Programming in R and Shiny

Martin Losert and Katharina Naumann

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

TquanT+ Seminar, Hungary, March 2019

Slides and exercises:
http://www.mathpsy.uni-tuebingen.de/RShiny.zip

```
library(fortunes)
fortune(52)
```

```
Can one be a good data analyst without being a half-good
programmer? The short answer to that is, 'No.' The long
answer to that is, 'No.'

   -- Frank Harrell
      1999 S-PLUS User Conference, New Orleans (October 1999)
```

"In the 20th century if you wanted to do statistics you had to be a bit of a mathematician, whether you wanted to or not; in the 21st century if you want to do statistics you have to be a bit of a programmer, whether you want to or not."

http://andrewgelman.com/2017/02/12/aaron-kaufman-reviews-luke-heatons-brief-history-mathematical-thought/

Contents

1 Know your editor

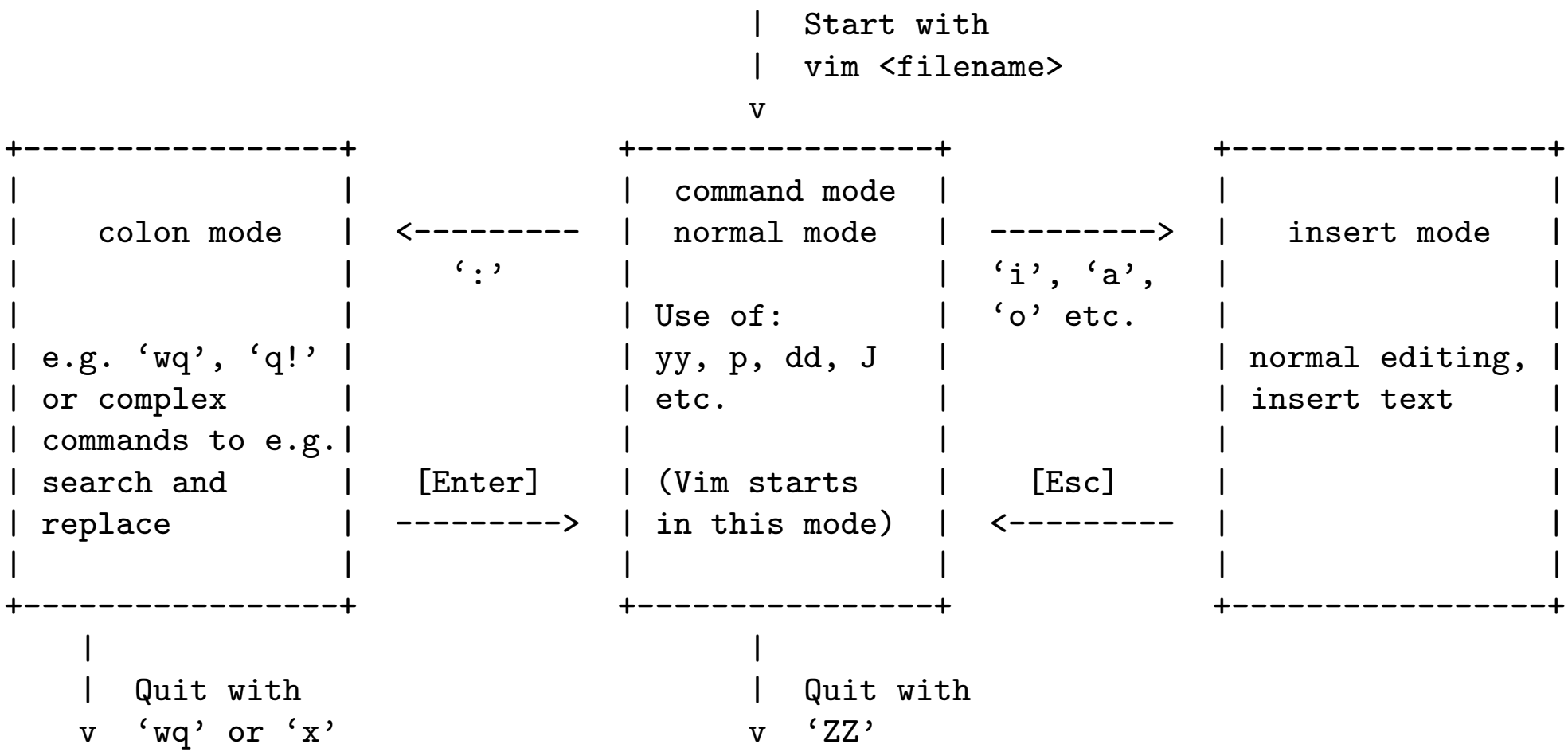
Vim editor

- Vi IMproved
- Based on UNIX text editor `vi`
- Open-source, free (charityware)
- "Vim is a highly configurable text editor built to enable efficient text editing." (`https://www.vim.org/about.php`)
- Cross-language text editor (R, HTML, LaTeX, C, Python, . . .)

Vim editor

- Modal editing
  1. Command mode or Normal mode
     Vim starts in command mode. Use of easy commands to do things like "delete line," "search word" etc. Pressing `Esc` switches back to command mode.
  2. Insert mode
     In insert mode you can insert text. Commands like 'i', 'I', 'a', 'A', 'o', 'O' switch to insert mode.
  3. Colon mode or ex mode
     Typing ':' (colon) switches to ex mode. Use of more complex commands like searching and replacing a certain word.
  4. Visual mode
     A variant of Command mode. Typing 'v', 'V', and CTRL-V (CTRL-Q on Windows) switches to characterwise, linewise, and blockwise visual mode, respectively. The text is highlighted according to the movement commands. Operator commands can then be applied on the highlighted text.

Vim: overview of the traditional modes



(Source: http://de.wikipedia.org/wiki/Vi)

Vim editor

Getting help

- Built-in documentation and help: `:help`
- E.g.: `:help vim-modes`
- Accessing the documentation for a command, topic, or option:

```
:h <command>
:h <topic>
:h <'option'> # single quotes are needed if there is
              # a command or topic with the same name
```

- Wiki: `http://vim.wikia.com/`

Vim editor

Setting options

- Settings can be:
  - set for the current session via `:set ...` (see `:help set`)
  - defined in a `vimrc` file to be read at the start of any VIM session via `set ...` (see `:help vimrc`)
- Example for an option that takes a value

```
set colorcolumn?       # Find current value
set colorcolumn=<Tab>  # Auto-complete current value
set colorcolumn=x      # Set value to x
set colorcolumn&       # Reset to default value
```

- Example for an option that is a binary switch (on/off)

```
set number?            # Find current value
set number             # Turn on
set nonumber           # Turn off
set number!            # Invert value
```

Vim editor

Common settings include:

```
set number              # line numbers on
set colorcolumn=81      # visual marker at 81st column
set autochdir           # working directory changes
                        # according to current file
```

Changing the font in VIM versions with a GUI:

```
:set guifont=*
```

Vim editor

Basic commands for opening, saving and closing files:

```
:edit [file]     # edit file in current tab
:tabedit [file] # edit file in new tab (same as :tabnew)

:w               # write file
:sav [filename] # save current file as [filename]

:q               # quit current window in current tab
                 # quit current tab if tab has only one window
                 # close VIM if there is only one tab
:q!              # quit without saving
```

Moving between windows and tabs:

```
gt                      # switch to tab to the right
gT                      # switch to tab to the left
[count]gt               # switch to tab [count]
CTRL-W [movement]       # switch between windows
```

2 Writing R Scripts

- Computing in R
- Getting help
- Session management and workflow

What is R?

- R is a free software environment for statistical computing and graphics.
- R and S-Plus are two implementations of the S language (Becker & Chambers, 1984).
- R was originally created by Ross Ihaka and Robert Gentleman in 1996 and is now being developed by an international team of scientists (R Core Team).
- Base R is a stable and mature system:
  - Yearly new R version with relatively minor changes
  - But: exponentially growing list of contributed extension packages
- R runs on Unix, Macintosh, and Windows systems.

R as a calculator

Simple mathematical operations may be entered directly at the command line.

```
2 + 2   # plus
3 - 7   # minus
2 * 2   # times
5 / 4   # divide
log(3)  # natural logarithm
exp(1)  # exponential function
```

Brackets can be used in longer expressions.

```
sqrt(2) * ( (5 - 1/6)^2 - pi/2^(1/3) )  # pi = 3.14
```

(R ignores any input after the # symbol.)

Assigning values

Results of operations and function calls may be stored in a variable. The assignment operator is <- (less than + minus).

```
x <- sqrt(2)
y <- x^2
```

In order to print the value of a variable, the variable name needs to be entered.

```
y
[1] 2
```

Valid variable names may consist of letters, numbers, underscores, and dots (e. g., `errors.item6`), but they may only start with letters or dots.
R is case sensitive (both for functions and variable names).

Vectorized arithmetic

Vectors are created using the function `c()` (combine).

```
weight <- c(60, 72, 57, 90, 95, 72)
weight
[1] 60 72 57 90 95 72
```

Mathematical operations may be applied to each element.

```
(weight - mean(weight))^2  # squared deviation
[1] 205.444444   5.444444 300.444444 245.444444
[5] 427.111111   5.444444
```

Many R functions accept vectors as arguments.

```
log(weight)
[1] 4.094345 4.276666 4.043051 4.499810 4.553877 4.276666
```

Packages

R comes with a standard library of packages

- Default (or 'base') packages: These are only updated with new releases of R
- Recommended packages: Available and updated at CRAN

A large part of the functionality of R is available in additional add-on packages.

```
library()                # which packages are installed?
library(nlme)            # load package nlme
search()                 # which packages are loaded?
detach(package:nlme)     # 'unload' package nlme
.libPaths()              # display or set search path
                         #   for packages
```

In order to install additional packages, the computer must be connected to the Internet. Packages are installed via

```
install.packages()
```

The help system in R

R offers a number of offline sources for getting help. The text-based documentation is called by `?`. For example, the documentation for the `rnorm()` function is accessed by

```
?rnorm  # or help(rnorm)
```

Words with syntactic meaning and special characters need to be enclosed in quotes

```
?"if"
```

At the bottom of the help page, there usually are examples that can be copied and pasted into the R console. Typing

```
apropos("norm")
```

displays functions in the currently *loaded* packages having the expression "norm" in their name.
The help system of all *installed* packages is searched for the expression "norm" via

```
help.search("norm")  # or ??"norm"
```

Documentation

R ships with several manuals. An HTML based (local) help system is available via

```
help.start()
```

The manuals may also be downloaded in pdf format from `https://cran.r-project.org/manuals.html`. Highly readable are

- An Introduction to R (with many examples)
- R Data Import/Export

Further sources of information

- Frequently Asked Questions
- FAQ for Windows

These are available via `help.start()`, too.

Resources on the Internet

A wealth of information may be found on the R website `http://www.r-project.org`.

Under R Project/Search

- R site search (within all default and CRAN packages), also via `RSiteSearch("topic")` from inside R
- Searchable mail archives (especially R help)

Under Documentation

- Manuals, FAQ
- The R Journal
- Books

At the Stack Exchange network, there are popular Q&A websites for R (`https://stackoverflow.com/tags/r/info`) and for applied statistics with R (`https://stats.stackexchange.com/tags/r/info`)

Books about R

Introductory books

- Dalgaard, P. (2008). *Introductory statistics with R.* New York: Springer.
- Verzani, J. (2005). *Using R for introductory statistics.* Boca Raton, FL: Chapman & Hall/CRC.

More advanced reading

- Venables, W. N. & Ripley, B. D. (2002). *Modern applied statistics with S.* New York: Springer.
- Wickham, H. (2014). *Advanced R.* Boca Raton, FL: Chapman & Hall/CRC. (`http://adv-r.had.co.nz/`)

Graphics

- Murrell, P. (2011). *R graphics.* Boca Raton, FL: Chapman & Hall/CRC.

Check the R website for more.

Working memory

All objects that are created during a session are kept in the working memory. Objects that are not needed any more should be deleted.

```
ls()            # list objects in working memory
rm(A)           # remove object A
rm(list=ls())   # remove all objects
```

When quitting a session (`q()`) the working memory may be written to a file, so the objects are available upon the next start of R.

Note that this is usually unnecessary. Rather write the R code that *created* the objects into a file (using a text editor). In this way, you will keep your entire analysis documented.[1]

[1] An even better way of documenting is to integrate the analysis into your report: R code is written directly into a LaTeX file; this file is processed using the `Sweave()` function and then compiled into a pdf report. This facilitates reproducible research: When the data change, the report (and the tables and figures) will change, too.

Working directory

When R is started, its working directory is set per default.

```
getwd()     # get working directory
[1] "D:/"
dir()       # list files in working directory
```

It is recommended to change the working directory to the current project directory. This keeps file paths short, and prevents you from accidentally overwriting data.

```
setwd("D:/projects/experiment3/analysis")  # set working
                                           #   directory
```

Executing R code

In order to execute the entire R code in a text file from inside R, type

```
source("filename.R")  # source file located in
                      #   R's working directory
```

When a set of functions is documented and (more or less) self-contained, one should consider writing an R package and submitting it to the Comprehensive R Archive Network (CRAN).

More information on programming is in the Writing R Extensions manual that ships with R.

R scripts: a note on style

Hints on "good" R style are given in the following documents:

- R Style Guide by Hadley Wickham
  `http://adv-r.had.co.nz/Style.html`
- Google's R style guide
  `https://google.github.io/styleguide/Rguide.xml`
- R coding conventions
  `http://www.aroma-project.org/developers/RCC`

These documents are opinionated, but readable. There is no official style guide by the R Core Team, although there are strong opinions about what constitutes good R style.

Consistency is usually considered an important criterion for good style.

3 Data structures in R (1)

Functions, vectors, arrays, factors

Functions and arguments

Many things in R are done using function calls.
Functions consist of

- a name
- a pair of brackets
- the arguments (none, one, or more)
- a return value (visible, invisible, `NULL`)

Example:

```
weight <- c(60, 72, 57, 90, 95, 72)
height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
plot(height, weight)
```

Arguments may be set to default values; what they are is documented in `?plot.default` (which `plot()` calls if no specific plot method exists for the first argument).

Functions and arguments

Arguments are passed

- either without name (in the defined order) → positional matching
- or with name (in arbitrary order) → keyword matching

```
plot(height, weight, pch=16, col="blue")
```

Even if no arguments are passed, the brackets need to be written, e. g., `ls()`, `dir()`, `getwd()`.

