



Applied Statistics: R

Semester 2018-I

Francisco Rosales Marticorena

Faculty of Economics & Finance

Table of Contents:

- 1 Introduction to R
- 2 Advanced Graphics
- 3 Data Management
- 4 Functions, Debugging & Condition Handling

Introduction to R

Week 1:

- 1 R basics, GitHub and \LaTeX
- 2 graphical methods in R

Week 2:

- 1 data management (import/export)
- 2 functions, debugging, condition handling

Week 3:

- 1 profiling, performance and parallelization
- 2 numeric methods and simulations

If possible there is time: Rcpp, building packages.

Classes and Coordination

Classes:

- One exam on mid-term week. 20% of your grade.
- More an R course than an “applied stats” course.
- 3 weeks, 5 hours \times week.
- TA comes the 3rd hour of the three-hour block.

Coordination:

- No Blackboard.
- Course repository [here](#).
- Use the Homework file to upload solutions of exercises.

Reading Material

- Hadley Wickham. **Advanced R**. The R series. CRC Press, 2015.
Available online: <http://adv-r.had.co.nz/>.
- Owen Jones, Robert Maillardet, and Andrew Robinson. **Introduction to scientific programming and simulation using R**. Chapman & Hall/CRC, 2009.
- Brian Everitt and Torsten Hothorn. **A handbook of statistical analyses using R**. Chapman & Hall/CRC, 2006.
- R Data Import/Export manual. Available online: <http://cran.r-project.org/doc/manuals/r-release/R-data.pdf>
- Deepayan Sarkar. **Lattice: multivariate data visualization with R**. Use R! Springer, 2008.
- Roger S Bivand, Edzer J Pebesma, and Virgilio Gómez-Rubio. **Applied spatial data analysis with R**, volume 10 of Use R! Springer, 2008.

Who are you?

- Ever used R?
- Text editor for R: Rstudio, emacs, Vim?
- Github, L^AT_EX?
- Operating system: Linux, Apple, Windows?
- Other programming languages: C, C++, Java, Python, Fortran, Julia, Matlab, Mathematica?

Heard of Laboratoria? kind of like that...

- Worldwide organization.
- Empowers women through coding.
- Third meet up on April 27th at UP Aula Magna H 15:00.
- Topic: Text-mining in Finance with R.
- Speaker: Leda Basombrío, Strategic Analysis Manager at BCP.

Not mandatory, but recommended. Not only for women.

help:	<code>?topic, help(topic), args(some function)</code>
assignments:	<code>x <- 5, x = 5, 5 -> x</code>
operators:	<code>+, -, *, /, ^, &, &&, , </code> ; see <code>help("+")</code>
comparisons:	<code>==, !=, >, >=, <, <=</code> ; see <code>help("==")</code>
loops:	<code>for, while, repeat</code>
comments:	everything that follows <code>#</code>

case sensitive: usage of CAPITAL and small letters matters!

Basic Data Structures

numeric (\mathbb{R}):	1, 301L, .141, 1.23e-3, NaN, Inf, -Inf
complex ($\mathbb{C} \setminus \mathbb{R}$)	1+0i, 1i, 3+5i
logical:	TRUE, FALSE, NA
character:	"hello", "I'm a string"
numeric:	no distinction between integers and doubles
missing:	stored as NA, and are logical.

Can check the basic structure with `str()`, `mode()` or `storage.mode()`.

```
str(1)          ## num 1
mode(1)         ## [1] "numeric"
storage.mode(1) ## [1] "double"
storage.mode(1L) ## [1] "integer"
mode(pi)        ## [1] "numeric"
storage.mode(pi) ## [1] "double"
str(LETTERS)    ## chr [1:26] "A" "B" "C" "D" "E" ...
```

Construction & Coercion

- vector can be constructed with `c()`
- coerced to a type `xxx` by `as.xxx()`
- when combining different data types, they will be coerced to the most flexible type
- coercion often happens automatically
- check if `xxx` is a specific type by `is.xxx()`

```
storage.mode(c(1,2L))      ## [1] "double"
storage.mode(c("a",1))     ## [1] "character"
x <- c(FALSE, TRUE, NA)
as.numeric(x)              ## [1] 0 1 NA
# Total number of TRUEs
sum(x, na.rm = TRUE)       ## [1] NA
```

In R, there three ways to represent “nothing”, but the reason for the missingness of the information can be distinguished:

NA	missing sample values
NaN	wrong math, e.g. $\log(-1)$, $1/0$
NULL	null pointer

```
c(3, NA)      ## [1] 3 NA
c(3, 0/0)     ## [1] 3 NaN
c(3, NULL)    ## [1] 3
max(3, NA)    ## [1] NA
```

Some math operations can be performed with `Inf` and `-Inf`:

```
max(3, Inf)
```

```
## [1] Inf
```

```
min(3, Inf)
```

```
## [1] 3
```

```
c(Inf + Inf, (-Inf) * Inf, Inf - Inf)
```

```
## [1] Inf -Inf NaN
```

Convolved Data Structures

- data structures in R can be organized by their dimensionality and if all their contents are of the same type (or not):

object	dimension	homogeneous	heterogeneous
1d	length()	atomic vector	list
2d	dim()	matrix	data frame
nd	dim()	array	–

- a `data.frame` is a matrix with different data type columns.
- a `list` can have different data type elements.

```
x <- matrix(1, nrow = 5, ncol = 2)
is.matrix(x)          ## [1] TRUE
as.vector(x)          ## [1] 1 1 1 1 1 1 1 1 1 1
x <- list(a = "Hallo", b = 1:10, pi = pi)
```

All objects can have arbitrary additional attributes, used to store metadata about the object.

