Computer-Assisted Research Methods

Florian Wickelmaier

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



Last modified: March 6, 2026

```
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 twentieth century, it was said that if you wanted to do statistics, you had to be a bit of a mathematician, whether you want to or not. In the twenty-first century, if you want to do statistics, you have to be a bit of a programmer, whether you want to or not."

https://andrewgelman.com/2017/05/14/computer-programming-prerequisite-learning-statistics/

Contents

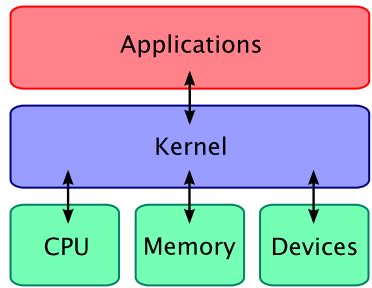
1. Know your OS/know your editor
2. Writing R Scripts
3. Data structures in R (1)
4. Data structures in R (2)
5. Data input and output
6. Programming in R
7. Graphics
8. Programming in Python and PsychoPy
9. Classical hypothesis tests
10. Generalized linear models
11. Simulation based procedures

1. Know your OS/know your editor
   - Kernel, shell, file system
   - Vim editor

Kernel and shell

A computer operating system (OS) consists of two major parts:

1. The kernel: bridge between applications and the actual data processing done at the hardware level.
2. A shell: interface that provides user access to the services of the kernel.



Shells are either command-line (CLI) based or graphical (GUI).

CLI-based shells (terminal, command prompt, console)

- `https://matt.might.net/articles/basic-unix/` (UNIX, Linux, Mac OS)
- `https://en.wikipedia.org/wiki/PowerShell` (Windows)

Shell commands

Accessing the file system

```
dir, ls  # list contents of directory
cd       # change directory
cp       # copy file(s)
man cp   # display help for cp command
mv       # move or rename file(s)
rm (-r)  # delete file(s) (and directories recursively)
mkdir    # create directory
notepad  # invoke a simple text editor
```

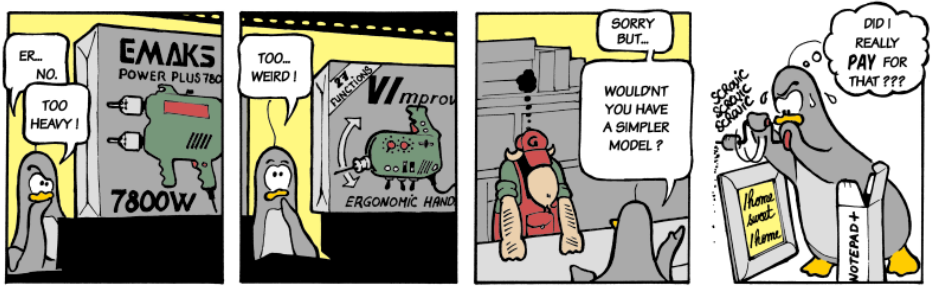
Wildcards

? one character, * zero or more characters, [a-z] characters a to z

Examples

```
ls *.txt                # list all files ending in .txt
cd ..                   # change into directory above
mv dat[1-3].csv ../foo  # move dat1.csv ... into folder foo
```

Vim editor



Copyright (c) 2007 Laurent Gregoire

- Vi IMproved
  "Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient."
- Cross-platform text editor (R, HTML, LaTeX, C, Python, ...)
- More information and download: `https://www.vim.org/`

Vim: modal editing

Command/Normal mode (Vim starts here)

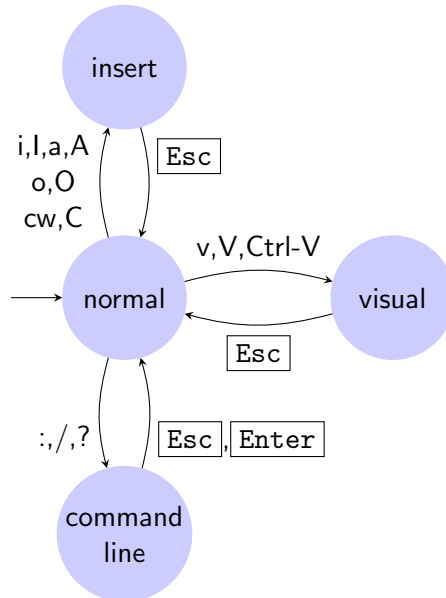
```
dd                  # delete line
yy                  # copy line
p                   # paste
3w                  # move three words
J                   # join lines
```

Insert mode: insert text

Colon/Command-line mode

