



Applied Statistics: R

Semester 2018-I

Francisco Rosales Marticorena

Faculty of Economics & Finance

Table of Contents:

- | | | | |
|---|-----------------------|---|------------------------------------|
| 1 | Introduction to R | 6 | Numerics & Simulations |
| 2 | Advanced Graphics | 7 | Object Oriented Programming |
| 3 | Data Management | 8 | R Interaction with Other Languages |
| 4 | Functions & Debugging | 9 | Building R Packages |
| 5 | Performance | | |

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`

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

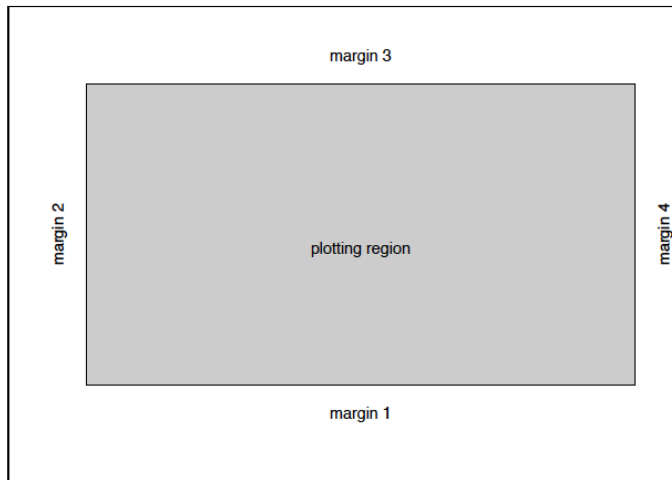
The most commonly used arguments in `par()`:

<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

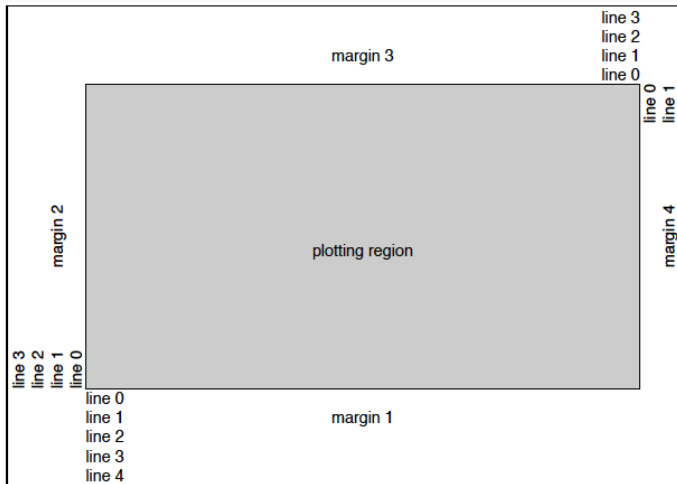
Graphical Parameter

```
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)

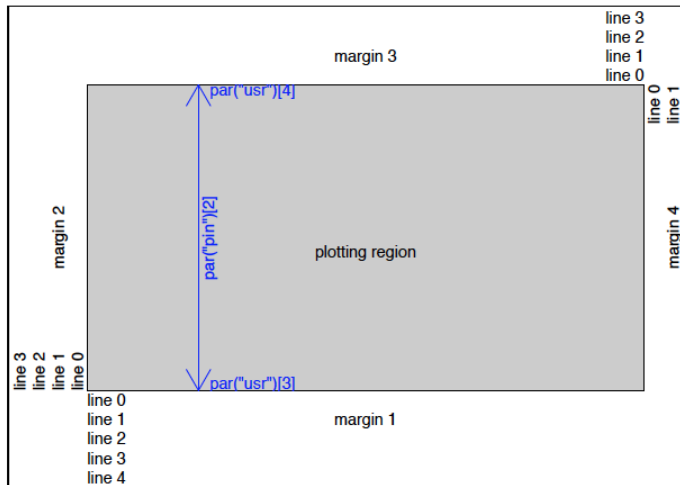
# example 2
set.seed(123)
x <- rnorm(100)
par(bg = "lightgreen")
hist(x, freq = FALSE, col = "red", las = 1,
     xlim = c(-5, 5), ylim = c(0, 0.6),
     main = "100 Draws from N(0,1)-distributed random variables")
curve(dnorm, from = -5, to = 5, add = TRUE, col = "blue", lwd = 3)
par(opar)
```