- can be thought of as a named list (with unique names); other frequently encountered attributes: “dimnames”, “names”, “class”(!)
- can be accessed all at once (as a list) with `attributes()`, or individually with `attr()`.
- arrays are simply vectors with a “dim” attribute.
- factor is a vector with the “levels” attribute
- `as.xxx()` functions delete all attributes including dimensionality

Attributes

```
x <- matrix(1:10, ncol = 5)
attributes(x)                ## $dim
                             ## [1] 2 5

rownames(x) <- c("Eins", "Zwei")
attributes(x)                ## $dim
                             ## [1] 2 5
                             ## $dimnames
                             ## $dimnames[[1]]
                             ## [1] "Eins" "Zwei"
                             ## $dimnames[[2]]
                             ## NULL

as.character(x)              ## [1] "1"  "2"  "3"  "4"  "5"...
attributes(as.character(x))  ## NULL
```


- There are three subsetting operators: `[`, `[[`, and `$`
- the three types of subsetting:
 - Positive integers return elements at the specified positions.
 - Logical vectors select elements where the corresponding logical value is `TRUE`; application of logical expressions.
 - character vectors to return elements with matching names.
- important differences in behaviour of different objects (e.g., vectors, lists, factors, matrices, and data frames).
- More advanced subsetting, in particular in combination with convoluted logical expressions, can be done using the functions `subset()` and `which()`.
- The default `drop=TRUE` simplifies the data type of the result.

Atomic Vectors

Use “[”-operator and number, logical vector or name of the element you want to pull out.

```
x <- c(2.1, 4.2, 3.3, 5.4)
x[c(3, 1)]          ## [1] 3.3 2.1
x[-c(3, 1)]         ## [1] 4.2 5.4
x[c(TRUE, TRUE, FALSE, FALSE)] ## [1] 2.1 4.2

(y <- setNames(x, letters[1:4])) ##  a    b    c    d
## 2.1 4.2 3.3 5.4
y[c("d", "c", "a")]      ##  d    c    a
## 5.4 3.3 2.1
```

Subsetting matrices and arrays with “[”-operator like vectors, while the dimension is separated by comma:

```
x <- matrix(1:10, ncol=2)
colnames(x) <- c("Eins", "Zwei")
```

```
x[1:2,]           # results a matrix
x[,"Zwei"]         # results a vector
x[,"Zwei", drop = FALSE] # results a matrix
x[-3,]            # everything, but not third row
```

Subsetting lists with "["-operator returns always a list, while [[, and \$ pull out elements of the list:

```
x <- list(a = "Hallo", b = 1:10, pi = pi)
```

```
x$a      # first element of the list
```

```
x[['a']]
```

```
x[[1]]
```

```
x[1]     # list with one element
```

```
x[2:3]   # list with two elements
```

```
x[[2:3]] # wrong result
```

Data Frames

Data frames possess the characteristics of both lists and matrices: if you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices.

```
iris[1:10,]           # data frame with 10 rows
iris[,1]              # numerical
iris$Sepal.Length     # the name
iris$Sepa1.Length     # Oops! what happened?
iris["Sepal.Length"]  # again first column
iris["Sepa1.Length"]  # Error:  undefined columns selected
iris[,1, drop = FALSE] # data frame with one column
```

- conditional evaluation: if, else, ifelse
- loops: for, while, repeat, switch

basic vocabulary:

if, &&, ||

for, while, repeat

next, break

switch

ifelse

```
if (<test>) {  
  <expression1>  
} else {  
  <expression2>  
}
```

- else block is optional
- <test> has to result in a value that is TRUE or FALSE
- only the first element of <test> is used, ow a warning is triggered
- to eval more than one statement use &, | or all() and any()
- can be nested

for Loop

```
for (<var> in <vector>) {  
  <expression>  
}
```

```
sum <- 0  
for(i in 1 : length(x)) {  
  sum <- sum + x[i]  
}
```

```
sum <- 0  
for(x_value in x) {  ## more efficient  
  sum <- sum + x_value  
}
```

Use `seq_along(x)` instead of `1:length(x)`

while Loop

```
while(<test>) {  
  <expression>  
}
```

E.g., the sum until the first NA:

```
sum <- 0  
i <- 1  
while((i <= length(x)) && !is.na(x[i])) {  
  sum <- sum + x[i]  
  i <- i + 1  
}
```

Be aware of infinite loops!

- next jumps to the next iteration in for or while loops
- break terminates for or while loops.

```
x <- c(1, 1, 1, NA, 2)
sum <- 0
for(val in x) {
  if(is.na(val)) break
  sum <- sum + val
}
```

```
x <- c(1, 1, 1, NA, 2)
sum <- 0
for(val in x) {
  if(is.na(val)) next
  sum <- sum + val
}
```

Style Example: Bad

```
fWLM<-function(y,X_mat,w){T0<-t(X_mat)%*%diag(w)%*%X_mat
t<-system.time({t_1<-solve(T0)%*%t(X_mat)%*%(w*y);
t2<-X_mat%*%t_1})
return(list(beta=t_1,hat=t2,stddev=sqrt(sum(w*(t2-y)^2))/
(length(y)-ncol(X_mat)), wts=w,t=t[[3]]))}
```

Style Example: Good

```
fit_weighted_lm <- function(response, design, weights) {  
  n_obs <- length(response)  
  n_coef <- ncol(design)  
  time_start <- Sys.time()  
  wcrossprod_design <- crossprod(design * weights, design)  
  weighted_response <- weights * response  
  coef <- solve(wcrossprod_design, t(design) %*% weighted_response)  
  time <- Sys.time() - time_start  
  fitted <- design %*% coef  
  residuals <- response - fitted  
  weighted_rss <- sum(weights * residuals^2)  
  sd_resid <- sqrt(weighted_rss / (n_obs - n_coef))  
  list(coef      = coef,  
        fitted   = fitted,  
        sd_resid = sd_resid,  
        weights  = weights,  
        time     = time)  
}
```

- find meaningful file names; if files need to be run in sequence, prefix them with numbers.

0-download.R

1-parse.R

2-explore.R

- avoid uppercase
- use an underscore to separate words within a name
- use nouns for variable names and verbs for function names

- strive for names that are concise and meaningful (this is not easy!).

