# R – a brief introduction

Nils Kammenhuber

Technische Universität München (в Новогархинске)

Gilberto Câmara

Instituto Nacional de Pesquisas Espaciais (São Tomé dos Campos)

# Original material

- Gilberto Câmara
  - Instituto Nacional de Pesquisas Espaciais
- Manfred Jobmann
  - Technische Universität München
- Johannes Freudenberg
  - Cincinnati Children's Hospital Medical Center
- Marcel Baumgartner
  - Nestec S.A.
- Jaeyong Lee
  - Penn State University
- Jennifer Urbano Blackford, Ph.D
  - Department of Psychiatry, Kennedy Center
- Wolfgang Huber

# History of R

- Statistical programming language “S” developed at Bell Labs since 1976 (at the same time as UNIX)
- Intended to interactively support research and data analysis projects
- Exclusively licensed to Insightful (“S-Plus”)
- “R”: Open source platform similar to S
  - Developed by R. Gentleman and R. Ihaka (University of Auckland, NZ) during the 1990s
  - Most S-plus programs will run on R without modification!

# What R is and what it is not

- R is
  - a programming language
  - a statistical package
  - an interpreter
  - Open Source
- R is not
  - a database
  - a collection of “black boxes”
  - a spreadsheet software package
  - commercially supported

# What R is

- Powerful tool for data analysis and statistics
  - Data handling and storage: numeric, textual
  - Powerful vector algebra, matrix algebra
  - High-level data analytic and statistical functions
  - Graphics, plotting
- Programming language
  - Language “built to deal with numbers”
  - Loops, branching, subroutines
  - Hash tables and regular expressions
  - Classes (“OO”)

# What R is not

- is not a database, but connects to DBMSs
- has no click-point user interfaces,
  but connects to Java, TclTk
- language interpreter can be very slow,
  but allows to call own C/C++ code
- no spreadsheet view of data,
  but connects to Excel/MsOffice
- no professional / commercial support

# R and statistics

- Packaging: a crucial infrastructure to efficiently produce, load and keep consistent software libraries from (many) different sources / authors
- Statistics: most packages deal with statistics and data analysis
- State of the art: many statistical researchers provide their methods as R packages

# Installation

- To obtain and install R on your computer
  - Go to http://cran.r-project.org/mirrors.html to choose a mirror near you
  - Click on your favorite operating system (Linux, Mac, or Windows)
  - Download and install the “base”
- To install additional packages
  - Start R on your computer
  - Choose the appropriate item from the “Packages” menu

# Getting started

- Call R from the shell:
  ```
  user@host$ R
  ```
- Leave R, go back to shell:
  ```
  > q()
  Save information (y/n/q)? y
  ```

## R: session management

- Your R objects are stored in a *workspace*
- To list the objects in your workspace (may be a lot):
  ```
  > ls()
  ```
- To remove objects which you don't need any more:

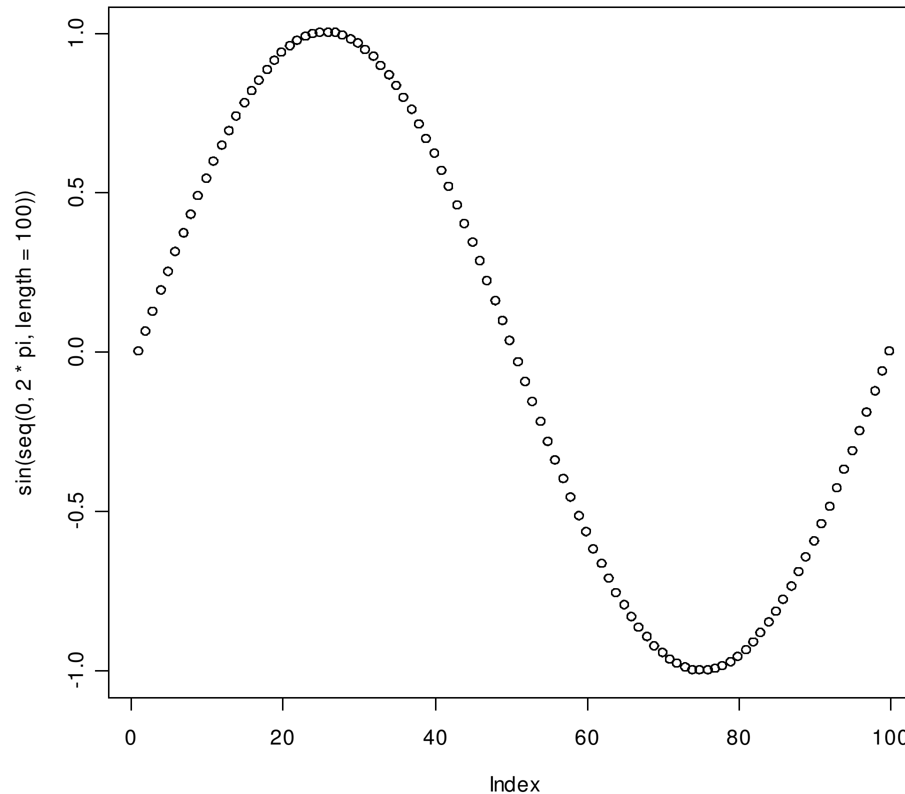
  ```
  > rm(weight, height, bmi)
  ```
- To remove ALL objects in your workspace:
  ```
  > rm(list=ls())
  ```
- To save your workspace to a file:
  ```
  > save.image()
  ```
- The default workspace file is `./.RData`

