

Lecture 01

An introduction to R

Csci E63 Big Data Analytics

Zoran B. Djordjević

About this course

- This is an IT (Information Technology) course.
- This is not a course in Statistics, Mathematics nor Artificial Intelligence.
- We will teach you how to install and use most important frameworks used for modern analysis of big data sets.
- We will demonstrate some sophisticated statistical and machine learning algorithms. However, they are not the main focus of this course.

Objectives of this lecture

- Review the basic concepts of R as a programming language.
- Get accustomed to a language with counter intuitive syntax.
- Gain skills in using a tool that large number of people and a large body of literature uses.

Why R

- Would you rather use T?
- It appears that R and not T is the mandatory entry on the resume of every data scientist.
- Who is a “data scientist”?
- *Definition: “a data scientist is a person who has R on her resume”.*
- Please note: You will become a Big Data scientist.
- You need a prototyping or modeling tool. Some people use MatLab, some use Python. Statisticians use R.
- U R a Statistician. R U not?
- Note: More and more statisticians are starting to use Python.

R, S and S-plus

- R has a long history.
- R is based on S, an interactive environment for data analysis developed at Bell Laboratories since 1976
 - 1988 - S2: RA Becker, JM Chambers, A Wilks
 - 1992 - S3: JM Chambers, TJ Hastie
 - 1998 - S4: JM Chambers
- Exclusively licensed by *AT&T/Lucent* to *Insightful Corporation*, Seattle WA. Product name: “S-plus”.
- Implementation languages are C, and Fortran.

See:

<http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html>

R, S and S-plus

- R: initially written by Ross Ihaka and Robert Gentleman at Dept. of Statistics of U of Auckland, New Zealand during 1990s.
- Since 1997: The language is maintained and developed by the international, “R-core” , team of some 15 people with access to common CVS archive.
- GNU General Public License (GPL)
 - can be used by anyone for any purpose
 - contagious
- Open Source
 - quality control!
 - efficient bug tracking and fixing system supported by the user community

Things R does and What R does not do

- data handling and storage: numeric, textual
- matrix algebra
- hash tables and regular expressions
- high-level data analytic and statistical functions
- classes (“Object Oriented”)
- graphics
- programming language: loops, branching, subroutines
- is not a database, but connects to DBMSs
- has no graphical user interfaces, however it connects to Java, TclTk and it has **R Studio**.
- language interpreters are not fast. However, R could be extended by compiled C/C++ code
- no spreadsheet view of data, but connects to MS Excel
- no professional / commercial support

Statistical Packages

- Packaging: a crucial infrastructure to efficiently produce, load and keep consistent software libraries from (many) different sources / authors
- Most R packages deal with statistics and data analysis
- Many statistical researchers publish their state of the art methods as R packages.
- Comprehensive R Archive Network (CRAN) is a place where you can fetch those packages for free. You can get truly powerful tools at CRAN.

R Learning Curve and Approach

- In R, simple things are simple and complex things are complex.
- Some complexity is somewhat artificial and is caused by sometimes difficult to penetrate terminology used by Statisticians. Once you understand the language of Statistics, R becomes more understandable.
- Some of R syntax will never become understandable to some people.
- R is an “experimental language”. The only way to figure out how some commands work is to experiment. Help is helpful.
- The learning curve is somewhat steep.
- The language is built for procedural processing. You run a command at a time. Examine results and run another command.
- You should not plan to build complex business applications with R. Some people do that, though.

Where to get R and R Studio

- Download R for Windows, Mac-OS or Linux from <http://cran.r-project.org/>
- If you like command line interface, you do not need more than that.
- If you prefer an IDE, download **R Studio** from <http://www.rstudio.com/>
- You will keep on fetching packages (libraries) from the CRAN <http://cran.r-project.org/> site.
- Run R installation first, then install R Studio. That is all.
- When we use R, we will use R Studio, except in rare circumstances.
- Eclipse plugins for R exist: <http://www.walware.de/goto/statet>

R Studio

The screenshot displays the RStudio environment with the following components:

- Source Editor:** Contains R code for installing packages and setting the working directory.

```
1 install.packages("arm")
2 install.packages("glmnet")
3 install.packages("igraph")
4 install.packages("lme4")
5 install.packages("lubridate")
6 install.packages("reshape")
7 install.packages("RJSONIO")
8 setwd('01-Introduction')
9 source('ufo_sightings.R')
10 setwd('..')
11
12 setwd('02-Exploration')
13 source('chapter02.R')
14 setwd('..')
15
```
- Console:** Shows the R startup message and the results of the commands entered in the source editor.

```
C:\CLASSES/code/R/
you are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

[workspace loaded from C:/CLASSES/code/R/.RData]

> ?R
No documentation for 'R' in specified packages and libraries:
you could try '??R'
> ??R
> ?manual
No documentation for 'manual' in specified packages and
libraries:
you could try '??manual'
> ??manual
> ??instruction
>
```
- Workspace:** Lists the objects in the current workspace.

Object	Details
M	3x4 integer matrix
fpd	20 obs. of 4 variables
fpd2	20 obs. of 4 variables
fpdatt	20 obs. of 4 variables
fpe	20 obs. of 3 variables
- Help Panel:** Displays the RStudio help page, titled "Statistical Data Analysis". It includes links to various resources:
 - 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](#)

R Studio

- The bottom left region is called Console. Console displays commands and results. You can type commands directly into the console.
- The top left region is the text editor. Open it with File > New File > R Script. Editor has syntax highlighting, parentheses matching and other features. You can open scripts (collections of commands) into text editor. You can highlight and execute individual commands or groups of commands from the text editor. You should get into the habit of always typing your commands in the editor.
- Top right region has command history and existing variables
- Bottom right region displays Help pages. Hit Help tab to get to the main help page.
- “An Introduction to R” and other manuals are decent reads.

R as a Calculator

```
> log2(32)
```

```
[1] 5
```

```
> sqrt(2)
```

```
[1] 1.414214
```

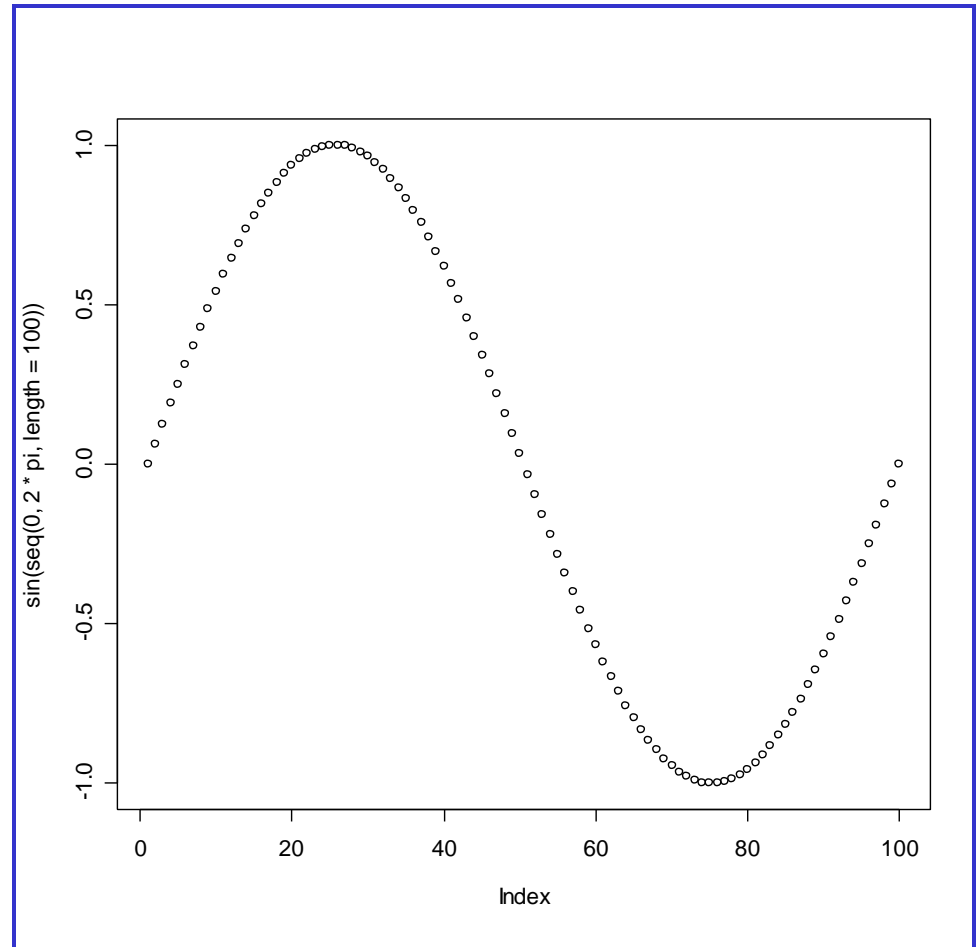
```
> seq(0, 5, length=6)
```

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

```
> plot(sin(seq(0, 2*pi, length=100)))
```

- To quit R, type

```
> q()
```



R Workspace

- When you close the R Studio or the R console window, the system asks if you want to save the workspace image.
- If you select to save the workspace all the objects in your current R session are saved in a binary file .Rdata located in the working directory of R.
- During your R session you can also explicitly save the workspace image to the current working directory
 - > `save.image()`
- To check what the current working directory is type
 - > `getwd()`
- To save to a specific file and specific location, type
 - > `save.image("C:\\CLASSES\\R\\2.RData")`
- To set the current working directory type
 - > `setwd("C:\\CLASSES\\R")`
- If you saved a workspace, the next time you start R, you can load(it).
 - > `load("C:\\CLASSES\\R\\2.RData")`
- All previously saved objects are available again.