# Good	# Bad
day_one	first_day_of_the_month
day_1	dayone
	djm1

- avoid using names of existing functions and variables.

```
# Bad
T <- FALSE
c <- 10
t <- temporal_variable
mean <- function(x) sum(x)
```

Formatting

- strive to limit your code to 80 characters per line.
- when indenting your code, use two spaces. Never use tabs.
- place spaces around all infix operators (=, +, -, <-, etc.), before parentheses, and after comma (just like in regular English)

Good

```
average <- mean(feet / 12 + inches, na.rm = TRUE)
```

Bad

```
average<-mean(feet/12+inches,na.rm=TRUE)
```

Good

```
if (debug) do(x)
```

```
plot(x, y)
```

Bad

```
if(debug)do(x)
```

```
plot (x, y)
```

- an opening (or closing) curly brace should always be followed by a new line, unless it's followed by `else`.

```
if (y == 0) {  
  log(x)  
} else {  
  y^x  
}
```

- use commented lines of `-` and `=` to break up your file into chunks.

```
# Load data -----  
# Plot data -----
```

- `formatR::tidy_source(source="input.R",file="output.R")`
cleans up and does some automatic formatting

Advanced Graphics

- creating interest and attention of the reader
- essential meaning can be visualized at a glance
- comprehensive picture of a problem gives more complete and balanced understanding
- the human visual system is very powerful in detecting patterns: outliers, diagnose models, search for perhaps unexpected phenomena

- A graphical device can be thought as a paper on which you can draw with different pens and colours, but nothing can be deleted.
- It can be opened more than one device, but there is only one active.
- There is no difference no matter which device is used.
- Typical steps to produce a graphic is:
 - 1 start device, e.g. `pdf('testgraphics.pdf')`
 - 2 generate graphic, e.g. `plot(1:10)`
 - 3 close device: `dev.off()`
- If no device is open, using a high-level graphics function will cause a device to be opened.

The following graphics devices are currently available:

- `pdf()`: write PDF graphics commands to a file; can be handy for distribution to cooperation partners, integration in PDF \LaTeX , or viewing many graphics
- `postscript()`: writes PostScript graphics commands to a file
- `bitmap()`: bitmap pseudo-device via ‘Ghostscript’ (if available).

Interactive plotting with GUI:

- `x11()`: The graphics device for the X11 windowing system
- `png()`: compressed Bitmap, without loss
- `jpeg()`: compressed Bitmap with information loss, optimized for pictures with many color shades

For more info see `?Devices`.

High-level Plots

High-level functions generate/initialize a graphic, e.g.:

<code>plot()</code>	depend of context
<code>barplot()</code>	Barplot
<code>boxplot()</code>	Boxplot
<code>coplot ()</code>	Conditioning plots
<code>contour()</code>	Contour line plot
<code>curve()</code>	Plotting functions
<code>dotchart()</code>	Dot Plots
<code>hist()</code>	Histogram
<code>image()</code>	Countour Plot (3. Dim. as color)
<code>mosaicplot()</code>	Mosaicplots (categorical data)
<code>pairs()</code>	Scatterplot matrix
<code>persp()</code>	perspective surface
<code>qqplot()</code>	QQ-Plot

High-level Plots

Many functions can be applied to different object types. They react in a “intelligent” way, so that a meaningful graphic can be found.

```
plot(trees)                                ## scatterplot matrix
plot(Volume ~ Girth, data = trees)         ## scatterplot

tree.lm <- lm(Volume ~ Girth, data = trees)
abline(tree.lm)                           ## regression line
plot(tree.lm)                             ## residual/diagnostic plots
boxplot(trees)                            ## boxplots
qqnorm(trees$Volume)                      ## quantile plot
```

- for nicer graphics, the `par` graphical parameters can be adapted
- some graphical parameters can be adjusted in high-level functions
- for help check `?plot`, `?plot.default`, or more comprehensive `?par`

Graphical Parameter

Some graphical parameters can be set in high-level functions like `plot()`. For example:

<code>axes</code>	should the axes be plotted?
<code>col</code>	color
<code>log</code>	logarithmic scale
<code>main, sub</code>	title and subtitle
<code>pch</code>	symbol for points
<code>type</code>	type (l=line, p=point, b=both, n=none)
<code>xlab, ylab</code>	x-/y-axis label <code>xlim, ylim</code>
<code>xlim, ylim</code>	x-/y-axis range

Graphical Parameter

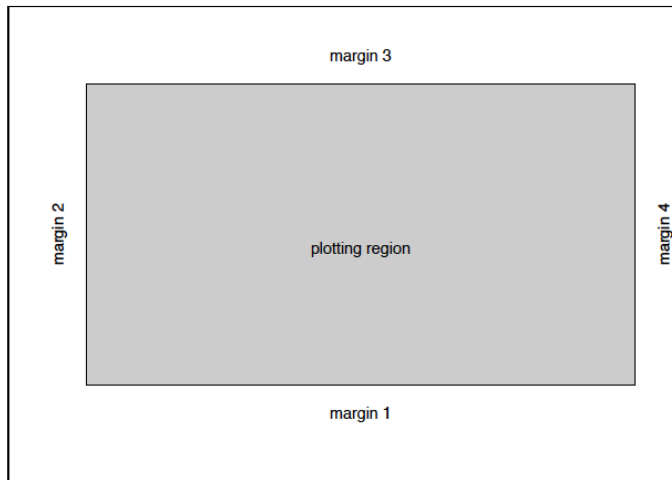
The most commonly used arguments in `par()`:

<code>bg</code>	background color
<code>cex</code>	size of a point or a letter
<code>las</code>	should labels be placed parallel wrt the axes
<code>lty, lwd</code>	line type (dashed, ...) and line width
<code>mar</code>	size of the margins
<code>mfc col, mfrow</code>	multiple plots in one device in rows/columns
<code>mfg</code>	which plot in a device should be chosen?
<code>oma</code>	size of the outer margins
<code>usr</code>	current extrema of the user coordinates
<code>xaxt, yaxt</code>	x-/y-axis scaling