```
:w                  # save file
:q                  # quit editor
:help vim-modes     # open help
:s/foo/bar/         # search for foo and
                    #   replace by bar
```

Visual mode: select areas of text



Vim editor

Getting help

- Interactive tutorial: `:Tutor`
- Built-in documentation and help: `:help`
- Help for a command, topic, or option: `:help <command>`
- Wiki: `https://vim.wikia.com/wiki/Vim_Tips_Wiki`

Setting options

```
:set textwidth=78        # break text after 78 characters
:set clipboard=unnamed   # copy to system clipboard
```

To make settings permanent, they can be stored in a `vimrc` file (`:help vimrc`), which is executed at start-up.

2 Writing R Scripts
- Computing in R
- Getting help
- Session management

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
  - Active development in thousands of contributed add-on 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 consist of letters, numbers, and dots (e. g., `errors.item6`), but they may not start with a number.

R is case sensitive (for both 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 are applied to each element.

```
(weight - mean(weight))^2  # squared deviation
[1] 205.4444   5.4444 300.4444 245.4444 427.1111   5.4444
```

Many R functions accept vectors as arguments.

```
log(weight)
[1] 4.0943 4.2767 4.0431 4.4998 4.5539 4.2767
```

The help system in R

R offers a number of sources for getting help.

The text-based documentation is called by ?

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

show the documentation of the `rnorm()` function or the `if` construct. At the bottom of the help page, there usually are examples that can be copied and pasted into the R console.

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

display functions having the expression "norm" in their name or in their documentation.

An HTML based local help system is available via

```
help.start()  # open a window in an HTML browser
```

Documentation locally available via `help.start()`

Manuals

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

Questions and answers

- Frequently Asked Questions
- FAQ for Windows

Technical papers

- Graphics, Sweave, ...

Resources on the Internet

R website: `https://www.r-project.org/`

- R mailing lists (especially R-help)
- The R Journal
- Books
- Many (shorter) documents, such as "Notes on the use of R for psychology experiments and questionnaires"

Stack Exchange network

- Programming in R
  `https://stackoverflow.com/tags/r`
- Applied statistics with R
  `https://stats.stackexchange.com/tags/r`

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.

Working directory

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

```
getwd()     # What is the current 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 scripts

All R code necessary to reproduce[1] an analysis should go into a text file. Such programs are called R scripts.

To execute the entire R code in a file from inside R, type

```
source("MyAnalysis.R")
```

To run an R script from outside R, enter

```
Rscript MyAnalysis.R
```

at the command line prompt of the shell.

[1] For even more reproducibility, integrate the analysis into your report: Write R code directly into a LaTeX report, process it using `Sweave()`, and compile it into a pdf document. With this workflow, when the data change, the report (and the tables and figures) will change, too.

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`
- R coding conventions
  `https://www.aroma-project.org/developers/RCC`

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

Consistency is usually considered a prerequisite for good style.

Packages

A large part of R's functionality is available in add-on packages. Some packages are pre-installed together with R.

```
library()              # Which packages are installed?
library(MASS)          # Load and attach package MASS
library(help = MASS)   # What functions does MASS provide?
search()               # Which packages are attached?
detach(package:MASS)   # Remove package MASS from search path
.libPaths()            # Display or set directory where to search for packages
MASS::mvrnorm()        # Call function without attaching package
```

Additional packages from R's web repository (`https://cran.r-project.org/`) are installed via

```
install.packages()
install.packages("xtable")  # install package xtable
```

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`)

Examples

```
mean(weight)
plot(height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91), 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 (`key = value`, in arbitrary order) → keyword matching

Example (positional and keyword arguments)

```
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.

Forward pipe

The return value of a function may serve as an argument for another function. The pipe operator passes the value on the left to the first argument of the function on the right.

```
nrow(subset(mtcars, cyl == 4))       # nesting
subset(mtcars, cyl == 4) |> nrow()   # piping
```

Use a named argument with the placeholder _ in the function call to specify where the value is to be inserted.

```
subset(mtcars, cyl == 4) |>
  aggregate(mpg ~ gear, data = _, mean)
#   gear    mpg
# 1    3 21.500
# 2    4 26.925
# 3    5 28.200
```

Vectors

Vectors are of a defined type.
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` and `FALSE` (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

- combining 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 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]    # 1st layer
# [1,]    1    2    3    4
# [2,]    5    6    7    8
# [3,]    9   10   11   12
# , , 2
#      [,1] [,2] [,3] [,4]    # 2nd layer
# [1,]    2    4    6    8
# [2,]   10   12   14   16
# [3,]   18   20   22   24
```

Multidimensional contingency tables: use `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()`.

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

Indexing vectors, matrices, and arrays

Elements of a vector can be accessed using `[i]` (see `?Extract`), where indices `i` may be numeric,

```
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
```

character, or logical vectors.