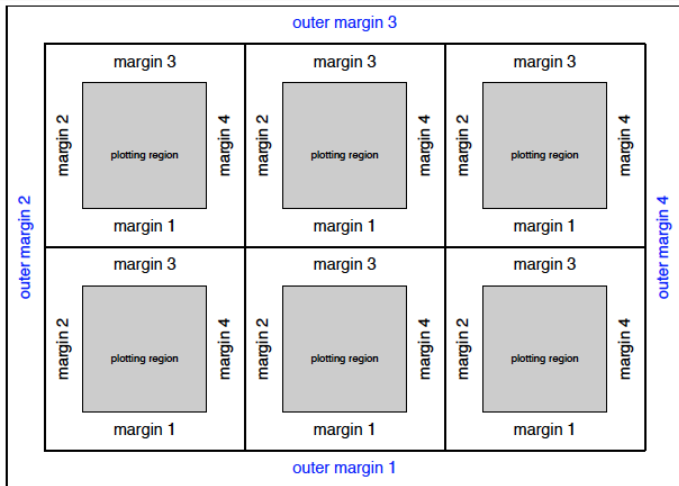
Device Control



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
```

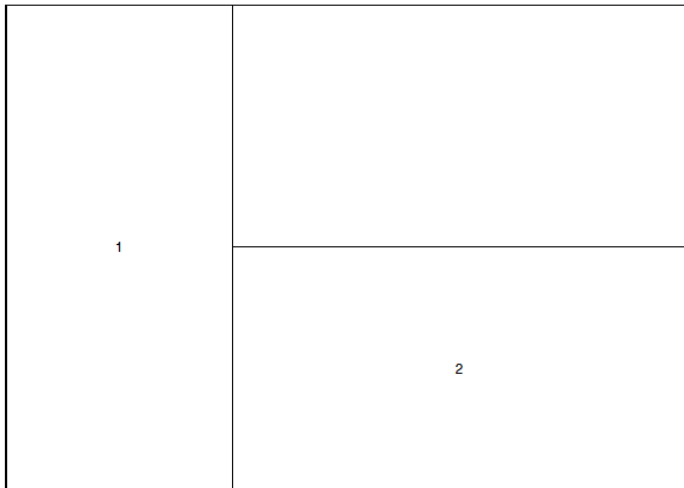
```
layout(m, widths=c(1,2))
```

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

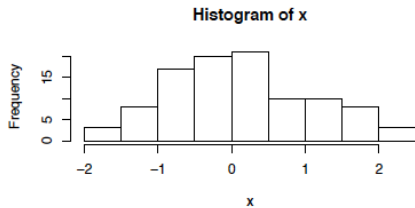
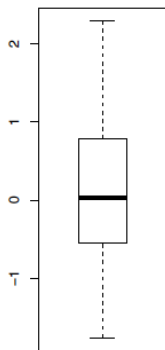
```
boxplot(x)
```

```
hist(x)
```

layout() Function



layout() Function



Low-level Graphics

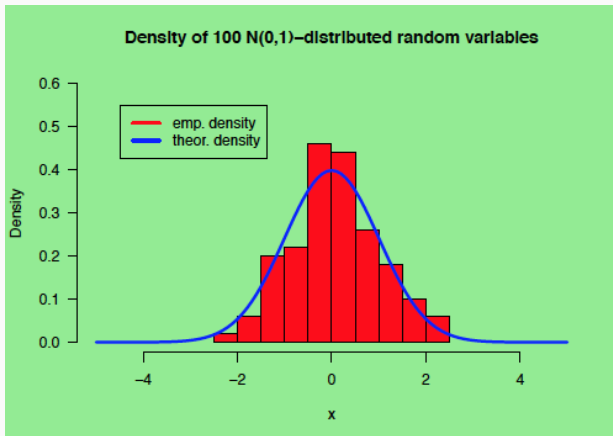
Low-level functions add elements to a (with high-level function) generated graphic, e.g., additional points, legends, etc.

<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.):
```

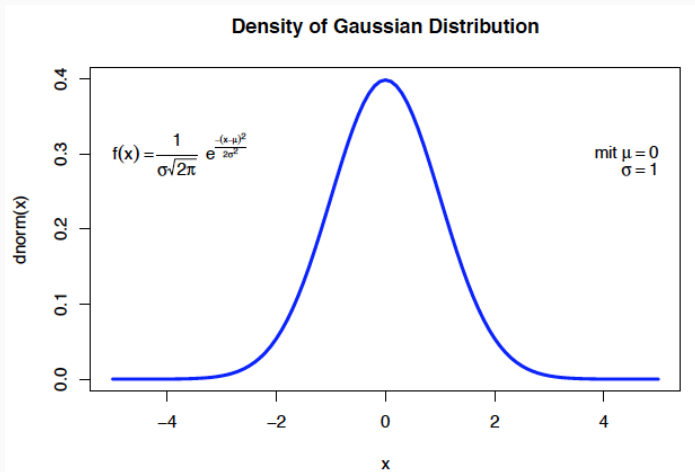
```
legend(-4.5, 0.55, legend = c("emp. density", "theor. density"),  
      col = c("red", "blue"), lwd = 3)
```



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)))
```

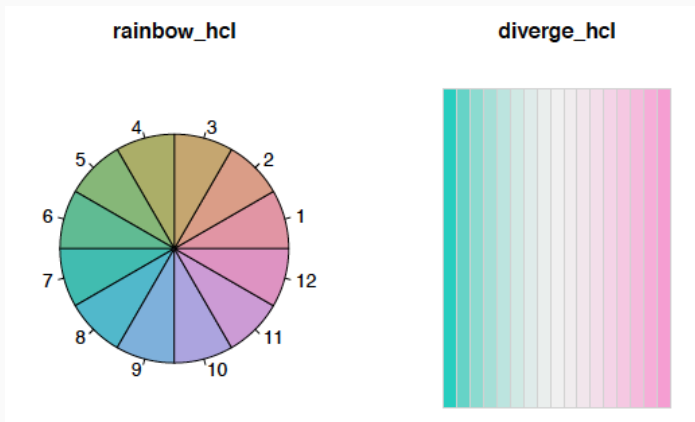


colorspace

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))
```

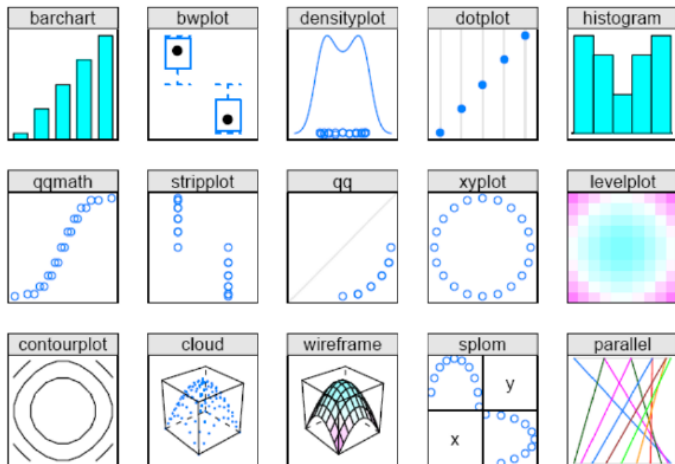


- trellis graphics is a family of techniques for viewing complicated data sets, that are based on basic concepts of human perception
- everything is possible (using a sufficient number of parameters)
- the trellis graphics system is in the `lattice` package
- the typical format is

```
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
```