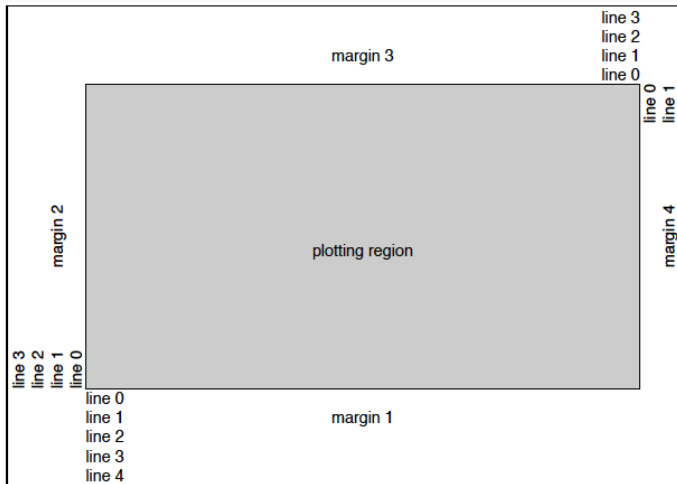
Graphical Parameter

```
opar <- par(mfrow = c(1, 1),bg = "White")
# example 1
par(mfrow = c(2, 2))
boxplot(trees, col = "blue")
hist(trees$Volume, las = 1)
qqnorm(trees$Volume, cex.axis = 2, pch = (trees$Girth > 14) + 8)
plot(trees$Girth,trees$Height,cex = scale(trees$Volume,center=FALSE))
par(opar)

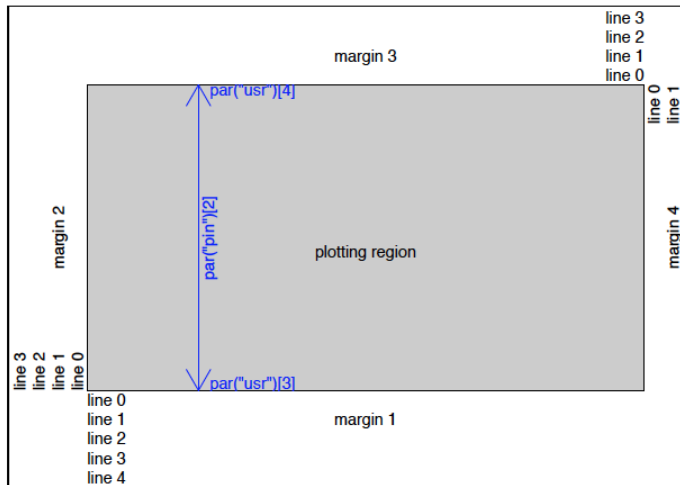
# example 2
set.seed(123)
x <- rnorm(100)
par(bg = "lightgreen")
hist(x, freq = FALSE, col = "red", las = 1,
     xlim = c(-5, 5), ylim = c(0, 0.6),
     main = "100 Draws from N(0,1)-distributed random variables")
curve(dnorm, from = -5, to = 5, add = TRUE, col = "blue", lwd = 3)
par(opar)
```



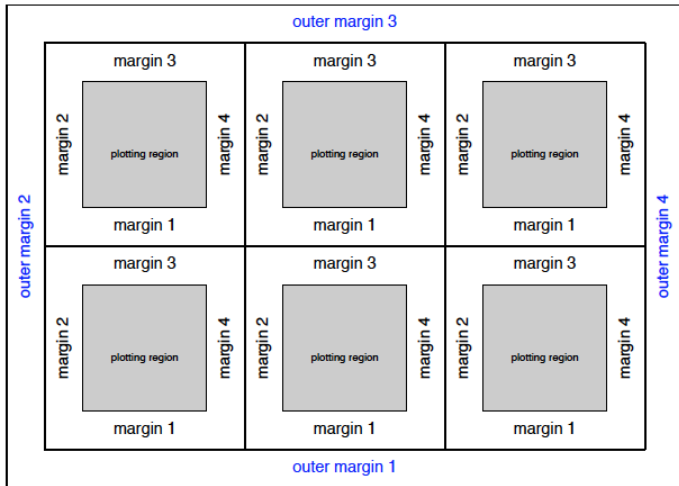
Device Control



Device Control



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



layout() Function

- layout() organizes independent plots on one plotting device, also in irregular grids
- boxes can have different widths
- neighboring boxes can be combined
- boxes can be left empty

```
m <- matrix(c(1,1,0,2), 2, 2)
```

```
m
```

```
##      [,1]  [,2]
```

```
## [1,]    1    0
```

```
## [2,]    1    2
```

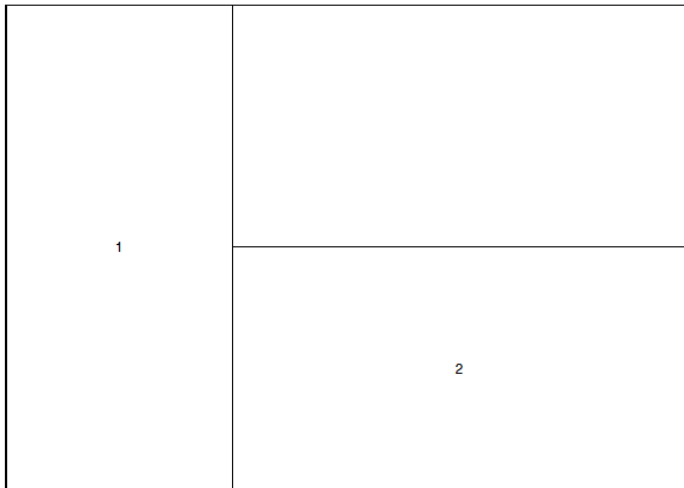
```
layout(m, widths=c(1,2))
```

```
x <- rnorm(100)
```

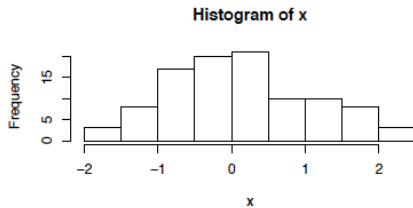
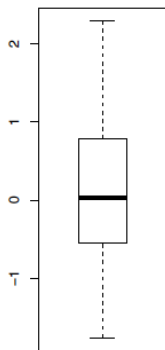
```
boxplot(x)
```

```
hist(x)
```

layout() Function



layout() Function



Low-level Graphics

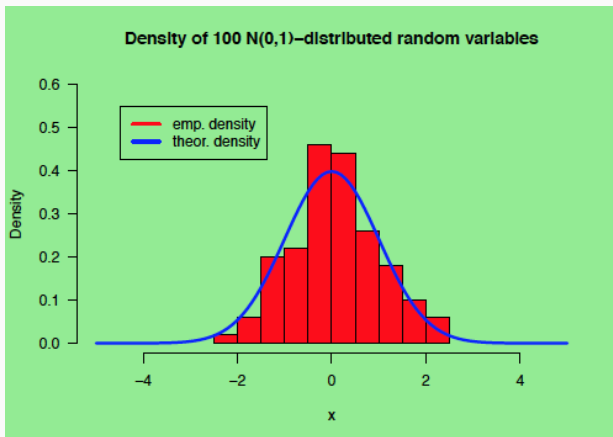
Low-level functions add elements to a (with high-level function) generated graphic, e.g., additional points, legends, etc.