# First steps: R as a calculator

```
> 5 + (6 + 7) * pi^2
[1] 133.3049
> log(exp(1))
[1] 1
> log(1000, 10)
[1] 3
> Sin(pi/3)^2 + cos(pi/3)^2
Error: couldn't find function "Sin"
> sin(pi/3)^2 + cos(pi/3)^2
[1] 1
```

# R as a calculator and function plotter

```
> 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)))
```



# Help and other resources

- Starting the R installation help pages

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

- In general:

```
> help(functionname)
```

- If you don't know the function you're looking for:

```
help.search(„quantile“)
```

- "What's in this variable"?

```
> class(variableInQuestion)
[1] "integer"
> summary(variableInQuestion)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.000   5.250   8.500   9.833  13.250  19.000
```

- www.r-project.org
  - CRAN.r-project.org: Additional packages, like www.CPAN.org for Perl

# Basic data types

# Objects

- Containers that contain data
- Types of objects:
  *vector, factor, array, matrix, dataframe, list, function*
- Attributes
  - mode: numeric, character (=string!), complex, logical
  - length: number of elements in object
- Creation
  - assign a value
  - create a blank object

# Identifiers (object names)

- must start with a letter (A-Z or a-z)
- can contain letters, digits (0-9), **periods (".")**
  - **Periods have no special meaning (I.e., unlike C or Java!)**
- case-sensitive:
  e.g., `mydata` different from `MyData`
- **do not use use underscore "_"!**

# Assignment

- “<–” used to indicate assignment

```
x <- 4711
x <- “hello world!”
x <- c(1,2,3,4,5,6,7)
x <- c(1:7)
x <- 1:4
```

- *note: as of version 1.4 “=“ is also a valid assignment operator*

# Basic (atomic) data types

- **Logical**

```
> x <- T; y <- F
> x; y
[1] TRUE
[1] FALSE
```

- **Numerical**

```
> a <- 5; b <- sqrt(2)
> a; b
[1] 5
[1] 1.414214
```

- **Strings** *(called "characters"!)*

```
> a <- "1"; b <- 1
> a; b
[1] "1"
[1] 1
> a <- “string"
> b <- "a"; c <- a
> a; b; c
[1] “string"
[1] "a"
[1] “string"
```

# But there is more!

R can handle “big chunks of numbers” in elegant ways:

- Vector
  - Ordered collection of data of the same data type
  - Example:
    - Download timestamps
    - last names of all students in this class
  - In R, a single number is a vector of length 1
- Matrix
  - Rectangular table of data of the same data type
  - Example: a table with marks for each student for each exercise
- Array
  - Higher dimensional matrix of data of the same data type
- (Lists, data frames, factors, function objects, … → later)

# Vectors

```
> Mydata<-c(2,3.5,-0.2)
```
Vector (c=“concatenate”)

```
> colours<-c(“Black",
“Red",“Yellow")
```
String vector

```
> x1 <- 25:30
> x1
[1] 25 26 27 28 29 30
```
Number sequence

```
> colours[1]
[1] “Black"
```
**Index starts with 1, not with 0!!!**

Addressing one element…

```
> x1[3:5]
[1] 27 28 29
```
…and multiple elements

# Vectors (continued)

- More examples with vectors:

```
> x <- c(5.2, 1.7, 6.3)
> log(x)
[1] 1.6486586 0.5306283 1.8405496
> y <- 1:5
> z <- seq(1, 1.4, by = 0.1)
> y + z
[1] 2.0 3.1 4.2 5.3 6.4
> length(y)
[1] 5
> mean(y + z)
[1] 4.2
```

# Subsetting

- 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", "a")
> x[1]                    # first (!) element
> x[3:5]                  # elements 3..5
> x[-(3:5)]                    # elements 1 and 2
> x[c(T, F, T, F, T, F, T, F)]    # even-index elements
> x[x <= "d"]             # elements "a"..."d","a"
```

# Typical operations on vector elements

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

```
> Mydata > 0
[1] TRUE TRUE FALSE
```

- Test on the elements

```
> Mydata[Mydata>0]
[1]  2  3.5
```

- Extract the positive elements

```
> Mydata[-c(1,3)]
[1]  3.5
```

- Remove the given elements

# More vector operations

```
> x <- c(5,-2,3,-7)
> y <- c(1,2,3,4)*10
> y
[1] 10 20 30 40
```

Multiplication on all the elements

```
> sort(x)
[1] -7 -2 3 5
```

Sorting a vector

```
> order(x)
[1] 4 2 3 1
```

Element order for sorting

```
> y[order(x)]
[1] 40 20 30 10
```

Operation on all the components

```
> rev(x)
[1] -7 3 -2 5
```

Reverse a vector

# Matrices

- Matrix: Rectangular table of data **of the same type**

```
> m <- matrix(1:12, 4, byrow = T); m
        [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> y <- -1:2
> m.new <- m + y
> t(m.new)
     [,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))
[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.matB <- matrix(x,ncol=2,
                   byrow=T)          By-row creation
> x.matB
     [,1] [,2]
[1,]    3   -1
[2,]    2    0
[3,]   -3    6
```

# Building subvectors and submatrices

```
> x.matB[,2]
[1] -1 0 6
```

2nd column

```
> x.matB[c(1,3),]

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

1st and 3rd lines

```
> x.mat[-2,]
          [,1] [,2]
[1,]        3    0
[2,]        2    6
```