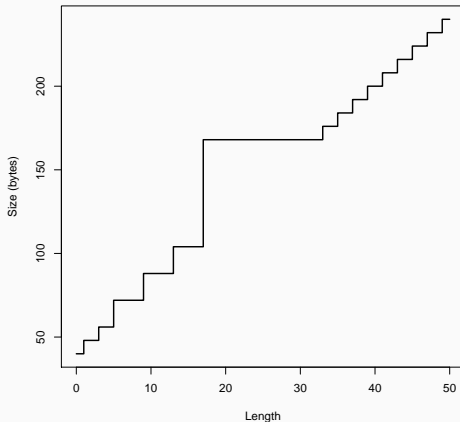
- some understanding of R's memory management will help you predict how much memory you will need
- `pryr::object_size()` tells you how much memory an object occupies (including environments)

```
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

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>

Performance

- useful to measure the runtime of code
- optimising code to make it run faster is an iterative process:
 - 1 find the biggest bottleneck (the slowest part of your code).
 - 2 try to eliminate it
 - 3 repeat until your code is “fast enough”
- parallelization saves time because you are using more of your computer's resources

- `system.time()` quick and dirty performance assessment
- `microbenchmark::` performance of a very small piece of code
- `Rprof()` detailed analysis
- `lineprof` and `shinr()`: interactive profiling with information about memory usage and graphical interface (see memory session)

system.time()

```
x <- rnorm(100000)
system.time({
  s <- 0
  for(i in 1:length(x)){s <- s + x[i]}
})
#   user   system elapsed
#  0.12    0.00    0.23
system.time({
  s <- 0
  for(v in x){s <- s + v }
})
#   user   system elapsed
#  0.089   0.000   0.076
```

`system.time()` returns the difference between the two `proc.time` calls. See `?proc.time()` for details.

Measurement of the performance of a very small piece of code

```
require(microbenchmark)
x <- runif(100)
microbenchmark(sqrt(x), x ^ 0.5 )
# Unit: nanoseconds
#      expr   min      lq    mean median      uq    max neval
# sqrt(x)  557   563.0   598.60    569   599.5   1559    100
#  x^0.5  2700  2737.0  2883.98   2762  2813.5  11999    100
```


- create a file with your function, e.g. Rprof.r:

```
foo <- function(reps = 20, n = 1e5){  
  for(r in seq_len(reps)) {  
    x <- rnorm(n)  
    o <- order(x)  
    x <- x[o]  
  }  
  invisible(NULL)  
}
```

- create a log file

```
Rprof("foo-prof.log", line.profiling = TRUE)  
foo()  
Rprof(NULL)
```

- check results with:

```
summaryRprof("foo-prof.log", lines = 'show')
```

lineprof::shine()

```
require(lineprof)
lineprof_example <- lineprof(foo())
shine(lineprof_example)
```

Why is R slow?

- R code is usually poorly written
- R language is informal and incomplete
- extreme dynamism, so everything can be modified after created
- lazy evaluation of function arguments: each additional argument to a function decreases its speed a little

Strategies to achieve better performance:

- 1 look for existing solutions
- 2 do less work
- 3 vectorize
- 4 parallelize
- 5 avoid copies, e.g. `c()`, `append()`, ...

Optimize code by doing less work

- use a function tailored to a more specific type of input or output, or a more specific problem, e.g. `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()` are faster than `apply()` type functions
- avoid functions which coerce their inputs into a specific type, if your input is not the right type
- functions will do less work if you give them more information about the problem, e.g.
 - `read.csv()`: specify known column types with `colClasses`
 - `cut()`: do not generate labels if you do not need them, e.g. use `labels = FALSE`

Vectorize

Vectorize means often to use *functionals*, which is a function that takes a function as an input and returns a vector as output

- most used: `lapply()`, `apply()`, and `sapply()`
- as an alternative to for loops
- reduce bugs in your code by better communicating intent
- many are written in C, and use special tricks to enhance performance

```
sqrt2 <- function(x){  
  out <- vector('numeric', length(x))  
  for(i in seq_along(x)){  
    out[i] <- sqrt(i)  
  }  
  return(out)  
}  
microbenchmark(sqrt2(1:10), sapply(1:10, sqrt))
```

Functionals

- base R functionals have grown over time and are not very consistent
- base R functionals only covers a partial set of possible combinations of input and output types:
- the `plyr` package provides consistently named functions and covers all combinations of input and output data structures:

	list	data frame	array
list	<code>lply()</code>	<code>ldply()</code>	<code>laply()</code>
data frame	<code>dlply()</code>	<code>ddply()</code>	<code>daply()</code>
array	<code>aply()</code>	<code>adply()</code>	<code>aaply()</code>

- parallelization uses multiple cores to work simultaneously on different parts of a problem
- does not reduce the computing time, but it saves your time because you are using more of your computer's resources
- Unix: simply substitute `parallel::mclapply()` for `lapply()`

```
x <- 1:1e4
system.time(lapply(x, rnorm))
#   user  system elapsed
# 4.625   0.083   4.709
system.time(mclapply(x, rnorm, mc.cores = 8))
#   user  system elapsed
# 1.072   0.557   1.623
```

- “Life is a bit harder in Windows” (Wickham): first set up a local cluster and then use `parallel::parLapply()`
- for platform-independent implementations check for instance source code of `boot::boot()`

- Hadley Wickham. *Advanced R*. The R series. CRC Press, 2015. Chapter 11: Functionals, Chapter 16: Performance, Chapter 17: Optimising Code, Chapter 18 Memory.
- Floreal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek: *Evaluating the Design of the R Language*. Available online: <http://r.cs.purdue.edu/pub/ecoop12.pdf>
- Norman Matloff (2011): *The Art of R Programming*.

Numerics & Simulations

Representation of numbers

- Computers use indicators to encode information (1 for ON, 0 for OFF)
- One indicator is called a bit, eight bits are one byte, ...
- The way how numbers are represented depends on your computer, R has no influence on that

Representation of integers

The *sign-and-magnitude scheme*

- Use one bit for the sign (\pm) and the rest for the magnitude
- For k bits an integer is represented as

$$\pm b_{k-2} \dots b_2 b_1 b_0,$$

where $b_i \in \{0, 1\}, i = 1, 2, \dots$. This number is translated to

$$\pm(2^0 b_0 + 2^1 b_1 + \dots + 2^{k-2} b_{k-2})$$

- Example: represent -77 using the former scheme for $k = 8$

$$-77 = -(2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 + 2^3 \cdot 1 + 2^4 \cdot 0 + 2^5 \cdot 0 + 2^6 \cdot 1)$$

Hence, the resulting representation is -1001101 .

Representation of integers

Properties and extensions

- Integers range symmetrically from $-(2^{k-1} + 1)$ to $2^{k-1} - 1$
- Two representations of 0 (\pm)
 - 1 The *biased scheme* (see references):
avoids -0 , but addition of integers is complex and slow
 - 2 The *two's complement scheme*:
uses the binary representation for positive integers and represents $-1, -2, \dots, -2^{k-1}$ via $2^k - 1, 2^k - 2, \dots, 2^k - 2^{k-1}$
- Integers have infinite cardinality. In R, there is a “largest” integer