R looks like it is written in Java, Help

- R gets confused if you use a path like
`c:\mydocuments\myfile.txt`
- This is because R sees "\" as an escape character. Instead, use
`c:\\my documents\\myfile.txt`
or
`c:/mydocuments/myfile.txt`
- *Most R functions are overloaded.*
- *It is hard to remember all variations, use*
`help(functionName)` , *e.g.*
`> help(save)`

Objects

- Objects in R have names.
 - Names start with a letter (A-Z or a-z),
 - can contain letters, digits (0-9), and/or periods “.”
 - case-sensitive `mydata` is different from `MyData`
 - do not use underscore “_”
- Types of objects: `vector`, `factor`, `array`, `matrix`, `data.frame`, `list`, `functions`, `classes`
- Objects have attributes
 - mode: numeric, character, complex, logical
 - length: number of elements in the object

Assignment and Variable Types

- R uses operators `->` or `<-` to assign values.
- `Sign =` is a substitute for `<-`
- Atomic variables could be numeric, character or logical

```
> a <- 49 -> z
```

numeric

```
> sqrt(a)
```

```
[1] 7
```

```
> a = "The dog ate my homework"
```

character
string

```
> sub("dog", "cat", a)
```

```
[1] "The cat ate my homework"
```

```
> a = (1+1==3)
```

logical

```
> a
```

```
[1] FALSE
```

Variables could be Missing Values

- Variables of each data type (numeric, character, logical) can also take the value **NA**: not available.
 - NA is not the same as 0
 - NA is not the same as ""
 - NA is not the same as FALSE
- Operations (calculations, comparisons) that involve NA may or may not produce NA:

```
> NA==1
```

```
[1] NA
```

```
> 1+NA
```

```
[1] NA
```

```
> max(c(NA, 4, 7))
```

```
[1] NA
```

```
> max(c(NA, 4, 7), na.rm=T)
```

```
[1] 7
```

```
> NA | TRUE
```

```
[1] TRUE
```

```
> NA & TRUE
```

```
[1] NA
```

na.rm is a logical indicator, means remove missing values

Predefined functions

- R comes with some 30+ packages. To see which ones are there, at the command prompt, type

```
> library()
```
- R comes with a set of useful demos. To see which ones are there, at the command prompt, type

```
> demo()
```
- R also comes with a number of datasets which are used in some demos and come quite handy for testing your own procedures. To see a list of provided datasets, type

```
> data()
```
- To see all variables in the workspace, type

```
> ls()
```
- To remove some of those variables, provide a comma separated list of variables to function `rm()`, e.g.

```
> rm(d, fpdat, z)
```

Functions and Operators

- **Functions** do things with data. Functions have
 - “Inputs”: i.e. function arguments (0,1,2,...) and an
 - “Output”: function result (exactly one)

We could **define functions, for example** `add()` :

```
> add = function(a,b)
  { result = a+b           // "+" in the left column,
    return(result) }      // indicates continuation of
                          // a command

> add(2,3)
> [1] 5
```

- **Operators** are short-cut writings for frequently used functions of one or two arguments.

Examples: + - * / ! & | % %

Functions and Operators

- Functions do things with data
 - “Input”: function arguments (0,1,2,...)
 - “Output”: function result (exactly one)

Exceptions to the rule:

- Functions may also use data that sits around in other places, not just in their argument list: “scoping rules”
- Functions may also do other things than returning a result, e.g., plot something on the screen: “side effects”
- Functions could be a target of an assignment.

Vectors, Matrices, Arrays

- R deals with data and data are usually organized as vectors, matrixes or arrays.
- Vector
 - Ordered collection of data of the same data type
 - Example:
 - last names of all students in this class
 - A single number is a vector of length 1
- Matrix
 - Rectangular table of data of the same type
 - Example
 - Mean intensities of all genes measured during a microarray experiment
- Array
 - Higher dimensional matrix

Vectors, Matrices and Arrays

- **Vector:** an ordered collection of data of the same type

```
> a = c(1, 2, 3)
```

```
> a*2          // operation on a vector
```

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

- Function `c()`, constructs vectors. You may remember it as `concatenate()`, if you do not find that confusing, or `combine()` if less confusing
- In R, a single number is the special case of a vector with 1 element.
- Other vector types: character strings, logical vectors

Vectors

```
> Mydata <- c(2,3.5,-0.2) #Vector constructed with c() function
> Colors <-
  c("Red","Green","Red") #Character vector

> x1 <- 25:30 #Vector constructed with Range operator :
> x1
[1] 25 26 27 28 29 30 # Number sequences

> Colors[2]
[1] "Green" #Select One element with []
# Note, R counts from 1 not 0

> x1[3:5]
[1] 27 28 29 # Selected several elements with range
# of index values
```


Operations on Vector Elements

```
> Mydata
```

```
[1] 2 3.5 -0.2
```

Print vector Mydata

```
> Mydata > 0
```

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

Logical test on the elements

produces a vector of logical values

Extract the positive elements using

logical vector as indexes

```
> Mydata[Mydata>0]
```

```
[1] 2 3.5
```

```
> Mydata[-c(1,3)]
```

```
[1] 3.5
```

Minus sign in front of an index or

indexes removes element(s)

Operations on Vectors

```
> x <- c(5, -2, 3, -7)
```

```
> y <- c(1, 2, 3, 4) * 10
```

Operation on all the elements

```
> y
```

```
[1] 10 20 30 40
```

```
> sort(x)
```

Sorting a vector

```
[1] -7 -2 3 5
```

```
> order(x)
```

order() tells us how are elements

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

ordered in sorting

```
> y[order(x)]
```

```
[1] 40 20 30 10
```

Print elements of y using

indexes produced by order(x)

```
> rev(x)
```

Reverse vector x

```
[1] -7 3 -2 5
```

Matrixes

- Matrix: Rectangular table of data of the same type

```
> m <- matrix(1:12, nrow=4, byrow = TRUE); m
      [,1] [,2] [,3] # byrow=T means that matrix
[1,]     1     2     3 # is filled by rows, otherwise
[2,]     4     5     6 # is filled by columns
[3,]     7     8     9
[4,]    10    11    12
> y <- -1:2          ## vector y will be added to
                    ## every column of m
> m.new <- m + y      ## m.new is a new matrix
> t(m.new)           ## transpose function t()
      [,1] [,2] [,3] [,4]
[1,]     0     4     8    12
[2,]     1     5     9    13
[3,]     2     6    10    14
> dim(m)
[1]  4  3
> dim(t(m.new))      ## dot in m.new is just part of the name
[1]  3  4
```

Matrices

Matrix: Rectangular table of data of the same type

```
> x <- c(3,-1,2,0,-3,6)
> x.mat <- matrix(x,ncol=2)      ## Matrix with 2 cols
> x.mat
```

```
      [,1] [,2]
[1,]     3     0
[2,]    -1    -3
[3,]     2     6
```

```
> x.mat <- matrix(x,ncol=2, byrow=T)
## By row creation
> x.mat
## means the sequence
## is broken into rows
```

```
      [,1] [,2]
[1,]     3    -1
[2,]     2     0
[3,]    -3     6
```

Fetching Data from Matrices

```
> x.mat[,2]          ## 2nd col  
[1] -1 0 6
```

```
> x.mat[c(1,3),]     ## 1st and 3rd rows
```

```
      [,1] [,2]  
[1,]    3  -1  
[2,]   -3    6
```

```
> x.mat[-2,]         ## Give us matrix, exclude 2nd row
```

```
      [,1] [,2]  
[1,]    3  -1  
[2,]   -3    6
```

Dealing with Matrices

```
> dim(x.mat)
```

Dimension

```
[1] 3 2
```

```
> t(x.mat)
```

Transpose

```
      [,1] [,2] [,3]
[1,]     3     2    -3
[2,]    -1     0     6
```

```
> x.mat %*% t(x.mat)
```

Matrix Multiplication

```
      [,1] [,2] [,3]
[1,]    10     6   -15
[2,]     6     4    -6
[3,]   -15    -6    45
```

•Quick quiz: `x <- c(1,2,3)` a vertical or horizontal vector

```
> solve(a)
```

Inverse of a square matrix

```
> eigen()
```

Eigenvectors and eigenvalues

Missing Values, again

- R is designed to handle statistical data and therefore predestined to deal with missing values
- NA are values that are “not available”