Everything but the 2nd line

# Dealing with matrices

```
> dim(x.mat)                    Dimension (I.e., size)
[1] 3 2
> t(x.mat)                      Transposition
     [,1] [,2] [,3]
[1,]    3    2   -3
[2,]   -1    0    6


> x.mat %*% t(x.mat)            Matrix multiplication; also
see %o%
     [,1] [,2] [,3]
[1,]   10    6  -15
[2,]    6    4   -6
[3,]  -15   -6   45

> solve()              Inverse of a square matrix
> eigen()              Eigenvectors and eigenvalues
```

# Special values (1/3)

- R is designed to handle statistical data
- => Has to deal with missing / undefined / special values
- Multiple ways of missing values
  - NA: not available
  - NaN: not a number
  - Inf, -Inf: inifinity
- Different from Perl: **NaN ≠ Inf ≠ NA ≠ FALSE ≠ “” ≠ 0** (pairwise)
- NA also may appear as Boolean value
  I.e., boolean value in R ∈ {`TRUE, FALSE, NA`}

# Special values (2/3)

- `NA`: Numbers that are "not available"

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

- `NaN`: "Not a number"

```
> 0/0
[1] NaN
```

- `Inf, -Inf`: *inifinite*

```
> log(0)
[1] -Inf
```

# Special values (3/3)

Odd (but logical) interactions with equality tests, etc:

```
> 3 == 3
[1] TRUE
> 3 == NA
[1] NA                           #but not “TRUE”!
> NA == NA
[1] NA
> NaN == NaN
[1] NA
> 99999 >= Inf
[1] FALSE
> Inf == Inf
[1] TRUE
```

Lists

# Lists (1/4)

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.**

```
> doe = list(name="john",age=28,married=F)
> doe$name
[1] "john"
> doe$age
[1] 28
```

Typically, vector/matrix elements are accessed by their index (=an integer), list elements by their name (=a string).
**But both types support both access methods.**

# Lists (2/4)

- A list is an object consisting of objects called *components*.
- Components of a list **don't need** to be of the same mode or type:
  - `list1 <- list(1, 2, TRUE, "a string", 17)`
  - `list2 <- list(l1, 23, l1)` # lists within lists: possible
- A component of a list can be referred either as
  - `listname[[index]]`

  Or as:
  - `listname$componentname`

# Lists (3/4)

- The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely.
- Syntactic quicksand:
  - aa[[1]] is the first component of aa
  - aa[1] is the sublist consisting of the first component of aa only.
- There *are* functions whose return value is a list (and not a vector / matrix / array)

# Lists are very flexible

```
> 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

> my.list <- list(component1=c(5,4,-1),component2=c
("X1","X2","X3"))
> my.list$component2[2:3]
[1] "X2" "X3"
```

# Lists: Session

```
> Empl <- list(employee="Anna", spouse="Fred", children=3,
   child.ages=c(3,7,9))
> Empl[[1]]              # You'd achieve the same with: Empl$employee
"Anna"
> Empl[[4]][2]
7                        # You'd achieve the same with: Empl$child.ages[2]
> Empl$child.a
[1] 3 7 9                # You can shortcut child.ages as child.a
> Empl[4]                # a sublist consisting of the 4th component
   of Empl
$child.ages
[1] 3 7 9
> names(Empl)
[1] "employee"      "spouse"          "children"
   "child.ages"
> unlist(Empl)  # converts it to a vector. Mixed types will be
   converted to strings, giving a string vector.
```

# Back to matrices:
# Naming elements *of a matrix*

```
> x.mat
      [,1]  [,2]
[1,]  3     -1
[2,]  2     0
[3,]  -3    6

> dimnames(x.mat) <- list(c("Line1","Line2",“xyz"),
c(“col1",“col2"))
                                                 #assign
names to rows/columns of matrix

> x.mat
      col1  col2
Line1    3    -1
Line2    2     0
xyz     -3     6
```