```
> .Machine$integer.max  
[1] 2147483647
```

- The sets of real numbers represented by a computer $\mathbb{F} \subset \mathbb{R}$ is the union of the zero element and any number of the form

$$x = \pm(m/\beta^t)\beta^e,$$

where all constants are integers, $\beta \geq 2$ is the *base*, $t \geq 1$ is the *precision*, m lie in $1 \leq m \leq \beta^t$, and e is arbitrary. Usually $\pm(m/\beta^t)$ is called *mantissa* and e is called the *the exponent*.

Representation of real numbers

Properties:

Computers need to limit the size of the mantissa and exponent. In double precision

- use 8 bytes (i.e. 64 bits) in total
- 1 bit for the sign
- 52 bits for the mantissa
- 11 bits for the exponent (representation via *biased scheme*, ranges from -1022 to 1024)
- There is also a “largest” real

```
> .Machine$double.xmin
```

```
[1] 2.225074e-308
```

```
> .Machine$double.xmax
```

```
[1] 1.797693e+308
```

Examples:

- Underflow/ overflow

```
> 2^1023 + 2^1022 + 2^1021
```

```
[1] 1.572981e+308
```

```
> 2^1023 + 2^1022 + 2^1022
```

```
[1] Inf
```

- “Asymmetry”

```
> 2^(-1074) == 0
```

```
[1] FALSE
```

```
> 2^(-1075) == 0
```

```
[1] TRUE
```

```
> 1 / 2^(-1074)
```

```
[1] Inf
```


- Machine epsilon (smallest x , s.t. $1 + x$ can be distinguished from 1)

```
> .Machine$double.eps  
[1] 2.220446e-16
```

```
> x <- 1 + 2 ^ -52
```

```
> x - 1 == 0
```

```
[1] FALSE
```

```
> y <- 1 + 2 ^ -53
```

```
> y - 1 == 0
```

```
[1] TRUE
```

Numerical errors occur all the time. E.g. there is no finite binary representation of 0.1.

Denote by \tilde{x} an approximation of x .

- The absolute error is defined as $|x - \tilde{x}|$
- The relative error is defined as $\frac{|x - \tilde{x}|}{x}$

Catastrophic cancellation describes a loss of accuracy with a relative error of 10^{-8} or larger due to error propagation. It can occur when subtracting numbers of similar size.

Example: $\sin(x) - x, x \rightarrow 0$

- Standard computation

$$\lim_{x \rightarrow 0} \frac{\sin(x)}{x} = 1 \Rightarrow \sin(x) \approx x|_{x \rightarrow 0}$$

- Taylor expansion of order 2

$$\sin(x) - x \approx -\frac{x^3}{6} + \frac{x^5}{120} = -\frac{x^3}{6} \left(1 - \frac{x^2}{20}\right)$$

```
> x = 2^-c(10, 20, 30)
> sin(x) - x
[1] -1.552204e-10 -1.445250e-19 0.000000e+00
> -x^3 / 6 * (1 - x^2 / 20)
[1] -1.552204e-10 -1.445603e-19 -1.346323e-28
```

- Absolute errors: $\approx 10^{-21}, 10^{-23}, 10^{-28}$
- Relative errors: $\approx 10^{-11}, 10^{-4}, \text{Inf}$
- Catastrophic cancellation at $x = 2^{-30}$!

Generalities

- For many mathematical and statistical problems there are no analytical solutions (or they are very hard to find)
- Examples are optimization, integration, solving (systems of) equations, differentiation, eigenvalue problems, . . .
- For most cases, numerical alternatives have been developed and implemented
- Key features of such algorithms are accuracy/precision and convergence (rate)/speed

Some important functions for numerical mathematics in R:

optimization	derivation	(system of) equations	integration	other
optim(ize)	deriv	solve	integrate	eigen
optimx	grad	polyroot	adaptIntegrate	qr
nlm	hessian	solveLP		

See <http://cran.r-project.org/web/views/Optimization.html> for more.

Note that also other functions like `glm` or `lme` use numerical techniques to optimize the likelihood with respect to the regression parameters.

Examples

Find the minimum of the function $f(x) = x^2$.

```
> my.square = function(x) { x^2 }  
> optimize(f = my.square, interval = c(-2, 2)) $minimum  
[1] -5.551115e-17  
$objective  
[1] 3.081488e-33  
> optimize(f = my.square, interval = c(2, 3))  
$minimum  
[1] 2.000066  
$objective  
[1] 4.000264
```

What is the value of $\int_{-2}^2 x^2 dx$?

```
> integrate(f = my.square, lower = -2, upper = 2)  
5.333333 with absolute error < 5.9e-14
```

Definition 6.1

A (Monte-Carlo)-Simulation is a numerical technique for conducting experiments on a computer. The term Monte-Carlo refers to the involvement of random experiments.

Application areas:

Simulation studies are performed when analytical results are hard or impossible to find to

- identify properties of estimators or test statistics (bias, variance, distribution, etc.)
- investigate consequences of violations of model assumptions
- find out about the influence of the sample size
- compare different models or estimators (in terms of bias, precision, computational time, etc.)

Rationale

- Estimators and test statistics have true sampling distributions (under certain assumptions)
- Knowing the distribution would answer all questions about the properties described above
- Approximate these distributions by conducting according random experiments very often (law of large numbers)

Usual setup

- Simulate K independent data sets under the conditions of interest
- Calculate the numerical values of the statistic T of interest for each data set, i.e. T_1, \dots, T_K
- Evaluate the properties of the results under the assumption that the distribution of T_1, \dots, T_K approximates the true distribution of the statistic

Stochastic distributions in R

In R, density (***d***), distribution (***p***), quantile (***q***), and (pseudo) random number generator (***r***) functions are already implemented.

function	distribution
beta()	beta
binom()	binomial
exp()	exponential
gamma()	gamma
hyper()	hypergeometric
logis()	logistic
lnorm()	lognormal
nbinom()	negativ binomial
norm()	normal
pois()	poisson
t()	t
unif()	uniform

Further notes

- The random numbers in R are not really random
- Use `set.seed` to make your results replicable