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

```
> x + 3
```

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

- “NaN” Not a number and “Inf” are somewhat different

```
> log(c(0, 1, 2))
```

```
[1]      -Inf 0.0000000 0.6931472
```

```
> 0/0
```

```
[1] NaN
```

Review of Subsetting

- It is often necessary to extract a subset of a vector or matrix
- R offers a couple of neat ways to do that

```
> x <- c("a", "b", "c", "d", "e", "f", "g", "h")
> x[1]
> x[3:5]
> x[-(3:5)]
> x[c(T, F, T, F, T, F, T, F)]
> x[x <= "d"]
> m[,2]
> m[3,]
```


Lists

- **Vector:** an ordered collection of data of the same type.

```
> a = c(7,5,1)
```

```
> a[2]
```

```
[1] 5
```

- **List:** an ordered collection of data of arbitrary types. Lists have elements, each of which can contain any type of R object, i.e. the elements of a list do not have to be of the same type.
- List elements are accessed through different indexing operations.

```
> doe = list(name="john", age=28, married=F)
```

```
> doe$name
```

```
[1] "john "
```

```
> doe[3]
```

```
[1] FALSE
```

Lists

- A component of a list can be referred as `aa[[I]]` or `aa$times`. Here `aa` is the name of a list, `I` is the position of the component we are extracting and `times` is the name of a component of `aa`.
- The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely.
- `aa[[1]]` is the first component of `aa`, while `aa[1]` is the sublist consisting of the first component of `aa` only.
- There are functions whose return value is a List. We have seen some of them: `eigen`, `max`, `min`, `svd`, ...

Lists are very flexible

- List can contain a numeric vector as one component and a character vector as the other. The following is a list with anonymous components

```
> my.list <- list(c(5,4,-1),c("X1","X2","X3"))
```

```
> my.list
```

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

```
[1] 5 4 -1
```

```
[[2]]:
```

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

```
> my.list[[1]]
```

```
[1] 5 4 -1
```

- You can name components of your list, and access them by their names

```
> my.list <- list(c1=c(5,4,-1),c2=c("X1","X2","X3"))
```

```
> my.list$c2[2:3]
```

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

Rename Components, Convert to Vector

```
Empl <- list(employee="Anna", spouse="Fred", children=3,  
             child.ages=c(4,7,9))
```

- You can change names of components of a list. Rather than `employee`, `spouse`, `children` and `child.ages`, those names could, for example, be the first 4 letters (a,b,c,d).
- Rename component names by assigning those letters to the function `names` (`Empl`), like

```
names(Empl) <- letters[1:4] # print new Empl to see the effect
```

- You can extend a list with new components

```
Empl <- c(Empl, service=8)
```

- You can concatenate lists

```
newList <- c(firstList, secondList)
```

- You can convert a list to a vector. Mixed types will be converted to character, giving a `character` vector

```
unlist(Empl).
```

Extracting a Slice of a List

- For example, the following variable `x` is a list containing copies of three vectors `n`, `s`, `b`, and a numeric value 3.

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc", "dd", "ee")
> b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
> x = list(n, s, b, 3) # x contains copies of n, s, b
```

- A slice is a copy of one or several components of a list.
- We retrieve a list slice with the single square bracket "`[]`" operator. The following is a slice containing the second member of `x`, which is a copy of `s`.

```
x[2]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"
```

- Please note that you cannot modify a slice

Slice with multiple components of a List

- With an index vector, we can retrieve a slice with multiple members. Here is a slice containing the second and fourth members of x.

```
> x[c(2, 4)]  
[[1]]  
[1] "aa" "bb" "cc" "dd" "ee"
```

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

Referencing a Component of a List

- To reference a member of the list and modify it, we have to use the double square bracket `[[]]` operator. `x[[2]]` is the second member of `x`. `x[[2]]` is a copy of `s`, but is not a slice containing `s`

```
> x[[2]]  
[1] "aa" "bb" "cc" "dd" "ee"
```

- We can modify the content of the referenced component directly.

```
> x[[2]][5] = "ff"  
> x[[2]]  
[1] "aa" "bb" "cc" "dd" "ff"
```

- If the referenced component is modified, the list `x` itself is modified

```
> x  
[[1]]  
[1] 2 3 5  
[[2]]  
[1] "aa" "bb" "cc" "dd" "ff"  
[[3]] [1] TRUE FALSE TRUE FALSE FALSE  
[[4]] [1] 3
```

Naming Rows and Columns of a Matrix

- Consider matrix **x.mat**:

```
> x.mat
```

	[,1]	[,2]
[1,]	3	-1
[2,]	2	0
[3,]	-3	6

- You can name rows and columns of a matrix using a list with two components (names of rows and names of columns)

```
➤ dimnames(x.mat) <-  
  list(c("R1", "R2", "R3"), c("C1", "C2"))
```

```
> x.mat
```

	C1	C2
R1	3	-1
R2	2	0
R3	-3	6

Factors

- A factor is a vector object used to specify a set of discrete values (categories, enumerations) appearing as results of certain measurement, e.g.

`Gender {male, female}, Income {low, medium, high}`, etc.

- For efficiency, factors are stored as numbers but have character labels for display. Efficiency is noticeable for large data sets.
- Consider a list of students in a class by gender:

```
class<-c("male", "female", "female", "male", "male", "female")
```

- Apply function `factor()` and place the result in a new variable

```
> student.gender <- factor(class)
```

```
> student.gender
```

```
[1] male female female male male female
```

```
Levels: female male
```

```
> str(student.gender)    # str() displays structure of object
```

```
Factor w/ 2 levels "female","male": 2 1 1 2 2 1
```

```
> class(student.gender)  [1] "factor"
```

```
> mode(student.gender)   [1] "numeric"
```

```
> levels(student.gender) [1] "female" "male"
```

```
> labels(student.gender) [1] "1" "2" "3" "4" "5" "6"
```

Factors and `tapply()`

- Let us look at grades in the same class

```
>class <- c("male","female","female","male","male","female")
>grades <- c(4,3,4,3,3,4)
```
- To calculate sample mean grade for each gender, we can use function `tapply()`

```
> grades.mean <- tapply(grades, student.gender, mean)
> grades.mean
> female male
3.666667 3.333333
```
- Function `tapply()` is used to apply a function, here `mean()`, to each group of components of the first argument, here `grades`, defined by the levels of the second component, `student.gender`.
- `tapply()` also works when its second argument is not a factor, e.g., `tapply(grades, class, mean)`.
- This is true for many similar functions, since arguments are coerced to factors when necessary (using `as.factor()`).

factor()

- The function `factor()` is used to encode a vector as a factor (a set of 'categories' or 'enumerated types'). If argument `ordered` is `TRUE`, the factor levels are assumed to be ordered.
`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

Usage

```
factor(x = character(), levels, labels = levels, exclude = NA,  
ordered = is.ordered(x))
```

x	a vector of data, usually taking a small number of distinct values
levels	an optional vector of the values that x might have taken. The default is the unique set of values taken by <code>as.character(x)</code> , sorted into increasing order of x
labels	either an optional vector of labels for the levels (in the same order as levels after removing those in <code>exclude</code>), or a character string of length 1.
exclude	a vector of values to be excluded when forming the set of levels
ordered	logical flag to determine if the levels should be regarded as ordered

Data Frames

- **Data Frame** represents the typical data table with rows and columns, like a spreadsheet. Data frame is the central type in R.
- Data within each column (variable, component) have the same type (e.g. number, text, logical). Different columns may have different types.
- Both rows and columns have human readable names
- The components must be vectors (numeric, character, or logical), factors, numeric matrices, lists, or other data frames.
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Numeric vectors, logicals and factors are included as is, and character vectors are coerced to be factors, whose levels are the unique values appearing in the vector.
- Vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.

Making Data Frames

- Let us add vector `income` to the description of our student class
- ```
> income <-c(45000, 34500, 67000, 42000, 81000, 53000)
```
- We will construct a data frame `students` using components (variables):  
`student.gender`, `grades` and `income`

```
> students <- data.frame(gender=student.gender,
grade=grades, householdincome=income)
```

```
> students
```

|   | gender | grade | householdincome |
|---|--------|-------|-----------------|
| 1 | male   | 4     | 45000           |
| 2 | female | 3     | 34500           |
| 3 | female | 4     | 67000           |
| 4 | male   | 3     | 42000           |
| 5 | male   | 3     | 81000           |
| 6 | female | 4     | 53000           |

# Data Frames

- You have noticed that names of data frame components: `gender`, `grade` and `housholdincome` appear as names of columns or variables in the printout of the students data frame.
- We could display those “column” names using function `names()`

```
> names(students)
```

```
[1] "gender" "grade" "householdincome"
```

- We could fetch rownames using `rownames()` or `row.names()`

```
> row.names(students)
```

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

- We do not treat students as numbers. Let us change `rownames`

```
> row.names(students) <- c("John", "Mary", "Dianna", "Bob", "Mike", "Joann")
```

- Now, when we ask for `rownames`, we get

```
> rownames(students)
```

```
[1] "John" "Mary" "Dianna" "Bob" "Mike" "Joann"
```

## Function `labels()`

- Function `labels()` returns names of both columns and rows