Entering only the name of a function without brackets will display the R code of that function.

Vectors

Vectors are of a defined mode and each element in that vector has that same mode, e.g.:

```
mode(weight)
[1] "numeric"
```

Other than `numeric` vectors there are

- `character` vectors with their elements in quotation marks

  ```
  c("subj01", "subj02", "subj03")
  [1] "subj01" "subj02" "subj03"
  ```

- `logical` vectors with values `TRUE`, `FALSE` and `NA` (missing values are indicated by `NA`)

  ```
  c(TRUE, TRUE, FALSE, TRUE)
  [1]  TRUE  TRUE FALSE  TRUE
  ```

  Logical vectors often result from a comparison

  ```
  weight > 60
  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE
  ```

Creating vectors

Vectors are created by

- concatenating elements using `c()`
- sequences

```
0:10                    # 11 numbers from 0 to 10
seq(1.65, 1.90, .05)    # sequence in steps of 0.05
```

- repeating elements

```
rep(1:5, 3)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

rep(c("high", "low"), each=3)
[1] "high" "high" "high" "low"  "low"  "low"

rep(c("f", "m"), c(5, 2))
[1] "f" "f" "f" "f" "f" "m" "m"

paste("stim", 1:5, sep="")
[1] "stim1" "stim2" "stim3" "stim4" "stim5"
```

Matrices and arrays

Generating a 3×4 matrix

```
A <- matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
      [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Labeling and transposing the matrix

```
rownames(A) <- c("a1", "a2", "a3")  # also: colnames()
t(A)                                # transpose
     a1 a2 a3
[1,]  1  5  9
[2,]  2  6 10
[3,]  3  7 11
[4,]  4  8 12
```

R implements linear algebra and matrix operations, such as matrix multiplication (using %*%) and inversion (using `solve()`).

Matrices and arrays

A matrix may be created by concatenating columns or rows.

```
cbind(a1=1:4, a2=5:8, a3=9:12)  # "column bind"
rbind(a1=1:4, a2=5:8, a3=9:12)  # "row bind"
```

Arrays are data structures having more than two dimensions.

```
array(c(A, 2*A), c(3, 4, 2))  # 3 x 4 x 2 array
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
, , 2
     [,1] [,2] [,3] [,4]
[1,]    2    4    6    8
[2,]   10   12   14   16
[3,]   18   20   22   24
```

Multidimensional contingency tables can be constructed using `table()`, `xtabs()`, and `ftable()`.

Factors

Factors are data structures for categorical variables, such as diagnosis, socio-economic status, or sex.

```
ses <- factor(c("low", "inter", "high"))
[1] low   inter high
Levels: high inter low
```

If factors are created from character vectors, then the factor levels are ordered alphabetically. The ordering of the levels may be changed using the `levels` argument.

```
ses2 <- factor(ses, levels=c("low", "inter", "high"))
[1] low   inter high
Levels: low inter high
```

Experimental factors may easily be generated by combining `factor()` and `rep()`.

```
factor(rep(1:2, each=10), labels=c("on", "off"))
```

4 Data structures in R (2)

Lists, data frames

Lists

It is often necessary to store an ordered collection of R objects with more than one mode in a single object: a list.

```
list1 <- list(w=weight, h=height, s=ses2, A=A)
```

The components of a list are extracted using $ and the name of the component, or using [[]] and either the the name or number of the component.

```
list1$h  # or list1["h"]] or list1[[2]]
[1] 1.75 1.80 1.65 1.90 1.74 1.91

list1$A  # or list1["A"]] or list1[[4]]
     [,1] [,2] [,3] [,4]
a1    1    2    3    4
a2    5    6    7    8
a3    9   10   11   12
```

The fundamental data structure: data frames

Data frames are lists that consist of vectors and factors of equal length. The rows in a data frame refer to one unit (observation or subject).

```
id <- factor(paste("s", 1:6, sep=""))
dat <- data.frame(id, weight, height)
  id weight height
1 s1     60   1.75
2 s2     72   1.80
3 s3     57   1.65
4 s4     90   1.90
5 s5     95   1.74
6 s6     72   1.91
```

Variables in a data frame are extracted by $.

```
dat$id
[1] s1 s2 s3 s4 s5 s6
Levels: s1 s2 s3 s4 s5 s6
```

Working with data frames

Frequently used functions (not only) for data frames (1)

```
dim(dat)      # show number of rows and columns
[1] 6 3

names(dat)    # variable names
[1] "id"     "weight" "height"

View(dat)     # open data viewer

plot(dat)     # pairwise plots
```

Working with data frames

Frequently used functions (not only) for data frames (2)

```
str(dat)      # show variables of dat
'data.frame':   6 obs. of  3 variables:
 $ id    : Factor w/ 6 levels "s1","s2","s3",..: 1 2 3 4 5 6
 $ weight: num  60 72 57 90 95 72
 $ height: num  1.75 1.8 1.65 1.9 1.74 1.91

summary(dat)  # descriptive statistics
  id         weight          height
 s1:1   Min.   :57.00   Min.   :1.650
 s2:1   1st Qu.:63.00   1st Qu.:1.742
 s3:1   Median :72.00   Median :1.775
 s4:1   Mean   :74.33   Mean   :1.792
 s5:1   3rd Qu.:85.50   3rd Qu.:1.875
 s6:1   Max.   :95.00   Max.   :1.910
```

Indexing variables

Elements of a variable can be accessed using `[]` (see `?Extract`).

```
weight[4]             # 4th element
weight[4] <- 92       # change 4th element
weight[c(1, 2, 6)]    # elements 1, 2, 6
weight[1:5]           # elements 1 to 5
weight[-3]            # without element 3
```

Indices may be logical.

```
weight[weight > 60]                    # values greater than 60
weight[weight > 60 & weight < 80]      # between 60 and 80

height[weight > 60 & weight < 80]
```

Logical expressions may contain AND `&`, OR `|`, EQUAL `==`, NOT EQUAL `!=`, and `<`, `<=`, `>`, `>=`, `%in%`.

For more information on logical expressions, see `?Comparison` (or `?">"`), `?Logic` (or `?"&"`), and `?match` (or `?"%in%"`).

Indexing data frames

Data frames have a row and a column index. Indices can be numeric or character vectors. Omitting one index selects all rows or columns, respectively.

```
dat[3, 2]                    # 3rd row, 2nd column
dat[1:4, ]                   # rows 1 to 4, all columns
dat[, c(2,4)]                # all rows, colums 2 and 4
dat[, "id"]                  # all rows, column "id"
dat[, c("id", "weight")] # all rows, columns "id" and "weight"
```

Using logical indices, specific rows may be selected for which certain conditions hold.

```
dat[dat$id == "s2", ]                    # all obs. of s2
dat[dat$id %in% c("s2","s5"),]           # all obs. of s2 and s5
dat[dat$id == "s2" | dat$id =="s5",]     # same as above
dat[dat$weight > 60, ]                   # all obs. above 60kg
```

Matrices and arrays have two or more indices.

```
array(c(A, 2*A), c(3, 4, 2))[, , 2]  # 2nd "slice"
```

Sorting

Data frames are most conveniently sorted using `order()` which returns the indices of the ordered variable.

```
order(dat$weight)                # 60 72 57 90 95 72
[1] 3 1 2 6 4 5

dat[order(dat$weight), ]         # sort by weight
  id weight height
3 s3     57   1.65
1 s1     60   1.75
2 s2     72   1.80
6 s6     72   1.91
4 s4     90   1.90
5 s5     95   1.74
```

`order()` accepts multiple arguments to allow for sorting by more than a single variable.

```
data[order(data$A, data$B), ]  # first sort by A, then by B
```

Aggregating

Frequently, one wants to aggregate a data frame, for example in order to average over repetitions or in order to calculate error sums etc. This may be achieved using the `aggregate()` function.

```
## Single response variable, single grouping variable
aggregate(y ~ x, data, FUN, ...)

## Multiple response and grouping variables
aggregate(cbind(y1, y2) ~ x1 + x2, data, FUN, ...)
```

The `y` variables are numeric data to be split into groups according to the grouping `x` variables (usually factors).

`FUN` is a function to compute the summary statistics.

Aggregating: example

```
head(esoph)  # smoking, alcohol and esophageal cancer
   agegp     alcgp    tobgp ncases ncontrols
1 25-34 0-39g/day 0-9g/day      0        40
2 25-34 0-39g/day    10-19      0        10
3 25-34 0-39g/day    20-29      0         6
4 25-34 0-39g/day      30+      0         5
5 25-34     40-79 0-9g/day      0        27
6 25-34     40-79    10-19      0         7
```

Aggregate across age groups:

```
aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, esoph, sum)
        alcgp    tobgp ncases ncontrols
1  0-39g/day 0-9g/day      9       261
2      40-79 0-9g/day     34       179
3     80-119 0-9g/day     19        61
4       120+ 0-9g/day     16        24
5  0-39g/day    10-19     10        84
6      40-79    10-19     17        85
...
```

Reshaping: long to wide

Frequently, data are in either long (one line per observation) or wide (one line per case) format. `reshape()` can be used to switch between both formats.

```
head(Indometh)          # long data format
    Subject time conc
1         1 0.25 1.50
2         1 0.50 0.94
3         1 0.75 0.78
4         1 1.00 0.48
...
## From long to wide
df.w <- reshape(Indometh, v.names="conc", timevar="time",
                idvar="Subject", direction="wide")
    Subject conc.0.25 conc.0.5 conc.0.75 conc.1 ...
1         1      1.50     0.94      0.78   0.48
12        2      2.03     1.63      0.71   0.70
23        3      2.72     1.49      1.16   0.80
...
```

Reshaping: wide to long

```
## From wide to long
df.l <- reshape(df.w, varying=list(2:12), v.names="conc",
  idvar="Subject", direction="long",
  times=c(.25, .5, .75, 1, 1.25, 2, 3, 4, 5, 6, 8))
       Subject time conc
  1.0.25       1 0.25 1.50
  2.0.25       2 0.25 2.03
  3.0.25       3 0.25 2.72
  4.0.25       4 0.25 1.85
  ...
## Optionally, re-order by subject
df.l[order(df.l$Subject), ]
       Subject time conc
  1.0.25       1 0.25 1.50
  1.0.5        1 0.50 0.94
  1.0.75       1 0.75 0.78
  1.1          1 1.00 0.48
  ...
```

5 Data input and output

- Reading data
- Data entry
- Writing data

Reading tabular text files

ASCII text files in tabular or spread sheet form (one line per observation, one column per variable) are read using `read.table()`.

```
dat <- read.table("D:/experiment/data.txt", header=TRUE)
```

This creates a data frame, numerical variables are converted to vectors, character variables to factors.

It is recommended to first open any file which (possibly) is a text file in your text editor (e.g., Vim) to find out which options to set when reading it into R:

Frequently used options

- `header`: variable names in the first line?
- `sep`: which delimiter between variables (white space, tabulator `\t`, comma, semicolon etc.)?
- `dec`: decimal point or comma?
- `skip`: number of lines to skip before reading data

Reading arbitrary text files

ASCII text files in arbitrary form are read using `readLines()`.

```
raw <- readLines("D:/experiment/rawdata.txt")
```

This creates a character vector with as many elements as lines of text.

The character vector is further processed using string functions or regular expressions (`?regex`).

Example:

```
readLines(file.path(Sys.getenv("R_HOME"), "doc", "THANKS"))
 [1] "R would not be what it is today without the invaluable help of these"
 [2] "people, who contributed by donating code, bug fixes and documentation:"
 [3] ""
 [4] "Valerio Aimale, Thomas Baier, Roger Bivand, Ben Bolker, David Brahm,"
 [5] "Goran Brostrom, Patrick Burns, Vince Carey, Saikat DebRoy, "
 [6] "Brian D'Urso, Lyndon Drake, Dirk Eddelbuettel, Claus Ekstrom,"
...
```

Reading files created by other software

Several possible strategies

- Save data as text file in other software. Then employ one of these functions:

```
read.table()  # decimal point "." and column separator " "
read.csv()    # decimal point "." and column separator ","
read.csv2()   # decimal point "," and column separator ";"
read.delim()  # decimal point "." and column separator TAB
read.delim2() # decimal point "," and column separator TAB
```

- Reading SPSS data files

```
library(foreign)  # load add-on package
dat <- read.spss("file.sav", to.data.frame=TRUE)
```

- Additional information (e. g., about Excel files, data bases) in R Data Import/Export manual

Combining data frames

Data frames having the same column names can be combined using `rbind()`.

