] 2 3 6
> grep("United", countries, value=TRUE)
[1] "United States"      "United Kingdom"      "United Arab Emirates"
```

Substitutions

Character substitution using sub() [only the first time]

```
> values<-c("1,700", "2,300")
> as.numeric(values)
[1] NA NA
Warning message:
NAs introduced by coercion
> as.numeric(gsub(",","",values))
[1] 1700 2300
```

Character substitution using gsub() [all times]

```
> values<-c("1,000,000", "2,000,000")
> sub(",","",values)
[1] "1000,000" "2000,000"
> gsub(",","",values)
[1] "1000000" "2000000"
```

Logical Vectors

```
> logical(3)
[1] FALSE FALSE FALSE

> as.logical(c(0:10))
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
   TRUE  TRUE  TRUE  TRUE
```

Logical Vectors

```
> x<-logical(10)
> x
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
    FALSE
> as.logical( 0:4 )
[1] FALSE  TRUE  TRUE  TRUE  TRUE
> as.logical( 0.000001:4 )
[1] TRUE TRUE TRUE TRUE
> as.logical( c('TRUE', 'FALSE') )
[1]  TRUE FALSE
> as.logical( c('TRUE', 'FALS') )
[1] TRUE   NA
> as.logical( c('TRUE', 'F') )
[1]  TRUE FALSE
> as.logical( c('TRUE', '0') )
[1] TRUE   NA
```

Logical Vectors

```
> 1:6<= 20
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> 1:6 <= 3
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
> 1:6 <= 1:6
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> 1:6 <= c(4,2,1,6,3,2)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
> 1:6 <= c(4,2,1,6,3)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
Warning message:
In 1:6 <= c(4, 2, 1, 6, 3) :
  longer object length is not a multiple of shorter object length
> 3*(1:6 <= c(4,2,1,6,3,2))
[1] 3 3 0 3 0 0
> sum(19:25)<= 20
[1] FALSE
```

Logical Vectors

```
> x<-  c(3, 9, 7, 1, 1, 5, 7, 2, 9, 1)
>y<-c(7, 9, 4, 1, 10, 4, 8, 10, 4, 8)
> x==y
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
> x!=y
 [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> !(x=y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
> !(x==y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
> (x==y)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> !(x==y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
```

Logical Vectors

```
> as.numeric(x ==1)
 [1] 0 0 0 1 1 0 0 0 0 1
> sum(as.numeric(x ==1))
[1] 3
> sum(as.numeric(x>3))
[1] 5
> sum(x>3)
[1] 5
> sum(!(x>3))
[1] 5
```

Logical Vectors

```
> (x<4)*2
 [1] 2 0 0 2 2 0 0 2 0 2
> x
 [1] 3 9 7 1 1 5 7 2 9 1
> x<4*2
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
> (x<4)*2
 [1] 2 0 0 2 2 0 0 2 0 2
> x<(4*2)
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
>
```

Logical Vectors

```
> y<- 10*(x>=5) + (-10)*(x<5)
> y
 [1] -10  10  10 -10 -10  10  10 -10  10 -10
> y<- 10*(x<3) + (-10)*(x>7) + 25*(x>=3)*(x<=7)
> y
 [1]  25 -10  25  10  10  25  25  10 -10  10
> (x>=3)*(x<=7)
 [1] 1 0 1 0 0 1 1 0 0 0
> (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
    FALSE FALSE
> (x>=3)&(x<=7) ==  (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
    FALSE FALSE
```

Logical Vectors

```
> (x>=3)*(x<=7)
 [1] 1 0 1 0 0 1 1 0 0 0
> (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
    FALSE
> (x>=3)&(x<=7) ==  (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
    FALSE
> (x>=3)&(x<=7) ==  as.logical((x>=3)*(x<=7))
 [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
    FALSE
> (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
    FALSE
> (x>=3)*(x<=7)
 [1] 1 0 1 0 0 1 1 0 0 0
```

Logical Vectors

```
> x1<-(x>=3)*(x<=7)
> x2<-(x>=3)&(x<=7)
> x1 == as.numeric(x2)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> sum(x1 == as.numeric(x2))
[1] 10
> sum(x1 == as.numeric(x2))==length(x1)
[1] TRUE
> all(x1 == as.numeric(x2))
[1] TRUE
> any(x1 == as.numeric(x2))
[1] TRUE
> sum(x1 == as.numeric(x2))>=1
[1] TRUE
```

Functions all & any

- all(x) => TRUE if all the elements of the vector x are TRUE
  Equivalent to sum(x)==length(x)

- any(x) => TRUE if at least one element of the vector x are TRUE
  Equivalent to sum(x)>0

```
> all( c(T,T,T,T,T) )
[1] TRUE
> all( c(T,T,T,T,F) )
[1] FALSE
> any( c(T,T,T,T,F) )
[1] TRUE
```

```
> any( c(T,F,F,F,F) )
[1] TRUE
> any( c(F,F,F,F,F) )
[1] FALSE
> all( c(F,F,F,F,F) )
[1] FALSE
```

Functions all & any

```
> test<-c(T,F,T,T,F,T,F)
> test
[1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE
> any(test)      # = TRUE - έστω και ένα να είναι TRUE τότε το
    αποτέλεσμα TRUE
[1] TRUE
> any( c(F,F,F))  # = FALSE- αφού όλα είναι false
[1] FALSE
> any( c(F,F,T))  # = TRUE - αφού ένα είναι T
[1] TRUE
> all( c(F,F,F))  # = FALSE - για να είναι T πρέπει όλα να είναι T
[1] FALSE
> all( c(F,F,T))  # = FALSE
[1] FALSE
```

Functions all & any

```
> all( c(F,T,T))  # = FALSE
[1] FALSE
> all( c(T,T,T))  # = TRUE εφόσον όλα είναι true
[1] TRUE
> sum( c(T,T,T))  # αθροισμα από true
[1] 3
> sum( c(T,T,T))==length(c(T,T,T))
# είναι το άθροισμα των true ίσο με το μήκος του διανύσματος δλδ είναι
    όλα true;
[1] TRUE
```

Functions all & any

```
> #
> # γενικότερη προσέγγιση μέσω διανύσματος
> #
> w2<-c(T,T,T)
> sum( w2)==length(w2)# ισοδύναμό του all
[1] TRUE
> all(w2)
[1] TRUE
> any(w2)
[1] TRUE
> sum( w2)>0 # ισοδύναμό του any
[1] TRUE
> !(sum( w2)==0) # ισοδύναμό του any
[1] TRUE
> !any(w2) # ολά false
[1] FALSE
```

Functions all & any

```
> math <- c(6, 8, 7, 6, 9, 3, 3, 6, 7, 4, 1, 2, 9, 9, 10, 6, 6, 4, 1, 7, 5, 7, 10, 4, 1, 5,
    10, 2, 1, 6, 9, 9, 10, 8, 10, 9, 8, 7, 6, 6, 8, 6, 9, 1, 4, 8, 3, 1, 5, 8, 10, 1, 4, 3, 4,
    6, 5, 1, 1, 2, 1, 9, 2, 5, 1, 1, 1, 5, 3, 7, 6, 10, 9, 2, 2, 10, 8, 5, 9, 9, 10, 4, 2, 10,
    5, 6, 3, 6, 8, 4, 1, 1, 6, 5, 8, 5, 9, 8, 5, 4)
> any(math<5) # υπάρχει έστω και ένας φοιτητής κάτω από τη βάση;
[1] TRUE
> any(math>8) # υπάρχει έστω και ένας φοιτητής πάνω από 8;
[1] TRUE
> all(math>=5) # περάσαν όλοι οι φοιτητές το μάθημα;
[1] FALSE
> table(math)  # δίνει τον πίνακα συχνοτήτων με τις βαθμολογίες
 1  2  3  4  5  6  7  8  9 10
15  7  6  9 11 14  6 10 12 10
```

Functions all & any

```
> yiannis <- TRUE
> #
> # πράξεις με all και λογικά διανύσματα
> all(math>=4) & yiannis
[1] FALSE
> all(math>=5) & yiannis
[1] FALSE
> all(math>=5) * yiannis
[1] 0
> all(math>=4) * yiannis
[1] 0
> all(math>=4) & !yiannis
[1] FALSE
>
```

Logical operators: & and |

```
> x <- round( rnorm(10, 70,10) ); x
 [1] 66 87 65 60 87 67 62 52 67 79
> t1<- (x<70);  t1
 [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
    FALSE
> !t1
 [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
    TRUE
> t2<- (x>70);  t2
 [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
    TRUE
> t1&t2
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
    FALSE
> t1|t2
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Combine logicals with other vectors

```
> c((x>=30),(x<=70))
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
    TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
[19]  TRUE FALSE
> c((x>=30),(x<=70), '1')
 [1] "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"
    "TRUE"  "TRUE"  "TRUE"  "FALSE" "TRUE"
[14] "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "TRUE"  "FALSE" "1"
> c((x>=30),(x<=70), 1)
 [1] 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1
> c((x>=30),(x<=70), 10)
 [1]  1  1  1  1  1  1  1  1  1  1  1  0  1  1  0  1  1  1  1  0 10
> c((x>=30),(x<=70), 10, '5fsd')
 [1] "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"
    "TRUE"  "TRUE"  "TRUE"  "FALSE" "TRUE"
[14] "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "TRUE"  "FALSE" "10"
    "5fsd"
```

The logical operator %in%

The logical operator %in% checks if an element is also an element of another object (e.g. vector)

```
> 3 %in% 1:10
[1] TRUE
> 15 %in% 1:10
[1] FALSE
> c(3,15)%in% 1:10
[1]  TRUE FALSE
> # is 'a' an element of vector 'letters'?
> 'a' %in% letters
[1] TRUE
>
```

The logical operator %in%

```
> z<-c("Grigoris", "Antonis", "Yiannis")
> z
[1] "Grigoris" "Antonis"  "Yiannis"
> NA %in% z
[1] FALSE
> z<-c(z,NA)
> NA %in% z
[1] TRUE
>
```

The logical operator %in%

```
> z<-c(10, 5, 9, 11, 11, 16, 7)
> z[3:7] %in% z
[1] TRUE TRUE TRUE TRUE TRUE
> all(z[3:7] %in% z) # checks if z[3:7] is a subvector of z
[1] TRUE
>
> 1:4 %in% z
[1] FALSE FALSE FALSE FALSE
```

- all(1:4 %in% z)
- [1] FALSE

```
> any(1:4 %in% z)
[1] FALSE
```

Factors

Categorical Variables:

```
> gender<-c('Male', 'Female', 'Male', 'Male', 'Female')
> gender
[1] "Male"   "Female" "Male"   "Male"   "Female"
> factor(gender)
[1] Male   Female Male   Male   Female
Levels: Female Male
> levels(factor(gender))
[1] "Female" "Male"
```

Ordinal Variables:

```
> opinion<-c('Low', 'Low', 'High', 'High', 'High',
'Medium')
> ordered(opinion, levels=c('Low', 'Medium', 'High'))
[1] Low    Low    High   High   High   Medium
Levels: Low < Medium < High
```

Factors

```
> gender<-c('Male', 'Female', 'Male', 'Male', 'Female'); gender
[1] "Male"   "Female" "Male"   "Male"   "Female"
> is.character(gender)
[1] TRUE
> x<-factor(gender); x
[1] Male   Female Male   Male   Female
Levels: Female Male
> is.character(x)
[1] FALSE
> is.numeric(x)
[1] FALSE
> is.factor(x)
[1] TRUE
```

```
> mode(x)
[1] "numeric"
> class(x)
[1] "factor"
```

Factors

```
> levels(x) <- NULL
Error in `levels<-.factor`(`*tmp*`, value = NULL) :
  number of levels differs
> levels(x) <- c(1,2)
> x
[1] 2 1 2 2 1
Levels: 1 2
```

Factors

```
> z<-c(1,2,3,4,1,1,1,2,3,4,1); factor(z)
 [1] 1 2 3 4 1 1 1 2 3 4 1
Levels: 1 2 3 4
> z0<-factor(z)
> levels(z0)
[1] "1" "2" "3" "4"
> levels(z0)<-paste('level',1:4,sep='')
> levels(z0)
[1] "level1" "level2" "level3" "level4“
> z0
 [1] level1 level2 level3 level4 level1 level1 level1 level2 level3 level4 level1
Levels: level1 level2 level3 level4
> factor(z, labels=c('l1', 'L2', 'l3', 'L4'))
 [1] l1 L2 l3 L4 l1 l1 l1 L2 l3 L4 l1
Levels: l1 L2 l3 L4
```

Factors

```
> opinion<-c('Low', 'Low', 'High', 'High', 'High', 'Medium')
> ordered(opinion, levels=c('Low', 'Medium', 'High'))
[1] Low    Low    High   High   High   Medium
Levels: Low < Medium < High
> ordered(rep(1:5,2), levels=c('Low', 'Medium', 'High'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: Low < Medium < High
> rep(1:5,2)
 [1] 1 2 3 4 5 1 2 3 4 5
> ordered(rep(1:5,2), levels=c('VL','L','M','H', 'VH'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=1:5, labels=c('VL','L','M','H', 'VH'))
 [1] VL L  M  H  VH VL L  M  H  VH
Levels: VL < L < M < H < VH
```

Factors

```
> ordered(rep(1:5,2), levels=1:5, labels=c('L','VL','M','H', 'VH'))
 [1] L  VL M  H  VH L  VL M  H  VH
Levels: L < VL < M < H < VH
> ordered(rep(1:5,2), levels=c(2,1,3:5), labels=c('VL','L','M','H', 'VH'))
 [1] L  VL M  H  VH L  VL M  H  VH
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=c('VL','L','M','H', 'VH'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=c(2,1,3:5), labels=c('VL','L','M','H', 'VH'))
 [1] L  VL M  H  VH L  VL M  H  VH
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=c(1:5), labels=c('VL','L','M','H', 'VH'))
 [1] VL L  M  H  VH VL L  M  H  VH
Levels: VL < L < M < H < VH
```

Two-dimensional matrices

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout.

Is a rectangular table of data of the same type (i.e. numeric, logical or character)

The function **matrix()** creates matrices.

```
> A<- matrix(c (1 ,3 ,2 ,4) , nrow=2 , ncol=2)
> A
[,1] [,2]
[1,] 1 2
[2,] 3 4
> A<- matrix(c (1 ,3 ,2 ,4) , 2 ,2) # equivalent syntax
```

Two-dimensional matrices

- The **nrow** and **ncol** arguments specify the dimension of the matrix.
- Only one dimension argument is needed, the other one will be automatically calculate by R.

```
A<- matrix(c (1 ,3 ,2 ,4) , ncol=2) # equivalent syntax
A<- matrix(c (1 ,3 ,2 ,4) , 2) # equivalent syntax
A<- matrix(c (1 ,3 ,2 ,4) , nrow=2) # equivalent syntax
```

```
> matrix( 1:6, 3 )
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> matrix( 1:6, ncol=3 )
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Examples on matrices

```
> matrix0 <- matrix(nrow=3, ncol=4)
> matrix0
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
 > x<-c(1:3,11:13,21:23,31:33)
 > x
 [1] 1 2 3 11 12 13 21 22 23 31 32 33
 > y<-matrix(x,ncol=3)
 > y
      [,1] [,2] [,3]
 [1,]    1   12   23
 [2,]    2   13   31
 [3,]    3   21   32
 [4,]   11   22   33
```

```
> y<-matrix(x,nrow=4)
> y
     [,1] [,2] [,3]
[1,]    1   12   23
[2,]    2   13   31
[3,]    3   21   32
[4,]   11   22   33
```

Entering the elements of a Matrix by rows

- In matrices, elements are introduced by columns i.e. first the first column, then the second etc.
- The **byrow** argument specifies how the matrix is to be filled.
- The default value for **byrow** is **FALSE** which means that by default the matrix will be filled column by column.
- If we want to fill the matrix row by row, we have to write in the function matrix() the argument `byrow=T`.

Entering the elements of a Matrix by rows

Example

```
> y<-matrix(x,ncol=3)
> y
     [,1] [,2] [,3]
[1,]    1   12   23
[2,]    2   13   31
[3,]    3   21   32
[4,]   11   22   33
```

```
> y<-matrix(x,ncol=3, byrow=T)
> y
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   11   12   13
[3,]   21   22   23
[4,]   31   32   33
```

Returning back to vectors

```
> x<-matrix(1:12,3)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> as.vector(x)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
> x<-matrix(1:12,3, byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> as.vector(x)
 [1]  1  5  9  2  6 10  3  7 11  4  8 12
> t(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> as.vector(t(x))
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Single column or row matrices

- To enter a $p \times 1$ column vector, simply enter a $p \times 1$ matrix.

```
> y<- matrix(c (1 ,2 ,3 ,4) ,4 ,1)
> y
[,1]
[1,] 1
[2,] 2
[3,] 3
[4,] 4
```

- Row vectors are, likewise, entered as $1 \times q$ matrices.

```
> x<- matrix(c (1 ,2 ,3 ,4) ,1 ,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

Main characteristics of matrices

```
> x<-c(1:3,11:13,21:23,31:33); y<-matrix(x,3); y
     [,1] [,2] [,3] [,4]
[1,]    1   11   21   31
[2,]    2   12   22   32
[3,]    3   13   23   33
```

Length

```
> length(y)
[1] 12
```

Dimension

```
> dim(y)
[1] 3 4
```

names, dimnames, rownames, colnames

```
names(y)
dimnames(y)
colnames(y)
rownames(y)
```

The dimension of a matrix

- The dim() function is used to list the dimensions of a matrix. The outcome is a vector with two elements.

- With command dim() we can create matrices from vectors. So, if in a vector add the dimension, then the elements of the vector will create a matrix with the same dimension, filling the column of the matrix consecutively.

- When the matrix we create has more elements than the vector, then R is recycling the elements automatically and print a warning on the screen.

The dimension of a matrix

```
> x<-1:6
> dim(x)<-c(3,2);  x
     [,1]  [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(x)<-c(2,3); x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(x)<-c(1,6);  x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
```

The dimension of a matrix

```
> matrix0 <- matrix( 1:20, 4,5 )
> dim(matrix0)     # vector with two elements - number of rows and number of columns
[1] 4 5
> dim(matrix0)[1]  # number of rows
[1] 4
> dim(matrix0)[2]  # number of columns
[1] 5
> nrow(matrix0) # number of rows
[1] 4
> ncol(matrix0) # number of columns
[1] 5
```

Dimension names of a matrix

- We can assign names to the rows and columns of the matrix with the utilization of dimnames() command.
- By this way we can access the elements by names.
- It is needed to specify the names for the rows and columns in that order in a list.
- The dimnames() of a matrix can be a list of the same length as the dimension of the matrix or NULL (no name). The components of the list are either a character vector with positive length of the appropriate dimension of the matrix or NULL. The list can have names.

Dimension names of a matrix

- If we want to delete a name from a specific row or column, we set it as NULL.
- The command dimnames can be included as a parameter in the matrix() command, matrix(..., dimnames = list, ...).

- If we want to name only the rows or the columns of a matrix we can use the rownames() or colnames() commands, respectively. The dimnames() command operates on both rows and columns at once.

Dimension names of a matrix

```
> x<-matrix(c(2.3,1.4,0,-12,5.27,6),ncol=3)
> dimnames(x) <- list( c("Row 1","Row 2"),c("Col 1", "Col 2","Col 3")); x
      Col 1  Col 2  Col 3
Row 1   2.3     0     5.27
Row 2   1.4   -12     6.00
> dimnames(x)<-list(NULL,c("Col 1","Col 2","Col 3")); x
   Col 1  Col 2 Col 3
[1,]   2.3    0    5.27
[2,]   1.4  -12    6.00
> dimnames(x)<-list( c("Row 1","Row 2"), NULL) ; x
     [,1] [,2] [,3]
Row 1  2.3    0 5.27
Row 2  1.4  -12 6.00
> dimnames(x)<-NULL; x
    [,1] [,2] [,3]
[1,]  2.3    0 5.27
[2,]  1.4  -12 6.00
```

Dimension names of a matrix

```
> colnames(x) <- paste( 'Col', 1:3, sep='' )
> dimnames(x)
[[1]]
NULL

[[2]]
[1] "Col1" "Col2" "Col3“

> x
    Col1 Col2 Col3
[1,]  2.3    0 5.27
[2,]  1.4  -12 6.00
```

```
> rownames(x) <- paste( 'R', 1:2, sep='' )
> dimnames(x)
[[1]]
[1] "R1" "R2“

[[2]]
[1] "Col1" "Col2" "Col3“

> x
   Col1 Col2 Col3
R1  2.3    0 5.27
R2  1.4  -12 6.00
```

Dimension names of a matrix

```
> names(dimnames(x))<-c('X', 'Y')
> x
   Y
X    Col1 Col2 Col3
  R1  2.3    0 5.27
  R2  1.4  -12 6.00
> dimnames(x)
$X
[1] "R1" "R2"

$Y
[1] "Col1" "Col2" "Col3"
```

Extracting individual elements of a matrix

- Individual elements of a matrix are referred to by their subscripts
- For example, consider a matrix A given below and we want the first element.
- To extract element $A_{1;1}$, we simply request $\mathrm{A}[1,1]$.

```
> A<- matrix( -5:6, ncol=4, nrow=3 );  A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]   -4   -1    2    5
[3,]   -3    0    3    6
> A[1,1]
[1] -5
```

```
> A[3,2]
[1] 0
> A[3,2]<-100000
> A
     [,1]   [,2] [,3] [,4]
[1,]   -5     -2    1    4
[2,]   -4     -1    2    5
[3,]   -3 100000    3    6
```

Extracting a row of a matrix

- To get an entire row of a matrix, you name the row and leave out the column.
- For example, in the matrix A below, to get the first row, just enter `A[1,]`.

```
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]   -4   -1    2    5
[3,]   -3    0    3    6
> A[1,]
[1] -5 -2  1  4
```

```
> A[2,]
[1] -4 -1  2  5
> is.vector(A[2,])
[1] TRUE
> is.matrix(A[2,])
[1] FALSE
```

Changing the values of a row of a matrix

```
> A[2,]
[1] -4 -1  2  5
> A[2,]<-0
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]    0    0    0    0
[3,]   -3    0    3    6
> A[2,] <- 1:4
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]    1    2    3    4
[3,]   -3    0    3    6
```

Extracting a Column of a Matrix

- To get an entire column of a matrix, you name the column and leave out the row.
- For example, in the matrix A below, to get the first column, just enter `A[,1]`.