```
> labels(students)
```

```
[[1]]
```

```
[1] "John" "Mary" "Dianna" "Bob" "Mike" "Joann"
```

```
[[2]]
```

```
[1] "gender" "grade" "householdincome"
```

# Characterizing Data Frames

- Data frames respond to standard inquiry functions:

```
> class(students)
```

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

```
> is.data.frame(students)
```

```
[1] TRUE
```

```
> mode(students)
```

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

```
> str(students)
```

```
'data.frame': 6 obs. of 3 variables:
```

```
$ gender :Factor w/ 2 levels "female","male":2 1 1 2 2 1
```

```
$ grade :num 4 3 4 3 3 4
```

```
$ householdincome:num 45000 34500 67000 42000 81000 53000
```

- Notice that function `str()` tells us the number of rows (observations) in the data frame and the number of variables (columns). Function `nrow()` does the same.



# Importing a Data Frame

- Function `data()`, run without an argument, lists all data sets provided with the standard distribution of R. Those are all data frames.
- Let us load `mtcars` data set (Motor Trend Cars Road Test) by passing the data set name to `data()` function:  
> `data(mtcars)`
- You can inspect the top of the data set with Unix like function `head()` and the bottom with function `tail()`

```
> tail(mtcars)
```

|                | mpg  | cyl | disp  | hp  | drat | wt    | qsec | vs | am | gear | carb |
|----------------|------|-----|-------|-----|------|-------|------|----|----|------|------|
| Porsche 914-2  | 26.0 | 4   | 120.3 | 91  | 4.43 | 2.140 | 16.7 | 0  | 1  | 5    | 2    |
| Lotus Europa   | 30.4 | 4   | 95.1  | 113 | 3.77 | 1.513 | 16.9 | 1  | 1  | 5    | 2    |
| Ford Pantera L | 15.8 | 8   | 351.0 | 264 | 4.22 | 3.170 | 14.5 | 0  | 1  | 5    | 4    |
| Ferrari Dino   | 19.7 | 6   | 145.0 | 175 | 3.62 | 2.770 | 15.5 | 0  | 1  | 5    | 6    |
| Maserati Bora  | 15.0 | 8   | 301.0 | 335 | 3.54 | 3.570 | 14.6 | 0  | 1  | 5    | 8    |
| Volvo 142E     | 21.4 | 4   | 121.0 | 109 | 4.11 | 2.780 | 18.6 | 1  | 1  | 4    | 2    |

# Subsetting

- You can select a single column of a data frame using \$ notation:

```
> head(mtcars$mpg)
```

```
[1] 21.0 21.0 22.8 21.4 18.7 18.1
```

- You can extract subsets of data frame data using bracket [] notation

```
> mtcars[1:5,3] # rows 1 to 5, column 3
```

```
[1] 160 160 108 258 360
```

```
> mtcars[1:5,"hp"] # rows 1 to 5 column "hp"
```

```
[1] 110 110 93 110 175
```

```
> mtcars[mtcars$mpg < 15, c("mpg", "gear")] # rows where mpg < 15
columns MPG and GEAR
```

```
 mpg gear
```

```
Duster 360 14.3 3
```

```
Cadillac Fleetwood 10.4 3
```

```
Lincoln Continental 10.4 3
```

```
> mtcars[c(1,2),] # rows 1 and 2, all columns
```

```
 mpg cyl disp hp drat wt qsec vs am gear carb
```

```
Mazda RX4 21 6 160 110 3.9 2.620 16.46 0 1 4 4
```

```
Mazda RX4 Wag 21 6 160 110 3.9 2.875 17.02 0 1 4 4
```

```
> mtcars[1,2] # Treating data.frame as a matrix
```

```
[1] 6 # element in the 1st row, 2nd column
```

```
> mtcars[3,1] # element in the 3rd row, 1st column
```

```
[1] 22.8
```

# Convert a Matrix into Data Frames

- Let us consider matrix `x.mat` with named rows and columns.  
We could verify the type (class) of the object

```
> x.mat
 C1 C2
R1 3 -1
R2 2 0
R3 -3 6
> class(x.mat)
[1] "matrix"
```

- The above matrix could be transformed into a data frame

```
> y <- data.frame(x.mat)
> class(y)
[1] "data.frame"
```

## `attach()` and `detach()` a Data Frame

- If you do a lot of work with one particular data frame, typing the name of the data frame followed by `$` sign followed by the variable name becomes tedious. If you type:

```
> attach(mtcars)
```

- Your data frame, `mtcars` in this case, becomes the default location for data lookups and from that point on you can refer to column `mtcars$gpm` just as `gpm`. The same applies for all columns of the attached data frame.
- Once you are done and would like to work with another data frame, you detach the data frame, e.g.

```
> detach(mtcars)
```

# Program Branching

R is programming language and it has standard tools for branching and decision making:

```
if (logical expression) {
 statements
} else {
 alternative statements
}
```

**else** branch is optional

# Grouped expressions in R

- Notice that we use parentheses to group expressions.
- We could also use “;” to indicate the end of a statement.

```
x = 1:11
if (length(x) <= 10)
{
 x <- c(x, 10:20);
 print(x)
} else {
 print(x[1])
}
```

- To run this set of statements, copy them to the text editor, highlight them all, and then apply CTRL Carriage Return

# Loops

- When the same or similar tasks need to be performed multiple times; for all elements of a list; for all columns of an array, etc. R uses for loops

```
for(i in 1:10) {
 print(i*i)
}
```

```
i=1
while(i<=10) {
 print(i*i);
 i=i+sqrt(i)
}
```

# User Defined Functions

- We have already seen that users could define functions. The general form of function definition is

```
name <- function(arguments) {
 expression
}
```

- For example, function **larger** is defined as

```
larger <- function(x, y=9) {
 if(any(x < 0)) return(NA)
 y.is.bigger <- y > x
 x[y.is.bigger] <- y[y.is.bigger]
 x
}
```

- Note,  $y=9$  provided the default value for parameter  $y$



# Missing Arguments in Functions

- If a function definition does not provide default arguments, or provide proper handling of missing arguments, function will return an error on missing argument

```
> add <- function(x, y=0) {x + y}
```

```
> add(4)
```

```
> add <- function(x, y) {
 if(missing(y)) x
 else x+y
}
```

```
> add(4)
```

# Variable Number of Arguments

- The special argument name “...” in the function definition will match any number of arguments in the call.
- Function `nargs()` returns the number of arguments in the current call.

# Variable Number of Arguments

```
> mean.of.all <- function(...) mean(c(...))
> mean.of.all(1:10, 20:100, 12:14)

> mean.of.means <- function(...)
{
 means <- numeric()
 for(x in list(...)) means <-
 c(means, mean(x))
 mean(means)
}
```

## Variable Number of Arguments, 2nd

```
mean.of.means <- function(...)
{
 n <- nargs()
 means <- numeric(n)
 all.x <- list(...)
 for(j in 1:n) means[j] <- mean(all.x[[j]])
 mean(means)
}
mean.of.means(1:10, 10:100)
```

# Mathematical operations

Standard operations: `+` `-` `*` `/`

Exponentiation: `2^5` or `2**5`

Integral Division: `8%/3` gives 2

Modulus: `7%5` gives 2)

Standard Functions : `abs()` , `sign()` , `log()` , `log10()` , `sqrt()` ,  
`exp()` , `sin()` , `cos()` , `tan()`  
`gamma()` , `lgamma()` , `choose()`

Rounding: `round(x, 3)` round with 3 decimal digits

`floor(2.5)` gives 2, `ceiling(2.5)` gives 3

## Some Useful functions

```
> seq(2,12,by=2)
```

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

```
> seq(4,5,length=5)
```

```
[1] 4.00 4.25 4.50 4.75 5.00
```

```
> rep(4,10)
```

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

```
> paste("V",1:5,sep="")
```

```
[1] "V1" "V2" "V3" "V4" "V5"
```

```
> LETTERS[1:7]
```

```
[1] "A" "B" "C" "D" "E" "F" "G"
```

## `lapply, sapply, apply`

- When the same or similar tasks need to be performed multiple times for all elements of a list or for all columns of an array. Easier and faster than “for” loops

`lapply( li, function )`

- To each element of the list `li`, apply function `function`. The result is a list whose elements are the individual `function` results.