```
dat <- rbind(dat1, dat2, dat3)
```

More flexibility is provided by `merge()`.

Example: merge by ID

```
dat <- merge(dat1, dat2, by="id", all=TRUE)
```

This creates a data frame with as many rows as *different* IDs. The new data frame includes all columns in `dat1` and `dat2`, possibly with missing values if IDs are in either `dat1` or `dat2`, but not in both.

Data entry

Small data vectors can be copied into the R console using `scan()`:

```
x <- scan(text = "<data>")     # for numeric input
y <- scan(text = "<data>",
          what = "character") # for character input
```

A simple spread-sheet editor is invoked by

```
dat <- edit(data.frame())      # start with empty dataframe
```

It may be more convenient to use external software (text editor, spread-sheet application) for data entry. Data preparation and validation, however, should be entirely done in R.

For reproducibility, note: only as little manual editing as necessary, but as much script-based data processing as possible!

Writing data

Writing text files

```
write.table(dat, "exp1_data.txt", ...)  # many optional
                                        #   arguments
write.csv(), write.csv2()
```

Text files are maximally portable (also across software packages), human readable, but may lose some features of the data structures (e. g., ordering of factor levels).

R binary files

```
save(dat, file = "exp1_data.rda")       # write binary file
load("exp1_data.rda")                   # recreate dat
```

R binary files are exact representations of the data structures, portable across platforms (within R only), but not human readable.

6 Programming in R

- Control statements
- Avoiding loops
- Writing functions

R programming traps

Learning to program in a new languange means making a lot of errors.

Some errors are unexpected both for beginners and for those who have programmed before in a different language.

Reading the FAQs, R help and R devel mailing lists (and Stackoverflow), the built-in documentation, and the manuals (Writing R Extensions), will help!

"The R Inferno" by Patrick Burns (`http://www.burns-stat.com/pages/Tutor/R_inferno.pdf`) is an entertaining and informative guide to the traps of the R language.

Conditional execution

Conditional execution of code is available in R via

```
if(expr_1) {
  expr_2
} else {
  expr_3
}
```

where `expr_1` must evaluate to a single logical value.

Additional `if` conditions can be implemended via `else if`:

```
if(expr_1) {
expr_2
} else if(expr3) {
expr_4
} else {
expr_5
}
```

where `expr_3` must also evaluate to a single logical value.

Conditional execution

Example

```
x <- 3
if(!is.na(x)) {y <- x^2} else {stop("x is missing")}
y
[1] 9
x <- NA
if(!is.na(x)) {y <- x^2} else {stop("x is missing")}
Fehler: x is missing
```

There is also a vectorized version of the if/else construct, the `ifelse()` function:

```
x <- 1:3
ifelse(x > 1, "larger than 1", "equal to 1")
[1] "equal to 1" "larger than 1"  "larger than 1"
```

Loops

There are for() and while() loops for repeated execution.

Example: Unbiased estimation of the population variance σ^2

Create vector `v` which will record the (uncorrected) estimated sample variance

```
v <- numeric(1000) # 1000 samples will be simulated
```

Draw 1000 samples, each with size $n = 10$, from a normal distribution $N(0, 5^2)$. Then, for each sample, assign the estimated sample variance to the component `v[i]` of the vector (`i` runs from 1 to 1000)

```
for(i in seq_along(v)) {
  x <- rnorm(n=10, mean=0, sd=5)
  v[i] <- sum((x - mean(x))^2) / (10-1)
}
mean(v) # approx. equal to 5^2
```

Loops

Warning: for() loops are used in R code much less often than in compiled languages. Code that takes a "whole object" view is likely to be both clearer and faster in R

Example for an unnecessary loop:

```
x <- rnorm(10)
y <- numeric(10)
for(i in seq_along(x)) {
  y[i] <- x[i] - mean(x)
}
```

Same return value, but clearer and faster calculation:

```
y <- x - mean(x)
```

Avoiding loops

The `apply()` family of functions may be used in many places where in traditional languages loops are employed.

Matrices and arrays: `apply()`

`apply()` is used to work vector-wise on matrices or arrays. Appropriate functions can be applied to the columns or rows of a matrix or array without explicitly writing code for a loop.

```
X <- matrix(rnorm(20), nrow = 5, ncol = 4)
apply(X, 2, max)      # maximum for each column
apply(X, 1, mean)     # mean of each row; rowMeans(x)
```

Avoiding loops

Data frames, lists and vectors: `lapply()` and `sapply()`

Using the function `lapply()` (`l` because the value returned is a list), a function can be applied element-wise to other objects, for example, data frames, lists, or simply vectors. The resulting list has as many elements as the original object to which the function is applied.

```
X_list <- as.list(as.data.frame(X))
apply(X_list, max)   # does NOT work, a list is not an array
lapply(X_list, max)  # same result as apply(X, 2, max)
```

The function `sapply()` (`s` for simplify) works like `lapply()` with the exception that it tries to simplify the value it returns. It might become a vector or a matrix.

```
sapply(iris, class)      # works column wise on data frame
                         # returns a character vector
Sepal.Length  Petal.Length  Petal.Width      Species
   "numeric"     "numeric"    "numeric"     "factor"
```

Avoiding loops

Repeated evaluation of the same expression: `replicate()`

`replicate()` is a wrapper function for `sapply()` designed to repeatedly evaluate exactly the same expression. It makes repeated random number generation very simple:

```
# Two alternative, rather complicated calls using sapply()
sapply(1:3, FUN = rnorm, n = 5)
sapply(1:3, function(x) rnorm(5))

# Same number generation, but simpler call using replicate()
replicate(3, rnorm(5))
```

```
           [,1]       [,2]       [,3]
[1,]  2.1990893 -0.1518117  1.4779021
[2,] -0.3680195 -0.5322069  0.5849734
[3,] -1.5413471  0.3421399 -0.9059937
[4,] -1.5915875 -0.3482059 -0.1155943
[5,]  0.1867226  1.2135592  1.1641212
```

Avoiding loops

Multiple expressions may be executed by `apply()`-type functions.

Example: Unbiased estimation of the population variance σ^2

```
# Using replicate()
v <- replicate(1000, {
  x <- rnorm(n=10, mean=0, sd=5)
  sum((x - mean(x))^2) / (10-1)
})
mean(v)

# Using sapply()
v <- sapply(1:1000, function(y) {
  x <- rnorm(n=10, mean=0, sd=5)
  sum((x - mean(x))^2) / (10-1)
})
mean(v)
```

Avoiding loops

Group-wise calculations: `tapply()`

`tapply()` (`t` for table) may be used (instead of `aggregate()`) to do group-wise calculations on vectors. Frequently it is employed to calculate group-wise means.

```
data(Oats, package="nlme")

# One factor
tapply(Oats$yield, Oats$Block, mean)

# Two factors
tapply(Oats$yield, list(Oats$Block, Oats$Variety), mean)
    Golden Rain Marvellous Victory
VI        90.25     109.00   89.50
V         95.50      85.25   92.00
III       86.75     118.50   82.50
IV       108.00      95.00   91.50
II       113.25     121.25   87.25
I        133.25     129.75  143.00
```

Writing functions

New functions may be defined interactively in a running R session using `function()`.

Example: a handmade t-test function (there is one in R already!)

```
twosam <- function(y1, y2) {                    # definition
  n1  <- length(y1); n2  <- length(y2)          # body
  yb1 <- mean(y1);   yb2 <- mean(y2)
  s1  <- var(y1);    s2  <- var(y2)
  s_p <- ((n1 - 1)*s1 + (n2 - 1)*s2)/(n1 + n2 - 2)
  tst <- (yb1 - yb2)/sqrt(s_p*(1/n1 + 1/n2))
  tst  # return value; exactly one object; is often a list
}
```

Calling the function

```
tstat <- twosam(data$male, data$female)
tstat
```

Named arguments

If there is a function `fun1` defined by

```
fun1 <- function(data, data.frame, graph, limit) { ... }
```

then the function may be invoked in several ways, for example

```
fun1(d, df, TRUE, 20)
fun1(d, df, graph=TRUE, limit=20)
fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

All of them are equivalent (cf. positional matching and keyword matching).

Defaults

In many cases, arguments can be given commonly appropriate default values, in which case they may be omitted from the call.

```
fun1 <- function(data, data.frame, graph=TRUE,
                 limit=20) { ... }
```

It could be called as

```
ans <- fun1(d, df)
```

which is now equivalent to the three cases above, or as

```
ans <- fun1(d, df, limit=10)
```

which changes one of the defaults.

Classes and methods

The return value of a function may have a specified class. The class of an object determines how it will be treated by so-called generic functions.

To find out which generic functions have a method for a given class, use:

```
methods(class = "nameOfClass")
```

Example:

```
methods(class = "factor")
[1] [
[4] [<-
[19] levels<-
[22] plot
[23] print
[29] summary
...
```

Classes and methods

The return value of a function may have a specified class. The class of an object determines how it will be treated by so-called generic functions.

Conversely, to find out which classes have a method for a generic function, use:

```
methods(genericFunction)
```

Example:

```
methods(print)
[1] print.acf*
[2] print.anova*
[3] print.aov*
...
```

Access the documentation (if available) for a method via e.g.

```
?print.class
```

A user-defined print method

A print method for the `twosam()` function could be

```
print.twsm <- function(obj) {
  cat("\nMy two-sample t test\n\n")
  cat("Value of test statistic: ", obj$tst, "\n")
  invisible(obj)  # return the object invisibly
}
```

(See `?cat` and `?Quotes` for message formatting.)

For this to work, the original function needs to return an object of class "twsm."

```
twosam <- function(y1, y2) {
  ...
  retval <- list(tst = tst)  # define return value
  class(retval) <- "twsm"    # assign class
  retval
}
```

Debugging

If an R program gives an error message, it may not be obvious where in the program the error occurred. The `traceback()` function lists the function calls that led to the last error in reverse order.

```
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }

foo()                   # call to function foo() gives an error
[1] 1
Error: object 'a.variable.which.does.not.exist' not found

traceback()             # find out where the error occurred
2: bar(2)
1: foo()
```

Debugging

When trying to find errors in the code, it may help to look inside a function while running it, and to execute it step by step. This can be done using `debug()`.

```
debug(twosam)
twosam(rnorm(10), rnorm(10) + 2)
...
Browse[1]>
```

At the prompt, any R code may be entered. Some useful commands:

- `ls()` displays what the function "sees" at each step
- `n` (or the return key) evaluates the next command of the function
- `Q` quits the function browser
- For more commands, see `?browser`

In order to disable the debugging mode for the function, type

```
undebug(twosam)
```

7 Graphics

Traditional graphics

Trellis graphics

R graphics systems

There are two graphics systems in R

- Traditional graphics
- Grid graphics

The former is easier to control, the latter provides more flexibility. The lattice package is based on the grid graphics system; it implements Trellis plots in R.

Both systems provide high-level functions that produce complete plots and low-level functions that add further output to an existing plot.

R graphics systems

Manual, documentation, examples

- See the "Graphical procedures" chapter in the Introduction to R manual for an introduction to the traditional graphics system.
- See Murrell (2011) for comprehensive documentation, and `http://www.stat.auckland.ac.nz/~paul/RG2e/` for examples.
- For a compendium of clean graphs by E.-J. Wagenmakers and Q. Gronau see `http://www.shinyapps.org/apps/RGraphCompendium/index.php`
- See `demo(graphics)` for built-in examples of high-level plotting functions.

Graphics devices

Graphics devices control the format in which a plot is produced.

When using R interactively, the standard output of graphics functions is on the screen.

```
x11()      # open new plotting window (X or Windows)
quartz()   # Mac OS Quartz window
```

File devices

```
postscript()    # vector graphics
pdf()
win.metafile() # Windows only; native format for Word etc.

png()           # bitmap graphics
tiff()
jpeg()
bmp()
```

Exporting graphics to files

Plots may be exported to various file formats. With vector graphics (such as eps or wmf) plots can be re-scaled without losing quality.

An eps graphic is produced by

```
postscript("myplot.eps", horizontal=FALSE, onefile=FALSE,
           height=3.375, width=3.375, pointsize=10)
## plotting commands ##
dev.off()
```

A Windows compatible wmf graphic is obtained by

```
win.metafile("myplot.wmf", height=3.375, width=3.375,
             pointsize=10)
## plotting commands ##
dev.off()
```

Traditional graphics functions

High-level functions

```
plot()      # scatter plot, specialized plot methods
boxplot()
hist()      # histogram
qqnorm()    # quantile-quantile plot
barplot()
curve()     # function plot
pie()       # pie chart
pairs()     # scatter plot matrix
persp()     # 3d plot
contour()   # contour plot
coplot()    # conditional plot
interaction.plot()
```

Many of these plot functions (e.g. `plot()`, `hist()`, `curve()`) may be added to an existing plot via the argument `add = TRUE`.

Traditional graphics functions

Low-level functions

```
points()    # add points
lines()     # add lines
rect()
polygon()
abline()    # add line with intercept a, slope b
arrows()    # useful for e.g. confidence intervals
text()      # add text in plotting region
mtext()     # add text in margins region
axis()      # customize axes
box()       # box around plot
legend()
```

Plot layout in traditional graphics

A figure consists of two regions

- the plotting region (containing the actual plot)
- the margins region (containing axes and labels)

Example: scatter plot

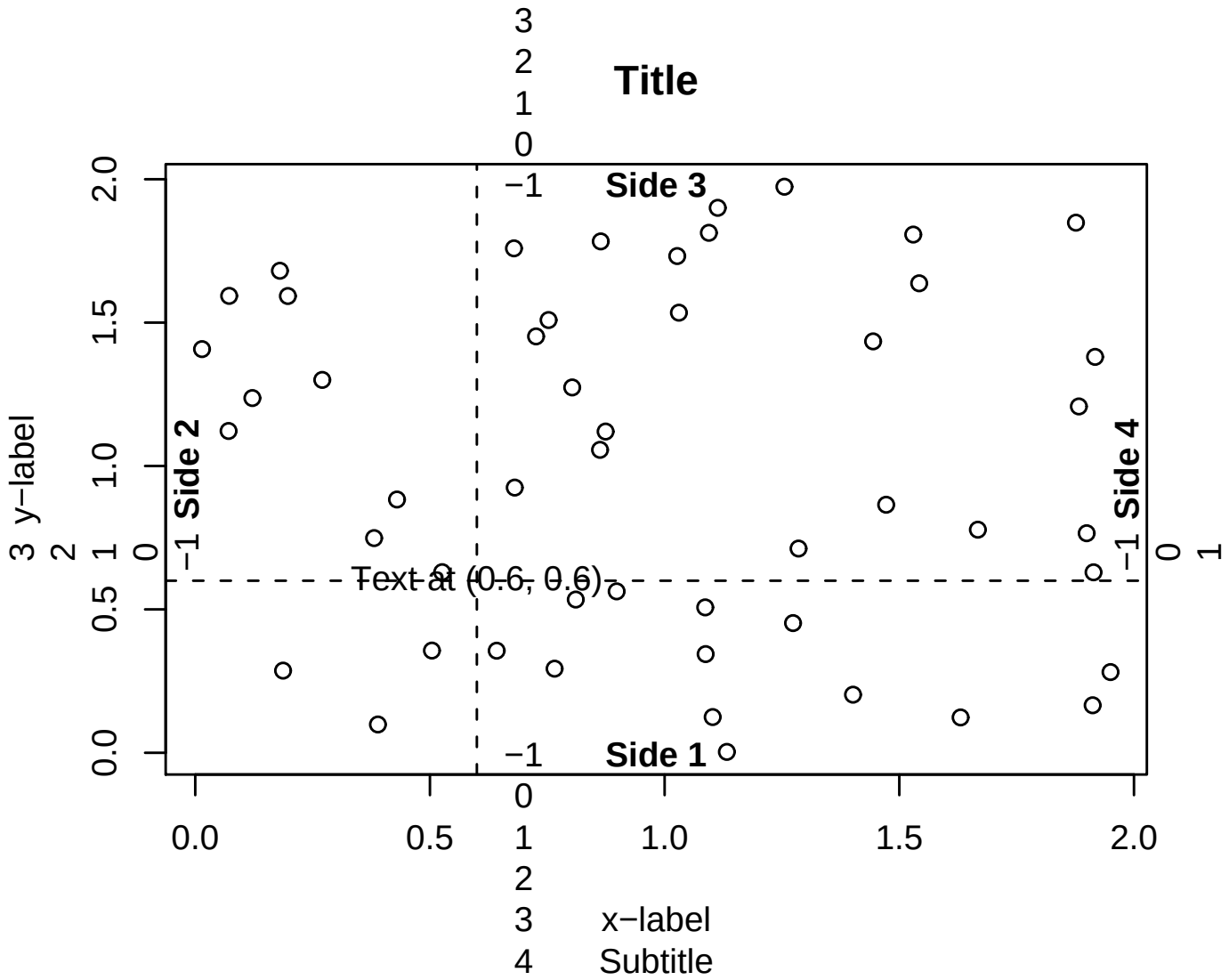
```
x <- runif(50, 0, 2)        # 50 uniform random numbers
y <- runif(50, 0, 2)
plot(x, y, main="Title", sub="Subtitle", xlab="x-label",
     ylab="y-label")        # produce plotting window
```

Writing text into plotting region

```
text(0.6, 0.6, "Text at (0.6, 0.6)")
abline(h=.6, v=.6, lty=2)  # horizont. and vertic. lines
```

Margins region

```
for(side in 1:4) mtext(-1:4, side=side, at=.7, line=-1:4)
mtext(paste("Side", 1:4), side=1:4, line=-1, font=2)
```



Stepwise plotting

In order to have full control over the details of a plot, it is recommended to build it step by step.