# R as a “better gnuplot”: Graphics in R

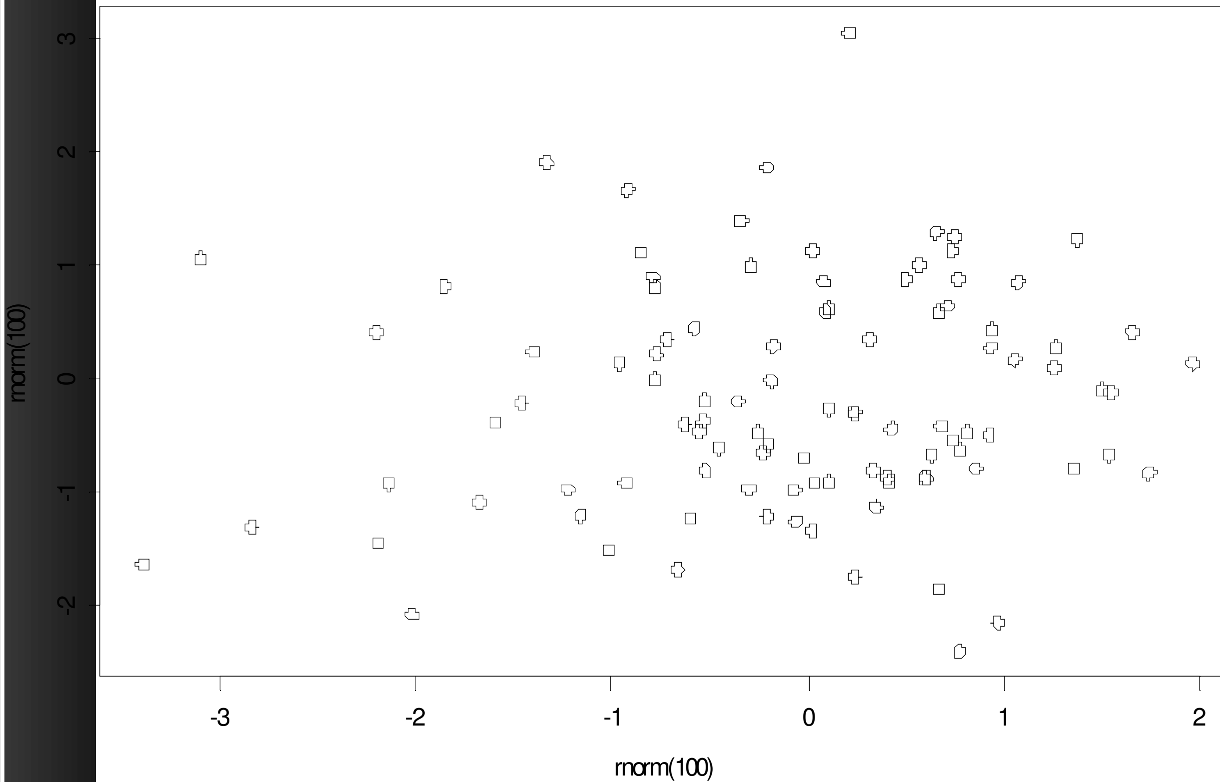
# plot(): Scatterplots

- A scatterplot is a standard two-dimensional (X,Y) plot
- Used to examine the relationship between two (continuous) variables

- If x and y are vectors, then
  `plot(x,y)` produces a **scatterplot** of x against y
  - I.e., do a point at coordinates (x[1], y[1]), then (x[2], y[2]), etc.
- `plot(y)` produces a **time series plot** if y is a numeric vector or time series object.
  - I.e., do a point a coordinates (1,y[1]), then (2, y[2]), etc.
- plot() takes lots of arguments to make it look fancier
  => help(plot)

# Example: Graphics with `plot()`

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




The function **`rnorm()`** Simulates a random normal distribution .

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

# Line plots

- Sometimes you don’t want just points
- solution:
```
> plot(dataX, dataY, type=“l”)
```
- Or, points and lines between them:
```
> plot(dataX, dataY, type=“b”)
```
- Beware: If dataX is not nicely sorted, the lines will jump erroneously across the coordinate system
  - try
    ```
    plot(rnorm(100,1,1), rnorm(100,1,1), type=“l”)
    ```
    and see what happens

# Graphical Parameters of `plot()`

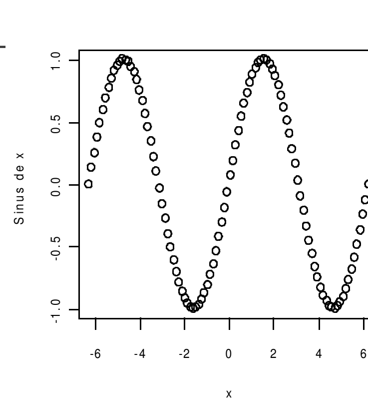
```
plot(x,y, …
  type = "c",              #c may be p (default), l,
  b,s,o,h,n.  Try it.
  pch="+",         # point type. Use character or
  numbers 1 – 18
  lty=1,           # line type (for type="l"). Use
  numbers.
  lwd=2,           # line width (for type="l"). Use
  numbers.
  axes = "L"               # L= F, T
  xlab ="string", ylab="string"           # Labels
  on axes
  sub = "string", main ="string"    #Subtitle for
  plot
  xlim = c(lo,hi), ylim= c(lo,hi)   #Ranges for axes
)
```

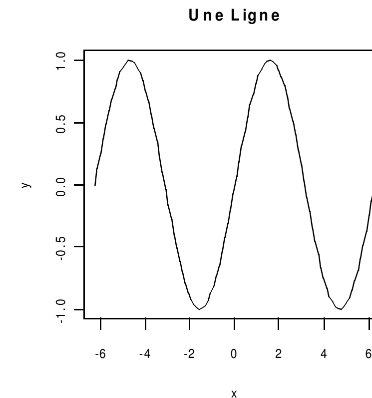
*And some more.*

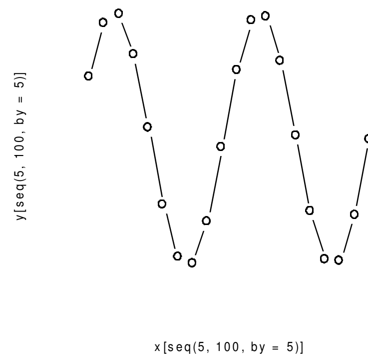
*Try it out, play around, read* `help(plot)`

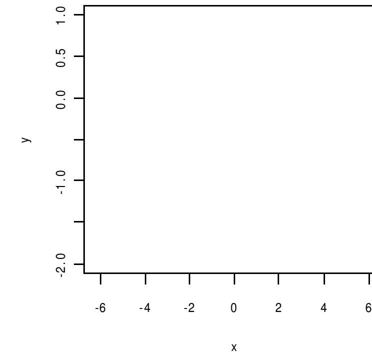
# More example graphics with `plot()`

```
> x <- seq(-2*pi,2*pi,length=100)
> y <- sin(x)

> par(mfrow=c(2,2))          #mu
> 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))
```