```
> set.seed(123)
> rnorm(3)
[1] -0.5604756 -0.2301775  1.5587083
> rnorm(3)
[1] 0.07050839 0.12928774 1.71506499
> set.seed(123)
> rnorm(3)
[1] -0.5604756 -0.2301775  1.5587083
```

Case Study

Investigate the behavior of the estimator for the linear regression model without intercept

$$y_i = \beta x_i + \epsilon_i, \quad i = 1, \dots, n, \quad (1)$$

when the distributional assumption $\epsilon_i \stackrel{iid}{\sim} \mathcal{N}(0, \sigma^2)$ is violated.

More precisely, simulate $K = 50$ independent data sets of sample size $n = 200$ with

- $x_i \sim \mathcal{N}(3, 2)$
- $\epsilon_i = u_i - v_i$, where $u_i \sim \mathcal{N}(0, 1^2)$ and $v_i \sim \mathcal{N}_+(0, 1^2)$
- $\beta = 2$
- y_i according to (1).

and estimate the covariate effect for each of the data sets. What are the approximate mean and variance of $\hat{\beta}$?

- Jones, Maillardet and Robinson (2009): **Scientific Programming and Simulation Using R**
- `http://cran.r-project.org/web/views/NumericalMathematics.html`
- `http://cran.r-project.org/web/views/Optimization.html`
- `http://cran.r-project.org/web/views/Distributions.html`

Object Oriented Programming

- in OOP, computer programs are made out of objects that interact with one another
- a class defines objects via attributes and their relationship to other classes.
- the class is also used when selecting methods, i.e. functions that behave differently depending on the class of their input.
- R has three OO systems: S3, S4, and the system of base types

- fundamental building blocks of a class system
 - definition of classes, generation of class instances
 - specification of inheritance
 - determination of a class and all parental classes of an object
 - access and modification of the class slots
 - class and method management
- generic functions are functions, that run different computations in dependence of the class of their arguments and even return different outputs

Base Types

- base types are the internal C-level types that underlie the other OO systems
- underlying every R object is a C structure that describes how that object is stored in memory
- you can determine an object's base type with `typeof()`
- the most common base types are atomic vectors and lists, but also encompass functions, environments, other more exotic objects
- functions that behave differently for different base types are almost always written in C

S3 Object System

- S3 is R's first and simplest OO system
- S3 is informal, ad-hoc, and minimal
- test if an object is an S3 object by using `pryr::otype()`
- test for a specific class by `class()`

```
df <- data.frame(x = 1:10, y = letters[1:10])  
otype(df)      # [1] "S3"  
otype(df$x)    # [1] "base"  
otype(df$y)    # [1] "S3"
```

- S3 objects at first glance

```
mean
# function (x, ...)
# UseMethod("mean")
# <bytecode: 0x2bc14a0>
# <environment: namespace:base>
```

- `pryr::ftype()` describes the object system. Default method is generic

```
ftype(mean) # [1] "s3"      "generic"
```

- you can recognise S3 methods by their names, e.g. the `Date` method for the `mean()` generic is called `mean.Date()`

- `methods()` returns all methods that belong to a generic or lists all generics that have a method for a given class

```
methods("mean")
methods(class = "ts")
```

Define S3 Classes and Create Objects

- can change the class of a base object.
- class names are usually lower case, and you should avoid “.”
- S3 has no checks for correctness

```
x <- 1:5
print(x)           # [1] 1 2 3 4 5
# construct object instance of class foo
class(x) <- "foo"
print(x)           # [1] 1 2 3 4 5
                   # attr("class")
                   # [1] "foo"
```

Create new Methods and Generics

- to add a new generic, create a function that calls UseMethod()
- to add a method, you just create a regular function with the correct generic.class name (adding a method to an existing generic works in the same way)

```
# add new generic and method for object foo
f <- function(x) UseMethod("f")
f.foo <- function(x) "Class foo"
f(x)                # [1] "Class foo"
# define existing print() generic for object foo
print.foo <- function(x, ...){
  cat("foo object of length", length(x), ":\n")
  print(unclass(x))
}
print(x)             # foo object of length 5 :
                    # [1] 1 2 3 4 5
```

S3 Problems

- S3 has no formal definition and therefore no checks for correctness, when construction objects, i.e. every object can get every class
- while you can change the type of an object, but you never should

```
mod <- lm(log(mpg) ~ log(displ), data = mtcars)
class(mod)
print(mod)
class(mod) <- "data.frame"
print(mod)
```

Not very helpful!

- the name convention for finding object-specific methods use the dot, which is not a special symbol

Does `t.test.default()` transpose an object from class `test.default`, or is it the default method for `t.test()`?

S4 Object System

- S4 works in a similar way to S3, but it adds formality and rigour
- classes have formal definitions which describe their fields and inheritance structures
- identify an S4 object by using `isS4()`, `pryr::otype()` or `pryr::ftype()`

```
require(stats4)
# From example(mle)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) - sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))
# An S4 object
isS4(fit)    # [1] TRUE
otype(fit)   # [1] "S4"
# An S4 generic
isS4(nobs)   # [1] TRUE
ftype(nobs)  # [1] "s4"      "generic"
```

Define S4 Classes and Create Objects

- define the representation of a class with `setClass()` with properties:

<code>name</code>	S4 class names use UpperCamelCase
<code>slots</code>	defines field names and permitted classes
<code>contains</code>	string giving the class it inherits from
<code>validity</code>	method that tests if an object is valid
<code>prototype</code>	object that defines default slot values

- create a new object with `new()` or a constructor function with the same name as the class (usually available)