Consider as an example the mean reaction times in a two-factorial experiment; the interaction is to be graphically displayed.

```
dat <- read.table(header=TRUE, text="
   A  B  rt
  a1 b1 825
  a1 b2 792
  a1 b3 840
  a2 b1 997
  a2 b2 902
  a2 b3 786
")
```

Stepwise plotting

First produce only the plotting region and the coordinate system.

```
plot(rt ~ as.numeric(B), dat, type="n", axes=FALSE,
     xlim=c(.8, 3.2), ylim=c(750, 1000),
     xlab="Difficulty", ylab="Mean reaction time (ms)")
```

Plot the data points separately for each level of factor *A*.

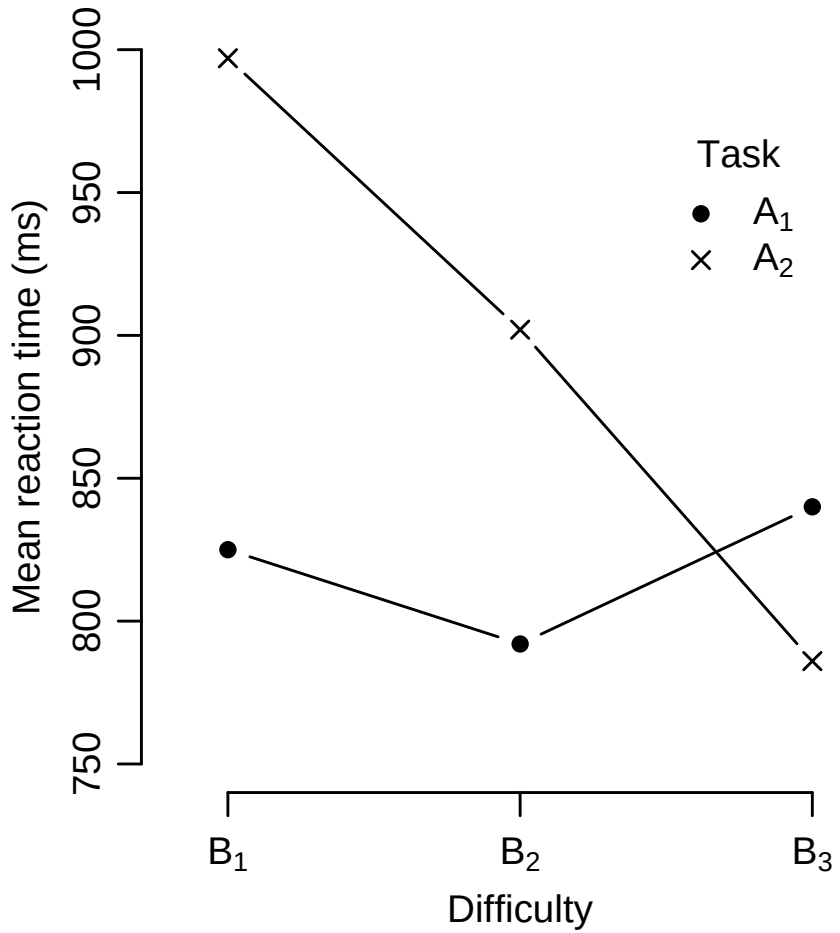
```
points(rt ~ as.numeric(B), dat[dat$A=="a1", ], type="b", pch=16)
points(rt ~ as.numeric(B), dat[dat$A=="a2", ], type="b", pch=4)
```

Add axes and a legend.

```
axis(side=1, at=1:3, expression(B[1], B[2], B[3]))
axis(side=2)
legend(2.5, 975, expression(A[1], A[2]), pch=c(16, 4),
       bty="n", title="Task")
```

Using `expression()`, mathematical expressions may be plotted (see `?plotmath` and `example(plotmath)`).

Stepwise plotting



- Error bars may be added using the `arrows()` function.
- Via `par()` many graphical parameters may be set (see `?par`), for example `par(mgp=c(2, .7, 0))` reduces the distance between labels and axes.

Graphical parameters

Some graphical parameters are accepted as arguments by high- and low-level plotting functions.

```
adj   # justification of text
bty   # box type for legend
cex   # size of text or data symbols (multiplier)
col   # color, see colors()
las   # rotation of text in margins
lty   # line type (solid, dashed, dotted, ...)
lwd   # line width
mgp   # placement of axis ticks and tick labels
pch   # data symbol type
tck   # length of axis ticks
type  # type of plot (points, lines, both, none)
```

Graphical parameters

Some graphical parameters may only be set by a call to `par()`.

```
mai    # size of figure margins (inches)
mar    # size of figure margins (lines of text)
mfrow  # number of sub-figures on a page
oma    # size of outer margins (lines of text)
omi    # size of outer margins (inches)
pty    # aspect ratio of plot region (square, maximal)
```

Parameters changed via `par()` are automatically reset when a new device is opened.

Setting and resetting parameters (for the active device):

```
# Change parameters while also saving the original parameters
# This works because par() returns the previously set values
op <- par(mfrow=c(1,2),          # two sub-figures side by side
          mai=c(.6,.6,.15,.1))  # margins (btm, lft, top, rgt)
# At end of plotting, reset to previous settings:
par(op)
```

Lattice graphics

The lattice package (Sarkar, 2008) implements Trellis plots in R. These are plots conditional on other variables. They are perfectly suited for visualizing complex relationships.

See `http://lmdvr.r-forge.r-project.org/` for examples.

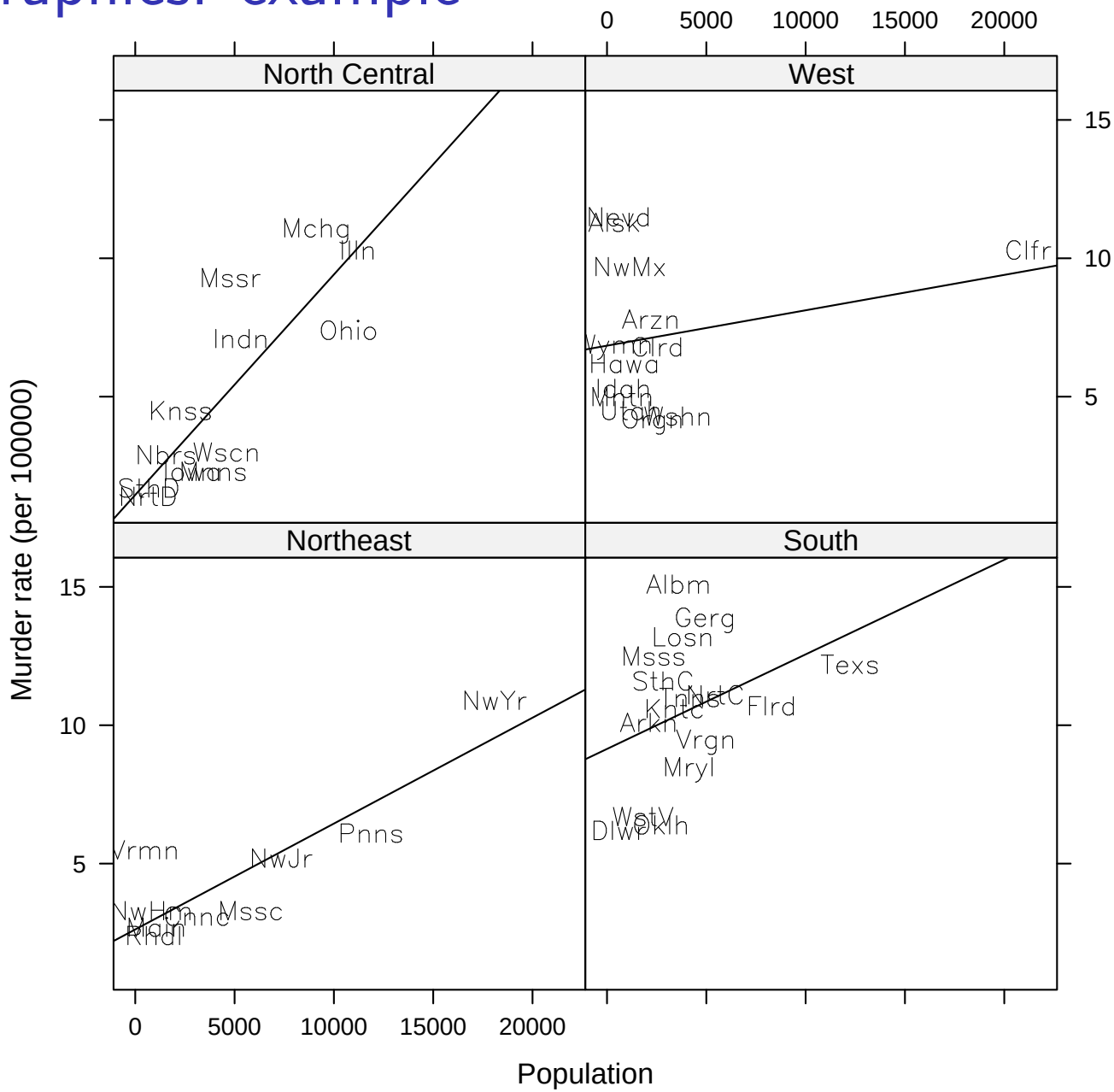
The lattice package is loaded into R by

```
library(lattice)
```

Example: murder rate in the USA

```
states <- data.frame(state.x77, state.name = state.name,
  state.region = state.region)   # built-in data sets
```

Lattice graphics: example



Lattice graphics functions

High-level functions

```
xyplot()        # scatter plot
bwplot()        # boxplot
histogram()
qqmath()        # quantile-quantile plot
barchart()
cloud()         # 3d scatter plot
wireframe()     # 3d surface
contourplot()   # contour plot
```

High level functions in lattice take a formula argument to indicate the conditioning variable across panels.

```
xyplot(Murder ~ Population | state.region, states)
```

The group argument may be used to define the grouping variable within panels.

```
xyplot(Murder ~ Population | state.region, states,
       group=state.name)
```

Panel functions and strip functions

Panel functions control what is drawn in each panel. High-level plot functions have a default panel function. Custom panel functions may be passed via the panel argument.

```
panel.xyplot()    # scatter plot
panel.abline()    # draw (regression) lines
panel.grid()      # draw grid lines
panel.points()    # draw points
panel.lines()     # draw lines
panel.arrows()    # draw arrows
panel.text()      # write text into panel
```

Strip functions control the appearance of the strip above the panels. The format of the strip text is changed using the `par.strip.text` argument.

Panel functions and strip functions

Define a function to be used as the `panel` argument. This function can consist of multiple panel functions.

```
plot.regression <- function(x,y) {
  # panel-specific scatter plot
  panel.xyplot(x, y)
  # panel-specific regression line
  panel.abline(lm(y ~ x))
  # grid for all panels; aligned with ticks
  panel.grid(-1, -1)
  # overall mean for all panels
  panel.abline(h=mean(states$Murder), lty=2)
}
```

Call high-level plot function and set `panel` accordingly

```
xyplot(Murder ~ Population | state.region,
       data = states,
       panel = plot.regression)
```

Fine control

```
xyplot(Murder ~ Population | state.region, states,
  groups = state.name,
  xlim = c(-5000, 25000),
  ylab = "Murder rate (per 100000)",
  # panel function
  panel = function(x, y, subscripts, groups) {
    panel.grid(-1, -1)
    panel.abline(lm(y ~ x))
    panel.abline(h=mean(states$Murder), lty=2
    # plot labels instead of scatter plot
    panel.text(x, y, labels=groups[subscripts], cex=1)
  },

  scales = list(tck=.5, cex=.7,
                x=list(at=seq(0, 20000, 5000))),

  strip = strip.custom(bg="gray96"),  # strip function
  par.strip.text = list(cex=.7)
)
```

8 Classical hypothesis tests

- Nominal data
- Metric response variables
- Ordinal response variables

Classical hypothesis testing in R

R is a free software environment for statistical computing and graphics.

The `stats` package, which is automatically loaded when R starts up, contains functions for statistical calculations and random number generation.

For a complete list of functions, see `library(help=stats)`.