```
> li = list("klaus", "martin", "georg")
> lapply(li, toupper)
[[1]]
[1] "KLAUS"
[[2]]
[1] "MARTIN"
[[3]]
[1] "GEORG"
```

## lapply, sapply, apply

`sapply( li, function )`

- Like `apply`, but tries to simplify the result, by converting it into a vector or array of appropriate size

```
> li = list("klaus", "martin", "georg")
```

```
> sapply(li, toupper)
```

```
[1] "KLAUS" "MARTIN" "GEORG"
```

```
> fct = function(x) { return(c(x, x*x, x*x*x)) }
```

```
> sapply(1:5, fct)
```

|      | [,1] | [,2] | [,3] | [,4] | [,5] |
|------|------|------|------|------|------|
| [1,] | 1    | 2    | 3    | 4    | 5    |
| [2,] | 1    | 4    | 9    | 16   | 25   |
| [3,] | 1    | 8    | 27   | 64   | 125  |



# apply

`apply( arr, margin, fct )`

- Applies the function **fct** along some dimensions of the array **arr**, according to **margin**, and returns a vector or array of the appropriate size; **margin**=1 indicates rows; **margin**=2, columns

```
> x
```

|      | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | 5    | 7    | 0    |
| [2,] | 7    | 9    | 8    |
| [3,] | 4    | 6    | 7    |
| [4,] | 6    | 3    | 5    |

```
> apply(x, 1, sum)
```

```
[1] 12 24 17 14
```

```
> apply(x, 2, sum)
```

```
[1] 22 25 20
```

# Hash Tables

- In vectors, lists, data frames and arrays, elements are stored one after another, and are accessed in that order by their offset (or: index), which is an integer number.
- Sometimes, consecutive integer numbers are not the “natural” way to access: e.g., gene names, oligo sequences
- E.g., if we want to look for a particular gene name in a long list or data frame with tens of thousands of genes, the linear search may be very slow.
- Solution: instead of list, use a [hash table](#). It sorts, stores and accesses its elements in a way similar to a telephone book.

# Hash Tables

- In R, a **hash table** is the same as a **workspace for variables**, which is the same as an **environment**.

```
> tab = new.env(hash=T)

> assign("cenp-e", list(cloneid=682777,
 description="putative kinetochore motor ..."), env=tab)

> assign("btk", list(cloneid=682638,
 fullname="Bruton agammaglobulinemia tyrosine kinase"), env=tab)

> ls(env=tab)
[1] "btk" "cenp-e"

> get("btk", env=tab)
$cloneid
[1] 682638
$fullname
[1] "Bruton agammaglobulinemia tyrosine kinase"
```

# Regular Expressions

- R provides text matching and replacement features in the form similar to the one found in many programming languages (Perl, Unix shells, Java)

```
> a = c("CENP-F", "Ly-9", "MLN50", "ZNF191", "CLH-17")
```

```
> grep("L", a)
[1] 2 3 5
```

```
> grep("L", a, value=T)
[1] "Ly-9" "MLN50" "CLH-17"
```

```
> grep("^L", a, value=T)
[1] "Ly-9"
```

```
> grep("[0-9]", a, value=T)
[1] "Ly-9" "MLN50" "ZNF191" "CLH-17"
```

```
> gsub("[0-9]", "X", a)
[1] "CENP-F" "Ly-X" "MLNXX" "ZNFXXX" "CLH-XX"
```

## Storing data

- Every R object can be stored into and restored from a file with the commands
  - “save” and “load”.
  - This uses the XDR (external data representation) standard of Sun Microsystems and others, and is portable between MS-Windows, Unix, Mac.
  - Make sure you can write to the directory
- 
- **`save(x, file="x.Rdata")`**
  - **`load("x.Rdata")`**
  - Rdata is a binary format. You can save as text.

# Importing and exporting data

- There are many ways to get data into R and out of R.
- Most programs (e.g. Excel), as well as humans, know how to deal with rectangular tables in the form of tab-delimited text files.

```
> x = read.delim("filename.txt")
```

```
also: read.table, read.csv
```

```
> write.table(x, file="x.txt", sep="\t")
```

## Importing data: caveats

- **Type conversions:** by default, the read functions try to guess and autoconvert the data types of the different columns (e.g. number, factor, character). There are options `as.is` and `colClasses` to control this
- **Special characters:** the delimiter character (space, comma, tabulator) and the end-of-line character cannot be part of a data field. To circumvent this, text may be “quoted”. However, if this option is used (the default), then the quote characters themselves cannot be part of a data field. Except if they themselves are within quotes...
- *Understand the conventions your input files use and set the quote options accordingly.*

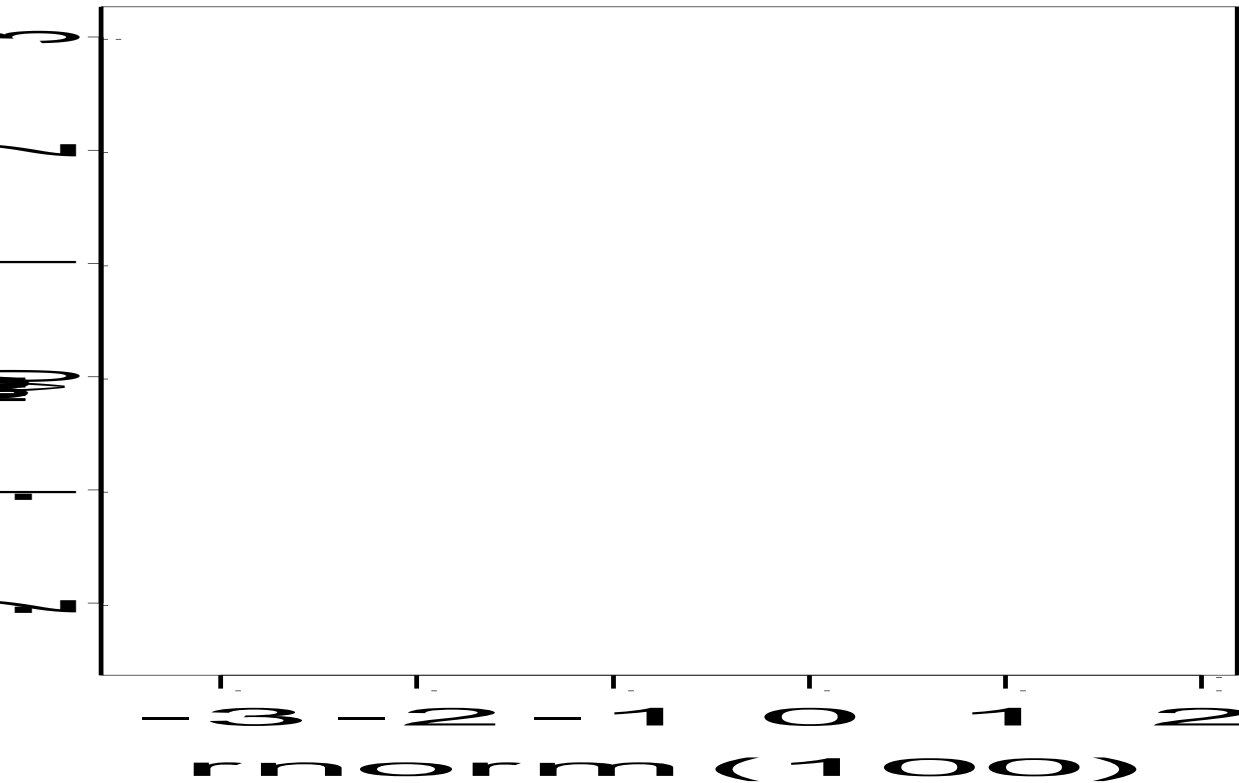
# plot()

- If `x` and `y` are vectors, `plot(x, y)` produces a scatterplot of `x` against `y`.
- `plot(x)` produces a time series plot if `x` is a numeric vector or time series object.
- `plot(df)`, `plot(~ expr)`, `plot(y ~ expr)`, where `df` is a data frame, `y` is any object, `expr` is a list of object names separated by `'+'` (e.g. `a + b + c`).
- The first two forms produce distributional plots of the variables in a data frame (first form) or of a number of named objects (second form). The third form plots `y` against every object named in `expr`.



# Graphics with `plot()`

```
> plot(rnorm(100), rnorm(100))
```



Function `rnorm()`  
Simulates a random  
normal distribution .

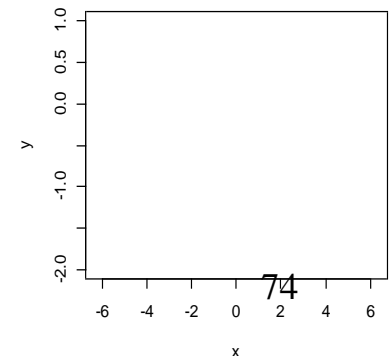
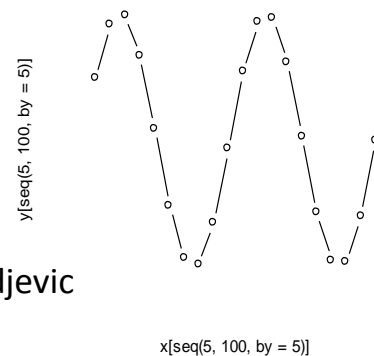
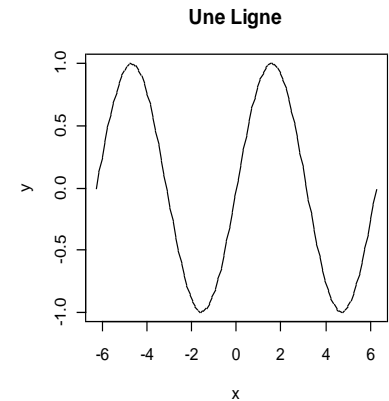
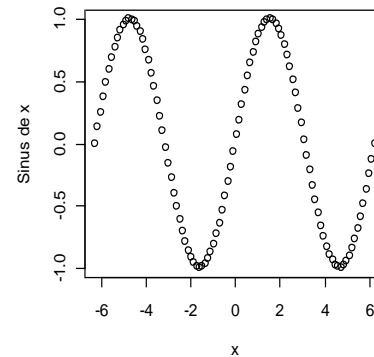
**Help** `?rnorm,`  
`?runif,`  
`?rexp,`  
`?binom, ...`

# Graphics with `plot()`