<code>abline()</code>	“intelligent” lines
<code>arrows()</code>	arrows
<code>axis()</code>	axes
<code>grid()</code>	gridlines
<code>legend()</code>	legend
<code>lines()</code>	(stepwise) lines
<code>mtext()</code>	text in margins
<code>points()</code>	points
<code>polygon()</code>	(filled) polygons
<code>segments()</code>	vector lines
<code>text()</code>	text
<code>title()</code>	title label

Low-level Graphics

```
# example 2 (cont.):
```

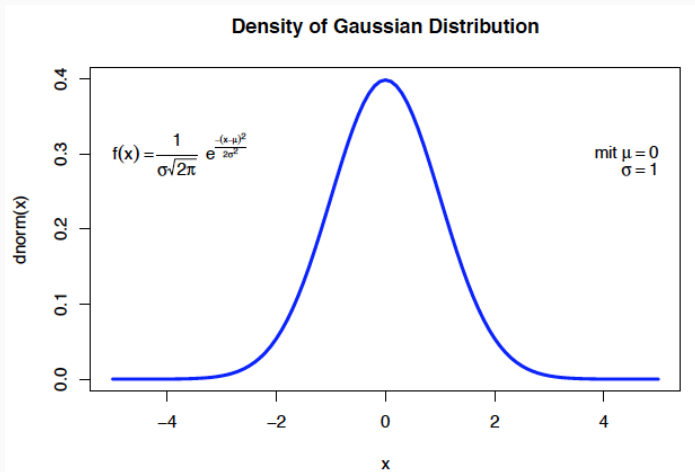
```
legend(-4.5, 0.55, legend = c("emp. density", "theor. density"),  
      col = c("red", "blue"), lwd = 3)
```



Mathematical Expressions

- mathematical notation and symbols formatted similar to \LaTeX code can be integrated in functions such as `axis()`, `legend()`, `mtext()`, `text()`, and `title()`
- For help check `?plotmath` or run `demo(plotmath)`

```
curve(dnorm, main = "Density of Gaussian Distribution",  
      from = -5, to = 5, col = "blue", lwd = 3)  
text(-3, 0.3,  
      expression(f(x) == frac(1, sigma * sqrt(2*pi)) ~  
                  e^{frac(-(x - mu)^2, 2 * sigma^2)}))  
text(4, 0.3, expression(paste("mit ", mu == 0)))  
sigma <- 1  
text(4.2, 0.28, bquote(sigma == .(sigma)))
```

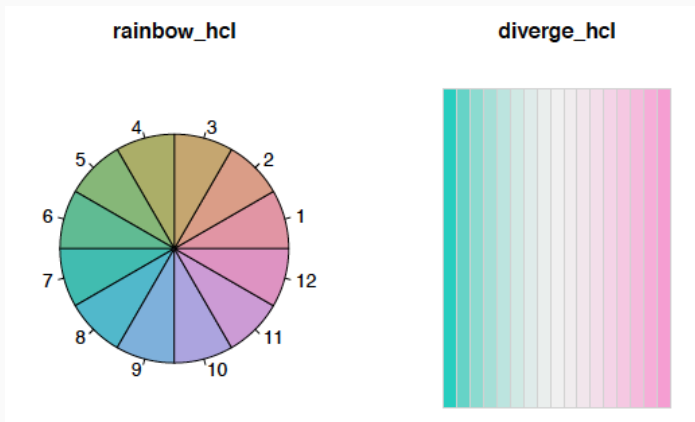


colorspace

The package `colorspace` provides various functions for perceptually-balanced color palettes

```
rainbow_hcl(12)
```

```
diverge_hcl(17, h = c(180, 330), c = 59, l = c(75, 95))
```

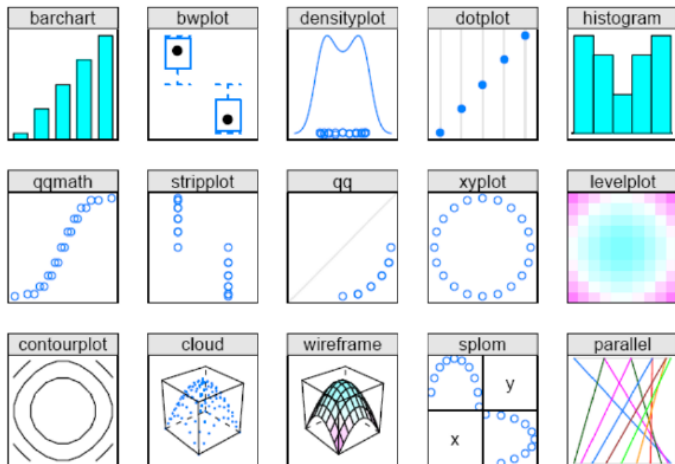


- trellis graphics is a family of techniques for viewing complicated data sets, that are based on basic concepts of human perception
- everything is possible (using a sufficient number of parameters)
- the trellis graphics system is in the `lattice` package
- the typical format is

```
graph_type(formula, data = )
```

<code>barchart()</code>	barplot
<code>bwplot()</code>	boxplot
<code>cloud()</code>	3D point clouds
<code>contourplot</code>	3D contour plot
<code>densityplot()</code>	kernel density plot
<code>dotplot()</code>	point plots
<code>histogram()</code>	histogram
<code>levelplot()</code>	levelplots
<code>panel.....()</code>	functions to add elements
<code>piechart()</code>	pie diagram
<code>print.trellis()</code>	plotting trellis object
<code>qq()</code>	QQ-plots
<code>stripplot</code>	strip plots
<code>wireframe()</code>	persp. 3D areas
<code>xyplot()</code>	scatterplot