```
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]   -4   -1    2    5
[3,]   -3    0    3    6
> A[,1]
[1] -5 -4 -3
```

```
> A[,4]
[1] 4 5 6
> is.vector(A[,4])
[1] TRUE
> is.matrix(A[,4])
[1] FALSE
```

Changing the values of a Column of a Matrix

```
> A[,4]
[1] 4 4 6
> A[,4]<-c(0,10,100)
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    0
[2,]    1    2    3   10
[3,]   -3    0    3  100
```

Extracting sub-matrices using numeric vectors

```
> x<-matrix(1:30, ncol=6)
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> index1<-c(2,5,4)
> index2<-c(1:2,6,2)
> x[index1,index2]
     [,1] [,2] [,3] [,4]
[1,]    2    7   27    7
[2,]    5   10   30   10
[3,]    4    9   29    9
```

Extracting sub-matrices using logical vectors

```
> x<-matrix(1:30, ncol=6)
> sel1<- c(F,F,T,T,F)
> sel2<-c(T,F,F,T,F,T)
> x[sel1,sel2]
     [,1] [,2] [,3]
[1,]    3   18   28
[2,]    4   19   29
> z1<-rnorm(5)
> z2<-rnorm(6)
> x[z1>0, z2<0]
     [,1] [,2] [,3]
[1,]    1   21   26
[2,]    2   22   27
[3,]    5   25   30
```

Removing rows and/or columns

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

```
> x[,-c(3,4)]
 [,1] [,2]
[1,]    1    2
[2,]    5    6
[3,]    9   10
> x[-c(1:2),-c(3,4)]
[1]  9 10
```

Submatrices using combinations of different types of syntax

```
> x<-matrix(1:30 ncol=6)
> x[ c(1,3), -2]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1   11   16   21   26
[2,]    3   13   18   23   28
> x[ c(T,T,F,F,F), c(3,1)]
     [,1] [,2]
[1,]   11    1
[2,]   12    2
```

```
> index<-1:6
> x[ -3, index>3]
     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   19   24   29
[4,]   20   25   30
```

Extracting elements under logical conditions

```
> x[ x>40 ]
integer(0)
> x[ x>20 ]
 [1] 21 22 23 24 25 26 27 28 29 30
> x[x>20]<-0
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16    0    0
[2,]    2    7   12   17    0    0
[3,]    3    8   13   18    0    0
[4,]    4    9   14   19    0    0
[5,]    5   10   15   20    0    0
```

Changing values of elements selected by using logical conditions

```
> z[z==4]
[1] 4 4 4 4 4
> z[z==4]<-0
> z
     [,1] [,2] [,3] [,4]
[1,]    0    2    7    5
[2,]    5    3    0    7
[3,]    5    5    7    7
[4,]    0    2    0    5
[5,]    0    3    7    6
```

```
> z[z==3]
[1] 3 3
> z[z==3]<-c(333,333333)
> z
     [,1]   [,2] [,3] [,4]
[1,]    0      2    7    5
[2,]    5    333    0    7
[3,]    5      5    7    7
[4,]    0      2    0    5
[5,]    0 333333    7    6
```

Which function in matrices

```
> x<-c(19, 14, 15, 14, 20, 20, 24, 19, 20, 15, 28, 17, 18, 21, 31,
+ 16, 17, 22, 18, 14, 13, 26, 20, 20, 21, 24, 13, 16, 27, 17)
> x<-matrix(x, ncol=6)
> z<-   x>20
> which(z)
[1]  7 11 14 15 18 22 25 26 29
# Δίνει τη θέση στο διάνυσμα όπου το z είναι TRUE (το αποτέλεσμα είναι
    ένα διάνυσμα)
> which(z, arr.ind=T)
     row col
[1,]   2   2
[2,]   1   3
[3,]   4   3
[4,]   5   3
[5,]   3   4
[6,]   2   5
[7,]   5   5
[8,]   1   6
[9,]   4   6
```

Δίνει τη θέση στον πίνακα όπου το z είναι TRUE (το αποτέλεσμα είναι ένα διάνυσμα)

Which function in matrices

```
> z
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] FALSE FALSE  TRUE FALSE FALSE  TRUE
[2,] FALSE  TRUE FALSE FALSE  TRUE FALSE
[3,] FALSE FALSE FALSE  TRUE FALSE FALSE
[4,] FALSE FALSE  TRUE FALSE FALSE  TRUE
[5,] FALSE FALSE  TRUE FALSE  TRUE FALSE
> dim(z)
[1] 5 6
> which(z)
[1]  7 11 14 15 18 22 25 26 29
```

```
> which(z, arr.ind=T)
     row col
[1,]   2   2
[2,]   1   3
[3,]   4   3
[4,]   5   3
[5,]   3   4
[6,]   2   5
[7,]   5   5
[8,]   1   6
[9,]   4   6
```

Creating matrices with same rows/columns

```
> z<-c(4,3,5,1,4,7,23)
> k<-10
> matrix(z,length(z),k)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    4    4    4    4    4    4    4    4    4    4
[2,]    3    3    3    3    3    3    3    3    3    3
[3,]    5    5    5    5    5    5    5    5    5    5
[4,]    1    1    1    1    1    1    1    1    1    1
[5,]    4    4    4    4    4    4    4    4    4    4
[6,]    7    7    7    7    7    7    7    7    7    7
[7,]   23   23   23   23   23   23   23   23   23   23
> k<-3
> matrix(z,k, length(z),byrow=T)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    4    3    5    1    4    7   23
[2,]    4    3    5    1    4    7   23
[3,]    4    3    5    1    4    7   23
```

Combining vector and matrices

- Occasionally, we need to build up matrices from smaller parts
- You can combine several matrices with the same number of columns by joining them as rows, using the `rbind()` command
- In similar fashion, you can combine several matrices with the same number of rows by joining them as columns, using the `cbind()` command.
- The `cbind()` and `rbind()` functions are used to append matrices together.

Combining vector and matrices

```
> x1<-1:5
> x2<-6:10
> cbind(x1,x2)
     x1 x2
[1,]  1  6
[2,]  2  7
[3,]  3  8
[4,]  4  9
[5,]  5 10
> rbind(x1,x2)
     [,1] [,2] [,3] [,4] [,5]
x1    1    2    3    4    5
x2    6    7    8    9   10
```

Combining vector and matrices

```
> z<-cbind(math,gender); z
      math gender
 [1,]    6      2
 [2,]    7      1
 [3,]    7      1
 [4,]   10      1
 [5,]    7      1
 [6,]    7      2
 [7,]    9      2
 [8,]    6      1
 [9,]   10      2
[10,]    9      1
> is.matrix(z)
[1] TRUE
> dim(z)
[1] 10  2
```

```
> z<-rbind(math,gender)
> is.matrix(z)
[1] TRUE
> dim(z)
[1]  2 10
> z
       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
math      6    7    7   10    7    7    9    6   10     9
gender    2    1    1    1    1    2    2    1    2     1
>
```

Combining vector and matrices

```
> x<-c(5,3,4,7)
> y<-c(-2,-1,0,4)
> cbind(x,y)
     x  y
[1,] 5 -2
[2,] 3 -1
[3,] 4  0
[4,] 7  4

> cbind(y,x)
     y x
[1,] -2 5
[2,] -1 3
[3,]  0 4
[4,]  4 7
```

```
> rbind(x,y,x)
  [,1] [,2] [,3] [,4]
x    5    3    4    7
y   -2   -1    0    4
x    5    3    4    7
```

Combining vector and matrices

```
> x<-c(5,3,4)
> y<-c(-2,-1,0,4,-2)
> cbind(x,y)
     x  y
[1,] 5 -2
[2,] 3 -1
[3,] 4  0
[4,] 5  4
[5,] 3 -2
Warning message:
In cbind(x, y) :
  number of rows of result is not a multiple of vector length (arg 1)
```

```
> cbind(y,x)
      y x
[1,] -2 5
[2,] -1 3
[3,]  0 4
[4,]  4 5
[5,] -2 3
Warning message:
In cbind(y, x) :
  number of rows of result is not a multiple of vector length (arg 2)
```

Diagonal Matrices

BE CAREFUL: The diag(x) command changes behavior according to the input

- **x=number** => Identity matrix of dimensions x times x
- x**=vector** => Diagonal matrix with the elements of x placed on the diagonal.
- **x=matrix** => vector with the diagonal elements of x

Diagonal Matrices

- **x=number** => Identity matrix of dimensions x times x

```
> diag(5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

Diagonal Matrices

- x=**vector** => Diagonal matrix with the elements of x placed on the diagonal.

```
> diag(1:5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
```

Diagonal Matrices

- **x=matrix** => vector with the diagonal elements of x

```
x<-matrix(1:30, ncol=6)
> diag( x )
[1]  1  7 13 19 25
```

diag(diag(x)) => ?

Elementwise numerical operations for matrices

Arithmetic Operators & matrices: +, - , *, /, %/%, %%

Can be used with matrices of the same dimension

Each operation is applied in each element corresponding to the same row and column

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4)
> x2<-matrix(1:12, ncol=3,nro=4)
> x1
     [,1] [,2] [,3]
[1,]    4    4    2
[2,]    3    7    7
[3,]    4    5    6
[4,]    6    6    7
> x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x1+x2
     [,1] [,2] [,3]
[1,]    5    9   11
[2,]    5   13   17
[3,]    7   12   17
[4,]   10   14   19
```

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4)
> x2<-matrix(1:12, ncol=3,nro=4)
> x1
     [,1] [,2] [,3]
[1,]    4    4    2
[2,]    3    7    7
[3,]    4    5    6
[4,]    6    6    7
> x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

```
> x1*x2
     [,1] [,2] [,3]
[1,]    4   20   18
[2,]    6   42   70
[3,]   12   35   66
[4,]   24   48   84
> x1/x2
        [,1]      [,2]      [,3]
[1,] 4.000000 0.8000000 0.2222222
[2,] 1.500000 1.1666667 0.7000000
[3,] 1.333333 0.7142857 0.5454545
[4,] 1.500000 0.7500000 0.5833333
```

Operators and functions for matrices

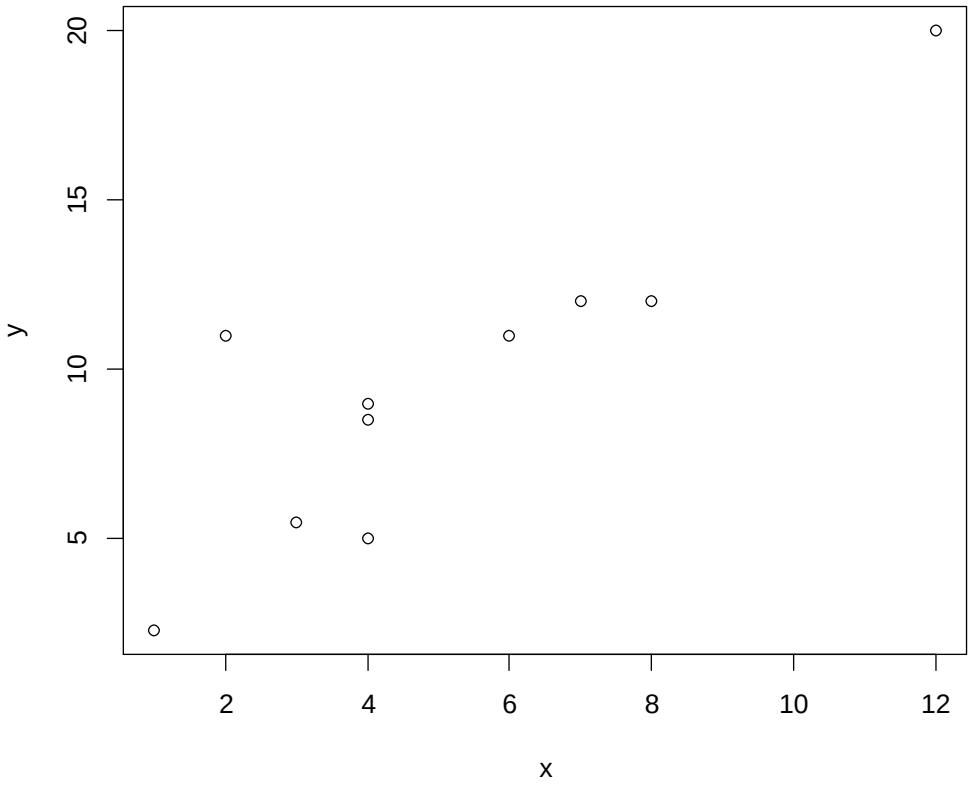
Operator	Description
%*%	Multiplication of Matrices (Πολλαπλασιασμός πινάκων)
t()	Transpose of a Matrix (ανάστροφος)
solve()	Inverse of a Matrix (αντίστροφος)
det() determinant()	Determinant of a matrix (ορίζουσα)

Operators and functions for matrices

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(x)
[1] 3 2
> y<-matrix(c(0,1,1,1), ncol=2)
> y
     [,1] [,2]
[1,]    0    1
[2,]    1    1
```

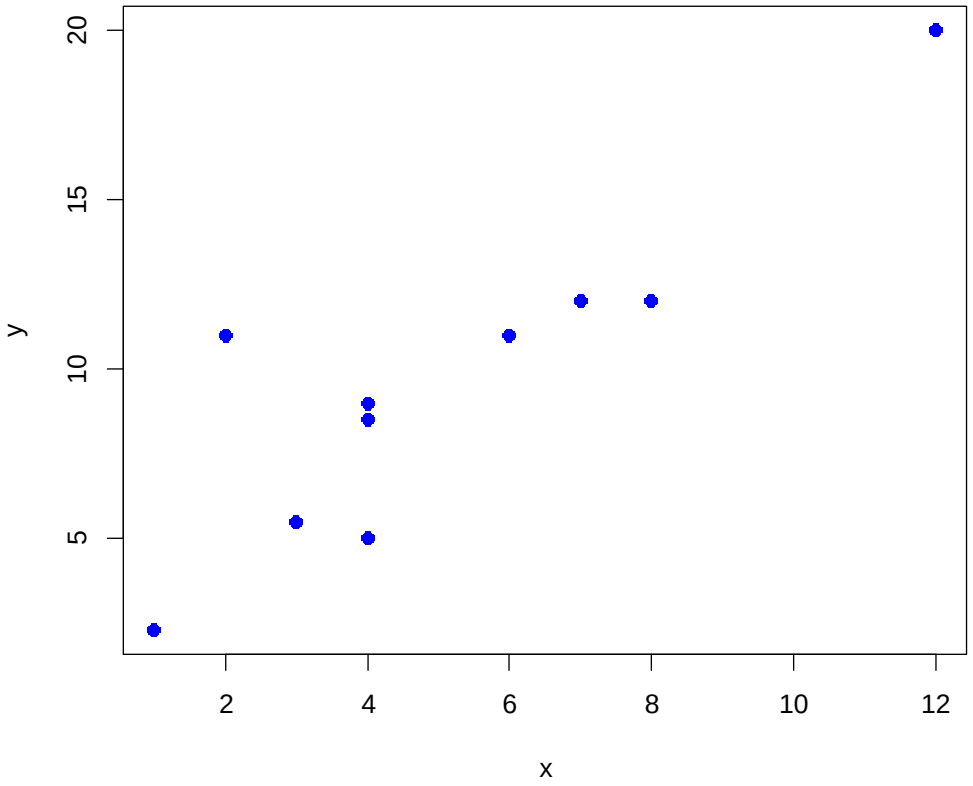
```
> x%*%y
     [,1] [,2]
[1,]    4    5
[2,]    5    7
[3,]    6    9
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> solve(y)
     [,1] [,2]
[1,]   -1    1
[2,]    1    0
```

Best fitted line using matrices



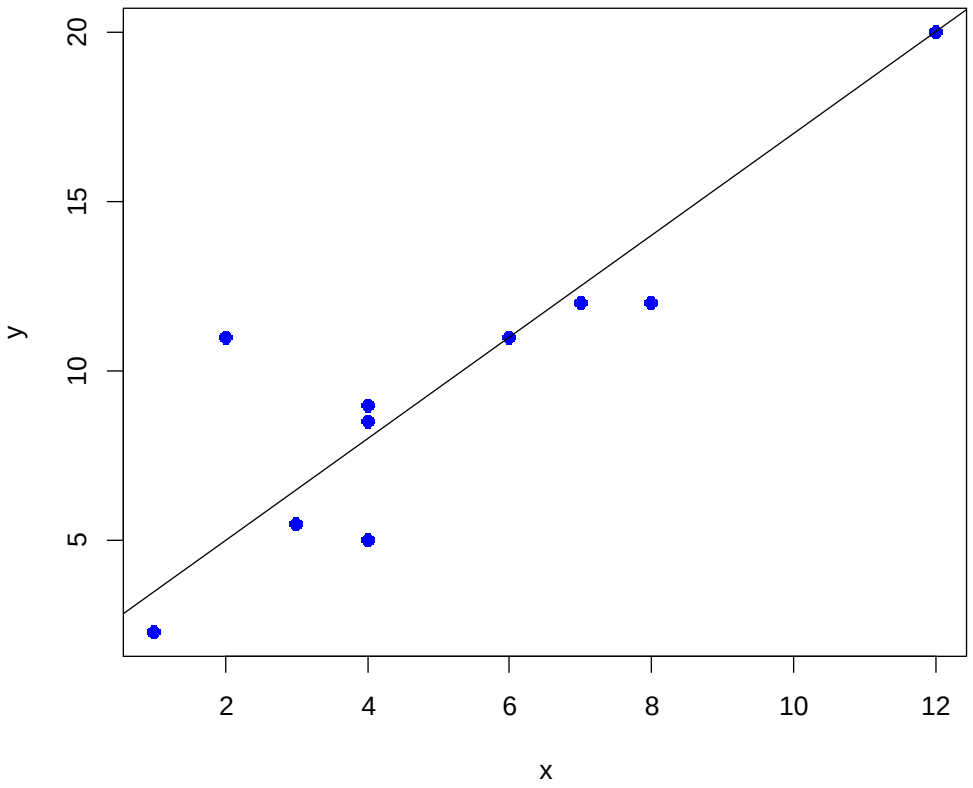

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
> plot(x,y)
# διάγραμμα σημείων
```

Best fitted line using matrices



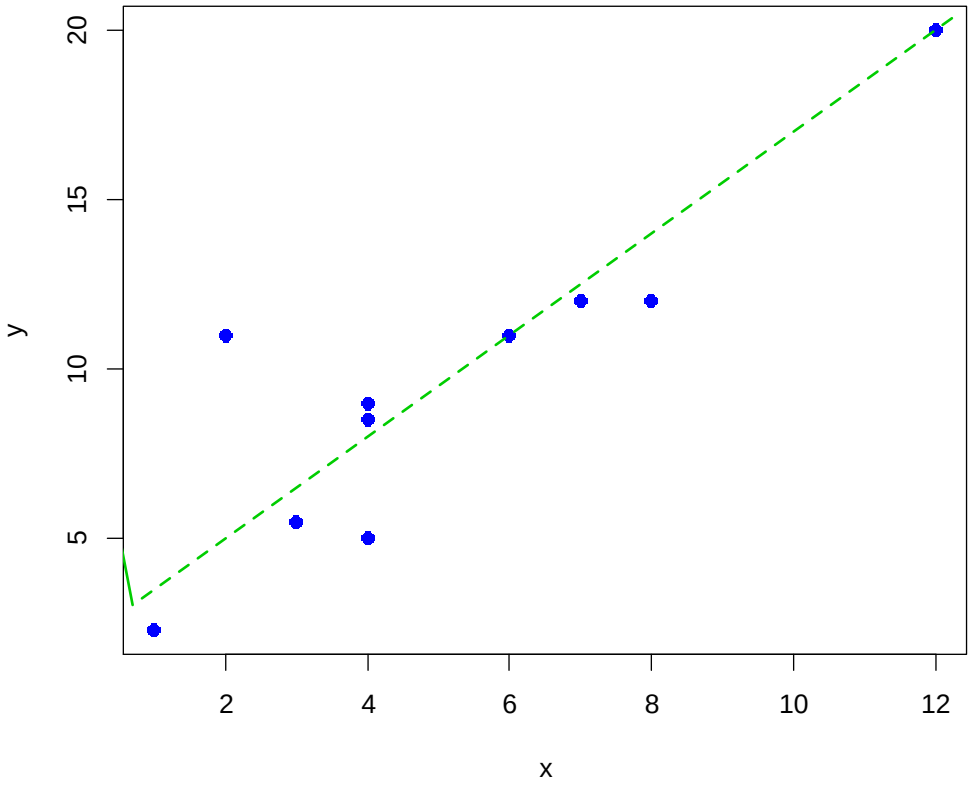

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')

# pch: type of point
# cex: size of points
# col: color
```

Best fitted line using matrices



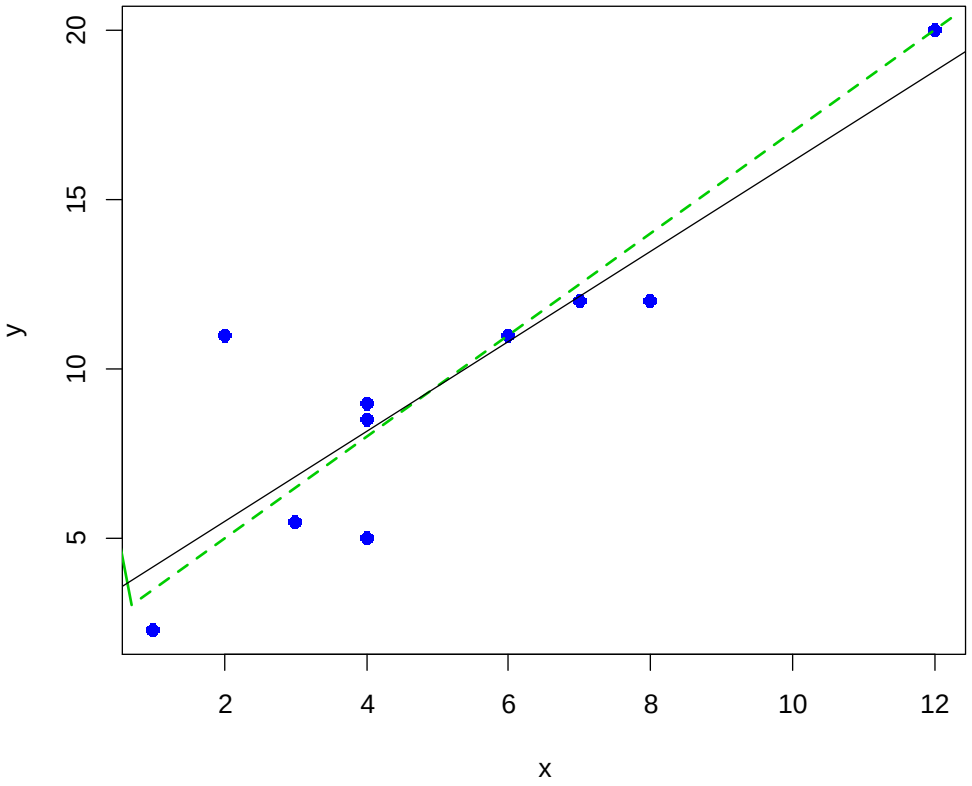

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5)
# adds the line y=a+bx
```

Best fitted line using matrices



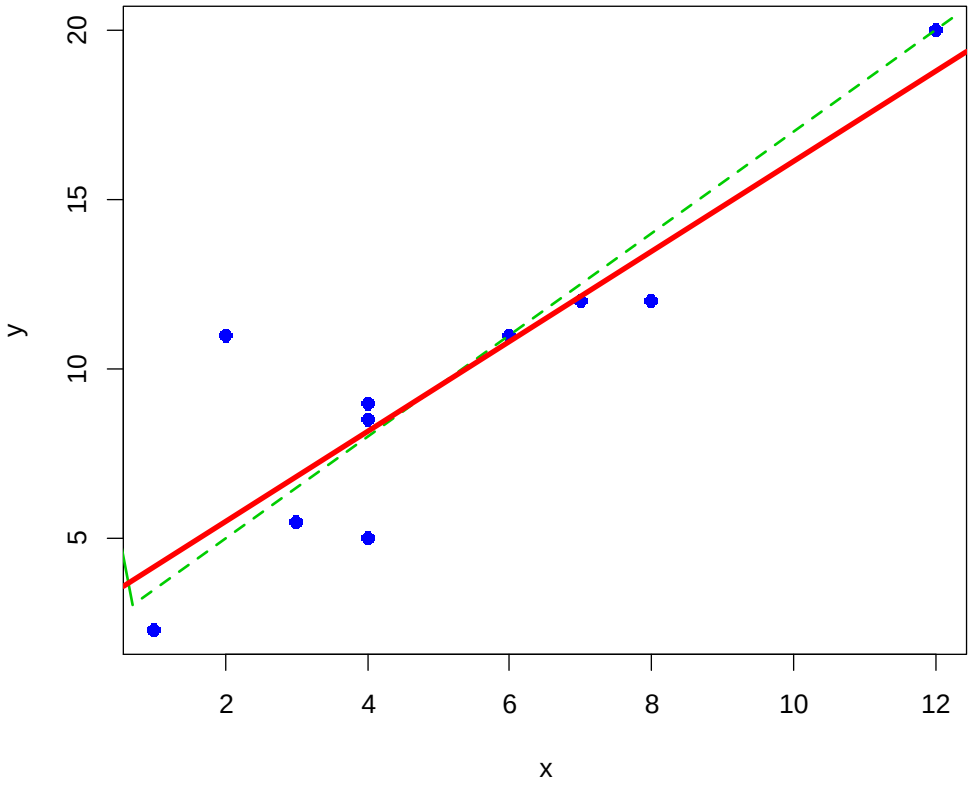

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
```

Best fitted line using matrices




```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
> lm(y~x)
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)          x
      2.876       1.324
> abline(lm(y~x)) # add lse line
```

Best fitted line using matrices

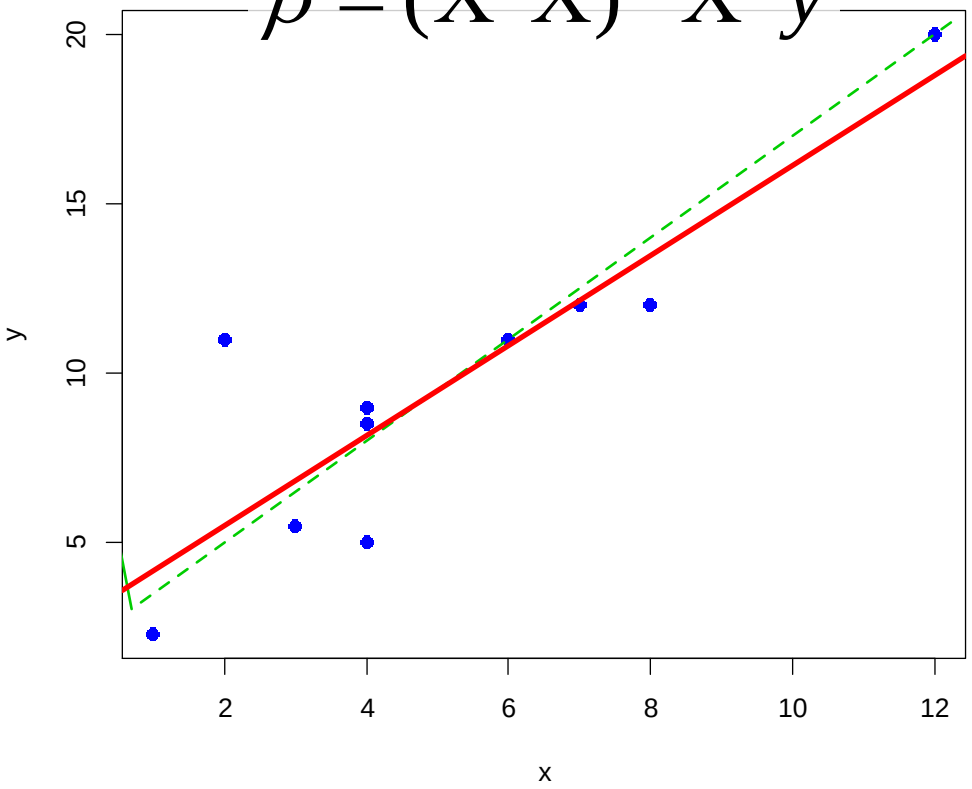



```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
abline(lm(y~x), col='red',
    lwd=3.5)
```