```
setClass("citizen",  
        slots = list(name = "character", age = "numeric",  
                      job = "character"),  
        prototype = list(name = "Anna", age=40,  
                          job="rocket scientist")  
)  
albert <- new("citizen", name = "Albert", age = 42,  
             job = "domestic servant")
```

Create new Methods and Generics

- `setGeneric()` creates a new generic or converts an existing function into a generic
- `setMethod()` takes the name of the generic, the classes the method should be associated with, and a function that implements the method
- `standardGeneric()` is the S4 equivalent to `UseMethod()` to create a new generic from scratch

```
setGeneric("union")      # [1] "union"
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
showMethods("union")     # x="ANY", y="ANY"
                        # x="data.frame", y="data.frame"
```


Picking a System

- majority of OO code that I have written in R is S3
- S3 is sufficient for fairly simple objects and methods for pre-existing generic functions like `print()`, `summary()`, and `plot()`
- S4 may be more appropriate for more complicated systems of interrelated objects
- good example for S4 is the `Matrix` package by Douglas Bates and Martin Maechler

S3/S4 Object System Comparison

	S3	S4
defintion	not neccessary	<code>setClass('class name',...)</code>
generation of instances	<code>class(object)<- 'class_name'</code>	<code>new('class_name')</code>
inheritance	vector of class_names (children before parents)	<code>contains='parental cl'</code> in definition
test class	<code>inherits(object,'class_name')</code>	<code>is(object,'class_name')</code>
access slots	depends on base type: for lists <code>\$</code> or <code>[[]</code> .	new operator: <code>@</code>
list methods	<code>methods()</code>	<code>showMethods()</code>

For S3 and S4, there are the following conventions

- constructor functions should be named like the class itself, e.g. `lm()`, with exception if a class is the return value of a number of functions
- standard methods, which are available supplied for many classes:

<code>print</code>	basic object information, also when using <code><RET></code> (S4 <code>show()</code>)
<code>summary</code>	more detailed description of the objects instance
<code>plot</code>	graphics

- every method should have the arguments of the corresponding generic (same order and defaults) and accept an arbitrary number of additional arguments (use `...`)

- Hadley Wickham. **Advanced R**. The R series. CRC Press, 2015. Chapter 7: OO field guide. Available online:
`http://adv-r.had.co.nz/00-essentials.html`
- Laurent Gatto. **A practical tutorial on S4 programming**. Available online:
`http://www.bioconductor.org/help/course-materials/2013/CSAMA2013/friday/afternoon/S4-tutorial.pdf`

R Interaction with Other Languages

Several programming languages can be included in R:

- C (nlm,optimize,...)
- FORTRAN (subroutines of fields package)
- C++ (Amelia, RcppDE)
- ⋮

C++ can be integrated with the add-on packages Rcpp. Why do we want to integrate C++?

- R is very slow for loops
- Parallelization is nice, but
 - does not reduce the total computational time
 - is not possible for iterative calculations

Determination of the reasons for chronic undernutrition in Peru.

Estimation of the model

$$\begin{aligned}\text{stunting} = & \text{sex} + \text{mEdu} + \text{PartnerEdu} + \text{householdmembers} \\ & + f(\text{breastfeeding}) + f(\text{cage}) + f(\text{mage}) + f(\text{mbmi}) \\ & + f(\text{mheight}) + f(\text{wealthscore}) + f_{\text{spat}}(\text{region})\end{aligned}$$

for $\tau = 5\%$ takes

- 226s with pure R code
- 24s with integration of C++ parts

Why R?

- Easy handling of input and output
- No compilation needed
- Good error detection
- Many statistical methods already implemented
- Easy/quick programming

Rcpp is a easy, nice and fast way to call C++ from R:

- Hadley Wickham. **Advanced R**. The R series. CRC Press, Boca Raton, 2015. Chapter 19: Rcpp
- Dirk Eddebuettel. **Seamless R and C++ Integration with Rcpp**. Springer, New York, 2013
- <http://www.rcpp.org>
- <http://cran.r-project.org/web/packages/Rcpp/index.html>
- <http://lists.r-forge.r-project.org/mailman/listinfo/rcpp-devel>

To learn C++:

- <http://www.learncpp.com/>
- <http://www.cplusplus.com/>

To run Rcpp a C++ compiler is required:

- Windows: Rtools
`http://cran.r-project.org/bin/windows/Rtools/`
- Mac: Xcode app store
- Linux: `sudo apt-get install r-base-dev`

Take care when installing Rtools:

- Path of R must not contain any spaces!
(C:\R\R-3.x.y)
- Path of Rtools must not contain any spaces!
(C:\R\Rtools\gcc-4.6.3\bin)
- Some useful tricks: `http://lists.r-forge.r-project.org/pipermail/rcpp-devel/2012-July/003979.html`

General Structure

General Structure:

```
cppFunction(  
  'variable-type-output function-name(variable-type-input variable-name)  
  ... ;  
  ... ;  
  return output;  
)'
```

Example:

```
library(Rcpp)  
cppFunction('double meanC(double x, double y, double z) {  
  double m = (x + y + z) / 3;  
  return m;  
)'  
meanC(1, 2, 8)
```

- Specify variable type, e.g. double, int,...
- End each statement with ;
- Use = for assignment not <-
- Return value with return ...;
- Comment with

```
// ... for one line
/* ...
... for multiple lines
... */
```

C++ Variable Types

R	Rcpp	Values
integer	int	\mathbb{Z}
numeric	double	\mathbb{R}
logical	bool	TRUE, FALSE, NA (R) true, false (C++)
character	String	e.g. "a", "text", "Word"
numeric vector	NumericVector	e.g. $(3.1, 3.2, 3.3)^T$
numeric matrix	NumericMatrix	e.g. $\begin{bmatrix} 3.1 & 3.2 & 3.3 \\ 7.1 & 8.2 & 9.3 \end{bmatrix}$

NumericVector and for-loop