```
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%`.

Matrices and arrays have two or more indices separated by commas.

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

4 Data structures in R (2)

Lists, data frames

Lists

A list stores different types of R objects into a single data structure.

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

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

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

list1$A  # or list1[["A"]], 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
              # might be of little help for large data sets

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 data frames

Data frames have a row and a column index. 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[, 3]      # all rows, 3rd column
dat[, "id"]   # all rows, id column
```

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

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

Sorting

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

```
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
sort_by(data, ~ A + B)         # R >= 4.4.0
```

Aggregating

Frequent use cases

- average over repetitions
- calculate error sums

Syntax

```
## 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: examples

Group means

```
aggregate(weight ~ group, PlantGrowth, mean)
#   group weight
# 1  ctrl  5.032
# 2  trt1  4.661
# 3  trt2  5.526
```

Cell counts

```
aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, esoph, sum)
#        alcgp    tobgp ncases ncontrols
# 1  0-39g/day 0-9g/day      9       252
# 2      40-79 0-9g/day     34       145
# 3     80-119 0-9g/day     19        42
# 4       120+ 0-9g/day     16         8
# 5  0-39g/day    10-19     10        74
# 6      40-79    10-19     17        68
# ...
```

Reshaping: from long to wide

Frequently, data are in either long (one row per observation) or wide (one row per case) format. `reshape()` switches between both formats.

```
head(Indometh)
#   Subject time conc
# 1       1 0.25 1.50  # long data format:
# 2       1 0.50 0.94  #  repeated measurements in separate rows
# 3       1 0.75 0.78
# 4       1 1.00 0.48
# ...
df.w <- reshape(Indometh, direction = "wide", v.names = "conc",
                timevar = "time", sep = "_", idvar = "Subject")
#    Subject conc_0.25 conc_0.5 conc_0.75 conc_1
# 1        1      1.50     0.94      0.78   0.48  # wide data format:
# 12       2      2.03     1.63      0.71   0.70  #   repeated measurements
# 23       3      2.72     1.49      1.16   0.80  #   in separate columns of
# ...                                             #   the same row
```

Reshaping: from wide to long

```
## Alternative 1: varying and v.names
df.l <- reshape(df.w, direction = "long", varying = 2:12, v.names = "conc")

## Alternative 2: varying and sep
df.l <- reshape(df.w, direction = "long", varying = 2:12, sep = "_",
                idvar = "Subject") |>
  sort_by(~ Subject)                    # optionally, sort by 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
# ...
```

More reshaping examples are in `vignette("reshape")`.

5 Data input and output
- Reading data
- Data entry
- Writing data

Reading tabular text files

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 character vectors (or factors, for `R < 4.0.0`).

Note the use of ordinary (forward) slashes / in the file path, also in Windows.

Frequently used options

- `header`: variable names in the first line?
- `sep`: which delimiter between variables (white space, tabulator `\t`, comma, etc.)?
- `dec`: decimal point or comma?

Reading arbitrary text files

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

```
file.path(Sys.getenv("R_HOME"), "COPYING") |> readLines()
# [1] "\t\t    GNU GENERAL PUBLIC LICENSE"
# [2] "\t\t       Version 2, June 1991"
# [3] ""
# [4] " Copyright (C) 1989, 1991 Free Software Foundation, Inc."
# [5] "          51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA"
# [6] " Everyone is permitted to copy and distribute verbatim copies"
# ...
```

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()
read.csv()
read.csv2()
read.delim()
read.delim2()
```

- Reading SPSS data files

```
dat <- foreign::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()            # expects numeric input
y <- scan(what = "")   # expects character input
```

A simple spread-sheet editor is invoked by

```
dat <- edit(data.frame())
```

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

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
- Writing functions

R programming

Learning to program in a new language means making errors. Some errors are unexpected both to beginners and to those who have programmed before in a different language.

Help with R programming

- An Introduction to R manual (Chap. 9 and 10)
- Advanced R: `http://adv-r.had.co.nz/Functionals.html`
- Stack Overflow: `https://stackoverflow.com/tags/r`

R programming traps

- The R Inferno: `https://www.burns-stat.com/pages/Tutor/R_inferno.pdf`
- Frustration: One Year With R: `https://github.com/ReeceGoding/Frustration-One-Year-With-R`

Conditional execution: distinction of cases

Conditional execution of code is available in R via