Best fitted line using matrices

We can calculate the least square error coefficients of the regression line by using the formula:

$$\hat{\beta} = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}y$$



y is the vector of length n with all values of y

X is a nx2 matrix with the first column equal to one and the second equal to the values of x

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}$$

Then

$$y = X\beta$$

With elements

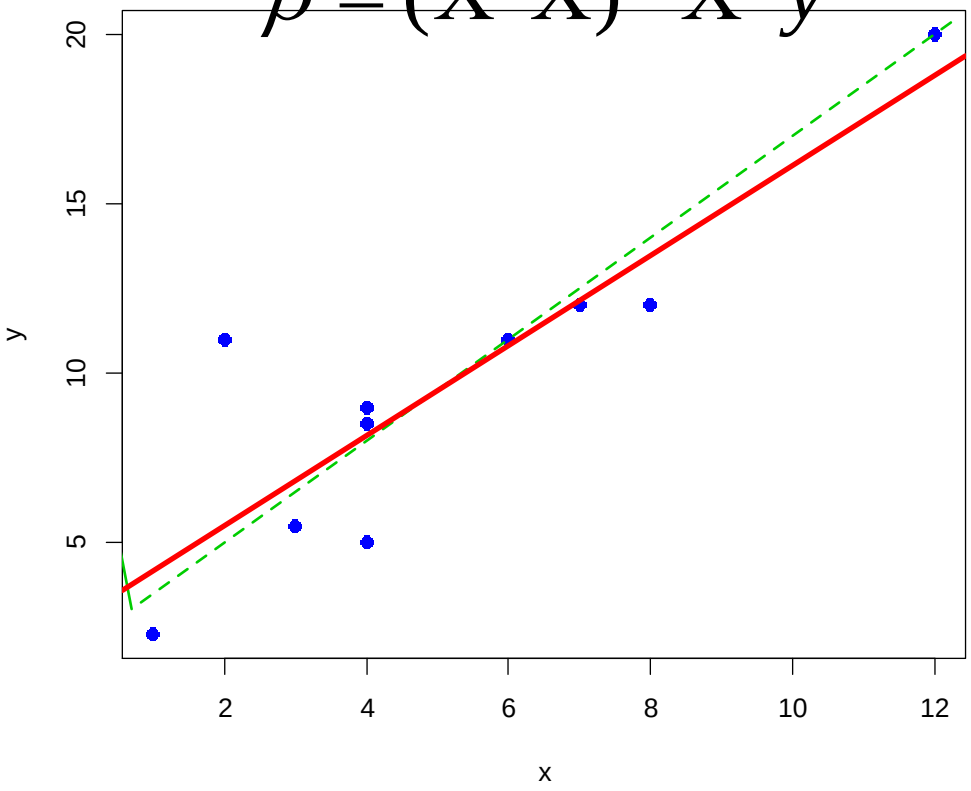
$$y_i = \beta_1 + \beta_2 x_i$$

Therefore β_1 is the intercept (a in abline) and β_2 is the slope (b in abline)

Best fitted line using matrices

We can calculate the least square error coefficients of the regression line by using the formula:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$




```
> x
 [1]  4  6  7 12  4  3  2  4  1  8
> X <- cbind(1,x)
> X
      x
 [1,] 1  4
 [2,] 1  6
 [3,] 1  7
 [4,] 1 12
 [5,] 1  4
 [6,] 1  3
 [7,] 1  2
 [8,] 1  4
 [9,] 1  1
[10,] 1  8
> as.vector(solve(t(X)%*%X)%*%t(X)%*%y)
[1] 2.876396 1.324236
> lm(y~x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      2.876        1.324
```

Logical matrices

```
> A <- round(matrix( rnorm(16), 4 ),1)
> A
     [,1] [,2] [,3] [,4]
[1,]  1.0  0.3  0.1 -1.7
[2,]  0.8  0.3  1.5 -1.0
[3,] -0.9 -0.6 -0.1  1.0
[4,] -0.3  0.8  0.5 -1.6
> logicA <- A>0
> logicA
      [,1]  [,2]  [,3]  [,4]
[1,]  TRUE  TRUE  TRUE FALSE
[2,]  TRUE  TRUE  TRUE FALSE
[3,] FALSE FALSE FALSE  TRUE
[4,] FALSE  TRUE  TRUE FALSE
>
```

```
> mode(A);class(A)
[1] "numeric"
[1] "matrix"
> mode(logicA);class(logicA)
[1] "logical"
[1] "matrix"
>
> logicB <- matrix( c(T,T,F,T), 2 )
> logicB
     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
```

Character matrices

```
> letters[1:12]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
> charA <- matrix( letters[1:12],3 )
> charA
     [,1] [,2] [,3] [,4]
[1,] "a"  "d"  "g"  "j"
[2,] "b"  "e"  "h"  "k"
[3,] "c"  "f"  "i"  "l"
>
> mode(charA); class(charA)
[1] "character"
[1] "matrix"
>
```

```
> logicB
      [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
> as.character(logicB)
[1] "TRUE"  "TRUE"  "FALSE" "TRUE"
> charB <- matrix( as.character(logicB),2 )
> charB
     [,1]   [,2]
[1,] "TRUE" "FALSE"
[2,] "TRUE" "TRUE
```

Sorting in matrices

```
> onoma <- c( 'Giorgos', 'Yiannis','Maria', 'Eleni' )
> age <- c( 31,21,22,17 )
> male <- c( 1,1,0,0 )
> A <- cbind(age,male)
> A
     age male
[1,]  31    1
[2,]  21    1
[3,]  22    0
[4,]  17    0
> rownames( A ) <- onoma
> A
        age male
Giorgos  31    1
Yiannis  21    1
Maria    22    0
Eleni    17    0
```

```
> B <- cbind(sort(A[,1]), A[,2])
> B
        [,1] [,2]
Eleni     17    1
Yiannis   21    1
Maria     22    0
Giorgos   31    0
```

```
> B <- cbind(sort(A[,1]), sort(A[,2]))
> B
        [,1] [,2]
Eleni     17    0
Yiannis   21    0
Maria     22    1
Giorgos   31    1
```

```
> A[order(A[,1]),]
        age male
Eleni    17    0
Yiannis  21    1
Maria    22    0
Giorgos  31    1
```

Sorting in matrices

```
> order(A[,1])
[1] 4 2 3 1
> i <- order(A[,1])
> A[i,]
         age male
Eleni     17    0
Yiannis   21    1
Maria     22    0
Giorgos   31    1
```

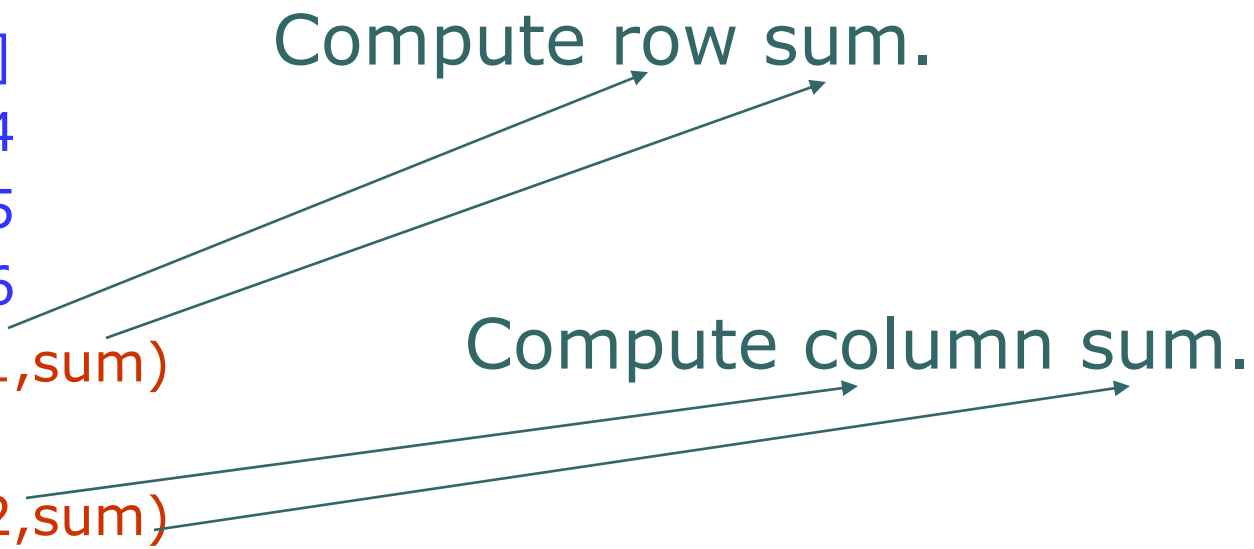
```
# Ordering by names
> rownames(A)
[1] "Giorgos" "Yiannis" "Maria"   "Eleni"
> sort(rownames(A))
[1] "Eleni"   "Giorgos" "Maria"   "Yiannis"
> i <- order(rownames(A))
> i
[1] 4 1 3 2
> A[i,]
         age male
Eleni     17    0
Giorgos   31    1
Maria     22    0
Yiannis   21    1
> A[i,1]
 Eleni Giorgos   Maria Yiannis
    17     31     22     21
> A[i,2]
 Eleni Giorgos   Maria Yiannis
    0      1      0      1
```

Sorting in matrices

```
> B<-t(A)
> B
     Giorgos Yiannis Maria Eleni
age       31       21    22    17
male       1        1     0     0
> i<-order( B[1,] )
> i
[1] 4 2 3 1
> B[,i]
     Eleni Yiannis Maria Giorgos
age     17       21    22       31
male     0        1     0        1
```

The apply() function

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(x,1,sum)
[1] 5 7 9
> apply(x,2,sum)
[1]  6 15
```



The apply() function

apply (X, MARGIN, FUN, ...)

- X is an array, including a matrix.
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

The apply() function

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
[1] 0.04868268 0.35743615 -0.09104379
[4] -0.05381370 -0.16552070 -0.18192493
[7] 0.10285727 0.36519270 0.14898850
[10] 0.26767260

> apply(x, 1, sum)
[1] -1.94843314 2.60601195 1.51772391
[4] -2.80386816 3.73728682 -1.69371360
[7] 0.02359932 3.91874808 -2.39902859
[10] 0.48685925 -1.77576824 -3.34016277
[13] 4.04101009 0.46515429 1.83687755
[16] 4.36744690 2.21993789 2.60983764
[19] -1.48607630 3.58709251
```

simulates from standard normal (more later)

The apply() function

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
            [,1]       [,2]       [,3]       [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686  ……….
75%  0.9258157  0.07065724  0.3050407 -0.06585436
```

Col/row sums & means

- rowSums (x) = apply(x, 1, sum)
- rowMeans (x) = apply(x, 1, mean)
- colSums (x) = apply(x, 2, sum)
- colMeans (x) = apply(x, 2, mean)

These shortcut functions are much faster, but you won't notice unless you're using a large matrix.

Arrays

- Arrays are matrices with 3 or more dimensions.
- To create them we use the function array().
- The dimension of the array is given by the parameter dim.
- For example if dim=c(2,3,4), we will have a 3 dimensional array of dimension 2×3×4.

Arrays

```
> X<-array(c(1:12,36:48),dim=c(2,3,4))
> X
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   36   38   40
[2,]   37   39   41

, , 4

     [,1] [,2] [,3]
[1,]   42   44   46
[2,]   43   45   47
```

Attributes of arrays

```
> length(X)
[1] 24
> mode(X)
[1] "numeric"
> class(X)
[1] "array"
> dim(X)
[1] 2 3 4
> dimnames(X)
NULL
```

Array dim & dimnames

These functions are used exactly in the same way as in matrices

dimnames have now length equal to the number of dimension (i.e. it is a list with 3 elements for X in our example and each element is a character vector with length equal to the corresponding elements of dim

Array dim & dimnames

```
> dim(X)
[1] 2 3 4
>
> dimnames(X)
NULL
> dimnames(X)[[1]] <- paste('row',1:2,sep='')
> X
, , 1
     [,1] [,2] [,3]
row1    1    3    5
row2    2    4    6

, , 2
     [,1] [,2] [,3]
row1    7    9   11
row2    8   10   12

, , 3
     [,1] [,2] [,3]
row1   36   38   40
row2   37   39   41

, , 4
     [,1] [,2] [,3]
row1   42   44   46
row2   43   45   47

> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
NULL

[[3]]
NULL
```

Array dim & dimnames

```
> dimnames(X)[[2]] <- paste('col',1:3,sep='')
> X
, , 1
     col1 col2 col3
row1    1    3    5
row2    2    4    6
 ……….
```

```
> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
NULL
```

```
> dimnames(X)[[3]] <- paste('tab',1:4,sep='')
> X
, , tab1

     col1 col2 col3
row1    1    3    5
row2    2    4    6
……….
```

```
> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

Array dim & dimnames

```
> attributes(X)
$dim
[1] 2 3 4

$dimnames
$dimnames[[1]]
[1] "row1" "row2"

$dimnames[[2]]
[1] "col1" "col2" "col3"

$dimnames[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

```
>  attributes(X)$dim
[1] 2 3 4
>  attributes(X)$dimnames
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

Array dim & dimnames

```
> dimnames(X) <- NULL
> attributes(X)
$dim
[1] 2 3 4

> dimnames(X)
NULL
> #
> # setting dimnames using lists
> dimnames(X)<-list(
+ c('Row1', 'Row2'),
+ c('Col1','Col2','Col3'),
+ c('Table1','Table2','Table3','Table4')  )
```

```
> X
, , Table1
     Col1 Col2 Col3
Row1    1    3    5
Row2    2    4    6
……….
, , Table4
     Col1 Col2 Col3
Row1   42   44   46
Row2   43   45   47

> dimnames(X)<-list( c('Row1', 'Row2'),
NULL, NULL )
> X
, , 1
     [,1] [,2] [,3]
Row1    1    3    5
Row2    2    4    6
……….
```

Array Subsetting (similar to matrices)

In order to obtain an element you need to include within brackets the same number as the dimensions

```
> X[2,3,1]
[1] 6
```

Leaving one dimension empty, it means that you require all elements of this dimension

```
> X[,3,1]
[1] 5 6
> X[2,,1]
[1] 2 4 6
> X[2,3,]
[1]  6 12 41 47
```

Array Subsetting (similar to matrices)

```
> X[2,,]
     [,1] [,2] [,3] [,4]
[1,]    2    8   37   43
[2,]    4   10   39   45
[3,]    6   12   41   47
> X[,3,]
     [,1] [,2] [,3] [,4]
[1,]    5   11   40   46
[2,]    6   12   41   47
> X[,,1]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> X[2:1,2:3,c(3,1)]
, , 1

     [,1] [,2]
[1,]   39   41
[2,]   38   40

, , 2

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

```
> X[ ,2, c(T,F,T,F)]
     [,1] [,2]
[1,]    3   38
[2,]    4   39
```

The apply() function for arrays

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]       [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748 0.04364908
```

The apply() function for arrays

```
> a[,,1]+a[,,2]+a[,,3]+a[,,4]
          [,1]     [,2]
[1,] 0.5026851 1.646615
[2,] 0.5753803 2.126716
> apply(a[,,1:4], c(1, 2), sum)
          [,1]     [,2]
[1,] 0.5026851 1.646615
[2,] 0.5753803 2.126716
> sum(a[,,1:4]) # sum is not the same
[1] 4.851396
```

The apply() function for arrays

```
> apply(a[,,1:4], 1, sum) # sums for all elements of each row
[1] 2.149300 2.702096
> apply(a[,,1:4], 2, sum) # sums for all elements of each column
[1] 1.078065 3.773331
> apply(a[,,1:4], 3, sum) # sums for all elements of each table
[1] 4.38399274 0.05762672 0.37768933 0.03208737
> a[,,1]
           [,1]       [,2]
[1,] 0.9356295  2.1267906
[2,] 1.5825914 -0.2610187
```

The apply() function for arrays

```
# this is not equivalent to apply
> a[,,sum(1:4)]
            [,1]       [,2]
[1,] -0.3119881  0.2235598
[2,]  1.6607486 -0.3690628
> sum(1:4)
[1] 10
> a[,,10]
            [,1]       [,2]
[1,] -0.3119881  0.2235598
[2,]  1.6607486 -0.3690628
```

More examples on arrays

```
> iris3[1:2,,]
, , Setosa
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      5.1      3.5      1.4      0.2
[2,]      4.9      3.0      1.4      0.2

, , Versicolor
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      7.0      3.2      4.7      1.4
[2,]      6.4      3.2      4.5      1.5

, , Virginica
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      6.3      3.3      6.0      2.5
[2,]      5.8      2.7      5.1      1.9
```

More examples on arrays

```
> dim(iris3)
[1] 50  4  3
> attributes(iris3)
$dim
[1] 50  4  3

$dimnames
$dimnames[[1]]
NULL

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

$dimnames[[3]]
[1] "Setosa"     "Versicolor" "Virginica“
```

```
> attributes(iris3)$dim
[1] 50  4  3
> dim(iris3)
[1] 50  4  3
> attributes(iris3)$dimnames
[[1]]
NULL
[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W.“
[[3]]
[1] "Setosa"     "Versicolor" "Virginica"

> dimnames(iris3)
> dimnames(iris3)[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
```

More examples on arrays

```
> a1<-array(1:24,dim=c(3,4,2))
> a1
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
> dim(a1)<-c(4,3,2)
> a1
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

More examples on arrays

```
> # arrays and matrices
> is.matrix(a1)
[1] FALSE
> is.matrix(a1[,,1])
[1] TRUE
> is.array(a1)
[1] TRUE
> as.matrix(a1)
      [,1]
 [1,]    1
 [2,]    2
.............
[23,]   23
[24,]   24
```

```
> x<-1:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
> dim(x)
NULL
> dim(x)<-c(3,4,2)
> x
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

More examples on arrays

```
> # arrays and matrices
> is.matrix(a1)
[1] FALSE
> is.matrix(a1[,,1])
[1] TRUE
> is.array(a1)
[1] TRUE
> as.matrix(a1)
      [,1]
 [1,]    1
 [2,]    2
.............
[23,]   23
[24,]   24
```

```
> x<-1:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
> dim(x)
NULL
> dim(x)<-c(3,4,2)
> x
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

Lists

Lists are object with elements other objects.

To create a list you can use the function list() giving as a main parameter the objects (members) you wish to be contained in your list, together with their names.

Lists

```
> Gender<-c('Male', 'Male', 'Male',
    'Female')
> Gender<-factor(Gender)
> Gender
[1] Male   Male   Male   Female
Levels: Female Male
> x<-1:10
> x
 [1] 1 2 3 4 5 6 7 8 9 10
> sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> y<-list(my_sample=sample, x=x,
    the_gender=Gender)
```

```
> y
$my_sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172

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

$the_gender
[1] Male   Male   Male
   Female
Levels: Female Male
```

Lists

With the symbols $ or [[ ]], we can extract an element of a list.

- **Name.of.list$object.name**

```
> y$x
[1]  1  2  3  4  5  6  7  8  9 10
```

- **Name.of.list[[#number of object#]]**

```
> y[[3]]
[1] Male   Male   Male   Female
Levels: Female Male
> y$x[1:3]
[1] 1 2 3
```

List attributes

```
> attributes(y)
$names
[1] "my_sample"  "x"          "the_gender"

> names(y)
[1] "my_sample"  "x"          "the_gender"
> length(y)
[1] 3
> mode(y)
[1] "list"
> class(y)
[1] "list"
```

Removing an object from a list

```
> test<-y
> names(test)
[1] "my_sample"  "x"          "the_gender"
> test$x<-NULL
> names(test)
[1] "my_sample"  "the_gender"
> test[[1]]
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> test[[2]]
[1] Male   Male   Male   Female
Levels: Female Male
```

Removing an object from a list

```
> test<-y
> names(test)
[1] "my_sample"  "x"          "the_gender"
> test[[2]]<-NULL
> names(test)
[1] "my_sample"  "the_gender"
> test[[1]]
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> test[[2]]
[1] Male   Male   Male   Female
Levels: Female Male
```

More examples on lists

```
> z<-list() # create any empty list
> z
list()
>
> z[[1]] <- 1:10
> z
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

> z[[3]]  <- letters[1:12]
> z
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
NULL

[[3]]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

```
> z[[2]]  <- matrix( rnorm(15), 3,5  )
> z
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
           [,1]       [,2]      [,3]        [,4]        [,5]
[1,] -0.5700250 0.09437889  0.327325  1.63758371 -1.27203113
[2,] -0.6386178 1.90702865  1.110245  0.81436137  0.06450305
[3,]  1.7010978 0.03081311 -1.072777 -0.01851825 -0.51079259

[[3]]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

More examples on lists

```
> z1<-list()
> z1[[4]] <- letters[1:12]
> z1
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

```
> names(z1)
NULL
> names(z1)<-c('a1','b2','nystazw', 'koimamai')
> z1
$a1
NULL

$b2
NULL

$nystazw
NULL

$koimamai
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"

>
```

Subsetting lists

```
> z[[2]] <- round(z[[2]],1)
> z[[3]][5]
[1] "e"
> z[[3]][3:5]
[1] "c" "d" "e"
> z[[2]][3:2,]
     [,1] [,2] [,3] [,4] [,5]
[1,]  1.7    0 -1.1  0.0 -0.5
[2,] 100.0  100  1.1  0.8  0.1
> z[[2]][,3:4]
    [,1] [,2]
[1,]  0.3  1.6
[2,]  1.1  0.8
[3,] -1.1  0.0
> dim(z[[2]])
[1] 3 5
```

```
> z[[2]][2,2]
[1] 100
> z[[2]][2,2]<-50
> z[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,] 100.0  100  0.3  1.6 -1.3
[2,] 100.0   50  1.1  0.8  0.1
[3,]  1.7    0 -1.1  0.0 -0.5
> z[[2]][1:2,1:2]
    [,1] [,2]
[1,]  100  100
[2,]  100   50
> z[[2]][1:2,1:2]<-100
> z[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,] 100.0  100  0.3  1.6 -1.3
[2,] 100.0  100  1.1  0.8  0.1
[3,]  1.7    0 -1.1  0.0 -0.5
```

Data Frames

- Data frames are used for storing statistical datasets.
- It has the tabular form of a matrix and hence some properties as the "dim" apply
- The difference from matrices, is that each column can be of different class.
- Moreover, they are lists of vectors of the same length.
- To create a data frame we use the function data.frame().

Data Frames

```
> Gender<-c('Male', 'Male', 'Male', 'Female')
> Gender<-factor(Gender)
> Gender
[1] Male   Male   Male   Female
Levels: Female Male
> Smoking<-c(T, T, F, F)
> Smoking<-factor(Smoking)
> Choresterol<-c(200, 220, 180, 172)
> Choresterol
[1] 200 220 180 172
> sample<-data.frame(Gender, Smoking, Choresterol)
> sample
  Gender Smoking Choresterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
```

Data Frames

as.data.frame() => converts an R object into a data frame.

row.names() => defines names for the rows (observations) of the data frame.

names() => defines names into the columns (variables) of the data frame.

Data Frames

```
> x<-matrix(c(1,1,200,1,1,220, 1,0,180,0,0,172),
      ncol=3, byrow=T)
> x
     [,1] [,2] [,3]
[1,]    1    1  200
[2,]    1    1  220
[3,]    1    0  180
[4,]    0    0  172
> x<-as.data.frame(x)
> x
  V1 V2  V3
1  1  1 200
2  1  1 220
3  1  0 180
4  0  0 172
> names(x)
[1] "V1" "V2" "V3"
> names(x)<-c('Gender', 'Smoking',
      'Choresterol')
```

```
 > x
  Gender Smoking Choresterol
1      1       1         200
2      1       1         220
3      1       0         180
4      0       0         172

> x<-data.frame(x, row.names=c('obs1', 'obs2',
      'obs3', 'obs4') )
> x
     Gender Smoking Choresterol
obs1      1       1         200
obs2      1       1         220
obs3      1       0         180
obs4      0       0         172
```

Extracting variables from data Frames

- **Name.of.data.frame$variable.name**

```
> x
  Gender Smoking Cholesterol
1      1       1         200
2      1       1         220
3      1       0         180
4      0       0         172
> x$Gender
[1] 1 1 1 0
> x$Smo
[1] 1 1 0 0
> x$Choles
[1] 200 220 180 172
```

  [using only the first characters will also work]

- **Name.of.data.frame[,#column number#]**

```
  > x[,1]
[1] 1 1 1 0
```

Extracting rows and sub-data

Similarly as in matrices

e.g.

x[3,] => 3rd row/observation

X[5:7, c(3,6,8)] => a new data.frame with values of variables X3, X6, X8 and for observations 5, 6, 7.