# Multiple data in one plot

- Scatter plot
  1. > plot(firstdataX, firstdataY, col=“red”, pty=“1”, …)
  2. > points(seconddataX, seconddataY, col=“blue”, pty=“2”)
  3. > points(thirddataX, thirddataY, col=“green”, pty=3)
- Line plot
  1. > plot(firstdataX, firstdataY, col=“red”, lty=“1”, …)
  2. > lines(seconddataX, seconddataY, col=“blue”, lty=“2”, …)
- Caution:
  - Only plot( ) command sets limits for axes!
  - Avoid using plot( …., xlim=c(bla,blubb), ylim=c (laber,rhabarber))
- (There are other ways to achieve this)

# Logarithmic scaling

- plot() can do logarithmic scaling
  - `plot(…. , log=“x”)`
  - `plot(…. , log=“y”)`
  - `plot(…. , log=“xy”)`
- Double-log scaling can help you to see more. Try:

```
> x <- 1:10
> x.rand <- 1.2^x + rexp(10,1)
> y <- 10*(21:30)
> y.rand <- 1.15^y + rexp(10, 20000)
> plot(x.rand, y.rand)
> plot(x.rand, y.rand, log=“xy”)
```

# More nicing up your graph

| | |
|---|---|
| `> axis(1,at=c(2,4,5), legend("A","B","C"))` | Axis details ("ticks", lEgend, …) Use `xaxt="n"` or `yaxt="n"` inside `plot()` |
| `> abline(lsfit(x,y))` | Add an adjustment |
| `> abline(0,1)` | add a line of slope 1 and intercept 0 |
| `> legend(locator(1),…)` | Legends: very flexible |

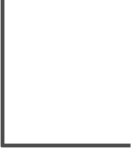
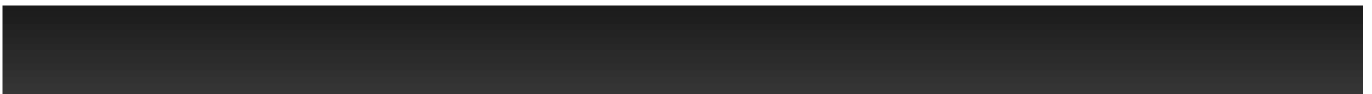
# Histogram

- A *histogram* is a special kind of bar plot
- It allows you to visualize the *distribution* of values for a numerical variable.  Naïvely:
  - Divide range of measurement values into, say, 10 so-called “bins”
  - Put all values from, say, 1-10 into bin 1, from 11-20 into bin 2, etc.
  - Count: how many values in bin 1?  In bin 2?  …
  - Then draw these counters
- 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: making a histogram

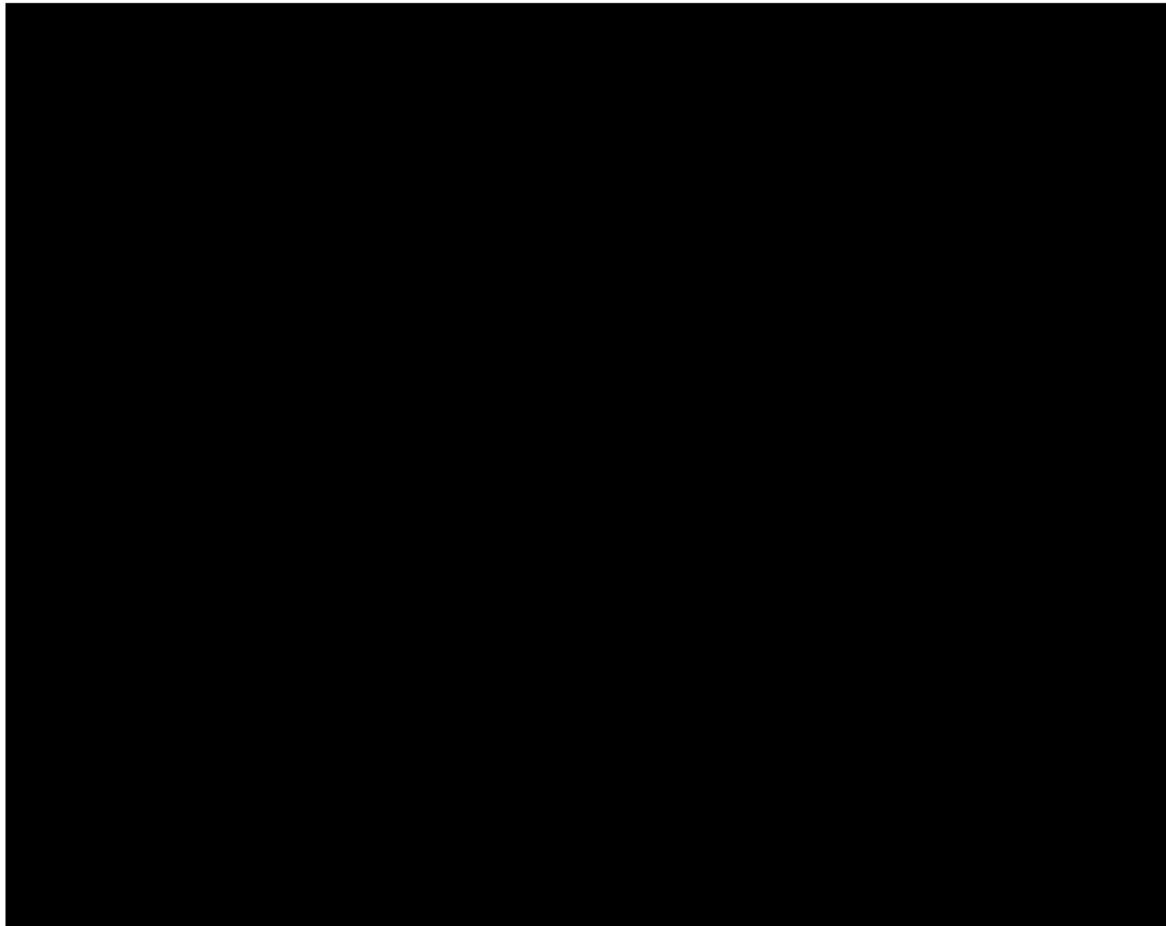
- Type `?hist` to view the help file
  - Note some important arguments, esp `breaks`
- Simulate some data, make histograms varying the number of bars (also called ‘bins’ or ‘cells’), e.g.