Trellis Graphics



```
require(lattice)
attach(mtcars)
# create factors with value labels
gear.f <- factor(gear, levels = c(3,4,5),
  labels = c("3gears", "4gears", "5gears"))
cyl.f <- factor(cyl, levels = c(4,6,8),
  labels = c("4cyl", "6cyl", "8cyl"))

# kernel density plot
densityplot(~mpg,
  main = "Density Plot",
  xlab = "Miles per Gallon")
# kernel density plots by factor level
densityplot(~mpg|cyl.f,
  main = "Density Plot by Number of Cylinders",
  xlab = "Miles per Gallon")
```

The basic R system does not allow many possibilities for interactive graphics. Some exceptions are:

- `identify()` identifies a selected data point, e.g.:

```
x <- rnorm(10)
plot(x)
identify(x)
```

- `locator()` returns the coordinates of a selected point, which can be used for instance for the interactive placing of labels:

```
plot(x)
legend(locator(1), legend = "A legend", pch = 1)
```

- First analysis: Brian Everitt and Torsten Hothorn. [A handbook of statistical analyses using R](#). Chapman & Hall/CRC, 2006
- Trellis: Deepayan Sarkar. [Lattice: multivariate data visualization with R](#). Use R! Springer, 2008
- Colorspace: Achim Zeileis, Kurt Hornik, and Paul Murrell. [Escaping RGBland: Selecting colors for statistical graphics](#). Computational Statistics & Data Analysis, 53:3259-3270, 2009. doi: 10.1016/j.csda.2008.11.033

Data Management

Data Import and Export

- can take more time than the statistical analysis itself
- majority of the data is spreadsheet-like data
- Easier to import badly formatted data than explaining what a “good” formatted data is
- R is no good to handle large-scale data
- In practice, you will be often faced with data corrections and updates

Data Import Vocabulary

```
# Reading data
data                # loads specified data sets
read.table          # Reads a file in table format
read.fwf            # Read a table of fixed width formatted data
load                # Reload 'RData' datasets
library(foreign)    # Read Data Stored by Minitab, SAS, SPSS, ...
```

Data Import Vocabulary

Files and directories

setwd, getwd	# set and get current working directory
dir	# names of files or directories in a directory
dirname	# returns the directory name
normalizePath	# convert file paths to canonical form
file.choose	# choose a file interactively
download.file	# download a file from the Internet

low-level interface to the computer's file system:

file.copy, file.create, file.remove, file.rename, dir.create,
file.exists, file.info

Import Spreadsheet-like Data

To read spreadsheet-like data you need the function `read.table()` and variations like `read.csv()`, ...

<code>fileEncoding:</code>	use <code>latin1</code> for Windows data, and <code>utf-8</code> for Unix data
<code>header:</code>	names of variables (columns) and observations (rows)
<code>sep:</code>	field separator vs. record separator
<code>quote:</code>	protecting separators appearing in strings
<code>na.strings:</code>	which string (or number) represents missing values?
<code>colClasses:</code>	which type of variable is contained in which column?
<code>skip:</code>	number of lines skipped before beginning to read data

Importing from other statistical systems

The package `require(foreign)` provides import facilities for files produced by other statistical systems:

<code>read.epiinfo:</code>	reads fixed-width text format .REC files as R data frames
<code>read.mtp:</code>	imports 'Minitab Portable Worksheet'
<code>read.xport:</code>	reads a file in SAS Transport (XPORT) format
<code>read.ssd:</code>	generates a SAS program to convert the ssd contents to SAS transport format
<code>read.spss:</code>	reads files created by SPSS's 'save'/'export' commands
<code>read.dta:</code>	imports binary files created by Stata
<code>read.S:</code>	reads S's 'data.dump' files

Data Export: Basic Vocabulary

```
# Output
print, cat          # print in console
dput                # Writes an ASCII text representation of
                    # an R object to a file
sink, capture.output # Evaluates its arguments with the output
                    # being returned as a character string or
                    # sent to a file.

# Writing data
write                # The data (usually a matrix) are written
                    # to file
write.table, write.csv # converts the object to a data frame and
                    # prints it to a file
save                 # writes an external representation of
                    # R objects
```

- exporting from R is easier than importing
- normally exporting a text or csv file with `write.table` or `write.csv` is good enough
- `sink` diverts the standard R output to a file, and thereby captures the output of (possibly implicit) print statements
- `foreign::write.foreign` writes a code file that will write this text file into another statistical package
- the precision is governed by the current setting of `options(digits)`: Export for report writing or further analysis?

Why use a database?

- R is not well suited to large data sets. Large data objects can cause R to run out of memory
- provide fast access to selected parts of large databases.
- powerful ways to summarize and cross-tabulate columns in databases.
- store data in more organized ways than the rectangular grid model of spreadsheets and R data frames.
- concurrent access from multiple clients running on multiple hosts while enforcing security constraints on access to the data.
- ability to act as a server to a wide range of clients.

R Database Interface Packages

R Database Interface Packages:

DBI	interface between R and relational DBMS
RJDBC	access to databases through the JDBC interface
RMySQL	interface to MySQL database
RODBC	ODBC database access
ROracle	Oracle database interface driver
RpgSQL	interface to PostgreSQL database
RSQLite	SQLite interface for R

Example for SQL queries:

```
SELECT State, Murder FROM USArrests WHERE Rape > 30 ORDER BY Murder
SELECT sex, COUNT(*) FROM student GROUP BY sex
SELECT sch, AVG(sestat) FROM student GROUP BY sch LIMIT 10
```

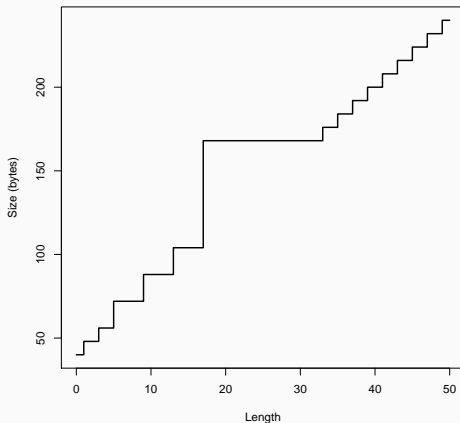
- some understanding of R's memory management will help you predict how much memory you will need
- `pryr::object_size()` tells you how much memory an object occupies (including environments)

```
require(pryr)
object_size(1:10)      # 88 B
object_size(mtcars)     # 6.74 kB
object_size(numeric()) # 40 B
```