```
cppFunction('double sumC(NumericVector x) {  
  int n = x.size();  
  double total = 0;  
  for(int i = 0; i < n; i++) {  
    total += x[i];  
  }  
  return total;  
'})
```

- Length of a vector with `.size()`
- Dimensions of a matrix with `.ncol()` and `.nrow()`
- Element of a vector/matrix with `()` or `[]` (only vector)
- First element of a vector has number 0 and Last element of a vector has number $n - 1$
- `i++` is a postfix operator for `i=i+1`
- `total += x[i]` \Leftrightarrow `total = total + x[i]`, similar are `-=`, `*=` and `/=`

Several methods for elementwise operations of vectors are implemented in RcppSugar:

- Basic operations:

`+`, `*`, `-`, `/`, `pow()`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`

- Standard functions:

`abs()`, `ceiling()`, `exp()`, `log()`, `log10()`, `signif()`,
`sin()`, `sqrt()`, . . .

- Basic statistics:

`mean()`, `min()`, `max()`, `sum()`, `sd()`, `var()`, `cumsum()`,
`diff()`, `pmin()`, `pmax()`, . . .

In RcppSugar these methods are only available for vectors and not for matrices!!!

Elementwise calculation like in R possible with RcppSugar, e.g.

$$\text{l2norm}(\mathbf{x}, \mathbf{y}) = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - y_i)^2}$$

```
cppFunction('List l2norm1(NumericVector x, NumericVector y) {  
  NumericVector dist = x - y;  
  NumericVector dist2 = pow(dist, 2);  
  double meandist = mean(dist2);  
  double edist = sqrt(meandist);  
  return List::create(Named("dist") = dist,  
                      Named("dist2") = dist2,  
                      Named("meandist") = meandist,  
                      Named("edist") = edist);  
}')  
  
cppFunction('double l2norm2(NumericVector x, NumericVector y) {  
  return sqrt(mean(pow((x - y), 2)));  
}')
```

- With inline additional features like RcppEigen can easily be included
- Eigen is a powerful C++ library for matrix operations and advanced routines

<http://eigen.tuxfamily.org/dox/index.html>

- Alternatively RcppArmadillo can be used
- <http://arma.sourceforge.net/>

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

```
library(Rcpp)
library(inline)
LMcpp <- '
using Eigen::MatrixXd;
using Eigen::VectorXd;
VectorXd y(as<VectorXd>(yy));
MatrixXd X(as<MatrixXd>(XX));
VectorXd b = ((X.adjoint() * X).inverse()) * X.adjoint() * y;
return List::create(Named("y")=y,Named("X")=X,Named("b")=b);
',

simpleLM <- cxxfunction(signature(yy = "numeric", XX = "matrix"),
                        LMcpp, plugin = "RcppEigen")
```

$$R^2 = \left(1 - \sum_{i=1}^n \hat{\epsilon}_i^2\right) / \left(\sum_{i=1}^n (y_i - \bar{y})^2\right)$$

For elementwise operations inside Eigen objects use array format

```
LMcpp <- 'using Eigen::MatrixXd;
         using Eigen::VectorXd;
         VectorXd y(as<VectorXd>(yy));
         MatrixXd X(as<MatrixXd>(XX));
         VectorXd b = ((X.adjoint() * X).inverse()) * X.adjoint() * y;
         VectorXd resid = y - X * b;
         VectorXd resid2 = pow(resid.array(),2);
         VectorXd diffY = y.array() - y.mean();
         VectorXd diffY2 = pow(diffY.array(),2);
         double R2 = 1 - (resid2.sum()) / (diffY2.sum());
         return List::create(Named("y")=y, Named("X")=X,
                             Named("b")=b, Named("R2")=R2);'
```

Manipulating the C++ object also changes the underlying R object, since the C++ object is a pointer to the R object.

```
manip_bad <- '  
  NumericVector invec(vx);  
  NumericVector outvec(vx);  
  for(int i = 0; i < invec.size(); i++) {  
    outvec[i] = log(invec[i]);  
  }  
  return outvec;  
,  
  
fun_bad <- cxxfunction(signature(vx="numeric"), manip_bad, "Rcpp")  
x <- seq(1.0, 3.0, by=1)  
fun_bad(x)      ## 0.0000000 0.6931472 1.0986123  
x               ## 0.0000000 0.6931472 1.0986123
```

The function `clone()` produces a hard copy of the input, thus the original input is not manipulated anymore.

```
manip_good <- '  
  NumericVector invec(vx);  
  NumericVector outvec = clone(vx);  
  for(int i = 0; i < invec.size(); i++) {  
    outvec[i] = log(invec[i]);  
  }  
  return outvec;  
,  
  
fun_good <- cxxfunction(signature(vx="numeric"), manip_good, "Rcpp")  
x <- seq(1.0, 3.0, by=1)  
fun_good(x)    ## 0.0000000 0.6931472 1.0986123  
x              ## 1 2 3
```

- Dirk Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Springer, New York, 2013
- Hadley Wickham. *Advanced R*. The R series. CRC Press, Boca Raton, 2015. Chapter 19: Rcpp
- <http://lists.r-forge.r-project.org/mailman/listinfo/rcpp-devel>
- <http://eigen.tuxfamily.org/>
- <http://arma.sourceforge.net/>

Building R Packages

Why Building R Packages?

- platform-independent distribution of R code
 - alpha/beta versions on R-forge or github
 - finished projects on CRAN or Bioconductor
- archiving R code for a specific project and software documentation
- reproducible research: distribute data and software accompanying a paper maintenance of dependencies, and automated loading of required external code
- CRAN uses R CMD check to test package on various platforms; packages are tested daily

Basic Structure

A basic (but good) R package has the following structure:

DESCRIPTION	what does the package? who can use it (license)? who is responsible (maintainer)?
NAMESPACE	which function should be seen by the user? which are internal?
R/	R functions
man/	documentation, help files with syntax similar to \LaTeX
data/	example data files

The standard structure can be obtained automatically using