Examples of functions to perform classical hypothesis testing in R

Nominal data

- `binom.test()` performs an exact test of a simple null hypothesis about the probability of success in a Bernoulli experiment
- `chisq.test()` performs chi-squared contingency table tests and goodness-of-fit tests

Metric response variables

- `cor.test()` for association between paired samples
- `t.test()` performs one- and two-sample t tests
- `var.test()` performs an F test to compare the variances of two samples from normal populations

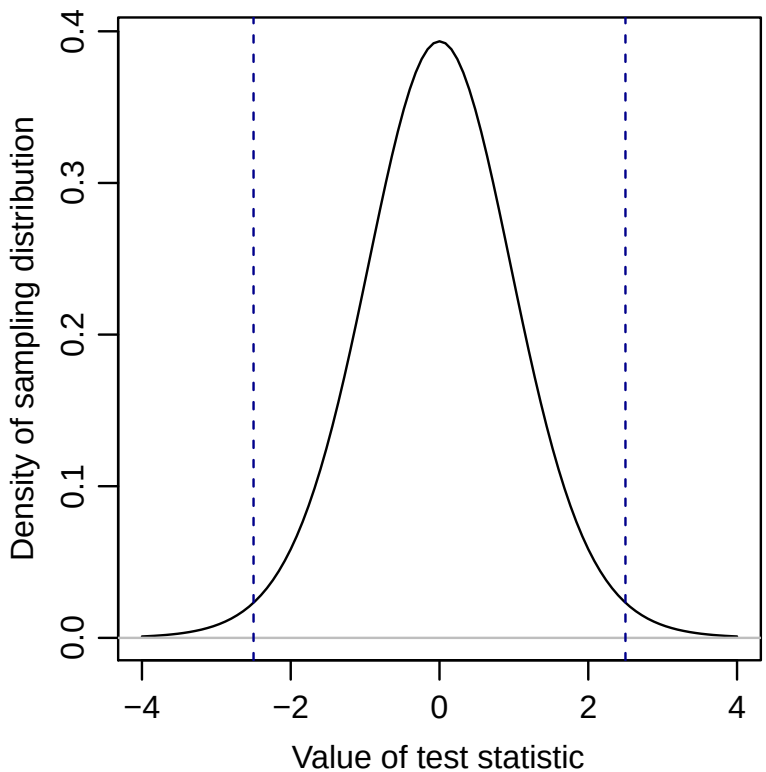
Ordinal response variables

- `wilcox.test()` performs one- and two-sample Wilcoxon tests

P-value and statistical decision rule

The p-value is the conditional probability of obtaining a test statistic at least as extreme as the one that was observed, given that the null hypothesis is true.

If the p-value is smaller than the probability α of a Type I decision error, then the null hypothesis is rejected.




```
curve(dt(x, 18), -4, 4)
abline(h=0, v=c(-2.5, 2.5))
```

Binomial test

- Assumptions
  $X_1, \ldots, X_n$ independent and identically Bernoulli distributed with parameter p
- Hypotheses
  1. H_0: $p = p_0$ H_1: $p \neq p_0$
  2. H_0: $p \geq p_0$ H_1: $p < p_0$
  3. H_0: $p \leq p_0$ H_1: $p > p_0$
- Test statistic

$$T = \sum_{i=1}^{n} X_i, \qquad T \sim B(n, p_0)$$

- R function

```
binom.test(8, 20, p = 0.25)
```

Chi-squared goodness-of-fit test

- Assumptions
  $X_i, \ldots, X_n$ independent and identically multinomially distributed
- Hypotheses
  - H_0: $P(X_i = k) = p_k$ for all $k = 1, \ldots, r$ $(i = 1, \ldots, n)$
  - H_1: $P(X_i = k) \neq p_k$ for at least one $k \in 1, \ldots, r$ $(i = 1, \ldots, n)$
- Test statistic

$$X^2 = \sum_{k=1}^{r} \frac{(Y_k - np_k)^2}{np_k}, \qquad X^2 \sim \chi^2(r-1)$$

- R function

```
tab <- xtabs(count ~ spray, InsectSprays)
chisq.test(tab)
```

Chi-squared test of homogeneity

- Assumptions

  $X_{j1}, \ldots, X_{jn_j}$ independent and identically multinomially distributed
- Hypotheses
  - H_0: $P(X_{ji} = k) = p_k$ for all $j = 1, \ldots, m$, $i = 1, \ldots, n_j$ and $k = 1, \ldots, r$
  - H_1: $P(X_{ji} = k) \neq P(X_{j'i} = k)$ for some $j \neq j' \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, r\}$
- Test statistic

$$X^2 = \sum_{j=1}^{m} \sum_{k=1}^{r} \frac{(x_{jk} - \hat{\mu}_{jk})^2}{\hat{\mu}_{jk}}, \qquad X^2 \sim \chi^2((m-1)(r-1))$$

- R function

```
tab <- xtabs(~ education + induced, infert)
chisq.test(tab)
```

Chi-squared test of independence

- Assumptions
  Independent pairs (X_i, Y_i), $i = 1, \ldots, n$, mit $X_i \in \{1, \ldots, m\}$ and $Y_i \in \{1, \ldots, r\}$
- Hypotheses
  - H_0: $P(X_i = j, Y_i = k) = P(X_i = j) \cdot P(Y_i = k)$ for all j, k and $i = 1, \ldots, n$
  - H_1: $P(X_i = j, Y_i = k) \neq P(X_i = j) \cdot P(Y_i = k)$ for some j, k and $i = 1, \ldots, n$
- Test statistic
  cf. test of homogeneity
- R function

```
chisq.test(HairEyeColor[,,"Female"])
```

Correlation test

- Assumptions
  Independent jointly normally distributed variables (X_i, Y_i), $i = 1, \ldots, n$
- Hypotheses
  1. H_0: $\rho_{XY} = 0$ H_1: $\rho_{XY} \neq 0$
  2. H_0: $\rho_{XY} \geq 0$ H_1: $\rho_{XY} < 0$
  3. H_0: $\rho_{XY} \leq 0$ H_1: $\rho_{XY} > 0$
- Test statistic

$$T = \frac{r_{XY}}{\sqrt{1 - r_{XY}^2}} \sqrt{n-2}, \qquad T \sim t(n-2)$$

- R function

```
cor.test(~ speed + dist, cars,
         alternative = "two.sided", method = "pearson")
```

Two-sample t test (independent)

- Assumptions
  Independent samples $X_1, \ldots, X_n \sim N(\mu_x, \sigma_x^2)$ and $Y_1, \ldots, Y_n \sim N(\mu_y, \sigma_y^2)$; σ_x^2, σ_y^2 unknown but equal
- Hypotheses
  1. H_0: $\mu_x - \mu_y = 0$ H_1: $\mu_x - \mu_y \neq 0$
  2. H_0: $\mu_x - \mu_y \geq 0$ H_1: $\mu_x - \mu_y < 0$
  3. H_0: $\mu_x - \mu_y \leq 0$ H_1: $\mu_x - \mu_y > 0$
- Test statistic

$$T = \frac{\bar{x} - \bar{y}}{\hat{\sigma}_{\bar{x}-\bar{y}}}, \qquad T \sim t(n + m - 2)$$

- R function

```
t.test(weight ~ group,
       PlantGrowth[PlantGrowth$group != "ctrl",],
       var.equal = TRUE)
```

Two-sample t test (dependent)

- Assumptions
  $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ with
  $D_1 = X_1 - Y_1, \ldots, D_n = X_n - Y_n$ independent and identically distributed with $N(\mu_d, \sigma_d^2)$ and unknown σ_d^2
- Hypotheses
  1. H_0: $\mu_d = 0$ H_1: $\mu_d \neq 0$
  2. H_0: $\mu_d \geq 0$ H_1: $\mu_d < 0$
  3. H_0: $\mu_d \leq 0$ H_1: $\mu_d > 0$
- Test statistic

$$T = \frac{\bar{d}}{\hat{\sigma}_{\bar{d}}}, \qquad T \sim t(n-1)$$

- R function

```
with(sleep,
     t.test(extra[group == 1],
            extra[group == 2], paired = TRUE))
```

Test of equal variances

- Assumptions
  Independent samples $X_1, \ldots, X_n \sim N(\mu_x, \sigma_x^2)$ and $Y_1, \ldots, Y_n \sim N(\mu_y, \sigma_y^2)$, unknown σ_x^2, σ_y^2
- Hypotheses
  1. H_0: $\sigma_x^2 = \sigma_y^2$ $\quad$ H_1: $\sigma_x^2 \neq \sigma_y^2$
  2. H_0: $\sigma_x^2 \geq \sigma_y^2$ $\quad$ H_1: $\sigma_x^2 < \sigma_y^2$
  3. H_0: $\sigma_x^2 \leq \sigma_y^2$ $\quad$ H_1: $\sigma_x^2 > \sigma_y^2$
- Test statistic

$$T = \frac{S_x^2/\sigma_x^2}{S_y^2/\sigma_y^2}, \qquad T \sim F(n-1, m-1)$$

- R function

```
var.test(weight ~ group,
         PlantGrowth[PlantGrowth$group != "ctrl",])
```

Wilcoxon rank-sum test (Mann-Whitney U test)

- Assumptions
  $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_m$ each independent and identically continuously distributed and $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ independent
- Hypotheses
  1. H_0: $\tilde{\mu}_x = \tilde{\mu}_y$ H_1: $\tilde{\mu}_x \neq \tilde{\mu}_y$
  2. H_0: $\tilde{\mu}_x \geq \tilde{\mu}_y$ H_1: $\tilde{\mu}_x < \tilde{\mu}_y$
  3. H_0: $\tilde{\mu}_x \leq \tilde{\mu}_y$ H_1: $\tilde{\mu}_x > \tilde{\mu}_y$
- Test statistic

$$U = \sum_{i=1}^{n} rk(X_i) - \frac{n(n+1)}{2}$$

  (distribution of Wilcoxon rank-sum statistic, see `?pwilcox`)
- R function

```
wilcox.test(weight ~ group,
            PlantGrowth[PlantGrowth$group != "ctrl",])
```

9 Linear models

- Formulae
- Overview
- Simple linear regression
- Multiple linear regression
- Generalized linear models

Formulae

Statistical models are represented by R formulae. R formulae usually correspond closely to the referring statistical models.

Formulae used in `lm()` and `glm()`, let A, B be fixed factors

`y ~ 1 + x`	$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$
`y ~ x`	(short form)
`y ~ 0 + x`	$y_i = \beta_1 x_i + \varepsilon_i$
`y ~ x + I(x^2)`	$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i$
`y ~ A + B`	$y_{ijk} = \mu + \alpha_i + \beta_j + \varepsilon_{ijk}$
`y ~ A*B`	$y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijk}$
`y ~ 1 + A + B + A:B`	(long form)

Formulae used in `aov()`, let P be a random factor

`y ~ A + Error(P/A)`	$y_{ij} = \mu + \alpha_j + \pi_i + \varepsilon_{ij}$
`y ~ 1 + A + Error(P + P:A)`	(long form)
`y ~ A*B + Error(P/(A*B))`	$y_{ijk} = \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk} + \pi_i + (\pi\alpha)_{ij} + (\pi\beta)_{ik} + \varepsilon_{ijk}$

Linear models

(More information is in the "Statistical models in R" chapter in the Introduction to R manual.)

Linear models of the form

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \varepsilon,$$

where y is assumed to be independent and normally distributed as $N(\mu, \sigma^2)$, are fit in R using `lm()`.

```
lm(y ~ x1 + x2 + ... + xk, data)
```

There are many extractor functions that work on lm objects.

Extractor functions for lm objects

```
coef()      # Extract the regression coefficients

summary()   # Print a comprehensive summary of the results of
            #   the regression analysis. This also returns a
            #   list from which further useful information
            #   can be extracted.

anova()     # Compare nested models and produce an analysis
            #   of variance table (for incremental F tests)

residuals() # Extract the (matrix of) residuals

fitted()    # Fitted values

plot()      # Produce four plots, showing residuals, fitted
            #   values and some diagnostics
```

Extractor functions for lm objects

```
predict()  # Either predicted values for the observed data,
           #   i.e. the fitted values
           #
           # Or a new data frame can be supplied having the
           #   same variables specified with the same labels
           #   as the original. The value is a vector or
           #   matrix of predicted values corresponding to the
           #   determining variable values in the data frame

vcov()     # Return the variance-covariance matrix of the
           #   main parameters of a fitted model object

model.matrix() # Return the design matrix
```

Coding of factors: contrasts

Factors with k levels are represented by $k - 1$ indicator variables.

Dummy coding (default): Indicators only take the values 0 and 1. If all dummies are zero, the reference level is indicated. Dummy coded effects are interpreted with respect to the reference level. Dummy coding is set in R by

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

Effect coding: Indicators take the values -1, 0, and 1. Effects are interpreted as a deviation from the mean. Effect coding is switched on in R by

```
options(contrasts = c("contr.sum", "contr.poly"))
```

Example

Math ability (amount of math problems solved in 2 minutes) as a linear function of working memory capacity (forwards letter span).

```
dat <- read.table("dataInverse.txt", header=TRUE)
dat <- na.omit(dat)
plot(Math ~ LetSp_f, dat)
```

Example

Fit the regression model and Wald tests of predictors

```
lm1 <- lm(Math ~ LetSp_f, dat)
summary(lm1)

Call:
lm(formula = Math ~ LetSp_f, data = dat)
Residuals:
     Min       1Q   Median       3Q      Max
-17.2196  -5.8799  -0.8799   5.3402  24.5603
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   4.7602     1.4474   3.289  0.00104 **
LetSp_f       1.7799     0.2474   7.195 1.22e-12 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 7.934 on 996 degrees of freedom
Multiple R-squared:  0.04941,   Adjusted R-squared:  0.04846
F-statistic: 51.77 on 1 and 996 DF,  p-value: 1.223e-12
```

Example

Coefficients

```
coef(lm1)
```

Predicted values

```
fitted(lm1)
```

Residuals

```
resid(lm1)
```

Confidence intervals for parameters

```
confint(lm1)
```

Add regression line to scatter plot

```
abline(lm1)
```

Example

Prediction and confidence intervals

For a given predictor x_0, we are often interested in a prediction for $Y_0 = \alpha + \beta \cdot x_0 + \varepsilon_0$. A reasonable point estimate is $\hat{Y}_0 = \hat{\alpha} + \hat{\beta} \cdot x_0$.