Data Frames

Similar functions used for matrices can be also used here.

e.g. cbind adds a new variable

```
> rbind(1,x)
     Gender Smoking Choresterol
1         1       1           1
obs1      1       1         200
obs2      1       1         220
obs3      1       0         180
obs4      0       0         172
```

e.g. rbind adds a new observation

```
> cbind(1,x)
      1 Gender Smoking Choresterol
obs1  1      1       1         200
obs2  1      1       1         220
obs3  1      1       0         180
obs4  1      0       0         172
```

Reading Data

Several functions for reading data into R.

- **read.table**: for reading tabular data from text files
- **read.csv**: similar as read.table – can be also saved from excel
- **scan**: reading a series of elements from file or keyboard
- **source**, for reading in R code files (inverse of **dump**)
- **dget**, for reading in R files (inverse of **dput**)
- **load**, for reading in saved workspaces
- **unserialize**, for reading single R objects in binary form
- **readLines**, for reading lines of a text file

Writing Data in a file

Several functions for writing data to files

- **write.table**: writes the data in a txt file in tabular format – can be recovered using read.table
- **dump** => dumps several R objects in a file with their names (in R syntax) – can be recovered using the source command or simply pasting back to an R console
- **dput** => saves only one object (in R syntax) without the name – can recovered using the dget function
- **save**: saves a workspace
- **serialize, writeLines**

The read.table() function

- Most commonly used function for reading data.
- Primary arguments:
  - file: the name of a file, or a connection
  - header: logical indicating if the file has a header line
  - sep: a string indicating how the columns are separated
  - nrows: the number of rows in the dataset

The read.table() function

- Secondary arguments:
  - colClasses: a character vector indicating the class of each column in the dataset
  - comment.char: a character string indicating the comment character
  - skip: the number of lines to skip from the beginning
  - stringsAsFactors: should character variables be coded as factors?

Reading & Writing Data

```
> Gender<-c("Male", "Male", "Male",
      "Female")
> write(Gender,file="g.txt", ncol=4)
> x
[1]  1  2  3  4  5  6  7  8  9 10
> write(x,file="x.txt",
      ncol=length(x))
> Smoking
   [1]  TRUE  TRUE FALSE FALSE
> write(Smoking,
      file="smoking.txt", ncol=4)
> X
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
[4,]    4   10
[5,]    5   11
[6,]    6   12
```

```
> write(t(X), "X.txt", ncol=2)
> sample
  Gender Smoking Cholesterol
1   Male    TRUE         200
2   Male    TRUE         220
3   Male   FALSE         180
4 Female   FALSE         172
> write.table(sample, file="sample.txt")
> x<-scan("x.txt")
Read 10 items
> x
[1]  1  2  3  4  5  6  7  8  9 10
> X<-matrix(scan("XX.txt"), ncol=2, byrow=T)
Read 12 items
> X
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
[6,]   11   12
```

Reading & Writing Data

```
> zz<-read.table("sample.txt",
     header=T)
> zz
    Gender Smoking Cholesterol
1   Male    TRUE        200
2   Male    TRUE        220
3   Male   FALSE        180
4 Female   FALSE        172
> zz<-read.table("sample.txt",
     header=T)
> zz
    Gender Smoking Cholesterol
1   Male    TRUE        200
2   Male    TRUE        220
3   Male   FALSE        180
4 Female   FALSE        172
```

```
> sapply(zz,mode)
    Gender     Smoking Cholesterol
 "numeric"   "logical"   "numeric"
> zz<-read.table("sample.txt", header=T,
     stringsAsFactors = FALSE)
> zz
  Gender Smoking Cholesterol
1   Male    TRUE        200
2   Male    TRUE        220
3   Male   FALSE        180
4 Female   FALSE        172
> sapply(zz,mode)
    Gender     Smoking Cholesterol
"character"   "logical"   "numeric"
```

The function read.table()

For small to moderately sized datasets, you can usually call **read.table** without specifying any other arguments

data <- read.table("foo.txt")

R will automatically

- skip lines that begin with a #
- finds out how many rows there are (and how much memory needs to be allocated)
- finds 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.

The function read.table()

With larger datasets:

- 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.

The function read.table()

- Use the colClasses argument. Considerably speeds up **read.table** (often twice as fast).
- If all of the columns are "numeric", for example, then you can just set colClasses = "numeric".
- A fast way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 10)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt", colClasses =
  classes)
```

- Set nrows. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay.

Textual Format data files

- **dump** and **dput** are useful because the resulting textual format is edit-able, 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 doesn't have to specify it all over again.
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem.
- Textual formats adhere to the "Unix philosophy".
- Downside: The format is not very space-efficient but they can be considerably compressed

dput() & dget()

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")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

dump()

Multiple objects can be deparsed using the **dump** function and recovered in using source.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
a b
1 1 a
> x
[1] "foo"
```

Package "foreign"

SPSS	read.spss()
S	read.S()
STATA	read.dta()
SAS	read.xport()
Epi Info	read.epiinfo()
Minitab	read.mtb()
Octave	read.octave()

If/else commands

Syntax	Description
if (A) B	Check if A is satisfied. If yes perform B
if (A) B1 else B2	Check if A is satisfied. If yes perform B1 else B2
ifelse(A, B1, B2)	Same as previous

if else statement in R

```
if(A)
 {
  A1
 } else
 {
  A2
 }
```

$$h(x) = \begin{cases} x^2 & , x \le 0.05 \\ 0.25 & , x > 0.05 \end{cases}$$

```
> x<-0.10
> if(x<=0.05){
            h<-x^2
  } else    {
            h<-0.25
  }
```

if else statement in R

```
if(A)
 {
  B1
 } else if(C)
 {
  B2
 } else
 {
  B3
 }
```

$$h(x) = \begin{cases} x^2, & x \le 0.05 \\ 0.25, & 0.05 < x \le 1 \\ 1, & x > 1 \end{cases}$$

```
> x<-0.10
> if(x<=0.05)
  {
          h<-x^2
  } else if(x>0.25 & x<=1)
  {
          h<-0.25
  } else
  {
          h<-1
  }
```

Loops in R

Syntax	Description
for(index in A) B	Performs B as long as index belongs to A
while(A) B	repetitive execution of B is to be carried out till it meets the constraint condition (not A)
repeat A	Same as while
break	Stops current loop
next	Stops current loop and starts next iteration

for loop

for(index in A) B

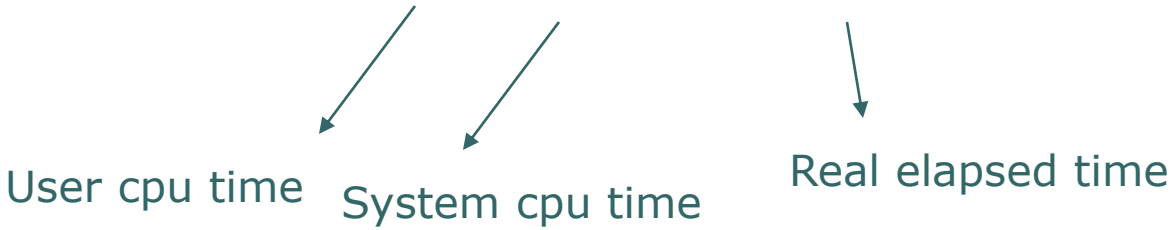
```
> x<-c(3,6,2,7)
> n<-length(x)
> proda<-1
> summ<-0
> for(i in 1:n){
    summ<-summ+x[i]
    proda<-proda*x[i]
  }
```

for loop

```
> A<-matrix(1:1000^2,ncol=1000,
    nrow=1000)
> summ<-0
> for(i in 1:1000){
     for(j in 1:1000){
        summ<-summ+A[i,j]
     }
  }
```

for loop

```
> system.time({summ<-0; for(i in 1:1000){for(j in
    1:1000){summ<-summ+A[i,j]}}})
   user  system elapsed
   1.94    0.00    1.96
>system.time({sum(as.numeric(apply(A,1,sum)))})
   user  system elapsed
   0.05    0.00    0.36
```



while & repeat

while(A) B

repeat(B; if(A) break)

Suppose we wish to apply Newton-Raphson method in order to find the solution of the equation $f(x) = x^3 + 2x^2 - 7 = 0$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$f'(x) = 3x^2 + 4x$$

while & repeat

```
> x<-1
> tolerance<-0.000001
> f<-x^3+2*x^2-7
> f.prime<-3*x^2+4*x
> while(abs(f)>tolerance)
 {
      x<-x-f/f.prime
      f<-x^3+2*x^2-7
      f.prime<-3*x^2+4*x
 }
```

```
> x<-1
> tolerance<-0.000001
> f<-x^3+2*x^2-7
> f.prime<-3*x^2+4*x
> repeat
   {
        x<-x-f/f.prime
        f<-x^3+2*x^2-7
        f.prime<-3*x^2+4*x
        if(abs(f) <= tolerance) break
   }
```

Functions in R

```
> x
 [1]  46 104  94 114  35  70 120  29  19 135 200 222
    89 100  55 214  15  81 118 193
> range<-function(x){
                  y<-max(x)-min(x)
                  return(y)
          }
> range(x)
[1] 207
```

Functions in R

```
> calc<-function(a,b=2){
      y<-a^b
      return(y)
  }
```

b=2 default value.

```
> calc(4)
[1] 16
> calc(4,3)
[1] 64
```

a=4, b=3

Functions in R

- **Example**: Suppose we wish to create our own function to calculate x! where x is a natural number

```
fact1<-function(x){
        y<-floor(x)
        if (y!=x | x<0){
                print("Your number
                    is not natural")
        } else {
                f<-1
                if (x<2) return(f)
                for (i in 2:x) {
                        f<-f*i
                }
                return(f)
        }
}
```

```
> fact1(3)
[1] 6
> fact1(1)
[1] 1
> fact1(0)
[1] 1
> fact1(4)
[1] 24
> fact1(2.3)
[1] "Your number is not natural"
```

Functions in R

```
fact2<-function(x){
        y<-floor(x)
        if (y!=x | x<0) {
                print("Your number
                is not natural")
        }else{
                f<-1
                t<-x
                while(t>1){
                        f<-f*t
                        t<-t-1
                }
                return(f)
        }
}
```

```
> fact2(3)
[1] 6
> fact2(1)
[1] 1
> fact2(0)
[1] 1
> fact2(4)
[1] 24
> fact2(2.3)
[1] "Your number is not natural"
```

Functions in R

```
fact3<-function(x){
        y<-floor(x)
        if (y!=x | x<0) {
                print("Your number is not
                                natural")
        }else{
                f<-1
                t<-x
                repeat{
                        if (t<2) break
                        f<-f*t
                        t<-t-1
                 }
                return(f)
        }
}
```

```
> fact3(3)
[1] 6
> fact3(1)
[1] 1
> fact3(0)
[1] 1
> fact3(4)
[1] 24
> fact3(2.3)
[1] "Your number is not natural"
```

Functions in R

Loops in R are time consuming.

We try to avoid for-loops using

- Apply
- Logical vectors to select sub-vectors or matrices
- Matrix operations where possible

  e.g.

  for(i in 1:length(y)) {if(y[i]<0} y[i]<-0}

  can be replaced by

  y[y<0]<-0

Functions in R

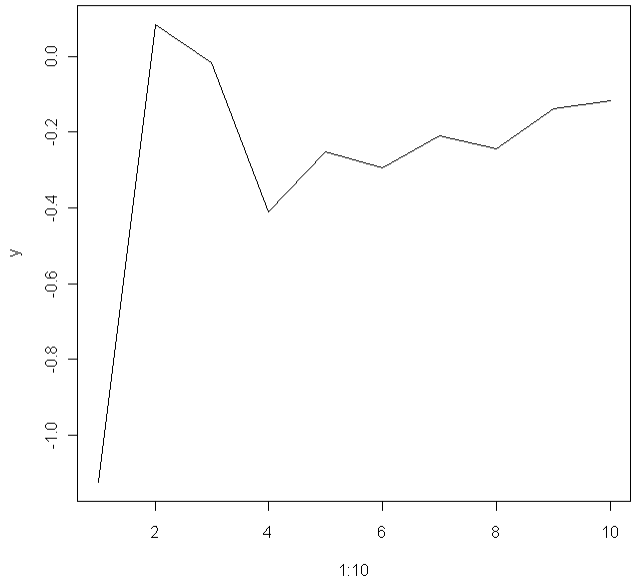
Additionally we could use the cumprod() [and cumsum] function of R, that calculates the cumulative product, i.e.

```
> cumprod(c(1,2,4))
[1] 1 2 8
```

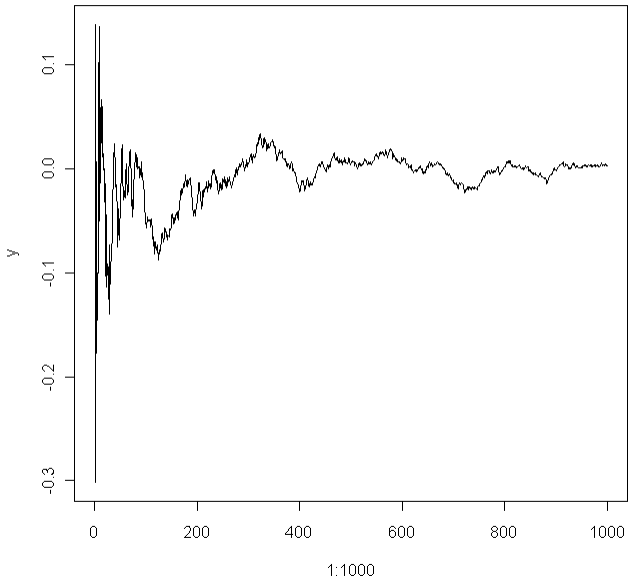
Sequential/Ergodic means using the cumsum function

```
x<- rnorm(10)
y<- cumsum(x)/1:length(x)
y
plot(1:lengt(x),y, type='l')
```

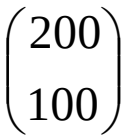
10 observations



1000 observations



Overflow Problems




```
> factorial(200)/(factorial(100)*factorial(100))
[1] NaN
Warning message:
In factorial(200) : value out of range in 'gammafn'
```

Overflow Problems

We have $\binom{200}{100} = \frac{101 \cdot \ldots \cdot 200}{1 \cdot \ldots \cdot 100}$

Thus in R

```
> prod(101:200)/prod(1:100)
[1] 9.054851e+58
```

$\longrightarrow \quad 9.05 \cdot 10^{58}$

or even better

```
> x<-1:100
> y<-101:200
> z<-y/x
> prod(z)
[1] 9.054851e+58
```

Overflow Problems

Another way to avoid overflow problems is by using logs.

$$\binom{200}{100} = \exp\left[\log\left\{\binom{200}{100}\right\}\right] = \exp[\log(200!) - 2\log(100!)]$$

But

$$\log(n!) = \sum_{i=1}^{n} \log(i)$$

```
> exp(sum(log(1:200))-2*sum(log(1:100)))
[1] 9.054851e+58
```

Other apply Functions

Avoiding loops with the following apply functions considerably speeds up computations

- **lapply**: Loop over a list and evaluate a function on each element.
- **sapply**: Same as lapply but try to simplify the result.
- **apply**: Apply a function over the margins of an array.
- **tapply**: Apply a function over subsets of a vector.
- **mapply**: Multivariate version of lapply.

lapply()

- **lapply** takes three arguments: a list X, a function (or the name of a function) FUN, and other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list.

```
> lapply
function (X, FUN, ...)
{
FUN <- match.fun(FUN)
if (!is.vector(X) || is.object(X))
X <- as.list(X)
.Internal(lapply(X, FUN))
}
```

The actual looping is done internally in C code.

lapply()

- **lapply** always returns a list, regardless of the class of the input.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3
$b
[1] 0.0296824
```

lapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),
                d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] 0.06082667
$c
[1] 1.467083
$d
[1] 5.074749
```

sapply()

- **sapply** will try to simplify the result of **lapply** if possible.
- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can’t figure things out, a list is returned.

sapply()

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),
                d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5
$b
[1] 0.06082667
$c
[1] 1.467083
$d
[1] 5.074749
```

sapply()

```
> sapply(x, mean)
a b c d
2.50000000 0.06082667 1.46708277
   5.07474950
> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric
   or logical: returning NA
```

tapply()

tapply is used to apply a function over subsets of a vector.

Syntax

tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)

- X is a vector.
- INDEX is a factor or a list of factors (or else they are coerced to factors).
- FUN is a function to be applied.
- ... contains other arguments to be passed FUN.
- simplify, should we simplify the result?

tapply()

Take group means.

simulates from uniform in (0,1) (more later)

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
1 2 3
0.1144464 0.5163468 1.2463678
```

Generate factors by specifying the pattern of their levels.

tapply()

- Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$‘1‘
[1] 0.1144464
$‘2‘
[1] 0.5163468
$‘3‘
[1] 1.246368
```

tapply()

- Find group ranges.

```
> tapply(x, f, range)
$‘1‘
[1] -1.097309 2.694970
$‘2‘
[1] 0.09479023 0.79107293
$‘3‘
[1] 0.4717443 2.5887025
```

split()

split takes a vector or other objects and splits it into groups determined by a factor or list of factors.

Syntax:

split(x, f, drop = FALSE, ...)

- x is a vector (or list) or data frame.
- f is a factor (or coerced to one) or a list of factors.
- drop indicates whether empty factors levels should be dropped.

split()

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$‘1‘
[1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
[5] 0.2849881 0.9383361 -1.0973089 2.6949703
[9] 1.5976789 -0.1321970
$‘2‘
[1] 0.09479023 0.79107293 0.45857419 0.74849293
[5] 0.34936491 0.35842084 0.78541705 0.57732081
[9] 0.46817559 0.53183823
$‘3‘
[1] 0.6795651 0.9293171 1.0318103 0.4717443
……
```

split()

Use **split** with **lapply**.

```
> lapply(split(x, f), mean)
$‘1‘
[1] 0.1144464
$‘2‘
[1] 0.5163468
$‘3‘
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
Ozone Solar.R Wind Temp Month Day
1 41 190 7.4 67 5 1
2 36 118 8.0 72 5 2
3 12 149 12.6 74 5 3
4 18 313 11.5 62 5 4
5 NA NA 14.3 56 5 5
6 28 NA 14.9 66 5 6
```

Splitting a Data Frame

```
> s <- split(airquality, airquality$Month)
> mycolmeans <-function(x) {
                    colMeans(x[, c("Ozone", "Solar.R", "Wind")])
                    }
> lapply(s, mycolmeans)
$‘5‘
Ozone Solar.R Wind
NA NA 11.62258
$‘6‘
Ozone Solar.R Wind
NA 190.16667 10.26667
$‘7‘
Ozone Solar.R Wind
NA 216.483871 8.941935
……
```

Splitting a Data Frame

```
> sapply(s, mycolmeans)
5 6 7 8 9
Ozone NA NA NA NA NA
Solar.R NA 190.16667 216.483871 NA 167.4333
Wind 11.62258 10.26667 8.941935 8.793548 10.1800

> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
na.rm = TRUE))
5 6 7 8 9
Ozone 23.61538 29.44444 59.115385 59.961538 31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind 11.62258 10.26667 8.941935 8.793548 10.18000
```

Splitting on more that one level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
[1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
[1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
[1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Splitting on more that one level

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))
List of 10
$ 1.1: num [1:2] -0.378 0.445
$ 2.1: num(0)
$ 1.2: num [1:2] 1.4066 0.0166
$ 2.2: num(0)
$ 1.3: num -0.355
$ 2.3: num 0.315
$ 1.4: num(0)
$ 2.4: num [1:2] -0.907 0.723
$ 1.5: num(0)
$ 2.5: num [1:2] 0.732 0.360
```

Splitting on more that one level

Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
$ 1.1: num [1:2] -0.378 0.445
$ 1.2: num [1:2] 1.4066 0.0166
$ 1.3: num -0.355
$ 2.3: num 0.315
$ 2.4: num [1:2] -0.907 0.723
$ 2.5: num [1:2] 0.732 0.360
```

by()

by(data, INDICES, FUN, ..., simplify = TRUE)

Arguments

- **data** an **R** object, normally a data frame, possibly a matrix.
- **INDICES** a factor or a list of factors, each of length nrow(data).
- **FUN** a function to be applied *to (usually data-frame)* subsets of data.
- ...further arguments to FUN.
- **simplify** logical: see tapply.

by()

```
require(stats)
by(warpbreaks[, 1:2], warpbreaks[,"tension"], summary)
```

```
warpbreaks[, "tension"]: L
     breaks       wool
 Min.   :14.00   A:9
 1st Qu.:26.00   B:9
 Median :29.50
 Mean   :36.39
 3rd Qu.:49.25
 Max.   :70.00
------------------------------------------------------------
warpbreaks[, "tension"]: M
     breaks       wool
 Min.   :12.00   A:9
 1st Qu.:18.25   B:9
 Median :27.00
 Mean   :26.39
 3rd Qu.:33.75
 Max.   :42.00
------------------------------------------------------------
warpbreaks[, "tension"]: H
     breaks       wool
 Min.   :10.00   A:9
 1st Qu.:15.25   B:9
 Median :20.50
 Mean   :21.67
 3rd Qu.:25.50
 Max.   :43.00
```

by()

by(warpbreaks[, 1], warpbreaks[, -1], summary)

```
wool: A
tension: L
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  25.00   26.00   51.00   44.56   54.00   70.00
------------------------------------------------
wool: B
tension: L
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.00   20.00   29.00   28.22   31.00   44.00
------------------------------------------------
wool: A
tension: M
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     12      18      21      24      30      36
------------------------------------------------
wool: B
```

by()

By does not work with functions for vectors

It needs functions for data.frames

```
> by(x,z,mean)
z: no
[1] NA
------------------------------------
z: yes
[1] NA
Warning messages:
1: In mean.default(data[x, , drop = FALSE], ...) :
  argument is not numeric or logical: returning NA
2: In mean.default(data[x, , drop = FALSE], ...) :
  argument is not numeric or logical: returning NA
```

by()

```
> x<-data.frame( variable1=rnorm(10), variable2=rnorm(10,2) )
> z<-gl(2,5)
> levels(z)<-c('no','yes')
> data.mean<-function( x, index ){
+ res<-sapply(x,mean)
+ return(res)
+ }
> by(x,z,data.mean)
z: no
variable1 variable2
0.2050054 2.2936324
------------------------------------------------------------
z: yes
 variable1  variable2
0.07653843 1.96158225
```

mapply()

- **mapply** is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

Syntax

mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

- FUN is a function to apply
- ... contains arguments to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

mapply()

The following is tedious to type

```
> list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1
[[2]]
[1] 2 2 2
[[3]]
[1] 3 3
[[4]]
[1] 4
```

Vectorizing a Function

```
> noise <- function(n, mean, sd) {
+                                   rnorm(n, mean, sd)
+                       }
> noise(5, 1, 2)
[1] 2.4831198 2.4790100 0.4855190 -1.2117759
[5] -0.2743532
> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266 4.2507057 1.1572738
[5] 3.7413584
```

Vectorizing a Function

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658
[[2]]
[1] 0.7113482 2.7555797
[[3]]
[1] 2.769527 1.643568 4.597882
[[4]]
[1] 4.476741 5.658653 3.962813 1.204284
[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

The above is the same as:

```
> list(noise(1, 1, 2), noise(2, 2, 2), noise(3, 3, 2), noise(4, 4, 2),
    noise(5, 5, 2))
```

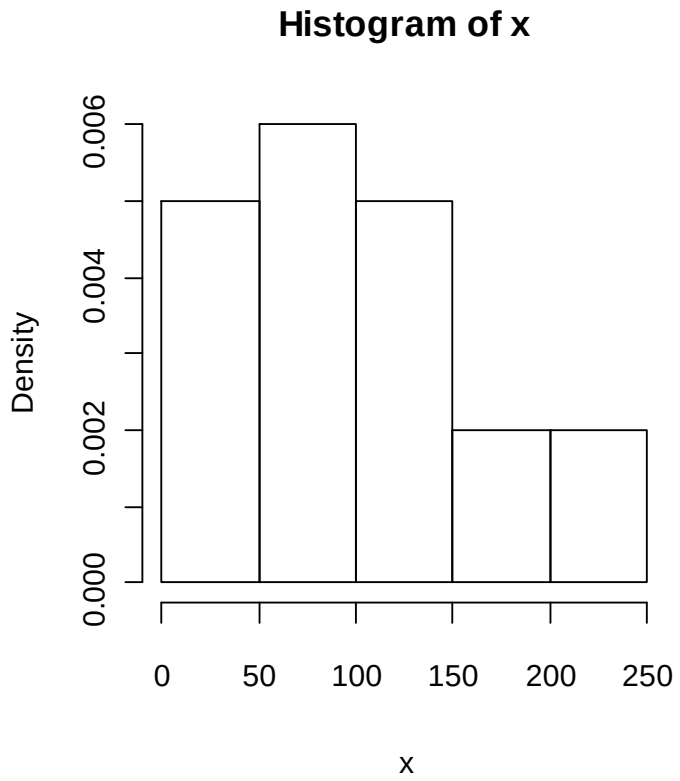
Numerical Measures – Quantitative Variables