```
utils::package.skeleton or tools in packages  
devtools::create or roxygen2.
```

Naming R Packages:

- can contain letters and numbers, but start with a letter
- avoid self-invented abbreviations, capital letters, ...
- should be identifiable in online search

R/ directory contains all R code:

- each function in a separate file (good for small packages)
- everything in one file (ok for small packages)
- group related functions in a file with meaningful names (best solution for larger projects)

Additional (optional) files in R packages:

<code>NEWS</code>	describes changings over different package versions
<code>demo/</code>	larger demos
<code>src/</code>	C, C++, FORTRAN source code
<code>inst/CITATION</code>	how should the user cite the package?
<code>inst/doc/</code>	vignette
<code>test/, inst/tests</code>	tests
<code>:</code>	
<code>:</code>	

Package:	name of the package
Title:	description of the package (one line, j 65 characters)
Description:	detailed description (one paragraph, multiple sentences)
Version:	version number formatwise major.minor-patchlevel or major.minor.patchlevel
Maintainer:	name and e-mail of a person who wants to take over the responsibility
License:	abbreviation of a software licence (GPL-2, BSD, MIT, ...)
Depends, Suggests, Imports, Enhances:	package dependencies URL: for website of a package
Collate:	order R files are loaded (default: alphabetically)

Package: mypackage

Title: What The Package Does (one line, title case required)

Version: 0.1

Authors@R: person("First", "Last",
 email="first.last@example.com",
 role=c("aut", "cre", "ctb"))

Description: What the package does (one paragraph)

Depends: R (>= 3.1.0)

License: What license is it under?

LazyData: true

- although the file looks like R code, it is not processed as R code
- specifies which variables in the package should be exported to make them available to package users, and which variables should be imported from other packages

```
import(foo, bar)      # all functions from foo and bar import
importFrom(foo, f, g) # selected functions f and g from foo
export(f, g)          # export functions f and g
```

- for packages with many variables to export it may be more convenient to specify the names to export with a regular expression

```
exportPattern("^{}[\\{}\\textbackslash\\textbackslash.]")
```

- assign the DLL reference for the C functions `myRoutine`, `myOtherRoutine` from DLL `foo`:

```
useDynLib(foo, myRoutine, myOtherRoutine)
```

- ensure that the generics are imported and register the methods using S3method directives
- the function print.foo does not need to be exported

```
# example samplingbook
export(Smean)
S3method(print, Smean)
export(htestestimate)
S3method(print, htestestimate)
.
.
# example ggplot2
S3methodautoplot,default)
exportautoplot)
importplyr)
importFrom(MASS,cov.trob)
.
.
```


- some additional steps are needed for packages which make use of S4 classes and methods
- package should depend on package methods (also DESCRIPTION file)
- you may need to import `graphics::plot` to make visible a function that can be converted into a implicit generic

```
import("methods")           # S4
import("ff")                 # namespaces from dependencies
import("igraph")
importFrom(graphics, "plot")  # S4 plot
exportPattern("^[:alpha:]]+") # regular pattern
exportMethods("GeneSNPsize", "plot", "show", "summary" )
exportClasses("GWASdata", "kernel", "pathway", "lkmt")
```

- R documentation format is very \LaTeX -like output (\LaTeX installation required)

```
\name{add}  
\alias{add}  
\title{Add together two numbers}  
\usage{ add(x, y) }  
\arguments{  
  \item{x}{A number}  
  \item{y}{A number}  
}  
\value{The sum of \code{x} and \code{y}}  
\description{ Add together two numbers }  
\examples{  
add(1, 1)  
add(10, 1)  
}
```

Building R Packages

For building a R package pkg run the following commands in your console:

R CMD SHLIB pkg	compiles C/C++/Fortran code in pkg/src
R CMD build pkg	generates package bundle pkg.tar.gz or pkg.zip
R CMD INSTALL pkg.tar.gz	installs package
R CMD check pkg.tar.gz	runs CRAN validity checks (is pkg valid?)

In windows, installation of Rtools is required:

On windows:

R CMD INSTALL --build pkg

R functions from devtools that simplifies R packaging:

<code>load_all()</code>	simulates installing and reloading your package
<code>document()</code>	updates documentation, file collation and NAMESPACE
<code>test()</code>	reloads your code, then runs all testthat tests.
<code>run</code>	<code>examples()</code> will run all examples to make sure they work.
<code>check_doc()</code>	runs most of the documentation checking components of R CMD check
<code>check()</code>	updates the documentation, then builds and checks the package
<code>build()</code> , <code>buildwin()</code>	builds a package file from package sources (only one R version)

The package `testthat` provides functions that make it easy to describe what you expect a function to do:

<code>expect_that</code>	describes expected result of your code (value, class, correct error message, computation time, etc.)
<code>test_that</code>	is grouping a number of expectation?s for one functions or a feature
<code>context()</code>	is grouping a number of content-related tests

```
require(testthat)
test_that("trigonometric functions match identities", {
  expect_that(sin(pi / 4), equals(1 / sqrt(2)))
  expect_that(cos(pi / 4), equals(1 / sqrt(2)))
  expect_that(tan(pi / 4), equals(1))
})
```

The package `testthat` easily integrates in your existing workflow, whether it's informal testing on the command line, building test suites or using R CMD check

- within your workflow:

```
testthat::test_file() # test in a file
testthat::test_dir()  # all ?test-? files in a directory
```

- within a package:

```
library(testthat)
library(yourpackage)
test_check("yourpackage")
test_package("yourpackage")
```

- using R CMD check:

- DESCRIPTION needs Suggests testthat
- test files in tests/testthat/ are automatically tested

There are three steps in the transformation from roxygen comments in your source file to human readable documentation:

- 1 add roxygen comments to your source file
- 2 `roxygen2::roxygenise()` or `devtools::document()` converts roxygen comments to `.Rd` files
- 3 R CMD check converts `.Rd` files to human readable documentation


```
add {rvest}
```

R Documentation

Add together two numbers

Description

Add together two numbers

Usage

```
add(x, y)
```

Arguments

x A number

y A number

Value

The sum of x and y

Examples

```
add(1, 1)  
add(10, 1)
```

Further Reading

- **Writing R Extensions manual:** <http://cran.r-project.org/doc/manuals/r-release/R-exts.html>
- Hadley Wickham (2015). **R packages**. O'Reilly Media. Available online: <http://r-pkgs.had.co.nz/>
- Friedrich Leisch. **Creating R Packages: A Tutorial:** <http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf>
- testthat: Hadley Wickham (2011). **testthat: Get Started with Testing**. The R Journal 3(1).
- roxygen2: <http://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html>
- Dirk Eddebuettel & Romain Francois: **Writing a package that uses Rcpp:** <http://dirk.eddebuettel.com/code/rcpp/Rcpp-package.pdf>