With the variances of $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\sigma}$ we can calculate the variance of the prediction error $Y_0 - \hat{Y}_0$ and the confidence interval for Y_0:

$$\hat{Y}_0 \pm \hat{\sigma} \cdot t_{1-\alpha/2}(n-2) \cdot \sqrt{1 + \frac{1}{n} + \frac{(x_0 - \bar{x})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}}$$

The confidence interval for the "true" regression line $\alpha + \beta\, x$ is not affected by ε_0 and its (constant) variance $\hat{\sigma}$, so the 1 underneath the square root is removed:

$$\hat{\alpha} + \hat{\beta}\, x \pm \hat{\sigma} \cdot t_{1-\alpha/2}(n-2) \cdot \sqrt{\frac{1}{n} + \frac{(x - \bar{x})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}}$$

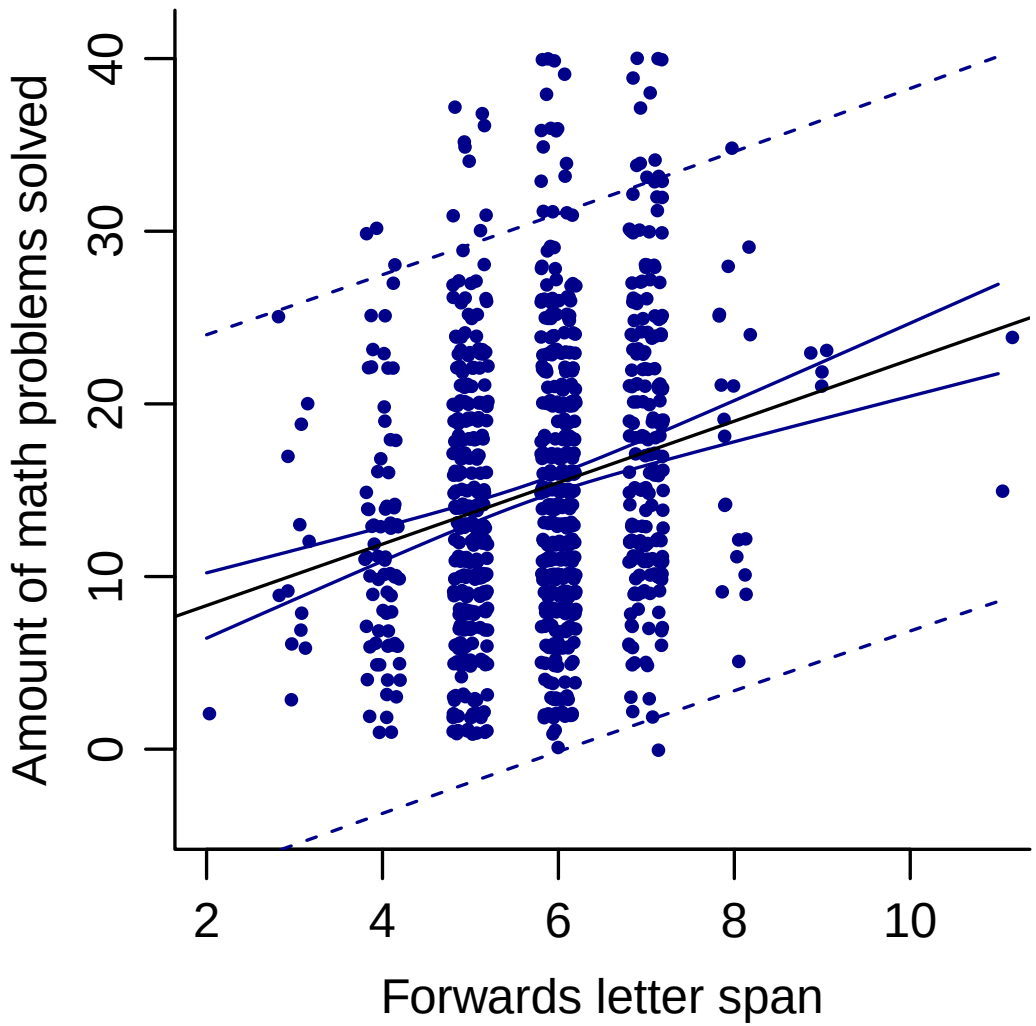
Example

Add confidence and prediction intervals

```
## define x values which will be supplied in a new dataframe
xval <- seq(2, 11, length.out=101)
## 95% confidence interval for the regression line
lines(xval, predict(lm1, data.frame(LetSp_f=xval),
      interval=c("confidence"))[,"lwr"])
lines(xval, predict(lm1, data.frame(LetSp_f=xval),
      interval=c("confidence"))[,"upr"])
## 95% prediction interval for the observations
lines(xval, predict(lm1, data.frame(LetSp_f=xval),
      interval=c("prediction"))[,"lwr"], lty=2)
lines(xval, predict(lm1, data.frame(LetSp_f=xval),
      interval=c("prediction"))[,"upr"], lty=2)
```

Model predictions and intervals

Scatter plot with fitted regression line, 95% confidence interval for the regression line, and 95% prediction interval for the observations



Multiple linear regression

More predictors is not always better!

The main issue is *multicollinearity*, which arises due to correlated predictors. Multicollinearity causes instability of the regression equation, may bias the estimated test statistics, and impedes parameter interpretation (when the size or even sign of a parameter estimate depends on the inclusion of other predictors).

So we need a way of testing whether adding predictors to the model is justified, whether two predictors have similar or different effects, etc.

Hypothesis tests

Parameter tests can be constructed via model comparison. These tests are appropriate for hierarchical ("nested") models, which are created by parameter restriction. Some restriction types are:

Parameter elimination (parameter is set to 0)

$$Y_i = \beta_0 + \beta_1 \cdot X_{i1} + \varepsilon_i \quad (0)$$
$$Y_i = \beta_0 + \beta_1 \cdot X_{i1} + \beta_2 \cdot X_{i2} + \varepsilon_i \quad (1)$$

Parameter fixation (parameter is set to k)

$$Y_i = \beta_0 + k \cdot X_{i1} + \varepsilon_i \quad (0)$$
$$Y_i = \beta_0 + \beta_1 \cdot X_{i1} + \varepsilon_i \quad (1)$$

Parameter equation (parameters are set to the same value)

$$Y_i = \beta_0 + \beta_3 \cdot X_{i1} + \beta_3 \cdot X_{i2} + \varepsilon_i \quad (0)$$
$$Y_i = \beta_0 + \beta_1 \cdot X_{i1} + \beta_2 \cdot X_{i2} + \varepsilon_i \quad (1)$$

In all three cases, models 0 and 1 are hierarchical models.

Hypothesis tests

Implementation of hierarchical models by parameter restriction

Parameter elimination (parameter is set to 0)

```
lm(y ~ x1)                # (0)
lm(y ~ x1 + x2)           # (1)
```

Parameter fixation (parameter is set to k)

```
lm(y ~ offset(k * x1))    # (0)
lm(y ~ x1)                # (1)
```

Parameter equation (parameters are set to the same value)
(for two continuous predictors x_1 and x_2)

```
lm(y ~ I(x1 + x2))        # (0)
lm(y ~ x1 + x2)           # (1)
```

Hypothesis tests

Incremental F test

Two hierarchical models can be compared using this test. Let's call the more complex model M_1 and the restricted model M_0. The restricted model may have parameter restrictions on more than one parameter.

Test statistic (q_0 and q_1 are equal to the number of parameters in M_0 and M_1, respectively):

$$F = \frac{(SS_{\hat{Y}1} - SS_{\hat{Y}0})/(q_1 - q_0)}{SS_{\hat{\varepsilon}1}/(N - q_1)} = \frac{(SS_{\hat{\varepsilon}0} - SS_{\hat{\varepsilon}1})/(q_1 - q_0)}{SS_{\hat{\varepsilon}1}/(N - q_1)}$$

$$= \frac{N - q_1}{q_1 - q_0} \cdot \frac{R_1^2 - R_0^2}{1 - R_1^2}$$

$$F \sim F(q_1 - q_0, N - q_1)$$

Hypothesis tests

Incremental F test

The null hypothesis H_0 is stated by the parameter restriction, i.e. for the three examples on the previous slides:

- Parameter elimination; $H_0 : \beta_2 = 0$
- Parameter fixation; $H_0 : \beta_1 = k$
- Parameter equation; $H_0 : \beta_1 = \beta_2$

We test whether the restricted model has to be rejected in favor of the complex model. If the restricted model explains significantly less variance than the complex model, it is — relative to the complex model — inappropriate for describing the observed data.

Hypothesis tests

Example: Incremental F test of parameter elimination

```
lm2 <- lm(Math ~ LetSp_f + LetSp_b, dat)
anova(lm1, lm2)

Analysis of Variance Table

Model 1: Math ~ LetSp_f
Model 2: Math ~ LetSp_f + LetSp_b
  Res.Df   RSS Df Sum of Sq      F    Pr(>F)
1    996 62696
2    995 61912  1    783.25 12.588 0.0004064 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Hypothesis tests

Example: Wald tests (note the equivalence to incr. F test above)

```
summary(lm2)

Call:
lm(formula = Math ~ LetSp_f + LetSp_b, data = dat)
Residuals:
     Min       1Q   Median       3Q      Max
-17.0125  -5.8373  -0.9911   5.1627  25.1627
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.5207     1.4809   2.378 0.017619 *
LetSp_f       1.3077     0.2796   4.676 3.33e-06 ***
LetSp_b       0.8676     0.2445   3.548 0.000406 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 7.888 on 995 degrees of freedom
Multiple R-squared:  0.06129,   Adjusted R-squared:  0.0594
F-statistic: 32.48 on 2 and 995 DF,  p-value: 2.16e-14
```

Generalized linear models

A generalized linear model is defined by

$$g(E(y)) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k,$$

where $g()$ is the link function that links the mean to the linear predictor. The response y is assumed to be independent and to follow a distribution from the exponential family.

In R, a GLM is fit by

```
glm(y ~ x1 + x2 + ... + xk, family(link), data)
```

Families

Each response distribution admits a variety of link functions to connect the mean with the linear predictor. Those automatically available are shown in the following table:

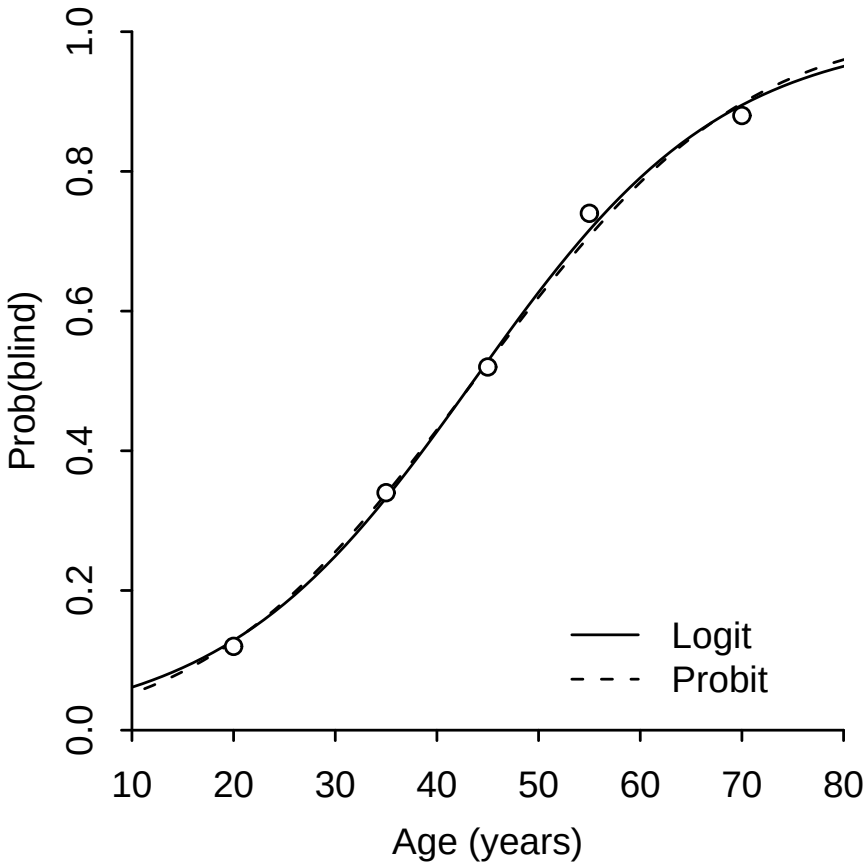
```
## Family name       Link functions
   binomial          logit, probit, log, cloglog
   gaussian          identity, log, inverse
   Gamma             identity, inverse, log
   inverse.gaussian  1/mu^2, identity, inverse, log
   poisson           log, identity, sqrt

   quasi             logit, probit, cloglog, identity,
                     inverse, log, 1/mu^2, sqrt
```

A GLM is a specific combination of a response distribution, a link function, and a linear predictor.

Binomial regression

Logit or probit models are special cases of GLMs for binomial response variables.

Artificial example: congenital eye disease



Logit model

$$\log \frac{p}{1-p} = \beta_0 + \beta_1 AGE$$

Probit model

$$\Phi^{-1}(p) = \beta_0 + \beta_1 AGE$$

Fitting binomial regression models

Fitting using `glm()`

```
dat <- data.frame(x = c(20,35,45,55,70), n = rep(50,5),
                  y = c(6,17,26,37,44))

glml <- glm(cbind(y, n - y) ~ x, binomial, dat)
glmp <- glm(cbind(y, n - y) ~ x, binomial(probit), dat)
```

Parameter estimates are obtained by

```
summary(glml)

             Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.53778    0.50232  -7.043 1.88e-12 ***
x            0.08114    0.01082   7.498 6.47e-14 ***
```

For the logit model, the parameter estimates may be interpreted as the log odds ratio: The odds of going blind are increased by a factor of $\exp(0.081) = 1.08$ when age increases by one year.

Goodness of fit and predictions

For grouped binomial data, the goodness of fit may be assessed using a likelihood ratio test (which is a generalization of the incremental F test).

```
## Compare to saturated model
glms <- glm(cbind(y, n - y) ~ factor(x), binomial, dat)
anova(glml, glms, test="Chisq")  # likelihood ratio test

Analysis of Deviance Table

Model 1: cbind(y, n - y) ~ x
Model 2: cbind(y, n - y) ~ factor(x)
  Resid. Df Resid. Dev Df Deviance Pr(>|Chi|)
1         3    0.31707
2         0  1.332e-15  3  0.31707    0.95679
```

Predictions based on new observations (see `?predict.glm`)

```
newx <- 0:100
predict(glml, data.frame(x = newx), type = "response")
```

⑩ Simulation based procedures

- Random numbers generation
- Resampling
- Parametric bootstrap

Random numbers generation

For drawing random numbers from a statistical distribution, the distribution name is prefixed by "r" (random deviate).

Examples (see `?Distributions` for a list of distributions):

```
rnorm(10)  # draw from standard normal distribution
rpois(10)  # draw from Poisson distribution
```

Sampling with or without replacement from a vector is performed by the `sample()` function.