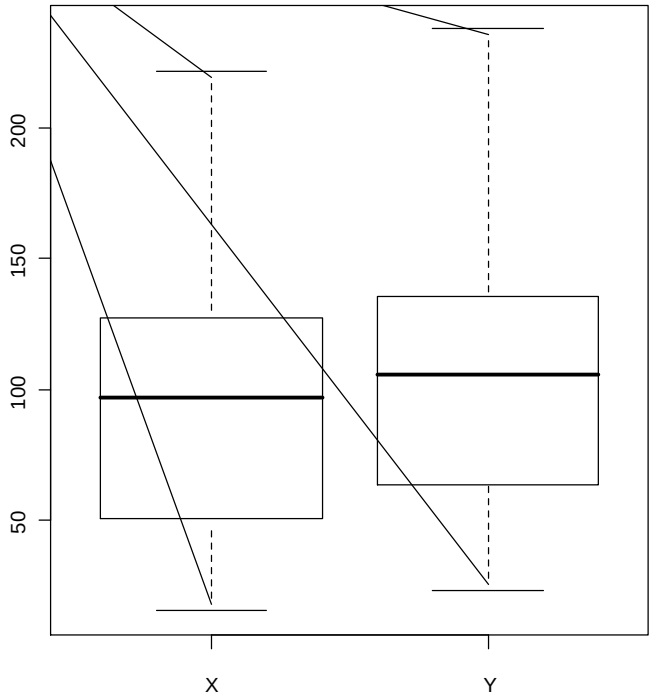
Function	Description
mean(x)	Sample Mean
min(x)	Minimum
max(x)	Maximum
median(x)	Median
var(x)	Variance
sd(x)	Standard Deviation
quantile(x,p)	Returns the p percentile. Για p=0.25 και p=0.75 we get the 1ο and 3ο quartile. Read the help in R for the different quantile algorithms (argument "types")

Graphical Methods – Quantitative Variables

- Histogram:
  - hist(x)
  - hist(x, nclass=10)
  - hist(x, breaks=seq(from=0,to=240,by=30))
  - hist(x, probability=T)
- Boxplot:
  - boxplot(x)
  - boxplot(x,y, names=c("X", "Y"))

Graphical Methods – Quantitative Variables





Descriptive Measures - Categorical Variables

```
> Transportation<-c("C", "C", "B", "M", "M", "C", "M", "M", "F",
    "C", "F", "B", "B", "M", "M", "C", "C", "C", "M", "C")
> Transportation<-factor(Transportation)
> Gender<-c(rep("M",10), rep("F", 10))
> Gender
 [1] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "F" "F" "F" "F"
    "F" "F" "F" "F" "F" "F"
> Gender<-factor(Gender)
> table(Transportation)
A
B C F M
3 8 2 7
> prop.table(table(Transportation))
A
   B    C    F    M
0.15 0.40 0.10 0.35
```

Descriptive Measures - Categorical Variables

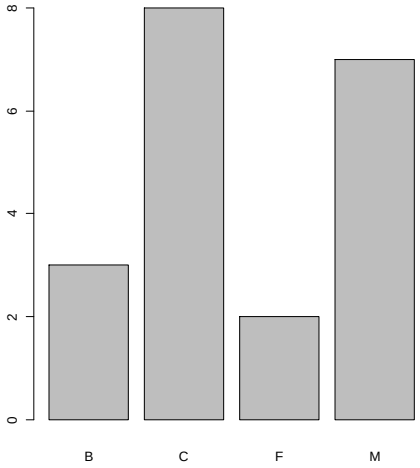
```
> mytable<-
      table(Transportation,Gender)
> mytable
   Gender
A   F M
  B 2 1
  C 4 4
  F 1 1
  M 3 4
> margin.table(mytable, 1)
A
B C F M
3 8 2 7
> margin.table(mytable, 2)
Gender
 F  M
10 10
```

```
> prop.table(mytable)
                Gender
Transportation      F    M
  B                 0.10 0.05
  C                 0.20 0.20
  F                 0.05 0.05
  M                 0.15 0.20
> prop.table(mytable, 1)
                Gender
Transportation   F         M
  B              0.6666667 0.3333333
  C              0.5000000 0.5000000
  F              0.5000000 0.5000000
  M              0.4285714 0.5714286
> prop.table(mytable, 2)
                Gender
Transportation     F   M
  B                0.2 0.1
  C                0.4 0.4
  F                0.1 0.1
  M                0.3 0.4
```

Descriptive Measures - Categorical Variables

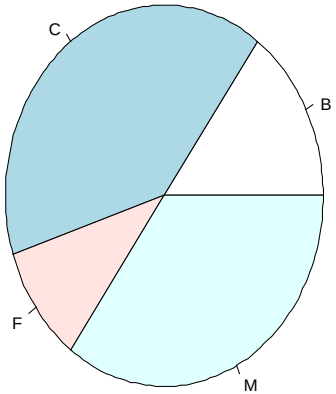
- **Barplot**

```
> AA<-table(A)
> AA
A
B C F M
3 8 2 7
> barplot(AA)
```



- **Piechart**

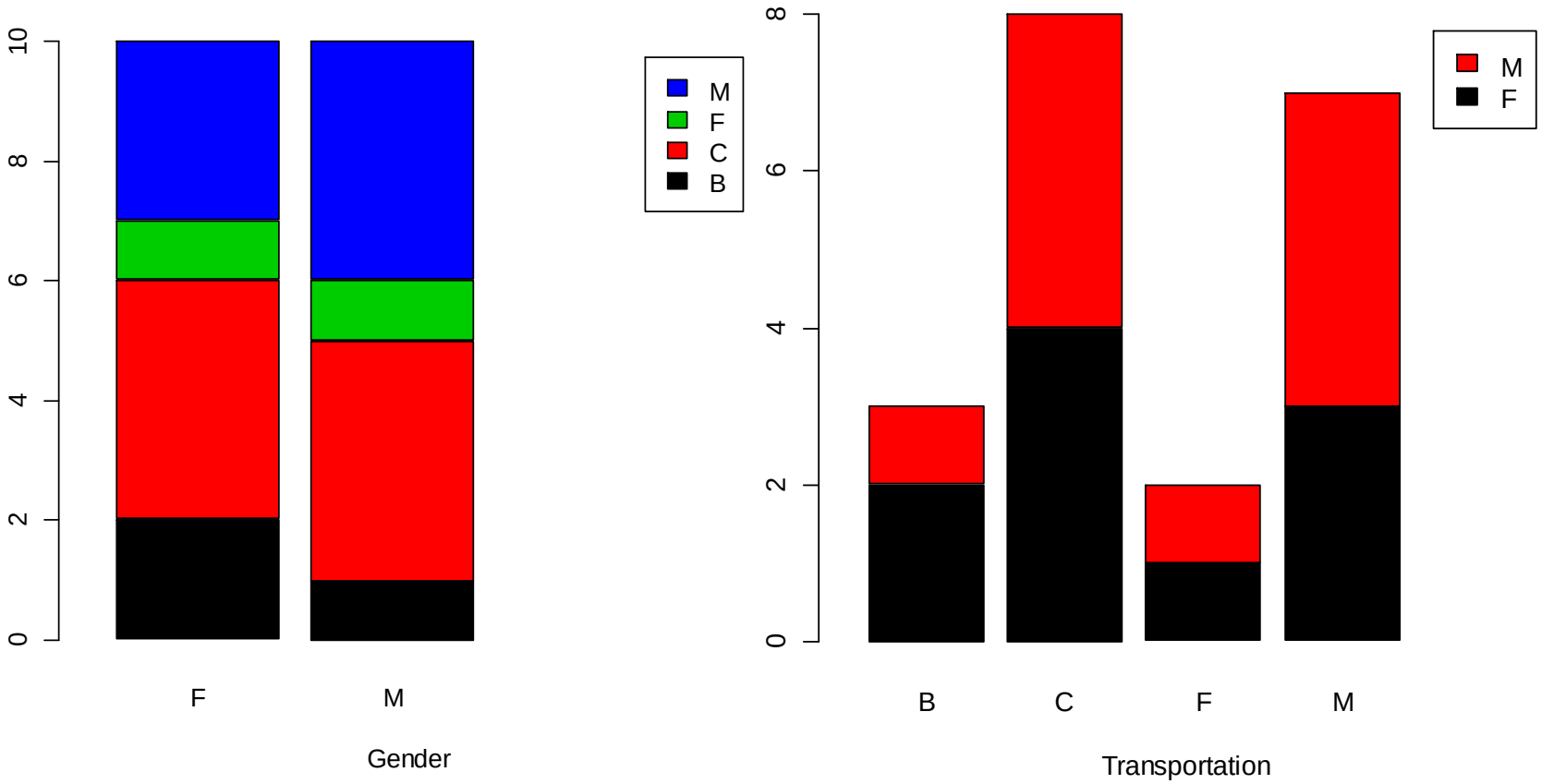
```
> pie(AA)
```



Stacked Barplots

```
> freq_table<-table(Transportation,Gender)
> barplot(freq_table, xlim=c(0,3), xlab="Gender",
    legend=levels(Transportation), col=1:4)
> freq_table<-table(Gender, Transportation)
> barplot(freq_table, width=0.85, xlim=c(0,5),
    xlab="Transportation", legend=levels(Gender),
    col=1:2)
```

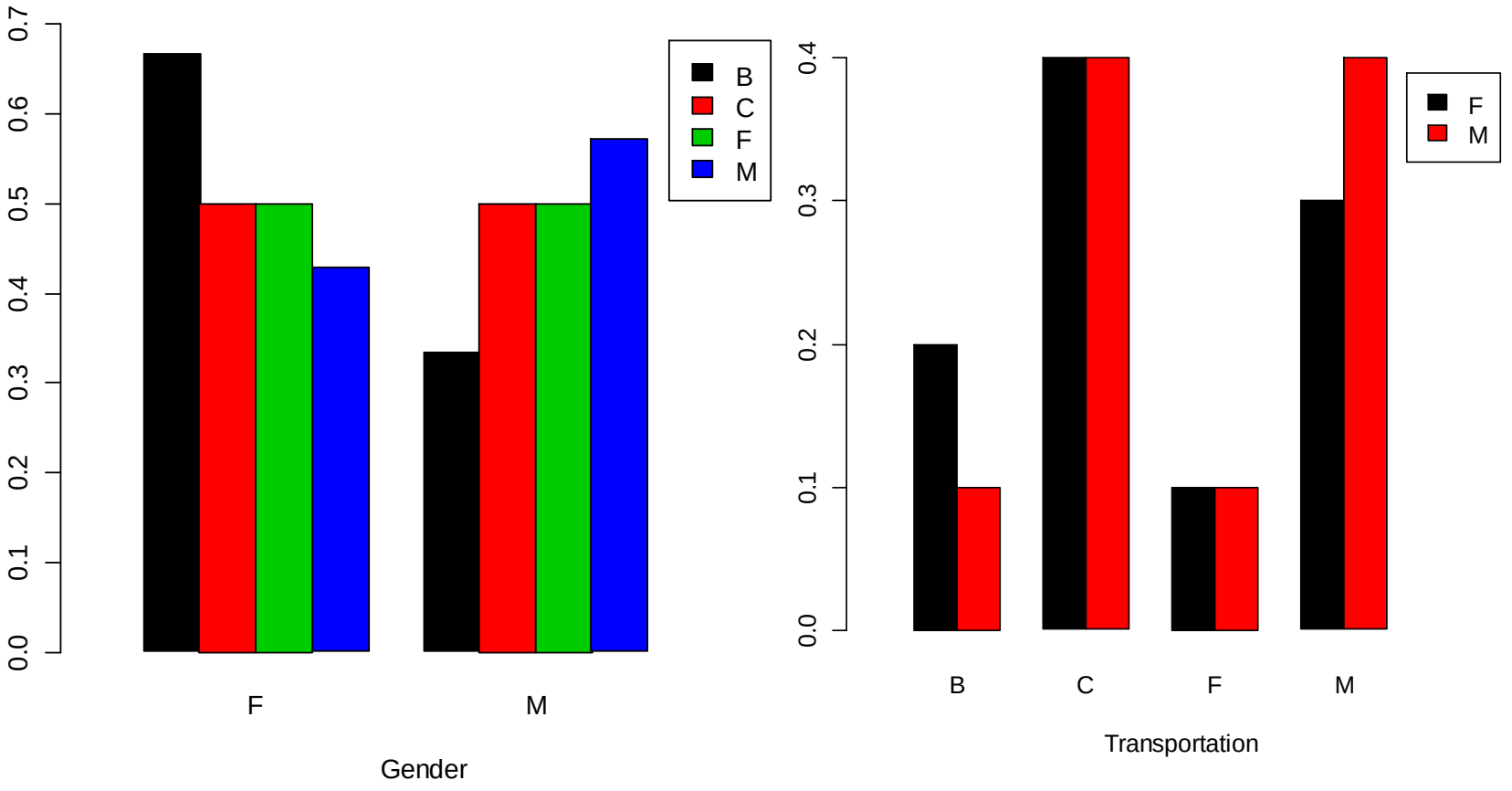
Stacked Barplots



Grouped Barplot

```
> freq_table<-table(Transportation,Gender)
> barplot(prop.table(freq_table,1), width=0.25, xlim=c(0,3),
    ylim=c(0,0.7), xlab="Gender",
    legend=levels(Transportation), beside=T, col=1:4)
> freq_table<-table(Gender, Transportation)
> barplot(prop.table(freq_table,1), width=0.25,
    xlim=c(0,3.6), xlab="Transportation",
    legend=levels(Gender), beside=T, col=1:2)
```

Grouped Barplot



Plotting

The plotting and graphics engine in R is encapsulated in a few base and recommend packages:

- **graphics**: contains plotting functions for the "base" graphing systems, including plot, hist, boxplot and many others.
- **lattice**: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like xyplot, bwplot, levelplot.
- **grid**: implements a different graphing system independent of the "base" system; the lattice package builds on top of grid; we seldom call functions from the grid package directly.
- **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

Base Graphics

plot(x, y): Main multiple purpose command

- plot(x, y) will launch a windows graphics device (if one is not already open) and draw the plot on the device.
- Many arguments for controlling the graph (title, x axis lable, y axis label, etc); see **help(par)** for details

Main Parameters of a Graph

- **main**: Title of the graph.
- **sub**: Subtitle of the graph.
- **xlab & ylab**: Titles of the axes.
- **xlim & ylim**: Range of the axes.

Base Graphics Parameters

The **par** function is used to specify global graphics parameters that affect all plots in an R session. These parameters can often be overridden as arguments to specific plotting functions.

- **pch**: the plotting symbol (default is open circle).
- **lty**: the line type (default is solid line), can be dashed, dotted, etc.
- **lwd**: the line width, specified as an integer multiple.
- **col**: the plotting color, specified as a number, string, or hex code; the colors function gives you a vector of colors by name.

Base Graphics Parameters

- **las**: the orientation of the axis labels on the plot.
- **bg**: the background color.
- **mar**: the margin size.
- **oma**: the outer margin size (default is 0 for all sides).
- **mfrow**: number of plots per row, column (plots are filled row-wise).
- **mfcol**: number of plots per row, column (plots are filled column-wise).

Base Graphics Parameters

- Some default values:

```
> par("lty")
[1] "solid"
> par("lwd")
[1] 1
> par("col")
[1] "black"
> par("pch")
[1] 1
> par("bg")
[1] "transparent"
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par("oma")
[1] 0 0 0 0
> par("mfrow")
[1] 1 1
> par("mfcol")
[1] 1 1
```

Important Base Plotting Functions

- **plot**: make a scatterplot, or other type of plot depending on the class of the object being plotted.
- **lines**: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots.
- **points**: add points to a plot.
- **text**: add text labels to a plot using specified x, y coordinates.
- **title**: add annotations to x, y axis labels, title, subtitle, outer margin.
- **mtext**: add arbitrary text to the margins (inner or outer) of the plot.
- **axis**: adding axis ticks/labels.

Text & Symbol Size

- **cex:** number indicating the amount by which plotting text and symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc.
- **cex.axis:** magnification of axis annotation relative to cex.
- **cex.lab:** magnification of x and y labels relative to cex.
- **cex.main:** magnification of titles relative to cex.
- **cex.sub:** magnification of subtitles relative to cex.

Axes

You can create custom axes using the **axis( )** function.

> axis(*side*, at=, labels=, pos=, lty=, col=, las=, tck=, ...)

- **lty:** line type.
- **col:** the line and tick mark color.
- **las:** labels are parallel (=0) or perpendicular(=2) to axis.

Axes

- **side:** an integer indicating the side of the graph to draw the axis (1=bottom, 2=left, 3=top, 4=right).
- **at:** a numeric vector indicating where tic marks should be drawn.
- **labels:** a character vector of labels to be placed at the tickmarks (if NULL, the *at* values will be used).
- **pos:** the coordinate at which the axis line is to be drawn.
  (i.e., the value on the other axis where it crosses).
- **tck:** length of tick mark as fraction of plotting region (negative number is outside graph, positive number is inside, 0 suppresses ticks, 1 creates gridlines) default is -0.01.

Axes

- The option **axes=FALSE** suppresses both x and y axes.
- **xaxt="n"** and **yaxt="n"** suppress the x and y axis respectively.
- **xlab=""** and **ylab=""** suppress the titles of the two axes.

Legend

legend() function => adds a legend in an existing graph

> legend(location, title, legend, ...)

location:

- Provide a vector with the **x,y coordinate** for the upper left hand corner of the legend.
- Use **locator(1)**, to interactively specify (using the mouse) location of the legend (i.e. the coordinates of the upper left hand corner).
- Use **keywords** "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "bottomright", or "center".
- If you use a keyword, you may want to use **inset=** to specify an amount to move the legend into the graph (as fraction of plot region).

Legend

Other arguments controlling the legend

- **title:** A character string for the legend title (optional).
- **legend:** A character vector with the labels.
- **...Other graph options**
  - **col =** colors of lines or points
  - **pch=** point symbols
  - **lwd=** line width
  - **lty=** line types/styles
  - **fill=** filled colored boxes (for barcharts and piecharts)

  All the above arguments should be vectors with length equal to the number of different lines, points or filled bars that appear in the associated graph.

Fonts

- **font:** Integer specifying font to use for text. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol.
- **font.axis:** font for axis annotation.
- **font.lab:** font for x and y labels.
- **font.main:** font for titles.
- **font.sub:** font for subtitles.
- **ps:** font point size (roughly 1/72 inch) text size=ps*cex

Fonts

family: font family for drawing text.

Standard values are "serif", "sans", "mono", "symbol".

Mapping is device dependent.

In windows,

mono => "TT Courier New",

serif => "TT Times New Roman",

sans => "TT Arial",

symbol => "TT Symbol"

(TT=True Type).

Margins & Graph Size

- **mar:** numerical vector indicating margin size c(bottom, left, top, right) in lines. default = c(5, 4, 4, 2) + 0.1.
- **mai:** numerical vector indicating margin size c(bottom, left, top, right) in inches.
- **pin:** plot dimensions (width, height) in inches.

Saving Graphs

- You can save the graph in a variety of formats from the menu

  File -> Save As.

- You can also save the graph via code using one of the following functions.
  - **pdf("mygraph.pdf"):** pdf file.
  - **win.metafile("mygraph.wmf"):** windows metafile.
  - **png("mygraph.png"):** pngfile.
  - **jpeg("mygraph.jpg"):** jpeg file.
  - **bmp("mygraph.bmp"):** bmp file.
  - **postscript("mygraph.ps"):** postscript file.

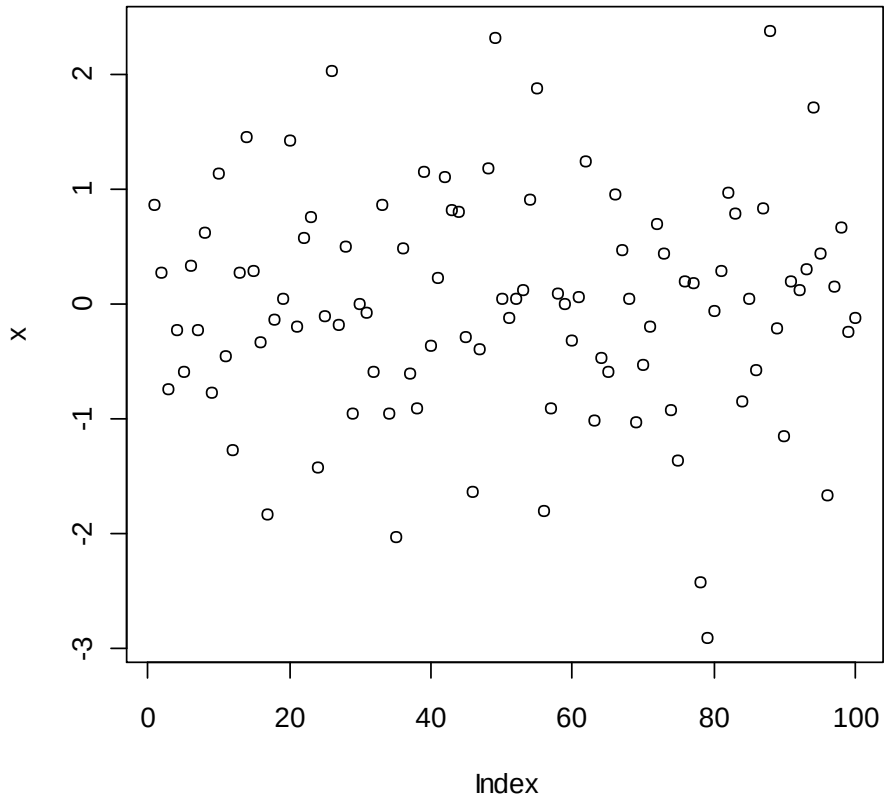
Useful Graphics Devices

The list of devices is found in ?Devices; there are also devices created by users on CRAN

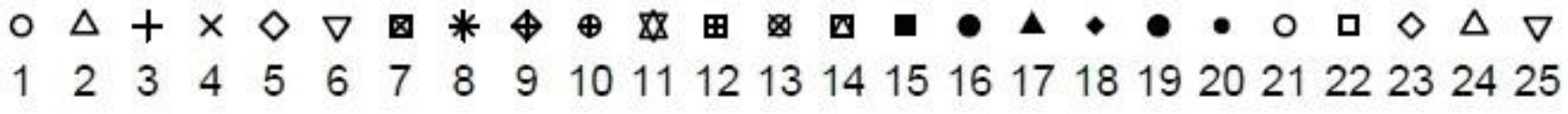
- windows The graphics device for Windows.
- png,jpeg, bmp, tiff : Standard image file devices
- postscript Writes PostScript graphics commands to a file
- pdf Write PDF graphics commands to a file