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

# R: setting your own breakpoints

```
> bps <- c(0,2,4,6,8,10,15,25)
> hist(simdata,breaks=bps)
```



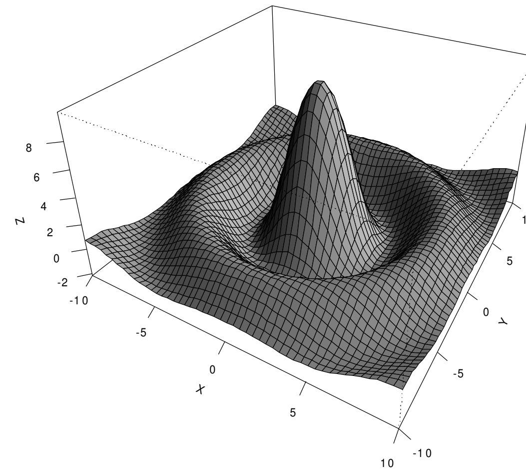
# Density plots

- Density: probability distribution
- Naïve view of density:
  - A “continuous”, “unbroken” histogram
  - “inifinite number of bins”, a bin is “inifinitesimally small”
  - Analogy: Histogram ~ sum, density ~ integral

- Calculate density and plot it

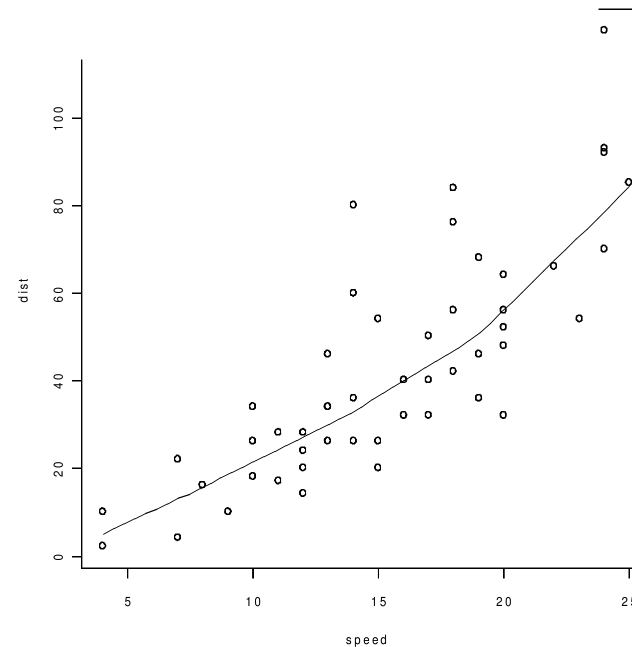
```
> x<-rnorm(200,0,1)        #create random numbers
> plot(density(x)) #compare this to:
> hist(x)
```

# Other graphical functions




See also:

```
barplot()
image()
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.

# Input / output

# Reading and writing files

- Different methods for input
  - Reading a vector (scan)
  - Reading a table (read.table, read.csv, …)
  - File handles
- Different methods for output
  - Writing single strings
  - Writing tables into a file (write.table)
  - Saving plots as PostScript, PNG, …
  - File handles

# Simple input

- Task: Read a file into a vector
- Input file looks like this:
  *1*
  *2*
  *17.5*
  *99*
- Read this into vector x:
  `x <- scan(“inputfile.txt”)`
- There are more options => `help(scan)`

# More complicated: Reading / writing tables

- Write a table into a file:
```
> x <- rnorm(100, 1, 1)
> write.table(x, file="numbers.txt")
```
    # There are more options => `help(write.table)`

- Read a table from a file:
```
> x <- read.table("in.txt", header=FALSE)
```
  # There are more options => `help(read.table)`

- Read a table from the Web:
```
> x <- read.table("http://www.net.in.tum.de/…"…)
```

# Universal: Using file handles

- File handles about as universal as in Perl
- Write two lines into a file:

```
> fh <- file(“output.txt”, “w”)    # w = write
> cat(“blah”, “blubb”, sep=“\n”, file=fh)
> close(fh)
```

- Write into a file and compress it using gzip:

```
> fh <- gzfile(“output.txt.gz”, “w”)
> cat(“blah blah blah”, … , file=fh)
```

- More examples: help(file)

- Also try “filenames” like *http://www.blabla.bla/data.gz*

# Graphical output: Saving your plots

- Output as (Encapsulated) PostScript:
```
> postscript("outputfile.eps")
> plot(data)              # You will not see this on screen!
> …                       # do some more graphics
> dev.off()               # write into file
```
  - There are many more options => help(postscript)
  - View the file using, e.g., `gv` program

- Output as PNG (bitmap):
  Simply replace `postscript()` above by `png()`:
```
> png("outputfile.png", width=800,
height=600, pointsize=12, bg="white")
```

# Useful built-in functions

# 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"
```