```
sample(1:5, size=10, replace=TRUE)
[1] 1 4 1 3 3 2 1 1 4 3
```

The random numbers generator in R is *seeded*: Upon restart of R, new random numbers are generated. To replicate the results of a simulation, the seed (starting value) can be set explicitly:

```
set.seed(1223)  # set seed, so on each run
runif(3)        #   random numbers will be identical
```

Simulated power

The power is the probability of a test to become significant given H_0 is false, i. e., to (correctly) detect an effect.

Power can be estimated by simulation: It is the proportion of significant tests for a specified effect size, variability, sample size, and α.

Example: Power of one-sample t test to detect an effect of 3 Hz at $\sigma = 8$ Hz, $n = 30$, and $\alpha = 5\%$

```
pval <- replicate(1000, {
  x <- rnorm(30, mean=443, sd=8)  # draw from effect distrib.
  t.test(x, mu=440)$p.value       # test against H0
})
mean(pval < 0.05)                 # simulated power
[1] 0.503
```

Bootstrap methods

Bootstrap: simulation methods for frequentist inference

The name is based on the famous story about the Baron Munchhausen, who was stuck with his horse in a swamp and got out by pulling on his own bootstraps (in German: am eigenen Schopf). We'll see that something similar to this can actually be done in statistics ...

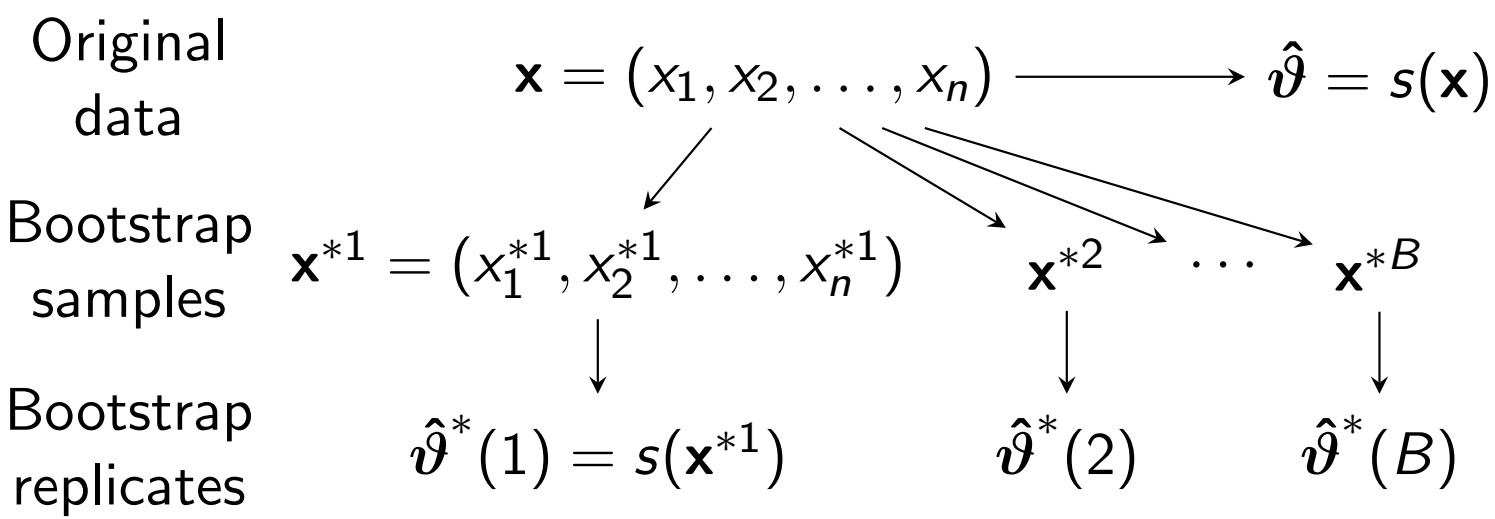
Useful when (Davison, 2006):

- standard assumptions invalid (n small, data not normal, ...)
- standard problem has non-standard twist
- complex problem has no (reliable) theory
- or (almost) anywhere else

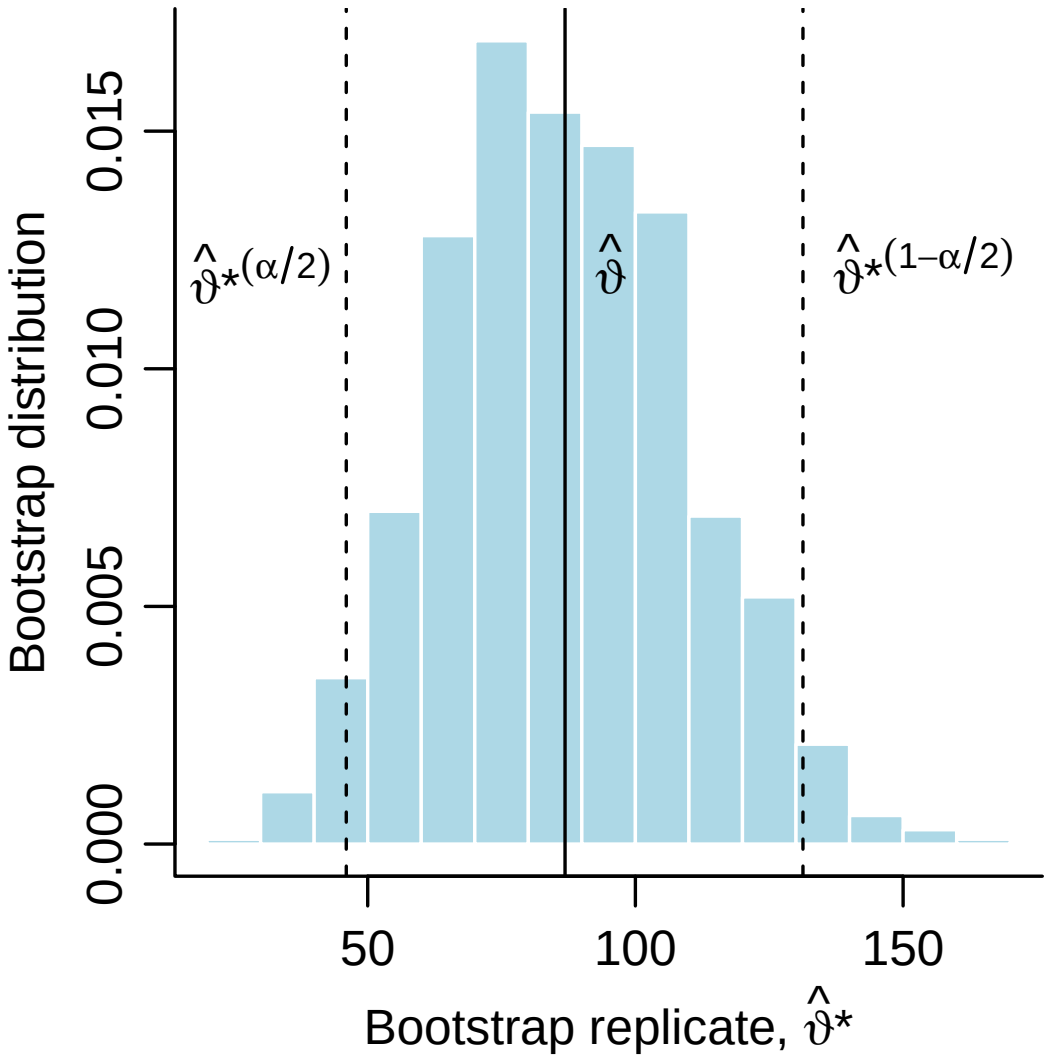
Bootstrap by resampling

Nonparametric bootstrap algorithm (Efron & Tibshirani, 1993):

- Select B bootstrap samples $\mathbf{x}^{*1}, \mathbf{x}^{*2}, \ldots, \mathbf{x}^{*B}$, each consisting of n data values drawn with replacement from the original data $\mathbf{x}$.
- Compute the bootstrap replicate $\hat{\boldsymbol{\vartheta}}^* = s(\mathbf{x}^*)$ of the statistic of interest $\hat{\boldsymbol{\vartheta}} = s(\mathbf{x})$ for each sample.
- Assess the variability of the statistic via the distribution of bootstrap replicates.



Bootstrap confidence intervals



For a given significance level α, the corresponding limits of the percentile interval of the bootstrap distribution are equal to its $\alpha/2$ and $1-\alpha/2$ quantiles.

This percentile interval is a $(1-\alpha)$ *bootstrap confidence interval* for ϑ.

Example: survival time of mice

```
## Mouse data (Efron & Tibshirani, 1993, p. 11)
mouse <- data.frame(
   grp = rep(c("trt", "ctl"), c(7, 9)),
  surv = c(94, 197, 16, 38, 99, 141, 23,              # trt
           52, 104, 146, 10, 50, 31, 40, 27, 46))     # ctl
## Observed difference of means
t.test(surv ~ grp, mouse, var.equal=TRUE)
meandiff <- with(mouse, diff(tapply(surv, grp, mean)))
## Resampling the observations and the difference of means
sam1 <- numeric(1000)
for(i in seq_along(sam1)) {
 trt <- sample(mouse[mouse$grp == "trt", "surv"], 7, replace=T)
 ctl <- sample(mouse[mouse$grp == "ctl", "surv"], 9, replace=T)
 sam1[i] <- mean(trt) - mean(ctl)
}
## 95% bootstrap confidence interval for difference of means
quantile(sam1, c(.025, .975))
     2.5%     97.5%
-22.28333  83.74048
```

Bootstrap based on a parametric model

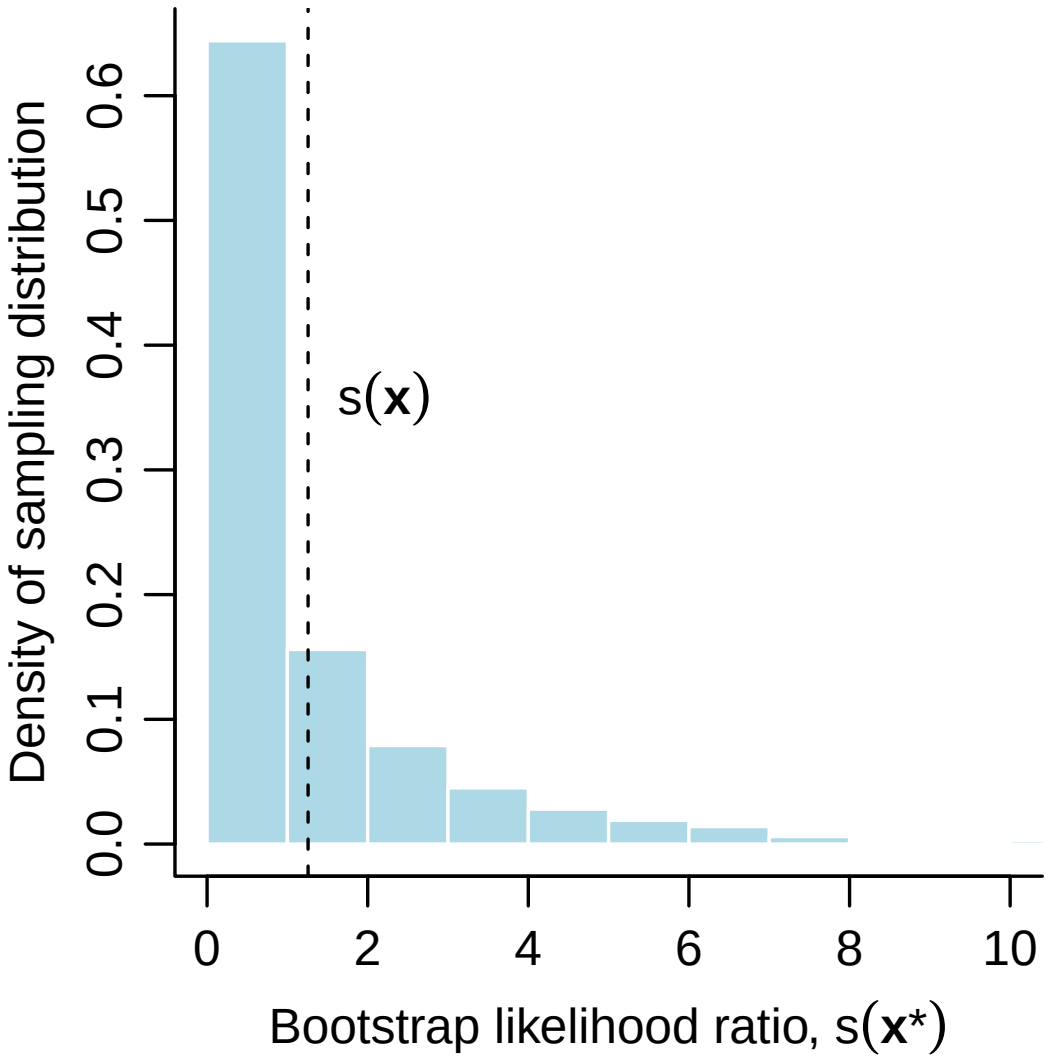
Parametric bootstrap algorithm for the likelihood ratio test:

- Fit a general (M_1) and a restricted model (M_0) to the original data $\boldsymbol{x}$. Compute the original likelihood ratio $s(\mathbf{x})$ between M_1 and M_0.
- Simulate B bootstrap samples $\mathbf{x}^{*B}$ based on the stochastic part of the restricted model: These are observations for which H_0 is true.
- For each sample, fit M_1 and M_0 and compute the bootstrap replicate $s(\mathbf{x}^*)$ of the likelihood ratio between them.
- Assess the significance of the original likelihood ratio via the sampling distribution of bootstrap replicates.

Example: survival time of mice revisited

```
## Model fit to original data
lm0 <- lm(surv ~ 1, mouse)    # H0: no difference between groups
lm1 <- lm(surv ~ grp, mouse) # H1: group effect
anova(lm0, lm1)               # original likelihood ratio
     F
1.2575

## Parametric bootstrap
sim1 <- numeric(1000)
for(i in seq_along(sim1)){
  surv0   <- simulate(lm0)$sim_1      # simulate from null model
  m0      <- lm(surv0 ~ 1, mouse)     # fit null model
  m1      <- lm(surv0 ~ grp, mouse)   # fit alternative model
  sim1[i] <- anova(m0, m1)$F[2]       # bootstrap likeli. ratio
}
```

Example: survival time of mice revisited



The bootstrap p-value is the proportion of bootstrap replicates that exceed the original likelihood ratio.