Plot

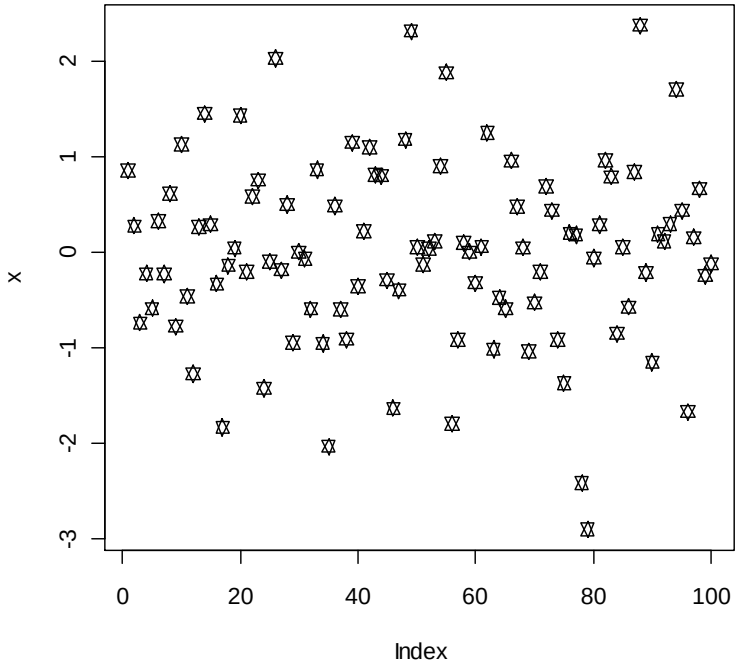
```
> plot(x)
```



Different plotting symbols

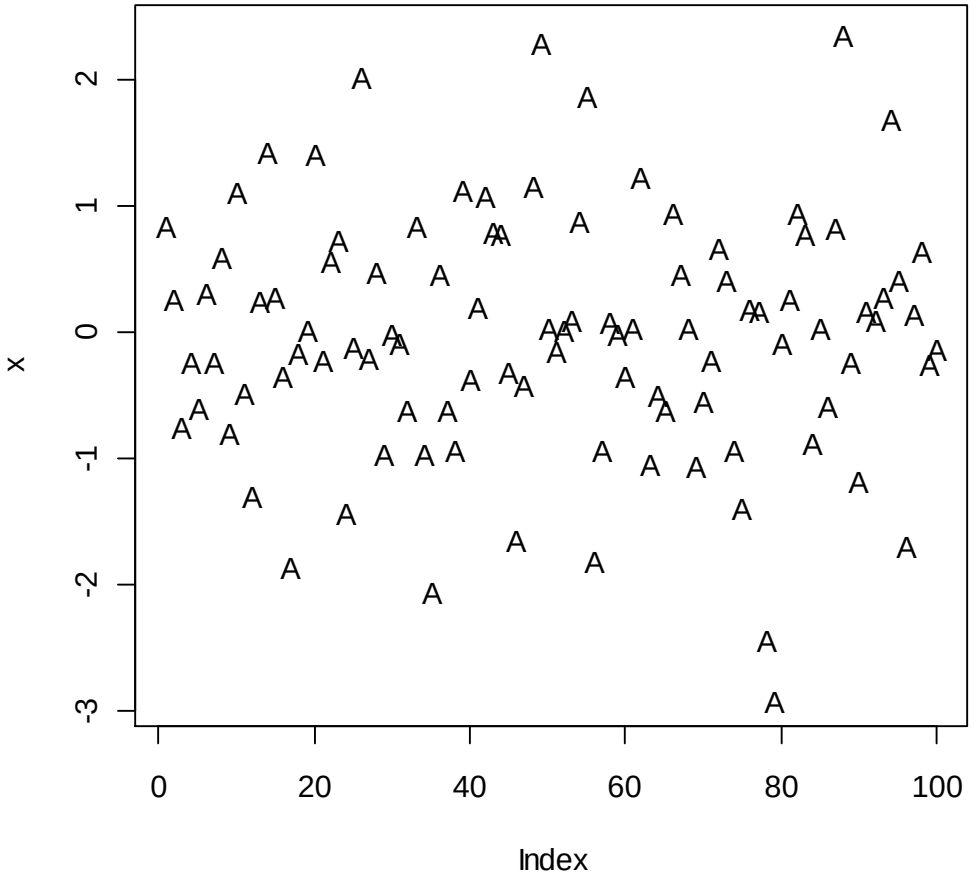



```
> plot(x, pch=11)
```



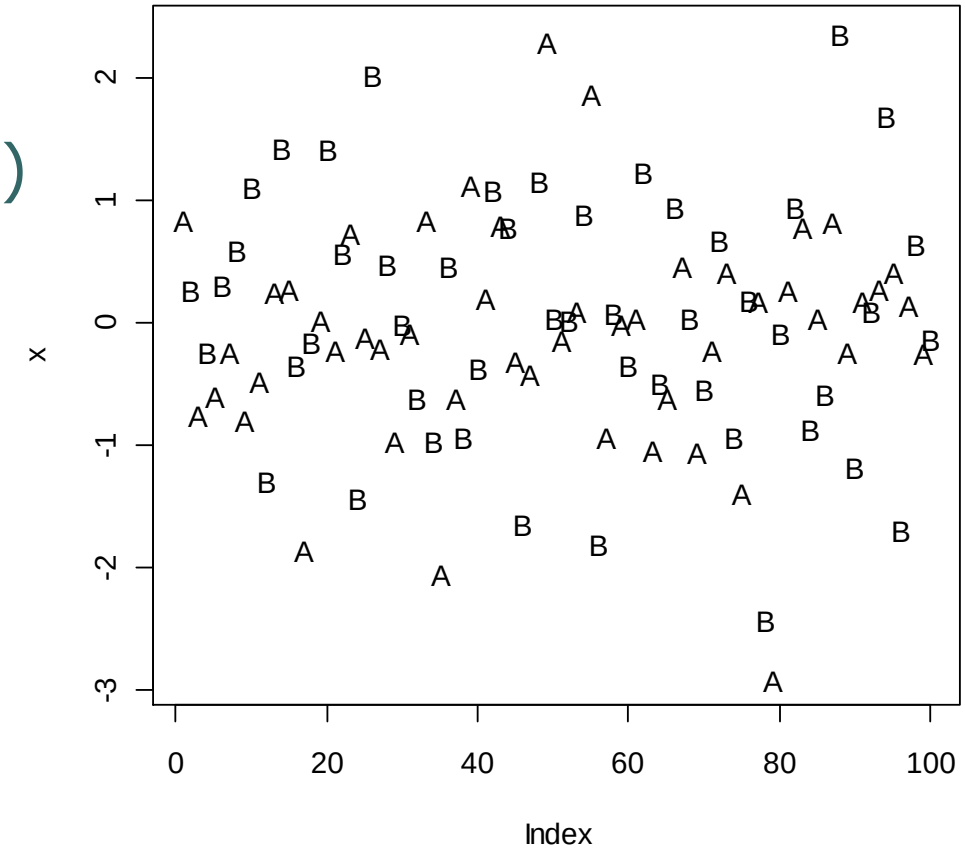
Different plotting symbols

```
> plot(x, pch='A')
```



Different plotting symbols

```
> plot(x, pch=c('A', 'B'))
```

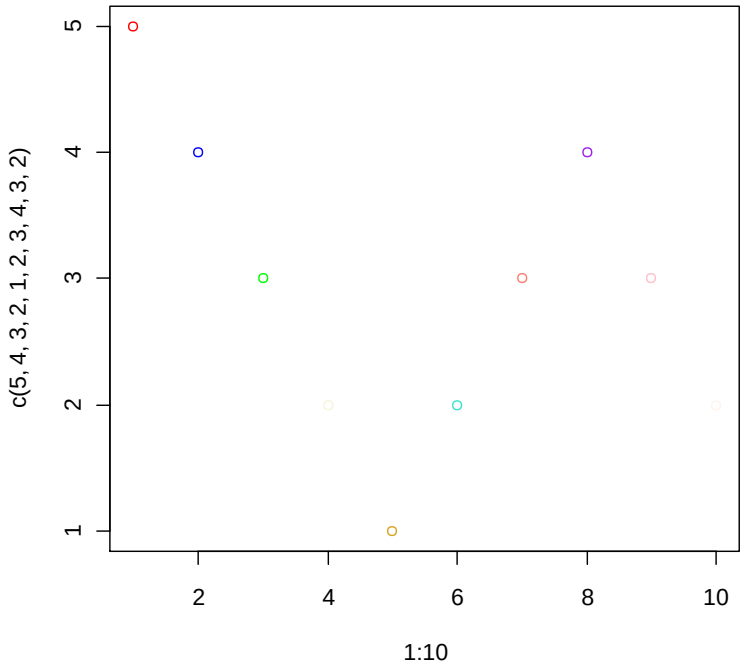


Colors

```
> colors()
  [1] "white"          "aliceblue"          "antiquewhite"
  [4] "antiquewhite1"  "antiquewhite2"  "antiquewhite3"
  [7] "antiquewhite4"  "aquamarine"     "aquamarine1"
 [10] "aquamarine2"    "aquamarine3"    "aquamarine4"
 [13] "azure"          "azure1"         "azure2"
 [16] "azure3"         "azure4"         "beige"
 [19] "bisque"         "bisque1"        "bisque2"
 [22] "bisque3"        "bisque4"        "black"
.....................................................
.....................................................
.....................................................
```

Colors

```
> plot(1:10, c(5, 4, 3, 2, 1, 2, 3, 4, 3, 2), col=c("red",
    "blue", "green", "beige", "goldenrod", "turquoise",
                "salmon", "purple", "pink", "seashell"))
```



Colors

```
> palette()
[1] "black" "red" "green3" "blue" "cyan" "magenta" "yellow"
[8] "gray"
```

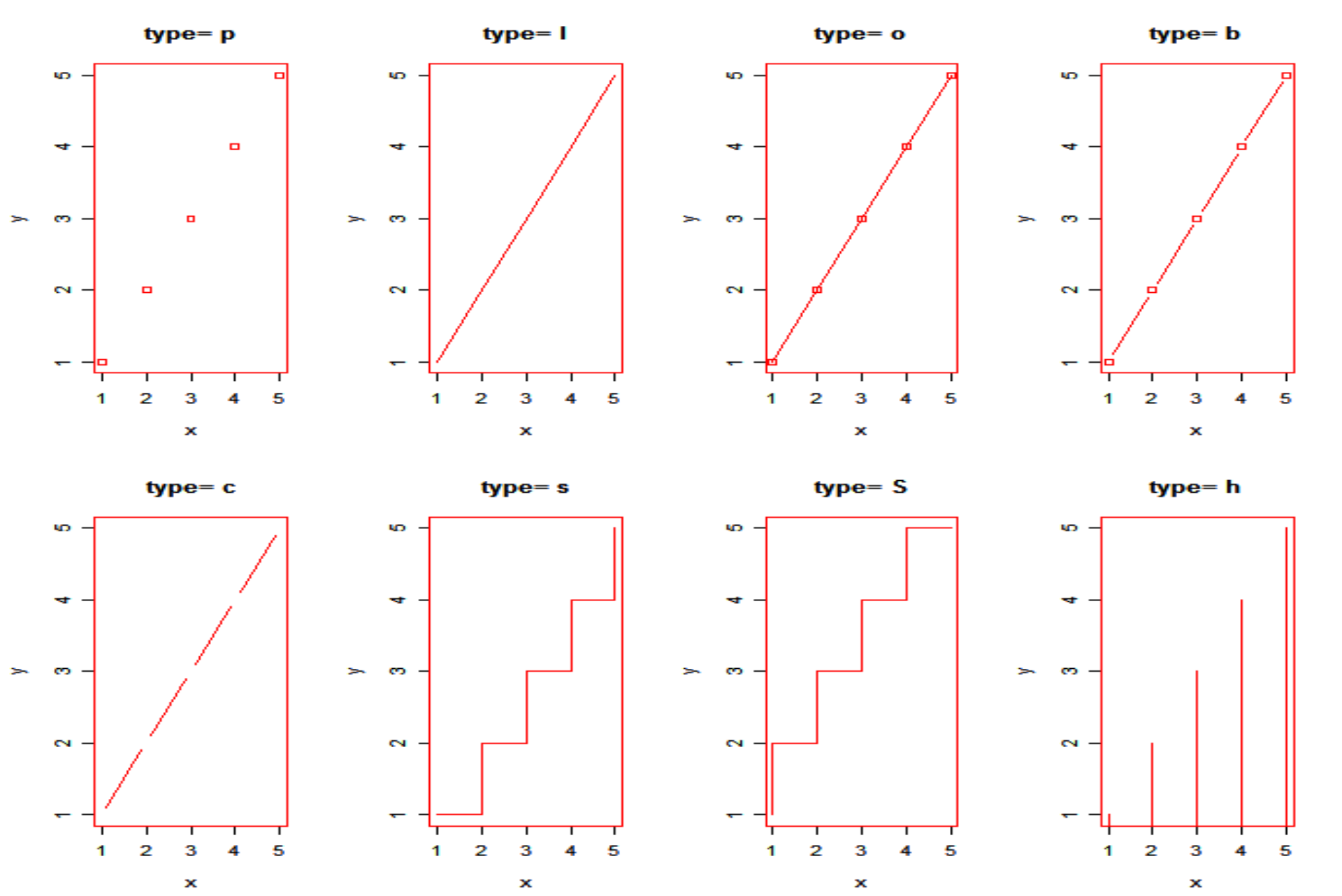
- col = 1 ⟷ col = "black"
- col = 2 ⟷ col = "red", etc

Type Arguments

- **"p":** Points
- **"l":** Lines
- **"b":** Both
- **"c":** The lines part alone of "b"
- **"o":** Both "overplotted"
- **"h":** Histogram like (or high-density) vertical lines
- **"n":** No plotting

Type Argument

```
>  x <- c(1:5); y <- x # create some data
    par(pch=22, col="red") # plotting symbol and color
    par(mfrow=c(2,4)) # all plots on one page
    opts = c("p","l","o","b","c","s","S","h")
    for(i in 1:length(opts)){
            heading = paste("type=",opts[i])
            plot(x, y, type="n", main=heading)
            lines(x, y, type=opts[i])
    }
```


type= p
type= l
type= o
type= b
type= c
type= s
type= S
type= h
x
y

Add to a graph

- **points(x,y):** Add points to an existing graph.
- **lines(x,y):** Add a line to an existing graph.

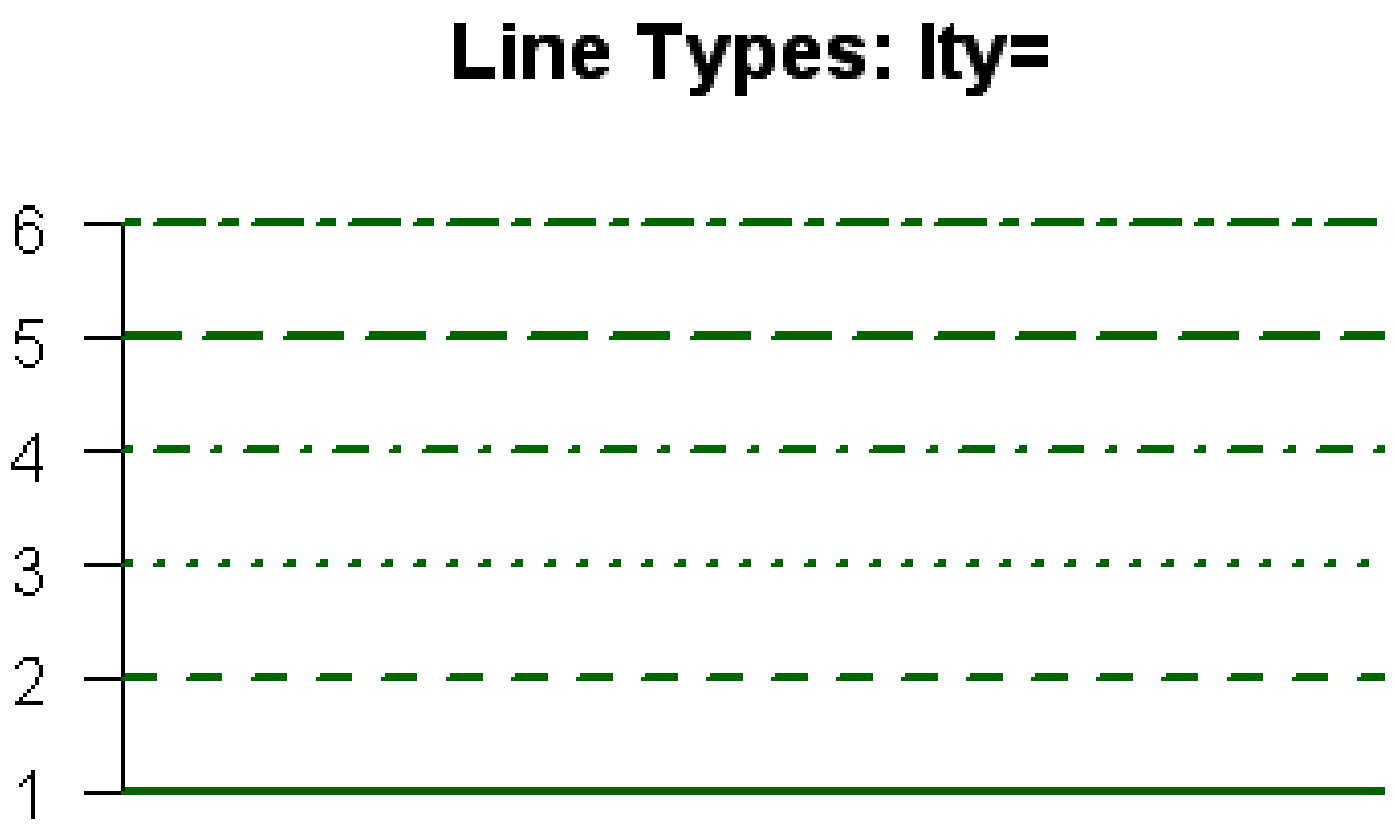
- **abline(a=2, b=4):** Add to the existing graph the line y = 2+4x.
- **abline(v=4):** Add to the existing graph the line x = 4.
- **abline(h=2):** Add to the existing graph the line y = 2.
- **abline(v=1:4):** Add to the existing graph the lines x = 1, x = 2, x = 3 & x = 4.
- **abline(lm(y~x)):** Least square regression line

Add to a graph

- **segments(x0, y0, x1,y1):** Draw line segments between pairs of points – from (x0, y0) to (x1, y1).
- **arrows(x0, y0, x1,y1):** Draw arrows from (x0, y0) to (x1, y1).
- Additional arguments here are:
  - (a) length: length of the edges of the arrow head (in inches),
  - (b) angle: angle from the shaft of the arrow to the edge of the arrow head,
  - (c) code: integer code, determining *kind* of arrows to be drawn.

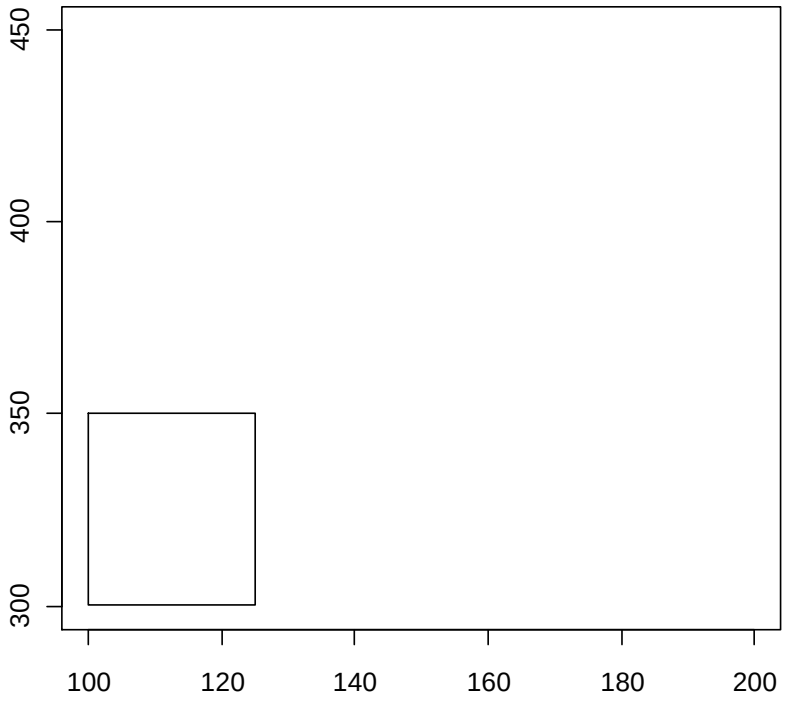
(x0, y0) to (x1, y1) => point coordinates

Line Types



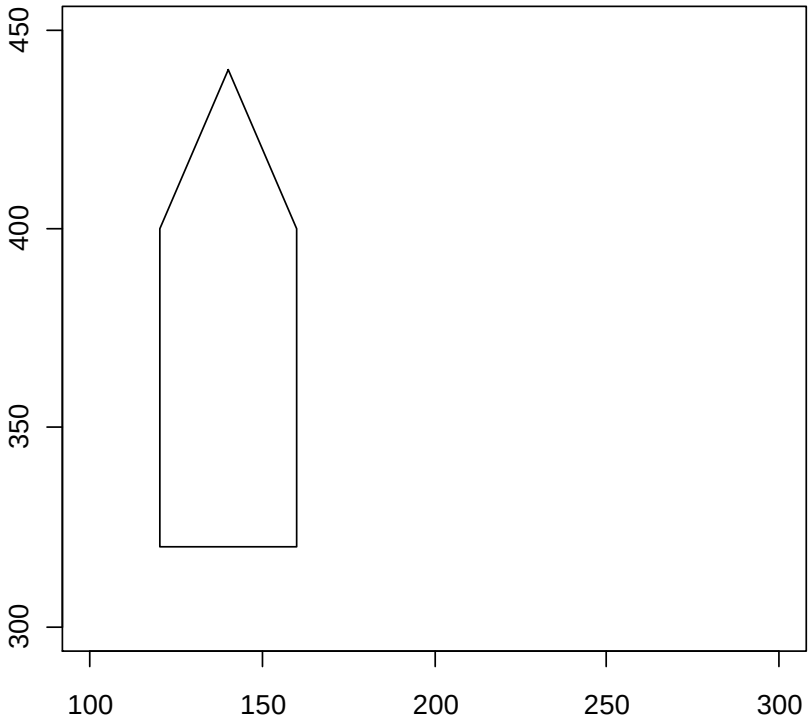
Rectangulars

```
> plot(c(100, 200), c(300, 450),
    type= "n", xlab="", ylab="")
> rect(100, 300, 125, 350)
```



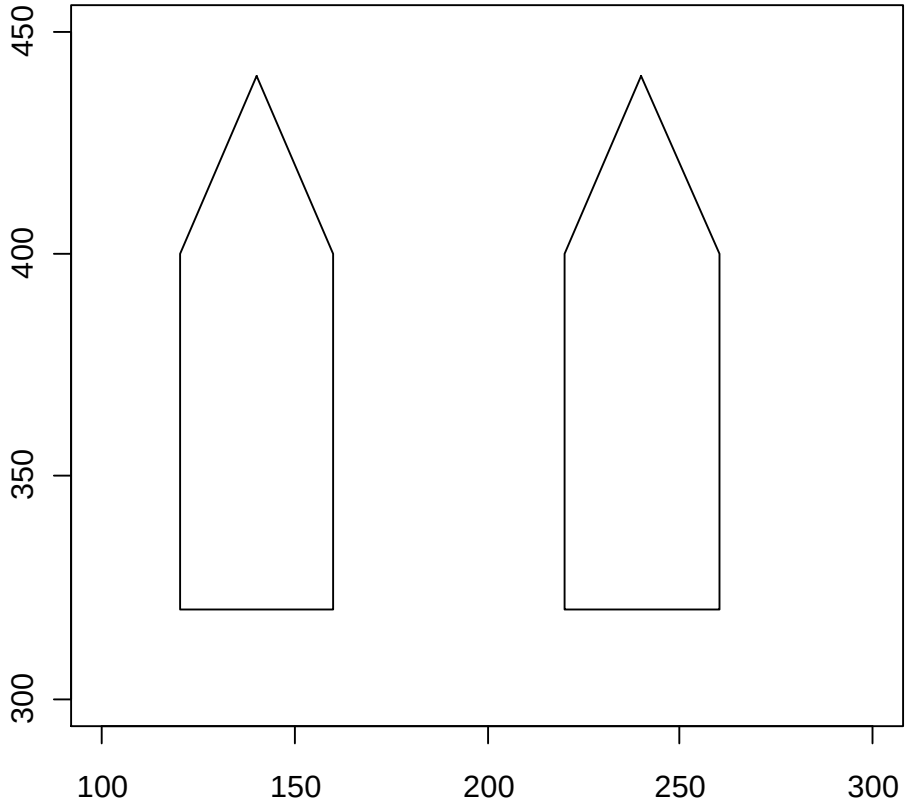
Polygons

```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> polygon(c(140, 120, 120, 160,
    160), c(440, 400, 320, 320,
    400))
```



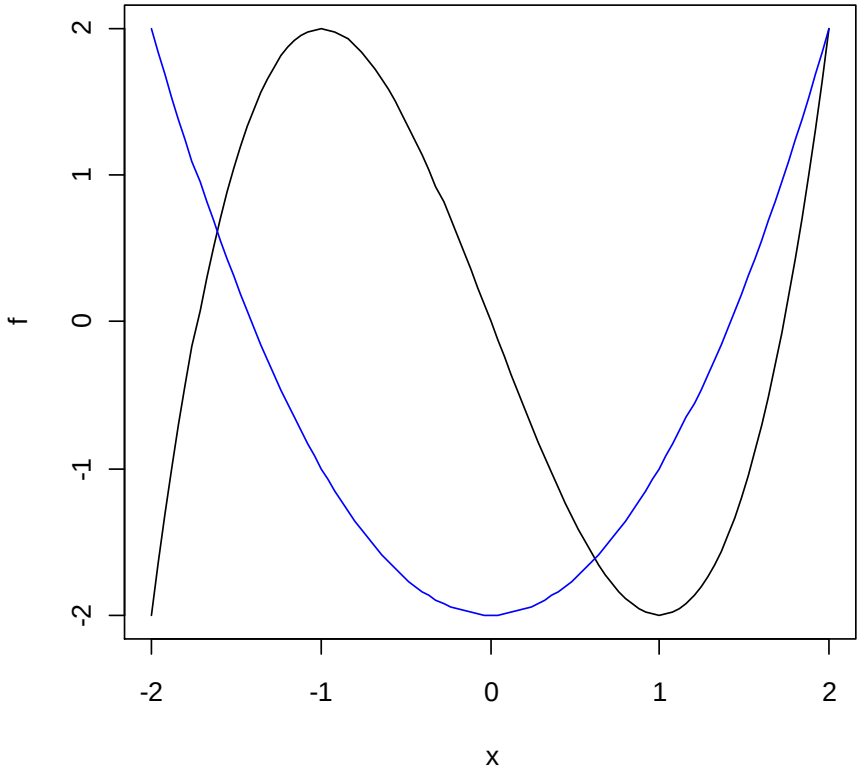
Polygons

```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> polygon(c(140, 120, 120, 160,
    160, NA, 240, 220, 220, 260,
    260),
  c(440, 400, 320, 320, 400, NA,
    440, 400, 320, 320, 400))
```