# Mathematical operations

Normal calculations : `+ - * /`
Powers: `2^5` or as well `2**5`
Integer division: `%/%`
Modulus: `%%` (`7%%5` gives `2`)

Standard functions: `abs(), sign(), log(), log10(), sqrt(), exp(), sin(), cos(), tan()`

To round: `round(x,3)` rounds to 3 figures after the point

And also: `floor(2.5)` gives 2, `ceiling(2.5)` gives 3

All this works for matrics, vectors, arrays etc. as well!

# Vector functions

```
> vec <- c(5,4,6,11,14,19)
> sum(vec)
[1] 59
> prod(vec)
[1] 351120
> mean(vec)
[1] 9.833333
> var(vec)
[1] 34.96667
> sd(vec)
[1] 5.913262
```

And also: `min()` `max()` `cummin()` `cummax()` `range()`

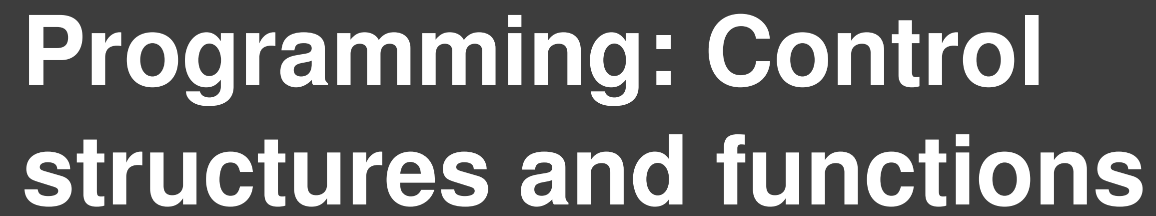
# Logical functions

R knows two logical values: **TRUE** (short **T**) et **FALSE** (short **F**). And **NA**.
Example:

| | |
|---|---|
| `==` | equals |
| `<` | less than |
| `>` | greater than |
| `<=` | less or equal |
| `>=` | greater or equal |
| `!=` | not equal |
| `&` | and |
| `\|` | or |

```
> 3 == 4
[1] FALSE
> 4 > 3
[1] TRUE

> x <- -4:3
> x > 1
[1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
> sum(x[x>1])
[1] 5
> sum(x>1)
[1] 2
```

Notez la différence !



# Programming: Control structures and functions

# Grouped expressions in R

```
x = 1:9

if (length(x) <= 10) {
  x <- c(x,10:20);      #append 10…20 to
  vector x
  print(x)
} else {
  print(x[1])
}
```

# Loops in R

```
list <- c(1,2,3,4,5,6,7,8,9,10)
for(i in list) {
    x[i] <- rnorm(1)
}

j = 1
while( j  < 10) {
  print(j)
  j <- j + 2
}
```

# Functions

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

```
> pleaseadd <- function(a,b) {
+ result <- a+b
+ return(result)
+ }
```

- Editing of functions:

  `> fix(pleaseadd)` # opens pleaseadd() in editor

  Editor to be used determined by shell variable $EDITOR

# Calling Conventions for Functions

- Two ways of submitting parameters
  - Arguments may be specified in the same order in which they occur in function definition
  - Arguments may be specified as name=value.
    Here, the ordering is irrelevant.
- Above two rules can be mixed!

```
> t.test(x1, y1, var.equal=F, conf.level=.99)
> t.test(var.equal=F, conf.level=.99, x1, y1)
```

# Missing Arguments

- R function can handle missing arguments two ways:
  - either by providing a default expression in the argument list of definition

  or
  - by testing explicitly for missing arguments

```
> 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.
- 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

```
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)
```

Even more datatypes:
Data frames and factors

# Data Frames (1/2)

- Vector: All components must be of same type
  List: Components may have different types
- Matrix: All components must be of same type
  => Is there an equivalent to a List?
- **Data frame:**
  - Data within each column must be of same type, but
  - Different columns may have different types (e.g., numbers, boolean,…)
  - Like a spreadsheet

Example:

```
> cw <- chickwts
> cw
   weight      feed
11    309       linseed
23    243       soybean
37    423       sunflower
…
```

# Data Frames (2/2)

- Data frame = special list with class “data.frame”.
  - But: restrictions on lists that may be made into data frames.
- Components must be
  - vectors (numeric, character, or logical)
  - Factors
  - numeric matrices
  - Lists
  - 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 and factors are included as-is
- Non-numeric 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.

# Subsetting in data frames (1/2)

Individual elements of a vector, matrix, array or data frame are accessed with “[ ]” by specifying their index, or their name

```
> cw = chickwts
> cw
   weight       feed
1     179       horsebean
11    309       linseed
23    243       soybean
...
> cw[3,2]
[1] horsebean
6 Levels: casein horsebean linseed ... sunflower
> cw [3,]
    weight      feed
37     423      sunflower
```

# Subsetting in data frames (2/2)

```
> an = Animals
> an
                         body  brain
Mountain beaver         1.350    8.1
Cow                   465.000  423.0
Grey wolf              36.330  119.5
> an [3,]
             body  brain
Grey wolf   36.33  119.5
```

# Labels in data frames