- for more information see also `?Memory`

Memory Management

```
sseq = function(n){object_size(seq_len(n))}  
sizes = sapply(1:50, sseq)  
plot(1:50, sizes, xlab = "Length", ylab = "Size (bytes)",  
     type = "s")
```



Memory Management

- `pryr::mem_used()` tells you the total size of all objects in memory.
`pryr::mem_change()` tells you how memory changes during code execution
- R has automatic garbage collection, but it is lazy, so that it won't ask for memory until it is actually needed, otherwise use `gc()`

```
mem_used()           # 44.6 MB
mem_change(x <- 1:1e6) # 4.01 MB
mem_change(rm(x))     # -4 MB
```

```
mem_change(x <- 1:1e6) # 4 MB
mem_change(y <- x)     # 1.74 kB
mem_change(rm(x))      # 1.62 kB
mem_change(rm(y))      # -4 MB
```

Object Storage

Workspace:	is your current R working environment and includes any user-defined objects (vectors, matrices, data frames, etc).
Binary files:	<code>save()</code> allows the explicit saving of functions and data in binary file, that can be loaded by <code>load()</code>
Source code:	<code>source()</code> accepts its input from the named file or URL and runs the script in the current session

Manage your Workspace

<code>ls()</code>	lists the objects in your workspace.
<code>list.files()</code>	lists the files located in the folder's workspace
<code>rm()</code>	removes objects from your workspace; <code>rm(list = ls())</code> removes them all.
<code>sessionInfo()</code>	gives information about your session, i.e., loaded packages, R version, etc.
<code>R.version</code>	provides information about the R version.

Publication Quality Output and Documentation of Analysis:

<code>knitr:</code>	enables integration of R code into \LaTeX , LyX, HTML
<code>xtable:</code>	converts some R objects into \LaTeX code.
<code>R2HTML:</code>	converts your output text, tables, graphs in HTML format
<code>odfWeave:</code>	has functions that allow you to embed R output in ODF
<code>SWordInstaller:</code>	allows you to add R output to Word
<code>R2PPT:</code>	provides wrappers for adding R output to PPTs.

- Sweave allows you to embed R code in \LaTeX , producing attractive reports if you know that markup language.
- R's ability to output results for publication quality reports is somewhat rudimentary
- typewrite the results from R can be laborious, time-consuming and is a potential source for errors
- integration of R code and report allows the reproducibility of the analysis and assures good scientific practice
- the reports are easily updated with corrections and extension of the underlying data

Sweave Example

```
\documentclass[a4paper]{article}
\title{Sweave Example 1}
\author{Friedrich Leisch}
\begin{document}
\maketitle
In this example we embed parts of the examples from the
\texttt{kruskal.test} help page into a \LaTeX{} document:
<<>>=
data(airquality)
kruskal.test(Ozone ~ Month, data = airquality)
@
which shows that the location parameter of the Ozone
distribution varies significantly from month to month.
\begin{center}
<<fig=TRUE,echo=FALSE>>=
boxplot(Ozone ~ Month, data = airquality)
@
\end{center}
\end{document}
```

Sweave Chunk Options

In recent versions of R the way to run Sweave from the command line:

```
R CMD Sweave example-1.Snw  
pdflatex example-1
```

The most important options for Sweave code chunks are:

echo:	logical (TRUE). Show code in output file?
eval:	logical (TRUE). Evaluate code?
results:	character string: verbatim, tex, or hide.
fig:	logical (FALSE). Graphics?
eps, pdf:	logical (TRUE), EPS/PDF-Datei?
width, height:	numerical (6), width and height of graphics

Defaults can be adapted using `SweaveOpts()`

- **R Data Import/Export manual**. Available online: <http://cran.r-project.org/doc/manuals/r-release/R-data.pdf>
- Hadley Wickham. **Advanced R**. The R series. CRC Press, 2015; Chapter 18: Memory.
- Sweave webpage by Friedrich Leisch
<http://www.statistik.lmu.de/~leisch/Sweave/>

Functions, Debugging & Condition Handling

Motivation: Functions in R

- R is a functional programming language. This means that it provides many tools for the creation and manipulation of functions.
- The structure of a function is given by

```
function(<arglist>){  
    <body>  
}
```

- the keyword `function` indicates the beginning of a function
- arguments `<arglist>` lists the arguments of the function
- `<body>` is a block of R commands which are executed by the function

Function Components

All R functions are objects and consist of three parts:

<code>body()</code>	code inside the function.
<code>formals()</code>	arguments that controls how you can call the function.
<code>environment()</code>	location of the function's variables.

```
f <- function(x){  
  x^2  
}  
f           # > function(x) x^2  
formals(f)  # > $x  
body(f)     # > x^2  
environment(f) # > <environment: R_GlobalEnv>
```

Function Arguments

- When calling a function you can specify arguments by position, by complete name, or by partial name.
- Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
- Avoid using positional matching for less commonly used arguments, and only use readable abbreviations with partial matching.

```
f <- function(abcdef, bcde1, bcde2) {  
  list(a = abcdef, b1 = bcde1, b2 = bcde2)  
}  
  
str(f(1, 2, 3))           # ok  
str(f(2, 3, abcdef = 1)) # ok  
str(f(2, 3, a = 1))       # ok  
str(f(1, 3, b = 1))       # Error
```

Default and Missing Arguments

- Function arguments in R can have default values.
- They can be also defined in terms of other arguments.
- You can determine if an argument was supplied or not with the `missing()` function.

```
myplot <- function(x, y, mycol = 'red') {  
  if(missing(y)) {  
    y <- x  
    x <- 1:length(y)  
  }  
  plot(x, y, col = mycol)  
}  
myplot(1:20)  
myplot(1:20, rnorm(20), mycol='darkgreen')
```

- `match.arg` matches arguments against a table of candidate values specified by choices

```
match.arg(arg, choices, several.ok = FALSE)
```

- missing argument will be replaced with the first candidate

```
tsex <- function(sex = c("Male", "Female")) {  
  match.arg(sex)  
}  
  
tsex("F")    # [1] "Female"  
tsex()       # [1] "Male"  
tsex("W")  
  
# Error in match.arg(sex) (from #2) :  
# 'arg' should be one of "Male", "Female"
```

Lazy Evaluation

- R function arguments are evaluated if they're actually used.

```
myfun <- function(x, y){  
  if(x < 0){  
    return(NaN)  
  }else{  
    return( y * log(x))  
  }  
}
```

```
myfun(-1)    # NaN
```

```
myfun(2,3)   # 2.079442
```

```
myfun(2)     # Error in myfun(2) : argument "y" is missing,  
             # with no default
```

- If you want to ensure that an argument is evaluated you can use `force()`
- More technically, an unevaluated argument is called a 'promise'.