```
x <- seq(-2*pi,2*pi,length=100)
y <- sin(x)
par(mfrow=c(2,2))
plot(x,y,xlab="x",ylab="Sin x")
plot(x,y,type="l", main="A Line")
```

```
plot(x[seq(5,100,by=5)],
 y[seq(5,100,by=5)],
 type="b",axes=F)
```

```
plot(x,y,type="n",
 ylim=c(-2,1))
par(mfrow=c(1,1))
```



@Zoran B. Djordjevic

## Graphical Parameters of `plot()`

`type = "c"`: `c = p` (default), `l`, `b`, `s`, `o`, `h`, `n`.

`pch = "+"` : character or numbers 1 – 18

`lty = 1` : numbers

`lwd = 2` : numbers

`axes = "L"`: `L = F, T`

`xlab = "string"`, `ylab = "string"`

`sub = "string"`, `main = "string"`

`xlim = c(lo,hi)`, `ylim = c(lo,hi)`

And some more.

# Graphical Parameters of `plot()`

```
x <- 1:10
y <- 2*x + rnorm(10,0,1)
plot(x,y,type="p") #Try l,b,s,o,h,n
axes=T, F
xlab="age", ylab="weight"
sub="sub title", main="main title"
xlim=c(0,12), ylim=c(-1,12)
```

# Other graphical functions

See also:

`barplot()`

`image()`

`hist()`

`pairs()`

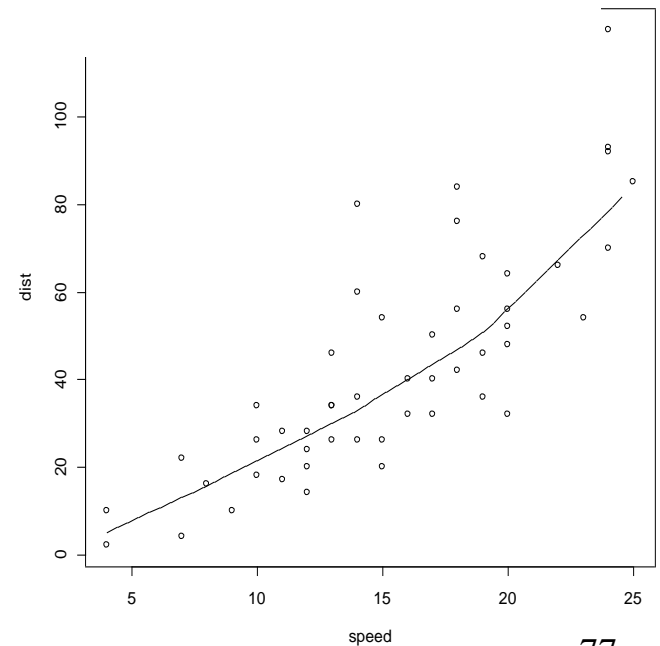
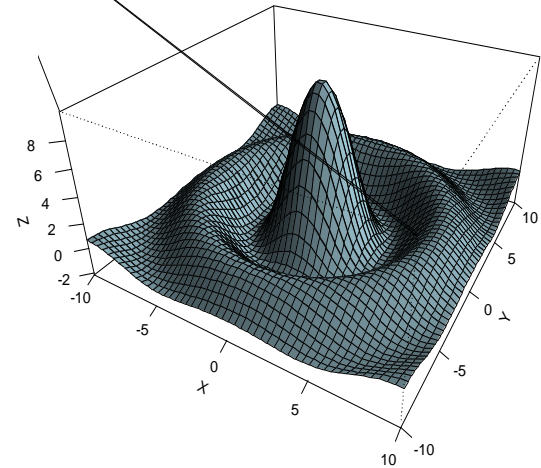
`persp()`

`piechart()`

`polygon()`

`library(modreg)`

`scatter.smooth()`



# Interactive Graphics Functions

- `locator(n, type="p")` : Waits for the user to select locations on the current plot using the left mouse button. This continues until `n` (default 500) points have been selected.
- `identify(x, y, labels)` : Allow the user to highlight any of the points defined by `x` and `y`.
- `text(x, y, "Hey")` : Write text at coordinate `x, y`.

## Plots for Multivariate Data

```
pairs(stack.x)
x <- 1:20/20
y <- 1:20/20
z <-
 outer(x,y,function(a,b){cos(10*a*b)/(1+
 a*b^2)})
contour(x,y,z)
persp(x,y,z)
image(x,y,z)
```

# Other graphical functions

```
> axis(1,at=c(2,4,5),
...)
 legend("A","B","C"))
```

Axis details ("ticks", legend,

Use `xaxt="n"` and `yaxt="n"` inside  
`plot()`

```
> lines(x,y,...)
```

Line plots

```
> abline(lsfitt(x,y))
> abline(0,1)
```

Add an adjustment  
add a line of slope 1 and  
intercept 0

```
> legend(locator(1),...)
```

Legends: very flexible



# Histogram

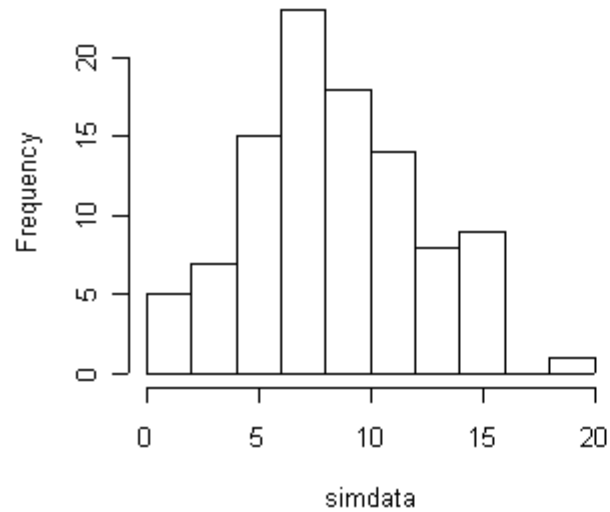
- A *histogram* is a special kind of bar plot
- It allows you to visualize the *distribution* of values for a numerical variable
- When drawn with a *density scale*:
  - the *AREA* (NOT height) of each bar is the proportion of observations in the interval
  - the *TOTAL AREA* is 100% (or 1)

# R: Histogram

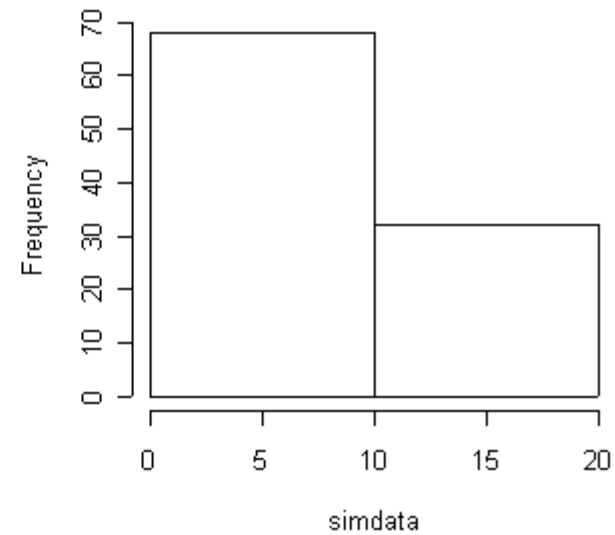
- Type **?hist** to view the help file
  - Note some important arguments, esp **breaks**
- Simulate some data, make histograms varying the number of bars (also called 'bins' or 'cells'), e.g.