```
> labels (an)
[[1]]
 [1] "Mountain beaver"  "Cow"
 [3] "Grey wolf"        "Goat"
 [5] "Guinea pig"       "Dipliodocus"
 [7] "Asian elephant"   "Donkey"
 [9] "Horse"            "Potar monkey"
[11] "Cat"              "Giraffe"
[13] "Gorilla"          "Human"
[15] "African elephant" "Triceratops"
[17] "Rhesus monkey"    "Kangaroo"
[19] "Golden hamster"   "Mouse"
[21] "Rabbit"           "Sheep"
[23] "Jaguar"           "Chimpanzee"
[25] "Rat"              "Brachiosaurus"
[27] "Mole"             "Pig"

[[2]]
[1] "body"  "brain"
```

# Factors

- A normal character string may contain arbitrary text
- A **factor** may only take pre-defined values
  - “Factor”: also called “category” or “enumerated type”
  - Similar to `enum` in C, C++ or Java 1.5
- help(factor)

# Hash tables

# Hash Tables

- In vectors, lists, dataframes, arrays:
  - elements stored one after another
  - accessed in that order by their index == integer
  - or by the name of their row / column
- Now think of Perl’s hash tables, or java.util.HashMap

- R has hash tables, too

# Hash Tables in R

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

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

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

Object orientation

# Object orientation

- primitive (or: atomic) data types in R are:

- *numeric (integer, double, complex)*
- *character*
- *logical*
- *function*
- out of these, vectors, matrices, arrays, lists can be built

# Object orientation

- Object: a collection of atomic variables and/or other objects that belong together
- Similar to the previous list examples, but there's more to it.

- Parlance:
  - class: the “abstract” definition of it
  - object: a concrete instance
  - method: other word for ‘function’
  - slot: a component of an object (I.e., object variable)

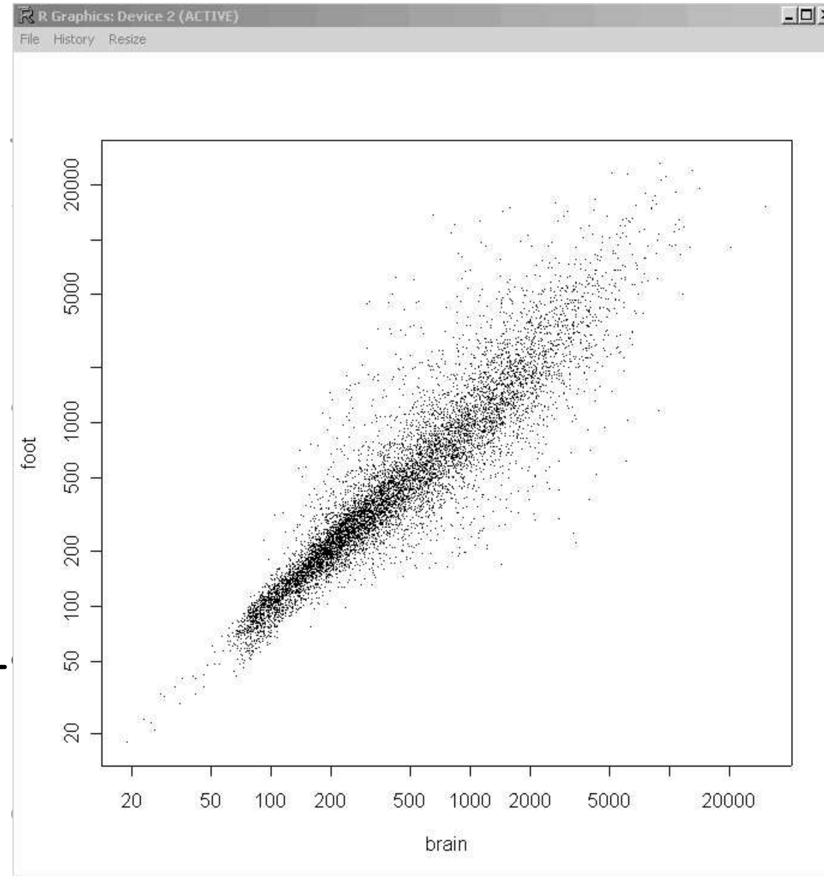
# Object orientation advantages

The usual suspects:

- Encapsulation (can use the objects and methods someone else has written without having to care about the internals)
- Generic functions (e.g. `plot, print`)
- Inheritance (hierarchical organization of complexity)

# Object orientation

```
library('methods')
setClass('microarray',                      ##
    representation(                         ##
        qua = 'matrix',
        samples = 'character',
        probes = 'vector'),
    prototype = list(                       ##
        qua = matrix(nrow=0, ncol=0)
        samples = character(0),
        probes = character(0)))

dat = read.delim('../data/alizadeh/l
z  = cbind(dat$CH1I, dat$CH2I)

setMethod('plot',                           ##
  signature(x='microarray'),                ##
  function(x, ...)
  plot(x@qua, xlab=x@samples[1], ylab=x@samples[2], pch='.', log='xy'))

ma = new('microarray',                      ## instantiate (construct)
        qua = z,
        samples = c('brain','foot'))
plot(ma)
```



# Object orientation in R

The `plot(pisa.linearmodel)` command is different from `plot(year,inclin)` .

`plot(pisa.linearmodel)`

R recognizes that `pisa.linearmodel` is a "lm" object.
Thus it uses `plot.lm()` .

Most R functions are object-oriented.

For more details see `?methods` and `?class`