```
if(expr1) {          # if expr1 is TRUE
  expr2              # then evaluate expr2
} else {             # else (if expr1 is FALSE)
  expr3              # then evaluate expr3
}
```

where `expr1` must evaluate to a single logical value.

More than two conditions can be implemented via `else if`

```
if(expr1) {
  expr2
} else if(expr3) {
  expr4
} else {
  expr5
}
```

Conditional execution

Example

```
x <- 3
if(!is.na(x)) {
  y <- x^2              # if x is not NA, calculate x^2
} else {
  stop("x is missing")  # else stop with an error message
}
```

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

```
## Check condition for each element
x <- 1:3
ifelse(x > 1, "larger than 1", "equal to 1")
[1] "equal to 1" "larger than 1"  "larger than 1"
```

Loops: repeating commands

The simplest way to repeatedly execute a command in R is by

```
replicate(how-often, command)
```

For more fine control, there are `for()` and `while()` loops.

Bad (but illustrative) example

```
x <- rnorm(10)
y <- numeric(10)          # allocate space for results
for(i in seq_along(y)) {  # for i = 1, 2, ..., 10
  y[i] <- x[i] - mean(x)  # replace element with result
}
```

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

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

Avoiding loops: functionals

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 for each row; rowMeans(X)
```

How would you solve the above tasks using a loop?

Avoiding loops

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

`lapply()` (`l` because the return value is a list) applies a function element-wise to other objects, for example, data frames, lists, or vectors. The resulting list has as many elements as the original object.

`sapply()` (`s` for simplify) works like `lapply()` but it tries to simplify its return value to a vector or a matrix.

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

lapply(iris, class)   # return class as a list
```

Avoiding loops

Group-wise calculations: `tapply()`

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

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

with(Oats, tapply(yield, list(Block, 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 are defined interactively using `function()`.

Example: a handmade t-test function (for illustration only)

```
twosam <- function(y1, y2) {                # definition
  n1  <- length(y1); n2 <- length(y2)       # body
  m1  <- mean(y1);   m2 <- mean(y2)
  s1  <- var(y1);    s2 <- var(y2)
  s   <- ((n1 - 1)*s1 + (n2 - 1)*s2)/(n1 + n2 - 2)
  tst <- (m1 - m2)/sqrt(s*(1/n1 + 1/n2))
  tst                # return value, can also be a list
}
```

Calling the function

```
tstat <- twosam(y1 = beaver1$temp, y2 = beaver2$temp)
tstat  # t = -15.94
```

Named arguments

If there is a function `f` defined by

```
f <- function(x, data, graph, limit) { ... }
```

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

```
f(x, df, TRUE, 20)
f(x, df, graph = TRUE, limit = 20)
f(x = x, limit = 20, graph = TRUE, data = 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 altogether from the call.

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

It could be called as

```
ans <- f(x, df)
```

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

```
ans <- f(x, 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.

For example, many classes have a tailored print method.

```
methods(print)
# [1] print.abbrev*       print.acf* print.anova print.Anova*
# [5] print.anova.loglm* print.aov* print.lm*    ...
```

Conversely, a class may have several associated methods.

```
methods(class = "lm")
#  [1] add1      alias      anova          case.names
#  [5] coerce    confint    cooks.distance deviance
#  [9] dfbeta    dfbetas    drop1          dummy.coef
# [13] effects   extractAIC family         formula
# [17] hatvalues influence  print          ...
```

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")  # display formatted output
  cat("Value of test statistic: ", obj$tst, "\n")
  invisible(obj)  # return the object
}
```

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, `ls()` will display what the function "sees" at each step. Hitting the return key will execute the next command of the function, `Q` will quit the function browser.

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

```
undebug(twosam)
```

Functions for string processing

```
substr(x, start, stop)          # extract substring
grep(pattern, x)                # match pattern
gsub(pattern, replacement, x)   # replace pattern
```

`pattern` is a regular expression (`?regex`), for example

```
foo         # match pattern foo
.           # match arbitrary character
[a-z0-9]    # match alphanumeric character
*, +, ?     # match previous character zero or more times,
            #   one or more times, at most once
^, $        # match start or end of a string
```

Example: Match string that starts with a or b and replace it by its starting letter.

```
gsub("(^[ab]).*", "\\1", c("aaa", "bcdef", "cba"))
[1] "a"   "b"   "cba"
```

Interactive regular expressions tutorial: `https://regexone.com/`

7 Graphics
- Traditional graphics
- Trellis graphics

R graphics systems

There are two graphics systems in R

- Traditional ("base") graphics
- Grid graphics

The former is easier to control, the latter provides more flexibility. The lattice and ggplot2 packages are based on the grid graphics system.