```
par(mfrow=c(2,2)) # set up multiple plots
simdata <- rchisq(100,8)
hist(simdata) # default number of bins
hist(simdata,breaks=2) # etc,4,20
```

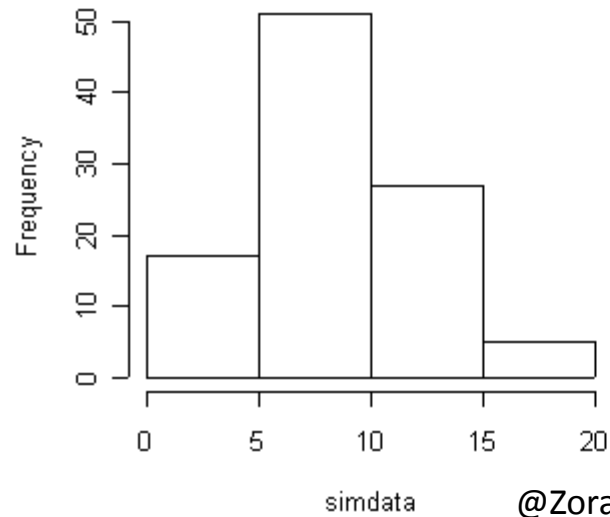
**Histogram of simdata**



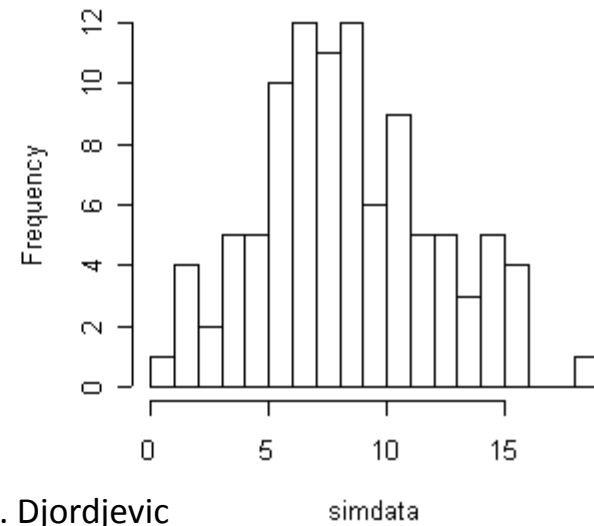
**Histogram of simdata**



**Histogram of simdata**



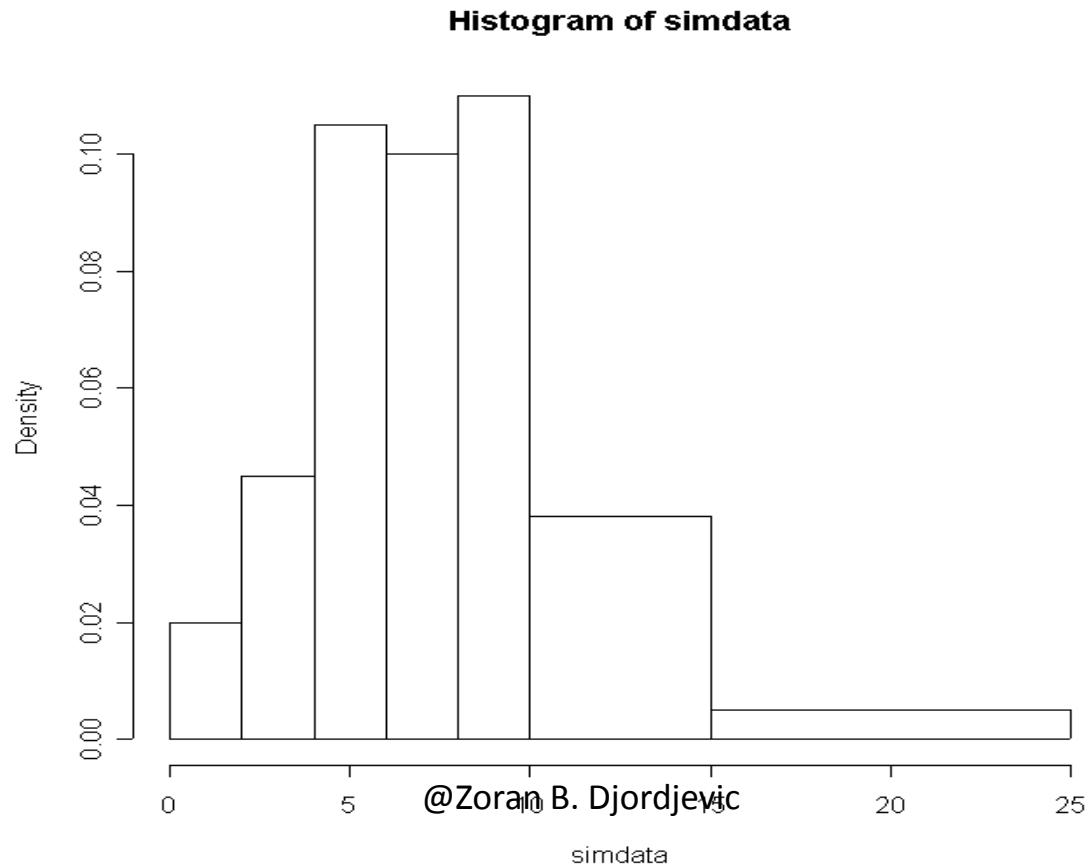
**Histogram of simdata**



## R: setting your own breakpoints

```
bps <- c(0, 2, 4, 6, 8, 10, 15, 25)
```

```
hist(simdata, breaks=bps)
```



# Scatterplot

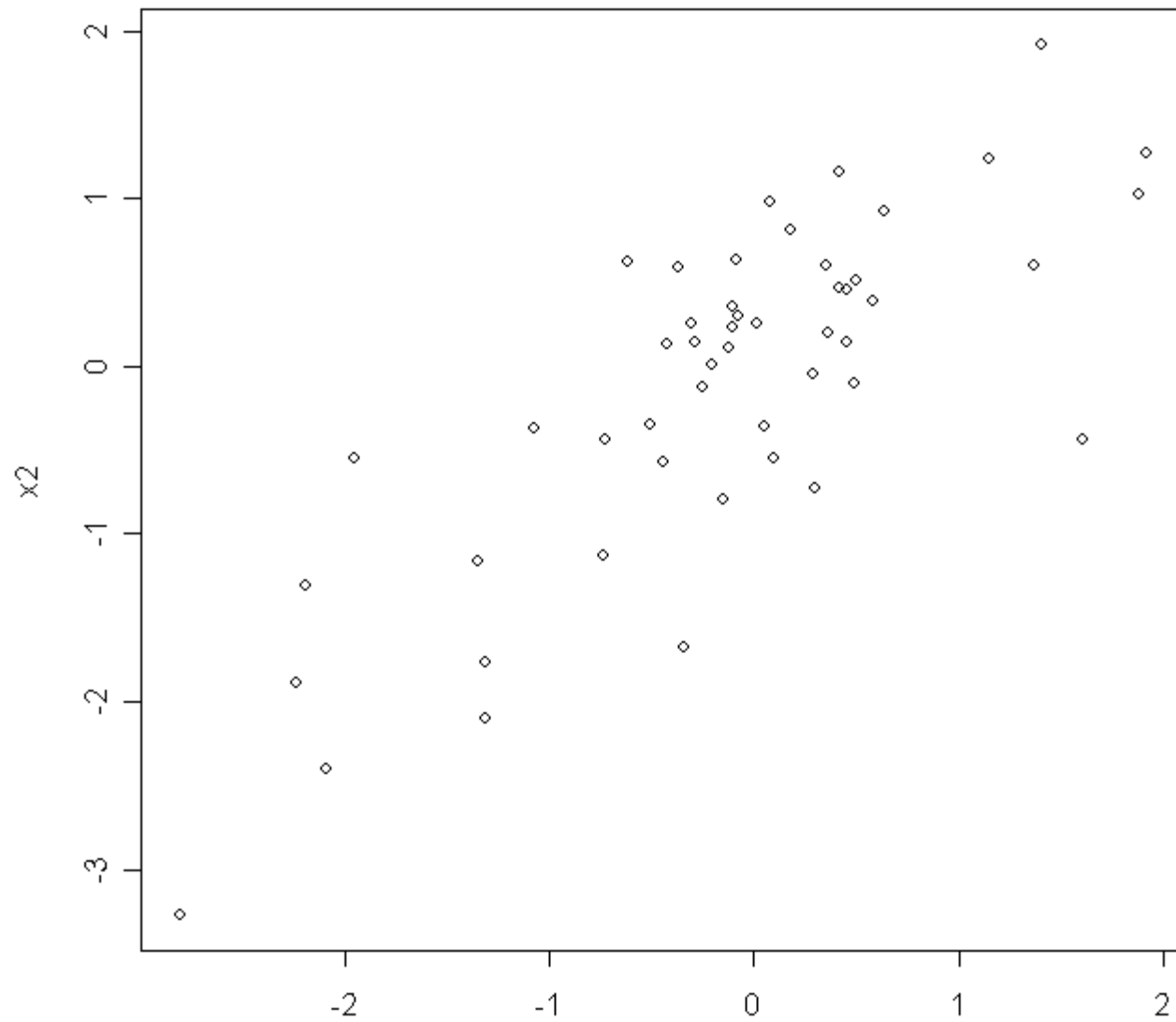
- A scatterplot is a standard two-dimensional (X,Y) plot
- Used to examine the relationship between two (continuous) variables
- It is often useful to plot values for a single variable against the order or time the values were obtained

# R: Scatterplot

- Type `?plot` to view the help file
  - For now we will focus on simple plots, but **R** allows extensive user control for highly customized plots
- Simulate a bivariate data set:

```
z1 <- rnorm(50)
z2 <- rnorm(50)
rho <- .75 # (or any number between -1 and 1)
x2<- rho*z1+sqrt(1-rho^2)*z2
plot(z1,x2)
```

**Scatterplot of X2 vs. Z1**



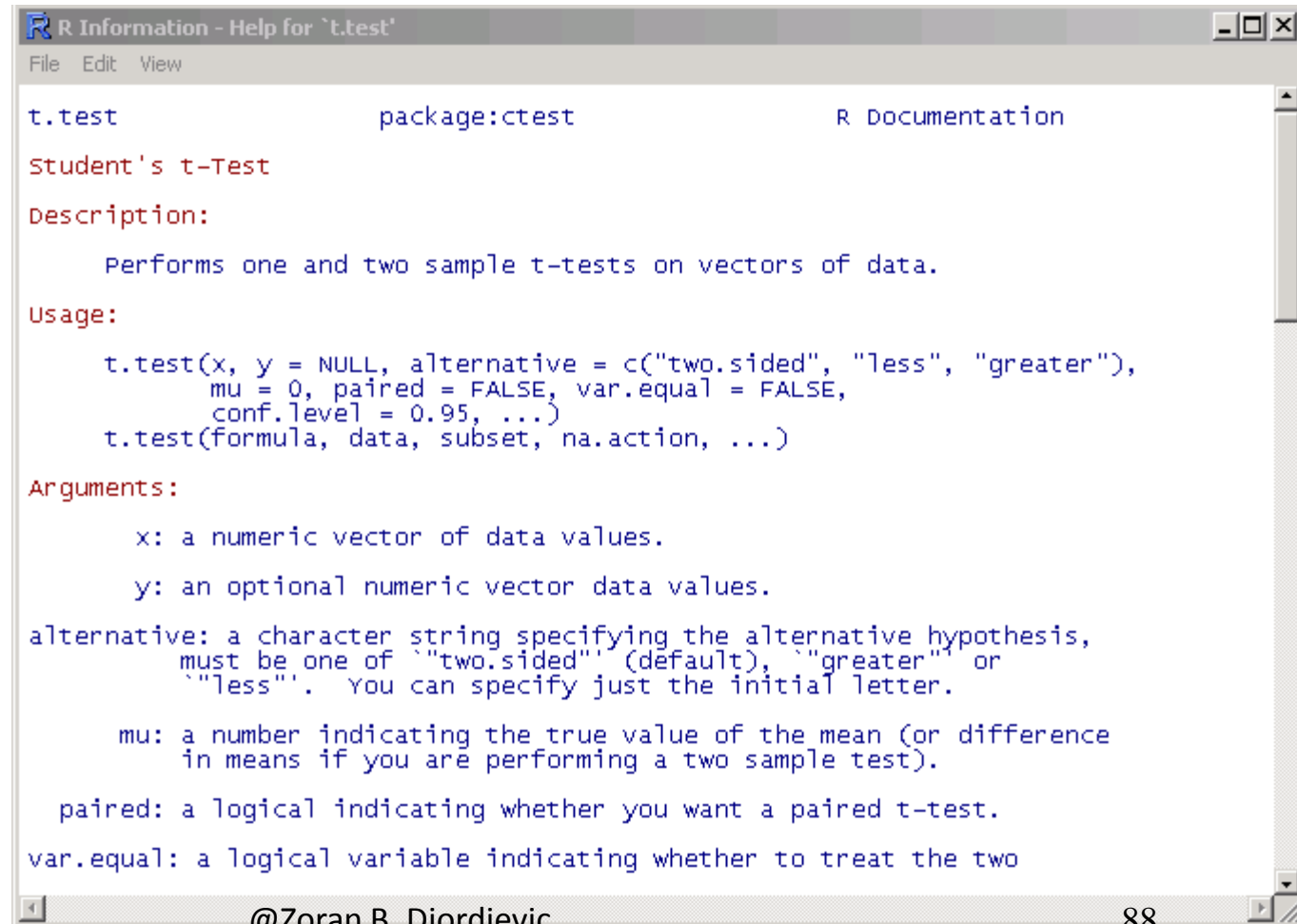
# Getting help

- Details about a specific command whose name you know (input arguments, options, algorithm, results):

**>? t.test**

**or**

**>help(t.test)**



```
R Information - Help for 't.test'
File Edit View

t.test package:ctest R Documentation

Student's t-Test

Description:
 Performs one and two sample t-tests on vectors of data.

Usage:
 t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"),
 mu = 0, paired = FALSE, var.equal = FALSE,
 conf.level = 0.95, ...)
 t.test(formula, data, subset, na.action, ...)

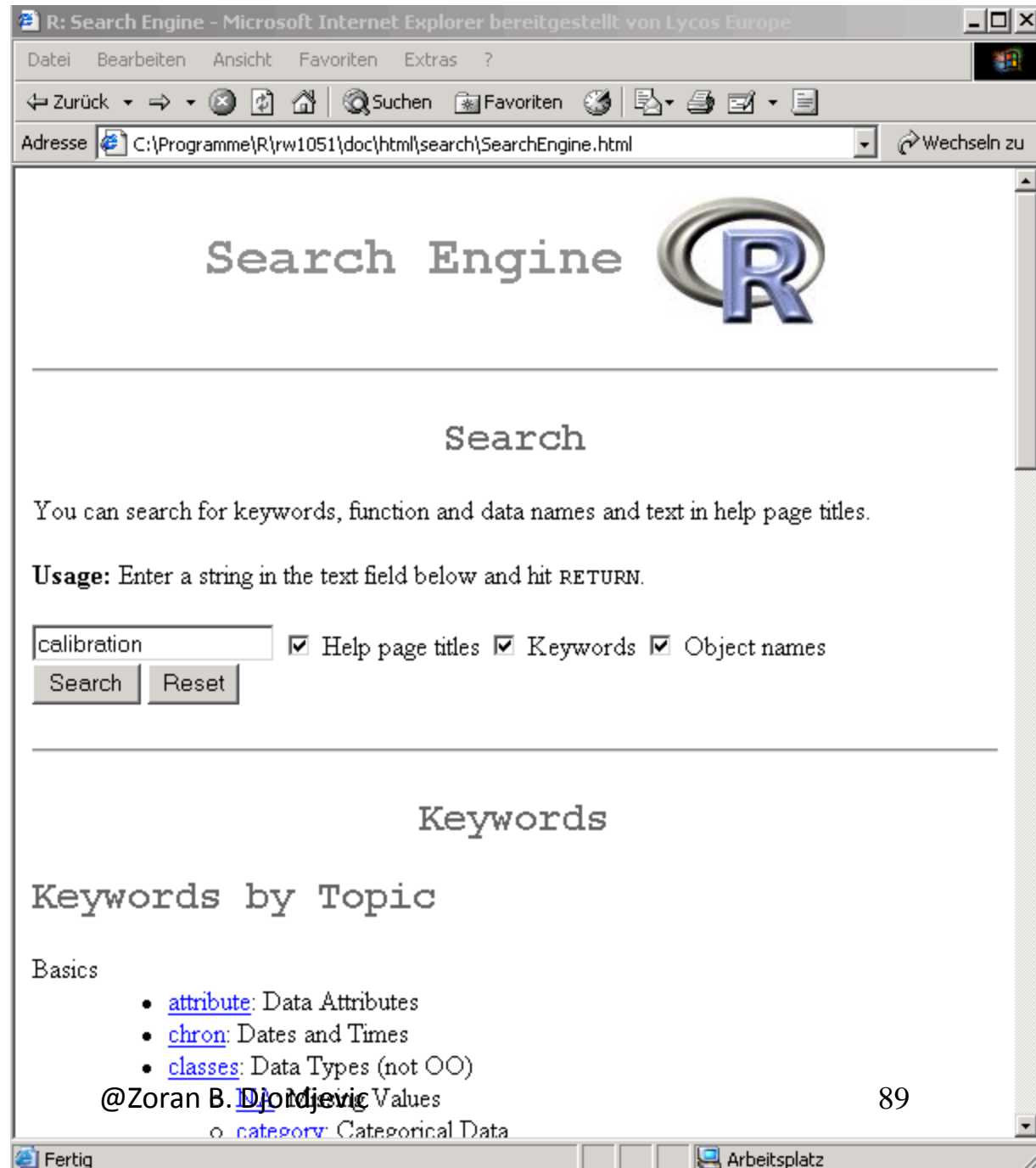
Arguments:
 x: a numeric vector of data values.
 y: an optional numeric vector data values.
 alternative: a character string specifying the alternative hypothesis,
 must be one of "two.sided" (default), "greater" or
 "less". You can specify just the initial letter.
 mu: a number indicating the true value of the mean (or difference
 in means if you are performing a two sample test).
 paired: a logical indicating whether you want a paired t-test.
 var.equal: a logical variable indicating whether to treat the two
```



- **Getting help**

HTML search  
engine

Search for topics  
with regular  
expressions:  
“help.search”



@Zoran B. Djurdjevic

# Install a new Package

# Web sites

- [www.r-project.org](http://www.r-project.org)
- [cran.r-project.org](http://cran.r-project.org)
- [www.bioconductor.org](http://www.bioconductor.org)
- Full text search:
  - [www.r-project.org](http://www.r-project.org)
  - or
  - [www.google.com](http://www.google.com)
- with '... site:.r-project.org' or other R-specific keywords