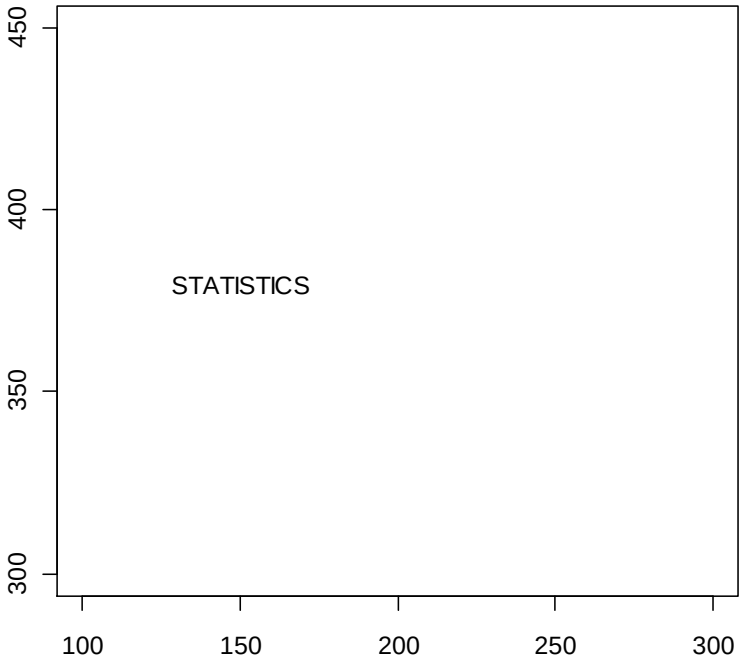
Curves

```
> curve(x^3 - 3*x, -2, 2,
    ylab="f")
> curve(x^2 - 2, add = TRUE, col
    = "blue")
```



Text

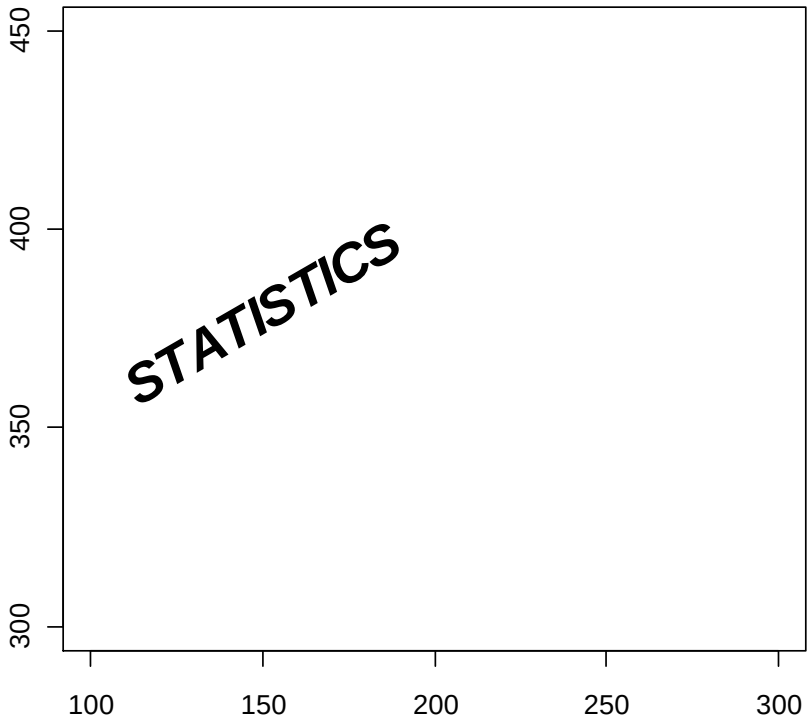
```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> text(150, 380, "STATISTICS")
```



Text

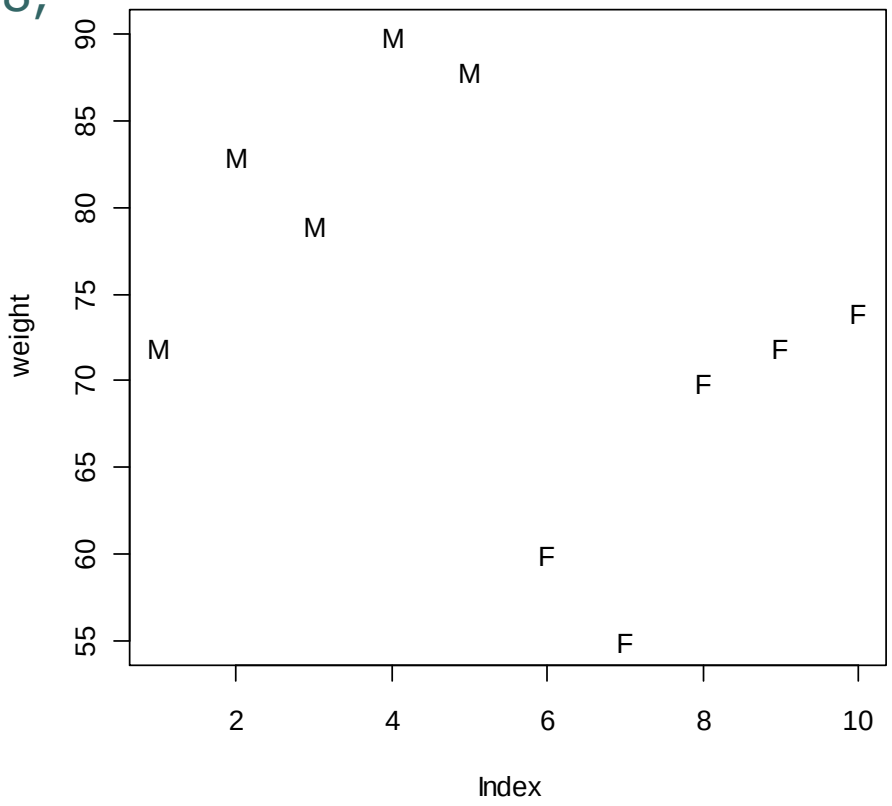
```
> plot(c(100, 300), c(300, 450),
    type= "n", xlab="", ylab="")
> text(150, 380, "STATISTICS",
    srt=30, cex=2, font=4)
```

rotation in degrees



Text

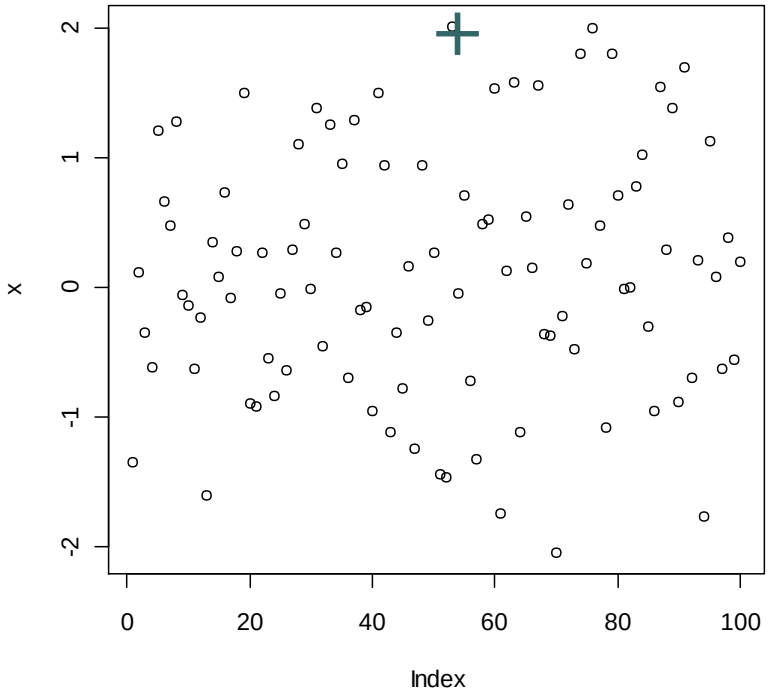
```
> weight<-c(72, 83, 79, 90, 88,
    60, 55, 70, 72, 74)
> gender<-rep(c("M", "F"),
    each=5)
> plot(weight, type="n")
> text(weight, label=gender)
```

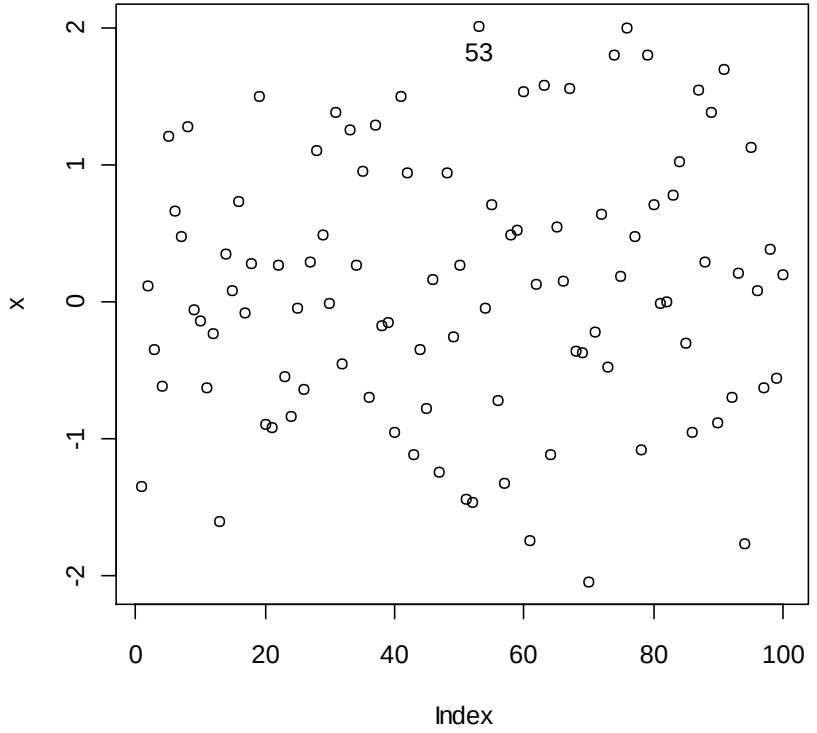


Identify Points in a Scatter plot

```
> plot(x)
> identify(x, n=1)
[1] 53
```

Identify Points in a Scatter plot





Multiple Graphs

- **R** makes it easy to combine multiple plots into one overall graph, using the **par( )** function.

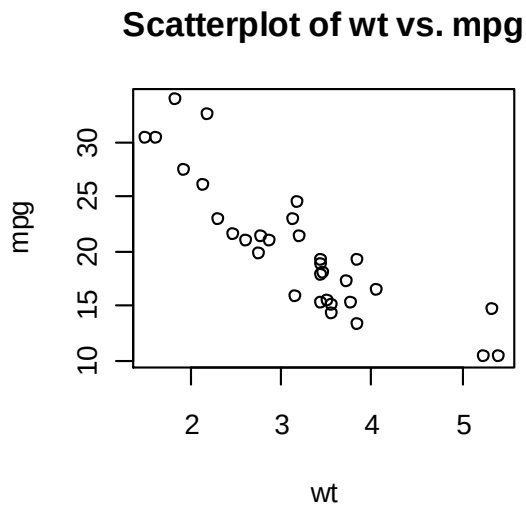
  Within par:

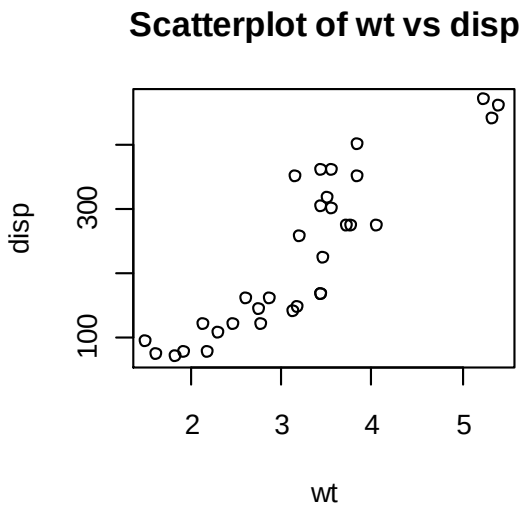
  - **mfrow=c(***nrows*, *ncols***)** creates a matrix of *nrows x ncols* plots that are filled in by row (in the current active device otherwise opens a window).
  - **mfcol=c(***nrows*, *ncols***)** same as mfrow but it fills in the matrix by columns.

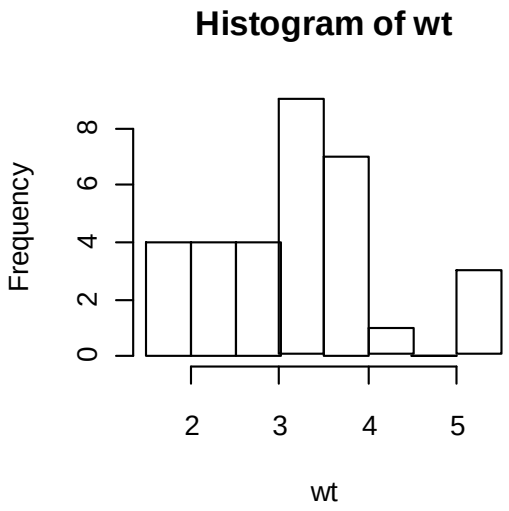
Multiple Graphs

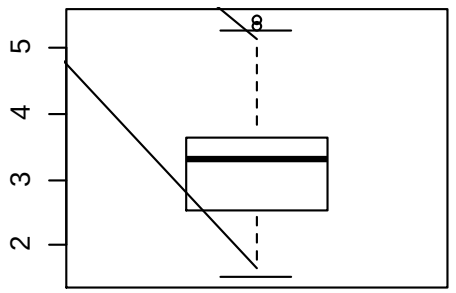
```
# 4 figures arranged in 2 rows and 2 columns
> attach(mtcars) # attaches dataset mtcars
> par(mfrow=c(2,2))
> plot(wt,mpg, main="Scatterplot of wt vs. mpg")
> plot(wt,disp, main="Scatterplot of wt vs disp")
> hist(wt, main="Histogram of wt")
> boxplot(wt, main="Boxplot of wt")
```

Multiple Graphs









Graphs in Higher Dimensions

- Install and Download library ElemStatLearn
- From the Menu or Type:

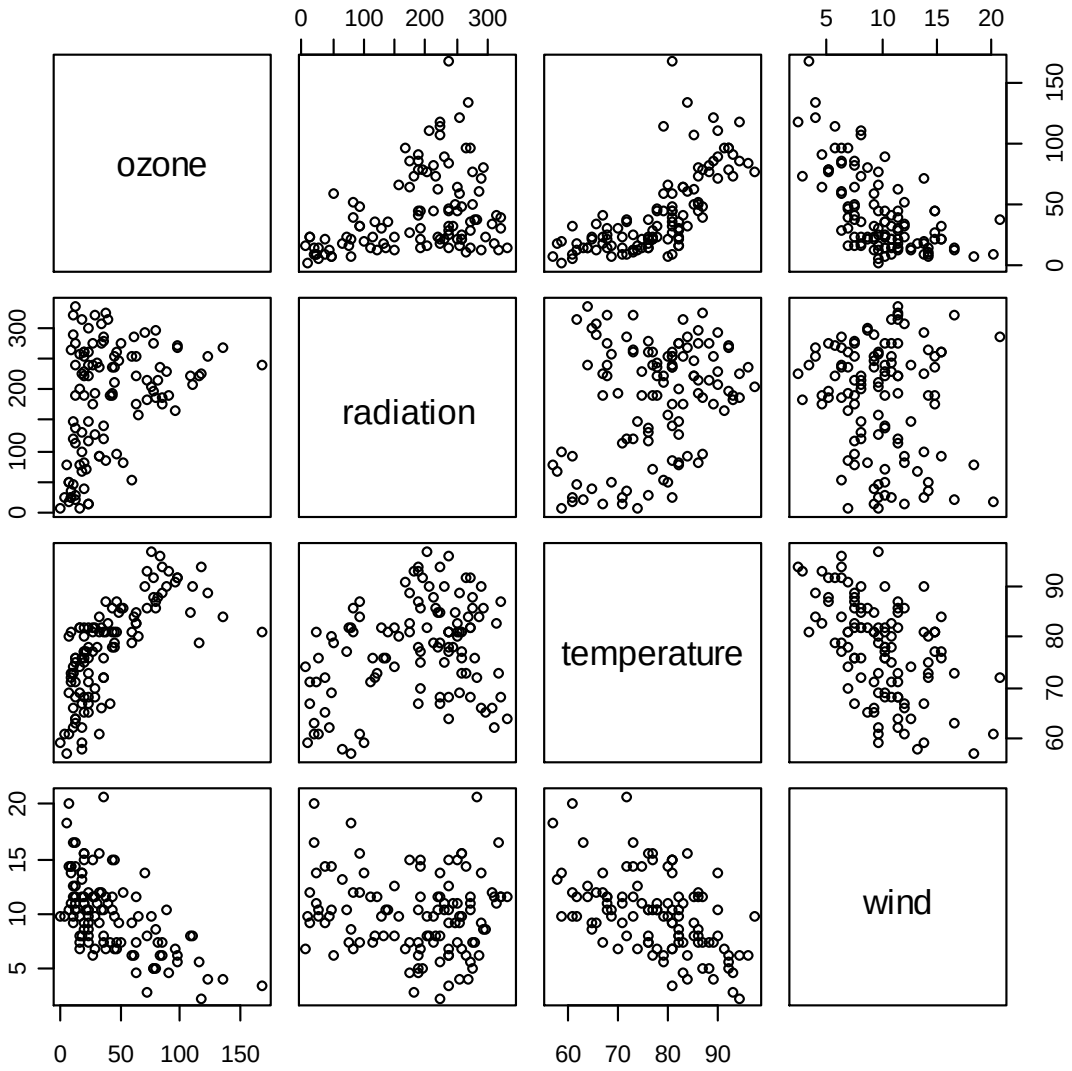
  > install.packages("ElemStatLearn")

  > library (ElemStatLearn)
- Data-set ozone:

  > data(ozone)

  > pairs(ozone)

Scatterplot Matrices

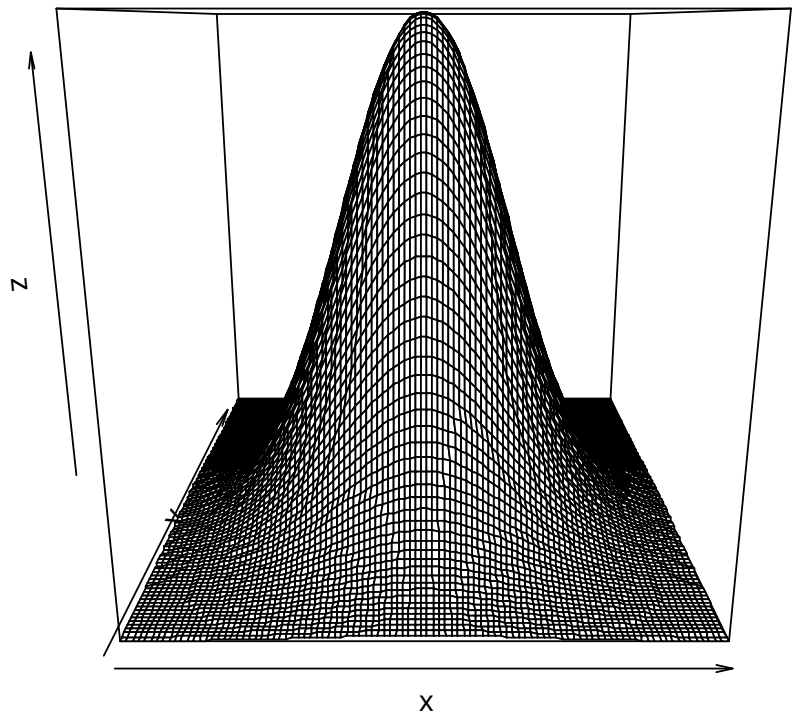


Perspective Plot

The bivariate normal density

```
> f <- function(x, y) {
    z<-(1/(2*pi))*exp(-0.5*(x^2+y^2))
    return(z)
  }
> y <- seq(-3,3, length.out=100)
> x <-seq(-3,3, length.out=100)
> z<-outer(x,y,f)   #compute density for all x, y
> persp(x,y,z)
```

Perspective Plot



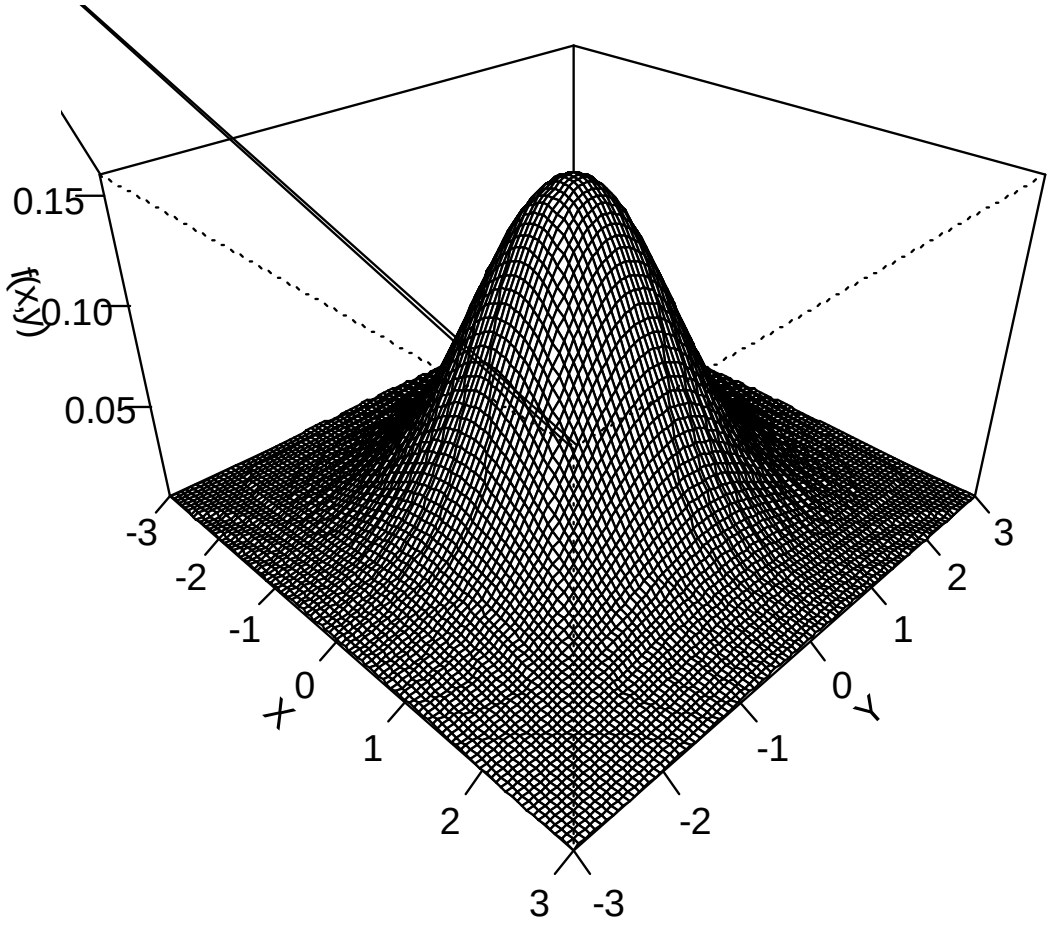
Perspective Plot

Syntax:

```
persp(x,y,z, theta=45, phi=30, expand=0.6,
    ticktype="detailed", xlab="X", ylab="Y", zlab="f(x,y)")
```

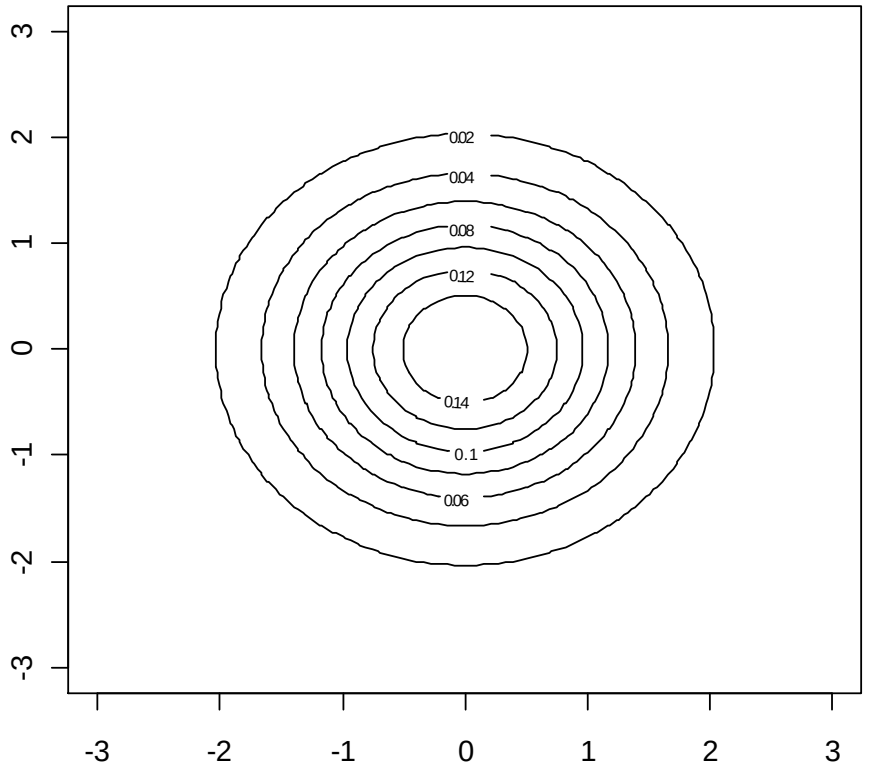
- **theta, Phi**: angles defining the viewing direction.