```
mean(sim1 >
  anova(lm0, lm1)$F[2])
[1] 0.298
```

11 Interactive web applications with Shiny

- Creating a Shiny app
- Customizing appearance
- Customizing reactions

Interactive web applications with Shiny

A Shiny app is an interactive web page that is created using R and functions from the Shiny package (and the many packages Shiny depends on).

Getting help

- `http://shiny.rstudio.com`
- `http://shiny.rstudio.com/tutorial/`
- `http://shiny.rstudio.com/gallery/`
- `http://shiny.rstudio.com/articles/`

Mailing list

- `https://groups.google.com/group/shiny-discuss`

Structure of a Shiny app

A Shiny app consists of

- a web page (HTML, CSS): the "user interface"
- an R script that runs on a server

Basic template

app.R

```
library(shiny)

ui <- fluidPage()                          # create web page
server <- function(input, output){}        # calculate output
                                           #   from user input
shinyApp(ui = ui, server = server)
```

App can be locally run by pasting code into R or by

```
runApp("path-to-dir/")
```

A simple example: histogram of random values

```
library(shiny)

## User interface with slider (input) and plot (output)
ui <- fluidPage(
  sliderInput("n", "Number of samples", 1, 100, 10),
  plotOutput("hist")
)

## Server function connecting input and output
server <- function(input, output){
  output$hist <- renderPlot({
    x <- rnorm(input$n)          # draw n random values
    hist(x)
  })
}

shinyApp(ui = ui, server = server)
```

The user interface: inputs and outputs

Building blocks of an interactive web page are elements the user interacts with; created by `*Input()` and `*Output()` functions.

```
sliderInput()           # Slider input widget
numericInput()          # Numeric input control
selectInput()           # Select list input control
checkboxInput()         # Checkbox input control
checkboxGroupInput()
dateInput()             # Date input
fileInput()             # File upload control
radioButtons()          # Radio buttons
textInput()             # Text input control
passwordInput()         # Password input control
actionButton()          # Action button
```

Every input needs a unique ID

```
sliderInput(inputId = "n", label = "Number of samples",
            min = 1, max = 100, value = 10)
```

The user interface: outputs

Outputs respond to changes of input values.

```
plotOutput()           # Plot output element
textOutput()           # Text output element
verbatimTextOutput()   # Verbatim text output element
tableOutput()          # Table output element
dataTableOutput()      # Data table output element
htmlOutput()           # HTML output element
uiOutput()             # user interface element
downloadButton()       # Download button
Progress()             # Reporting progress (object oriented)
withProgress()         # Reporting progress (functional)
outputOptions()        # Set options for an output object
```

Every output needs a unique ID

```
plotOutput(outputId = "hist")
```

The server function: assemble inputs into outputs

The server function takes two arguments

- the input list of reactive input values
- the output list of responding output objects

The names of the list elements must be identical to the input and output IDs in the user interface.

app.R

```
ui <- fluidpage(
sliderInput(inputId = "n", label = "Number of samples",
            min = 1, max = 100, value = 10)
plotOutput(outputId = "hist")
)
server <- function(input, output){
    ...
}
```

The server function: assemble inputs into outputs

How to create elements in the output list

1. Save objects to display to output$

app.R

```
ui <- fluidpage(
sliderInput(inputId = "n", label = "Number of samples",
            min = 1, max = 100, value = 10)
plotOutput(outputId = "hist")
)
server <- function(input, output){
  output$hist <- ...
}
```

The server function: assemble inputs into outputs

How to create elements in the output list

1. Save objects to display to `output$`
2. Build objects to display with `render*()`

app.R

```
ui <- fluidpage(
sliderInput(inputId = "n", label = "Number of samples",
            min = 1, max = 100, value = 10)
plotOutput(outputId = "hist")
)
server <- function(input, output){
  output$hist <- renderPlot({
    x <- rnorm(100)
    hist(x)
  })
}
```

Rendering output

```
renderPlot()        # Plot output
renderText()        # Text output
renderPrint()       # Printable output
renderTable()       # Table output
renderDataTable()   # Data table output
renderImage()       # Image file output
renderUI()          # UI (HTML) output
downloadHandler()   # File downloads
```

A `render*()` function

- creates an object to be displayed by a corresponding `*Output()` function
- takes as an argument an expression that creates the object (may be multiple lines of code included in `{}`)
- observes changes of reactive input values

The server function: assemble inputs into outputs

How to create elements in the output list

1. Save objects to display to `output$`
2. Build objects to display with `render*()`
3. Access reactive values via `input$`

app.R

```
ui <- fluidpage(
sliderInput(inputId = "n", label = "Number of samples",
            min = 1, max = 100, value = 10)
plotOutput(outputId = "hist")
)
server <- function(input, output){
  output$hist <- renderPlot({
    x <- rnorm(input$n)
    hist(x)
  })
}
```

Exercise

Expand the histogram-of-random-values example from the slides.

1. Add another input (`selectInput()` or `radioButtons()`) to let the user choose among drawing samples from three distributions:
   - normal
   - uniform (`runif()`)
   - exponential (`rexp()`)

   Hint: Use conditional execution (`if(){} else{}`) in the server function.

Customizing appearance

UI layout (to be used inside `fluidPage()`)

```
sidebarLayout(          # automatic layout with
  sidebarPanel(...),    #   sidebar and
  mainPanel(...)        #   main area
)
fluidRow(               # custom layout with
  column(6, ...),       #   two equal-width
  column(6, ...)        #   columns
)
wellPanel()             # color panel around inputs/outputs
```

HTML builder functions

```
builder (a, br, code, div, em, h1, h2, h3, h4, h5, h6,
         hr, img, p, pre, span, strong, tags)

HTML()          # mark characters as HTML

names(tags)     # all builder functions
```

Typesetting mathematical expressions

The MathJax library typesets Latex-style expressions on the fly.

Example

```
fluidPage(
  withMathJax(),
  "Here is an equation
   $$\\alpha + \\frac{1}{2},$$   <!-- displayed -->
   where \\(\\alpha = 2\\)."     <!-- inline -->
)
```

More information

- http://mathjax.org
- http://latex-project.org

Exercise

Expand the histogram-of-random-values example from the slides.

1. Add another input (`selectInput()` or `radioButtons()`) to let the user choose among drawing samples from three distributions:
   - normal
   - uniform (`runif()`)
   - exponential (`rexp()`)

   Hint: Use conditional execution (`if(){} else{}`) in the server function.
2. Add a layout: Try both, (1) an automatic layout using `sidebarLayout()`, `sidebarPanel()`, and `mainPanel()`; (2) a custom layout using `fluidRow()` and `column()`.

Customizing reactions using reactive constructs

Reactive values may only be called from inside a reactive context.

Reactivity is a two-step process

1. Reactive values notify objects that depend on them that they have changed (thus invalidating the objects).
2. Objects created by reactive functions respond (e. g. by re-running the code that creates the objects).

In addition to the `render*()` functions there are other reactive constructs that allow for finer control over reactions.

```
reactive()          # reactive expression
eventReactive()     # delay reactions
observe()           # reactive observer
observeEvent()      #   to run code on the server
reactiveValues()    # object for storing reactive values
isolate()           # non-reactive scope for an expression
```

Calculating values reactively

Reactive expressions: `reactive()`, `eventReactive()`

- Create an object to use in downstream code.
- This object is called like a function, and it returns a value.

Example

```
## hist() and summary() work on different data
server <- function(input, output){
  output$hist <- renderPlot(     hist(rnorm(input$n)) )
  output$desc <- renderPrint( summary(rnorm(input$n)) )
}
## hist() and summary() work on same data
server <- function(input, output){
  getdata <- reactive( rnorm(input$n) )
  output$hist <- renderPlot(     hist(getdata()) )
  output$desc <- renderPrint( summary(getdata()) )
}
```

Performing actions reactively

Reactive observers: `observe()`, `observeEvent()`

- Listen to a change of reactive values (often a button click).
- Are called for their side effects.

Examples

```
server <- function(input, output){
  observe({
    cat("The value of input$x is now ", input$x, "\n")
  })

  observeEvent(input$savebutton, {
    write.table(getdata(), file = "foo.txt")
  })
}
```

Reactive expressions versus observers

`reactive()` `eventReactive()`	`observe()` `observeEvent()`
Has to be called	Cannot be called
Returns a value	No return value
Lazy	Eager
Cached	–

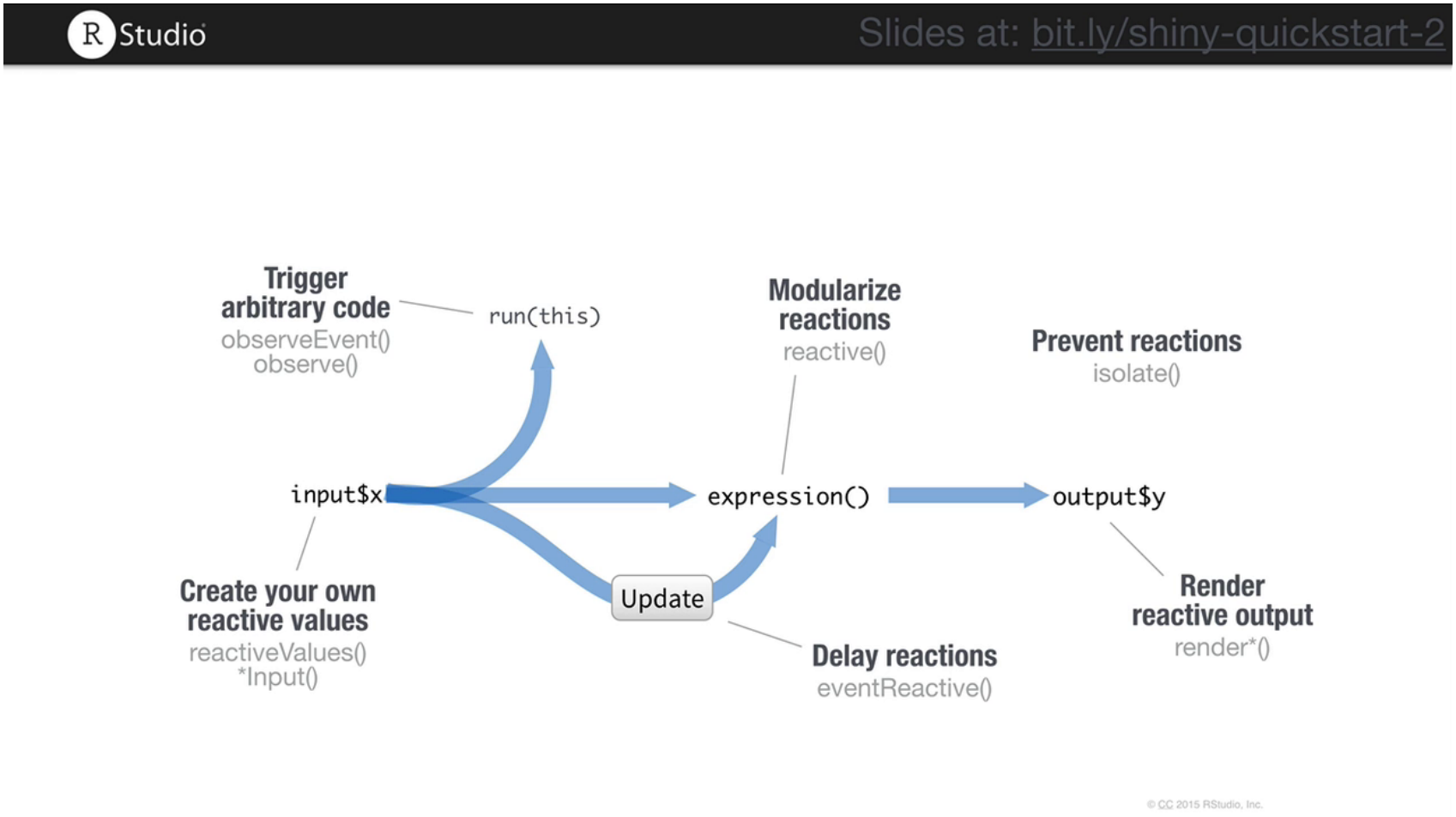
Used for different purposes

- `reactive()` is for calculating values, without side effects.
- `observe()` is for performing actions, with side effects.

Typical side effects

- save to file, upload to server
- setting `reactiveValues()`
- changing input elements: `updateSliderInput()` etc.

Customizing reactions using reactive constructs



References I

Becker, R. A., & Chambers, J. M. (1984). *S. An interactive environment for data analysis and graphics*. Monterey: Wadsworth and Brooks/Cole.

Dalgaard, P. (2008). *Introductory statistics with R* (2nd ed.). New York: Springer.

Davison, A. (2006). *Bootstrap methods and their application.* University Lecture. Retrieved from `https://www.researchgate.net/profile/Debashis_Kushary/publication/261638887_Bootstrap_Methods_and_Their_Application/links/54f5ccfe0cf27d8ed71cc1ae.pdf`

Efron, B., & Tibshirani, R. (1993). *An introduction to the bootstrap*. Boca Raton, FL: Chapman & Hall/CRC.

Ihaka, R., & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, *5*, 299–314.

Murrell, P. (2011). *R graphics*. Boca Raton, FL: Chapman & Hall/CRC.

R Core Team. (2018). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. (http://www.R-project.org/)

Sarkar, D. (2008). *Lattice: Multivariate data visualization with R*. New York: Springer. (http://lmdvr.r-forge.r-project.org/)

References II

Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S* (4th ed.). New York: Springer.

Verzani, J. (2005). *Using R for introductory statistics*. Boca Raton, FL: Chapman & Hall/CRC.

Wickham, H. (2014). *Advanced R*. Boca Raton, FL: Chapman & Hall/CRC.