The Argument ...

- Functions can have any number of arguments using the special argument “...”
- The argument “...” will match any arguments not otherwise matched, and can be easily passed on to other functions.
- Using “...” comes at a price – any misspelled arguments will not raise an error, and any arguments after “...” must be fully named.

```
myplot <- function(x, y, myarg, ...){  
  # optional calculation with x, y and myarg  
  # call standard plot function with additional  
  # unspecified arguments from ...  
  plot(x, y, ...)  
}
```


Connecting the “...”

- “...” can be passed to any number of functions, but the arguments are the same
- Different argument lists can be passed to different functions by using `do.call()`:

```
myfun <- function(x, fun2.args = NULL, fun3.args = NULL, ...){  
  # calculations  
  fun1(x, ...)  
  do.call(fun2, fun2.args)      # first arg is either function  
  do.call("fun3", fun3.args)    # or character  
  
  # further calculations  
}
```

Return Values

- The last expression evaluated in a function becomes the return value, the result of invoking the function.
- Functions can return only a single object, so that you have to combine a number of objects in a list.
- Functions can return invisible values, which are not printed out by default when you call the function.

```
hist_2by2 <- function(data, args, ...) {  
  opar <- par(no.readonly = TRUE)  
  on.exit(par(opar))  
  par(mfrow = c(2, 2))  
  # do histogram plots for all variables in the data frame  
  invisible(apply(data, MARGIN = 2, hist, ...))  
}  
hist_2by2(trees)
```

- With a good technique, you can debug a problem with just `print()`.
- Wickham [2015] provides an outline for a general procedure for debugging:
 - 1 realize that you have a bug: implement testing procedures!
 - 2 make it repeatable: create a minimal example
 - 3 figure out where it is: identify the line of code that's causing the bug
 - 4 fix it and test it: ensure you fixed the bug

Basic Debugging Vocabulary

Make the bug reproducible:

- make sure your workspace is empty
- set seed for drawing random numbers
- create a minimal example

Bug Identification

Binary search:	you repeatedly remove half of the code until you find the bug.
<code>traceback()</code> :	determines the sequence of calls that lead up to an error (call stack).
<code>browser()</code> , <code>debug()</code> :	starts an interactive console in the environment where the error occurred.
<code>options(error = c(NULL, browser, recover))</code>	

Bug Identification: traceback()

The first tool is the call stack, the sequence of calls that lead up to an error.

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
```

```
f(10)
```

```
# Error in "a" + d (from #1) : non-numeric argument to binary
# operator
```

```
traceback()
```

```
# 4: i(c) at #1
# 3: h(b) at #1
# 2: g(a) at #1
# 1: f(10)
```

Additional Notes on Debugging

- prevent bugs: Instead of trying to write one big function all at once, work interactively on small pieces.
- a function may generate an unexpected message. There is no built-in tool to help solve this problem, but it is possible to create one:

```
message2error <- function(code) {  
  withCallingHandlers(code, message = function(e) stop(e))  
}
```

- your code might crash R completely, which indicates a bug in the underlying C code.

- some error exceptions are no bugs:
 - numerical problems (over-/underflow)
 - unsuitable (user) inputs
 - convergence failure during iterative algorithms
- successful failure of code: When writing a function, you can often anticipate potential problems and communicate these to the user is the job of conditions:

`stop()` Fatal errors force all execution to terminate.

`warning()` Display potential problems

`message()` Give informative output. Also `texttt{suppressMessages()}`

The behavior of error and warning can be adapted with `options()`

stop()

With `stop()` fatal errors are raised and force all execution to terminate, when there is no way for a function to continue.

```
f <- function(x) {  
  if (!is.numeric(x))  
    stop("supplied x is not numeric.")  
  s <- sum(x)  
  message("sum = ", s)  
}  
f(1:10)    # 55  
sum("nonsense")  
# Error: invalid 'type' (character) of argument  
f("nonsense")  
# Error: supplied x is not numeric.
```


try-error()

- `try()` allows execution to continue even after an error has occurred.
- if unsuccessful it will return an (invisible) object of class "try-error"
- suppress the message with `try(..., silent = TRUE)`

```
f <- function(x, silent = TRUE) {  
  s <- try(sum(x), silent = silent)  
  if (inherits(s, "try-error")) {  
    warning("x of wrong type, returning NA.")  
    return(NA)  
  }  
  s  
}  
  
f(1:10)  
## [1] 55  
  
f("nonsense")  
## Warning: x of wrong type, returning NA.  
## [1] NA
```

tryCatch()

tryCatch() is a general tool for handling conditions: take different actions for errors, warnings, messages, and interrupts.

```
f <- function(x, silent = FALSE) {  
  s <- tryCatch(sum(x),  
    error = function(e) {  
      warning("x of wrong type, sum is NA.")  
      if (!silent) print(e)  
      return(NA)  
    },  
    finally = cat("buh bye!\n")  
  )  
  s  
}  
f("nonsense")  
# <simpleError in sum(x): invalid 'type' (character) of argument>  
# buh bye!  
# [1] NA  
# Warning: x of wrong type, sum is NA.
```

Further References

- Hadley Wickham. *Advanced R*. The R series. CRC Press, 2015. Chapter 6: Functions & Chapter 9: Exceptions and Debugging.
- Robert Gentleman and Luke Tierney. *A prototype of a condition system for R*. This describes an early version of R's condition system: <http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html>