Perspective Plots



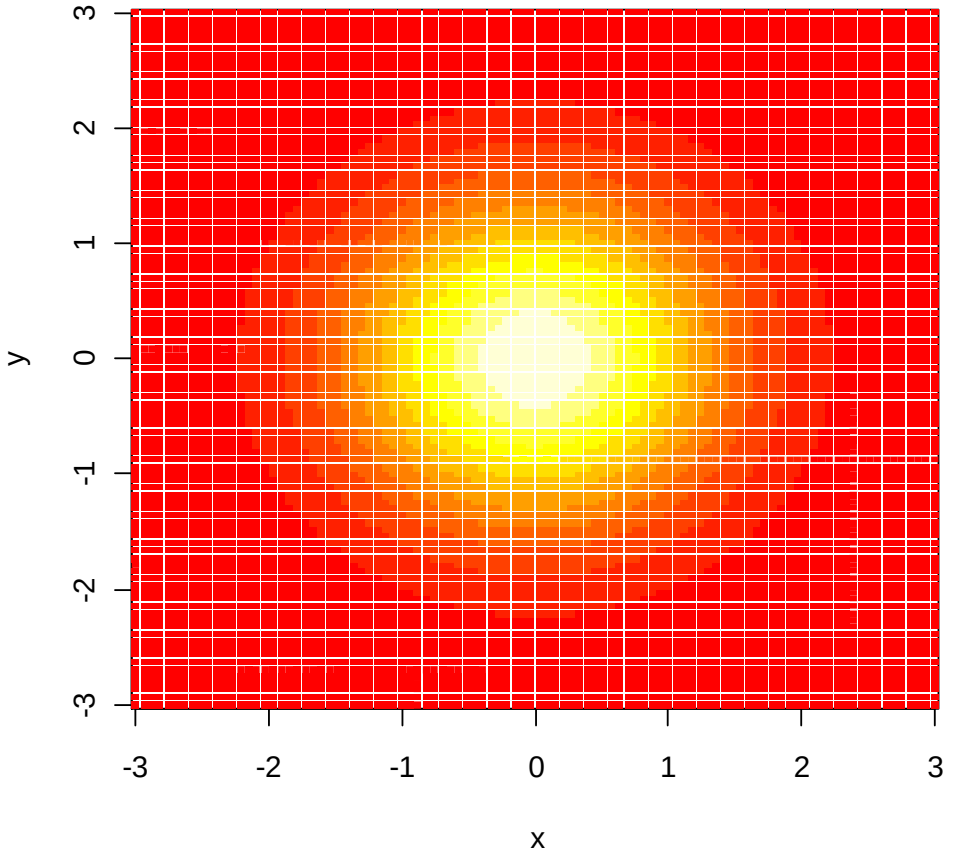
Contour Plots

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



Color Image

> image(x,y,z)



Mathematical Annotation

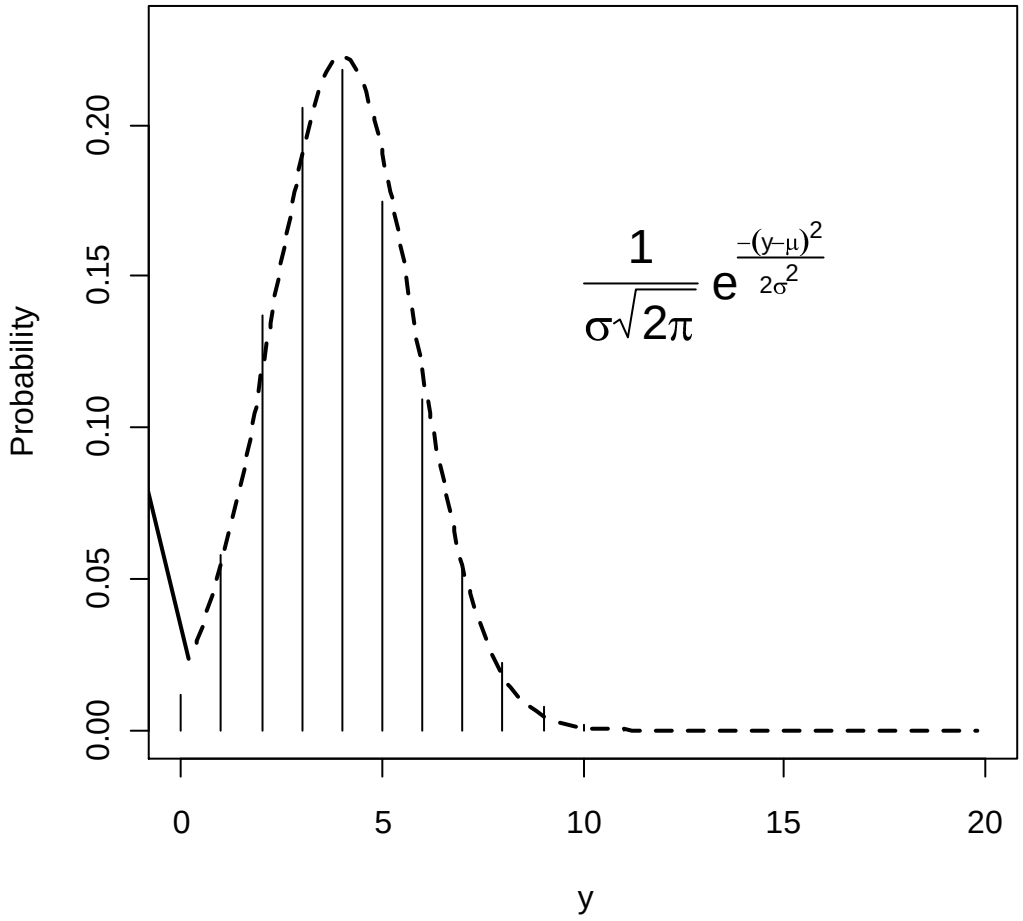
- If the text argument to one of the text-drawing functions (**text, mtext** & **axis**) in R is an expression, the argument is interpreted as a mathematical expression and the output will be formatted according to TeX-like rules.
- Expressions can also be used for titles, subtitles and x- and y-axis labels (but not for axis labels on persp plots).
- For more info type ?plotmath in R.

Mathematical Annotation

An example

```
> n<-20
> p<-0.2
> y<-0:20
> pr<-dbinom(y,n,p)
> plot(y,pr,type="h", xlim=c(0,20), ylim=c(0,0.23),
    ylab="Probability")
> mu = n * p; sigma = sqrt(n * p * (1 - p))
> curve(dnorm(x, mu, sigma), add=TRUE, lwd=2, lty=2)
> text(13, 0.15, expression(paste(frac(1,
    sigma*sqrt(2*pi)), " ", e^{frac(-(y-mu)^2,
    2*sigma^2)})), cex = 1.5)
```

Mathematical Annotation



Distributions in R

- Probability distribution functions usually have four functions associated with them.
- The functions are prefixed with a:
  - **d** for density.
  - **r** for random number generation.
  - **p** for cumulative distribution.
  - **q** for quantile function.

Distributions in R

command	distribution	command	distribution
beta	Beta	hyper	Hypergeometric
norm	Normal	unif	Uniform
pois	Poisson	cauchy	Cauchy
nbinom	Neg. Binomial	weibull	Weibull
gamma	Gamma	chisq	X^2
t	Student	exp	Exponential
binom	Binomial	geom	Geometric
f	Snedecor	mvnorm	Multivariate Normal

Distributions in R

Examples:

```
> pnorm(3,2,2)
[1] 0.6914625
```

Calculates the value of the cumulative distribution function of the Normal distribution with mean 2 and SD (**NOT VARIANCE**) 2 at x = 3.

```
> qgamma(0.3,1,1)
[1] 0.3566749
```

Finds the 0.3 percentile of the Gamma distribution with parameters 1 and 1.

```
> dt(2,3)
[1] 0.06750966
```

Calculates the value of the pdf of the Student distribution with 3 df at x = 2.

```
> runif(5,-2,2)
[1]  1.3448055 -0.4691324  1.2517269  1.5576504  0.9563447
```

Generates 5 uniform in (-2,2) variates.

Distributions in R

Arguments of the previous functions could be vectors.

```
> dexp(1:5,2)
[1] 2.706706e-01 3.663128e-02 4.957504e-03
    6.709253e-04 9.079986e-05
```

Calculates the value of the pdf of the exponential distribution with parameter 2 for x = 1, 2, 3, 4, 5.

Default Values in the Parameters: The command rnorm(50) generates 50 normal variates with $\mu = 0$ and $\sigma = 1$.

Distributions in R

If X~Student(10) then the P(X≤2) is equal to

```
> pt(2,10)
[1] 0.963306
```

while the P(X>2) is equal to

```
> pt(2,10, lower.tail=FALSE)
[1] 0.03669402
```

Distributions in R

All functions (except the random generators) are also available in log scale

```
> pnorm(3,2,2)
[1] 0.6914625
> pnorm(3,2,2, log=T)
[1] -0.3689464
> dt(2,3)
[1] 0.06750966
> dt(2,3, log=T)
[1] -2.695485
```

The normal distribution in R

Four functions:

- dnorm(x, mean = 0, sd = 1, log = FALSE).
- pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE).
- qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE).
- rnorm(n, mean = 0, sd = 1).

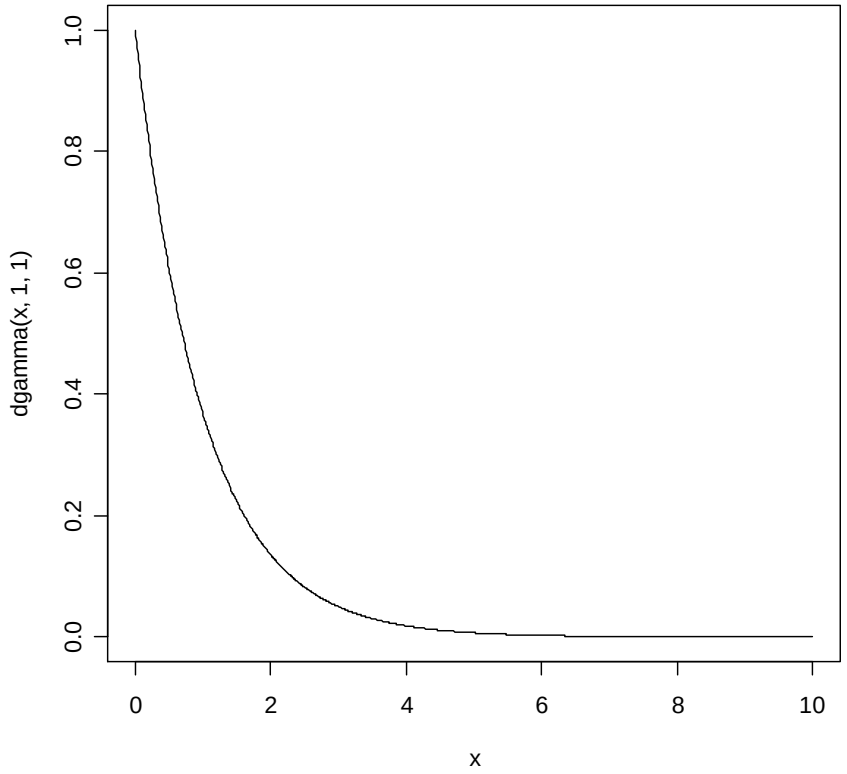
If Φ is the cumulative distribution function for a standard Normal distribution, then pnorm(q) = $\Phi(q)$ and qnorm(p) = $\Phi^{-1}(p)$.

Plotting the probability density function

Plot a pdf or pmf:

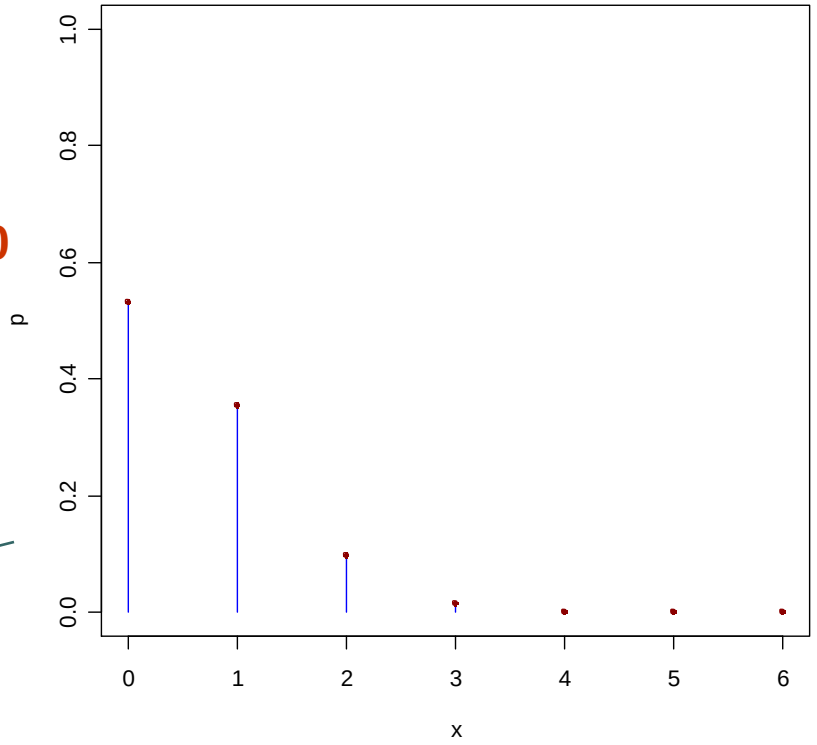
```
> x<-seq(0,10, 0.01)
> plot(x, dgamma(x,1,1),
    type='l')
```

Gamma distribution with parameters (1,1)



Plotting the probability (mass) function

```
> n<-6
> p<-0.1
> x<-0:6
> pr<-dbinom(x,n,p)
>plot(x,pr,type="h",xlim=c(0,6),ylim=c(0
    ,1), col="blue",ylab="p")
>points(x,pr,pch=20,col="dark red")
```



Binomial distribution with n=6 και p=0.1

Simulation of random variables: The Normal

```
> x <- rnorm(10)
> x
[1] 1.38380206 0.48772671 0.53403109 0.66721944
[5] 0.01585029 0.37945986 1.31096736 0.55330472
[9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
[1] 23.38812 20.16846 21.87999 20.73813 19.59020
[6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
18.32 19.73 20.55 20.67 21.67 23.39
```

Simulation of random variables: set.seed

Setting the random number seed with set.seed ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808
[5] 0.3295078
> rnorm(5)
[1] -0.8204684 0.4874291 0.7383247 0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808
[5] 0.3295078
```

Simulation of random variables: The Poisson

```
> rpois(10, 1)
[1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
[1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
[1] 20 11 21 20 20 21 17 15 24 20
> ppois(2, 2) ## Cumulative distribution
[1] 0.6766764 ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347 ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662 ## Pr(x <= 6)
```

Simulate from a Linear Model

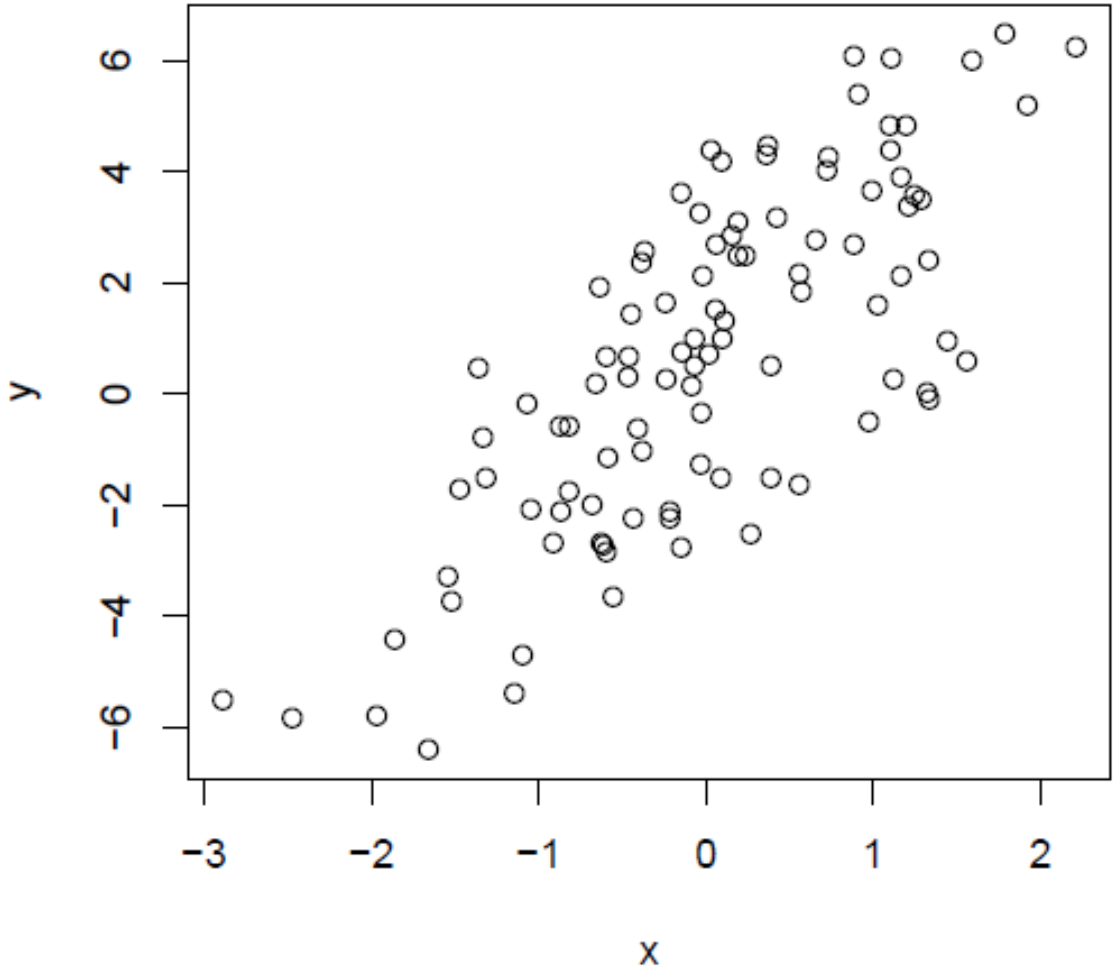
Suppose we want to simulate from a linear regression model:

$$\mathbf{y = \beta_0 + \beta_1 x + \varepsilon, \text{ with } \varepsilon \sim N(0; 2^2)}.$$

Assume $X \sim N(0; 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-6.4080 -1.5400 0.6789 0.6893 2.9300 6.5050
> plot(x, y)
```

Simulate from a Linear Model

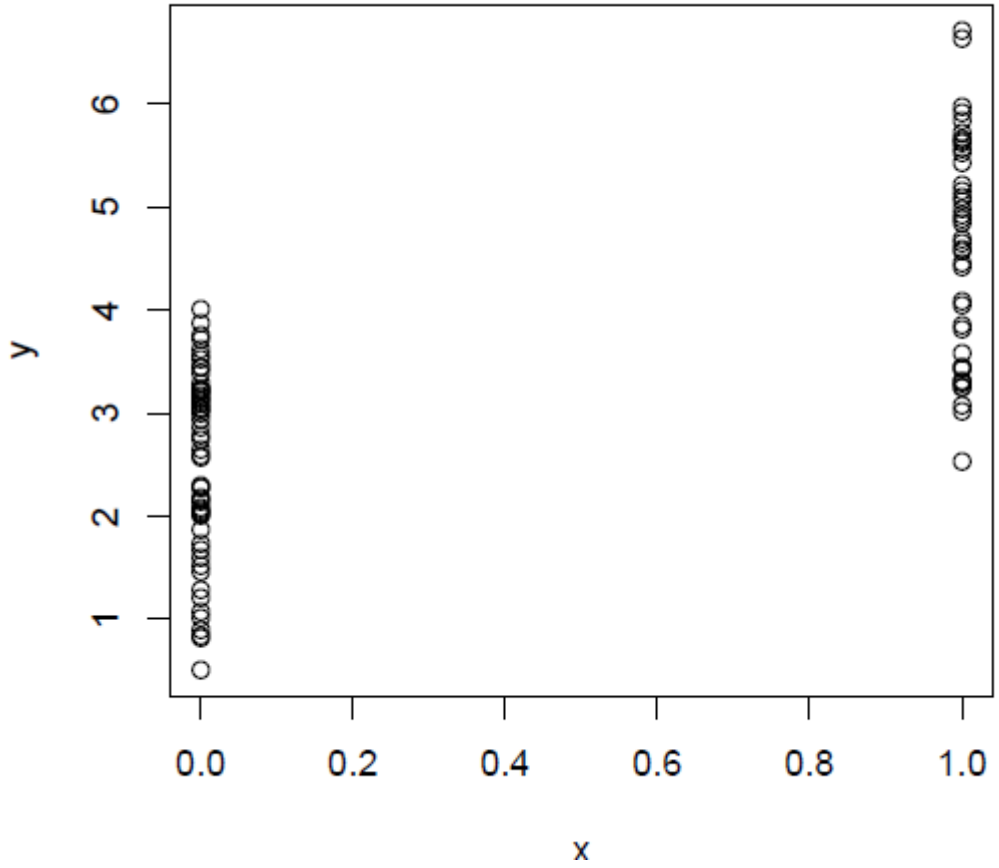


Simulate from a Linear Model

What if X is binary?

```
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-3.4940 -0.1409 1.5770 1.4320 2.8400 6.9410
> plot(x, y)
```

Simulate from a Linear Model



Simulate from a Poisson Model

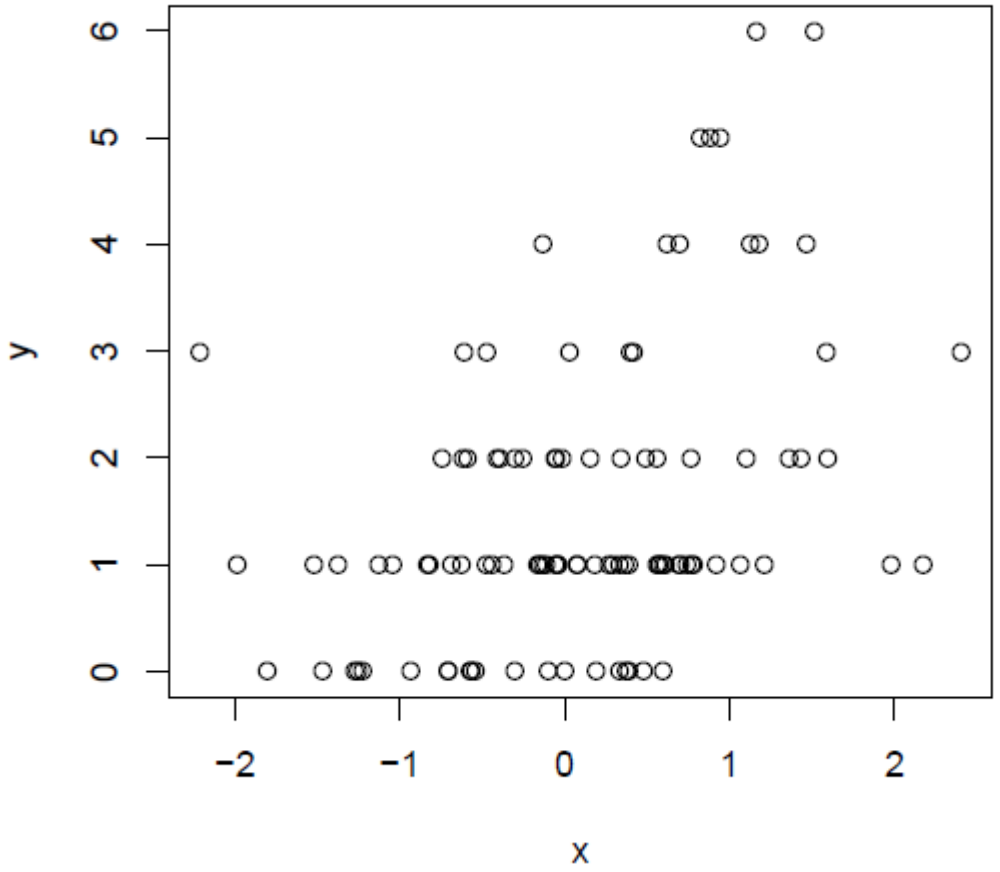
Suppose we want to simulate from a Poisson regression model

$$\mathbf{Y \sim Poisson(\mu) \text{ with } \log(\mu) = \beta_0 + \beta_1 x.}$$

Assume $X \sim N(0; 1^2)$ and $\beta_0 = 0.5$ and $\beta_1 = 0.3$.

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.00 1.00 1.00 1.55 2.00 6.00
> plot(x, y)
```

Simulate from a Poisson Model



Sampling

The **sample** function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

Sampling without replacement:

```
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
```

Sampling

Sampling a random permutation

```
> sample(1:10) ## permutation
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
```

Sampling with replacement

```
> sample(1:10, replace = TRUE) ## Sample
  w/replacement
[1] 2 9 7 8 2 8 5 9 7 8
> sample(1:10, 3, replace = TRUE)
[1] 3 9 1
```

Sampling with replacement and unequal probabilities

```
> sample(1:5, replace = TRUE, prob=c(0.3, 0.3, 0.4,
  0,0))
[1] 3 1 2 2 1
```

Conclusion