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

See Murrell (2011) for comprehensive documentation, and `https://www.stat.auckland.ac.nz/~paul/RG2e/` for examples.

See the "Graphical procedures" chapter in the Introduction to R manual for an introduction to the traditional graphics system.

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

```
pdf()            # vector graphics
svg()
postscript()
win.metafile()   # Windows only

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

Exporting graphics to files

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

A pdf graphic is produced by

```
pdf("myplot.pdf", height = 3.375, width = 3.375,  # unit: inch
    pointsize = 10)
## Put your plotting commands here ##
dev.off()
```

A Windows compatible wmf graphic is obtained by

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

Traditional graphics functions

High-level functions

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

See `demo(graphics)` for some examples of these functions.

Traditional graphics functions

Low-level functions

```
points()    # add points
lines()     # add lines
rect()
polygon()
abline()    # add line with intercept a, slope b
arrows()
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
```

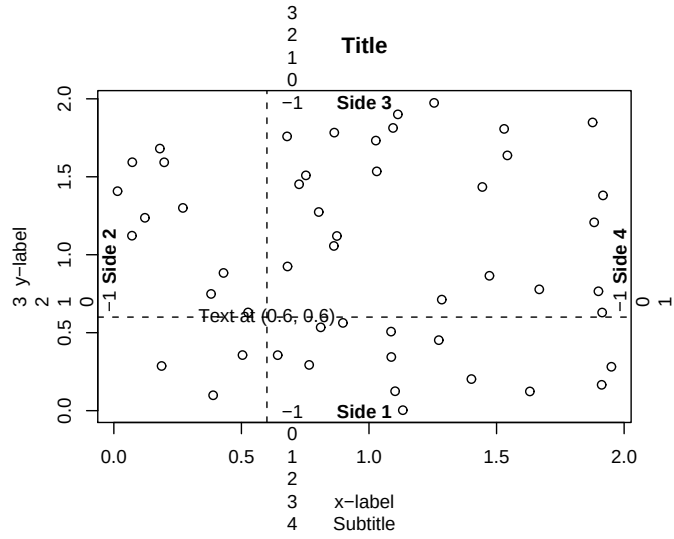
Writing text into plotting region

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

Margins region

Writing text into margins region

```
for(side in 1:4) {
  mtext(-1:4,
        side = side,
        line = -1:4,
        at = .7)
}
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: build it step by step.

Example: interaction plot

```
## Mean reaction times in a two-factorial experiment
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
")
dat$A <- factor(dat$A)
dat$B <- factor(dat$B)
```

Stepwise plotting

First, only produce the plotting region and the coordinate system.

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

Next, plot the data points separately for each level of factor *A*.

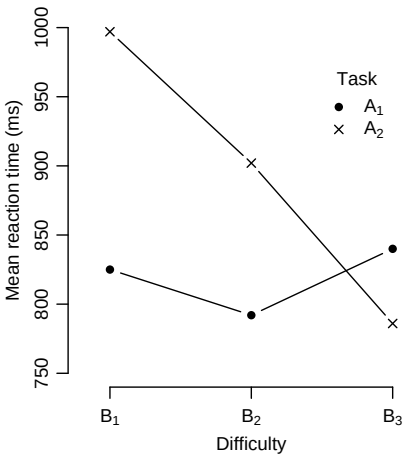
```
4  points(rt ~ as.numeric(B), dat[dat$A == "a1", ], type = "b", pch = 16)
5  points(rt ~ as.numeric(B), dat[dat$A == "a2", ], type = "b", pch = 4)
```

Then, add axes and a legend.

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

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

Stepwise plotting



Graphics tips

- 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 white space 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()` before the high-level plotting function is called.

```
mai     # size of figure margins (inches)
mar     # size of figure margins (lines of text)
mfrow   # number of sub-figures on a page:
        #   par(mfrow = c(1, 2)) creates two sub-figures in a row
omi     # size of outer margins (inches)
oma     # size of outer margins (lines of text)
pty     # aspect ratio of plot region (square, maximal)
```

Example

```
par(mai = c(0.7, 0.7, 0.1, 0.1))  # set figure margins (bottom, le, top, ri)
```

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

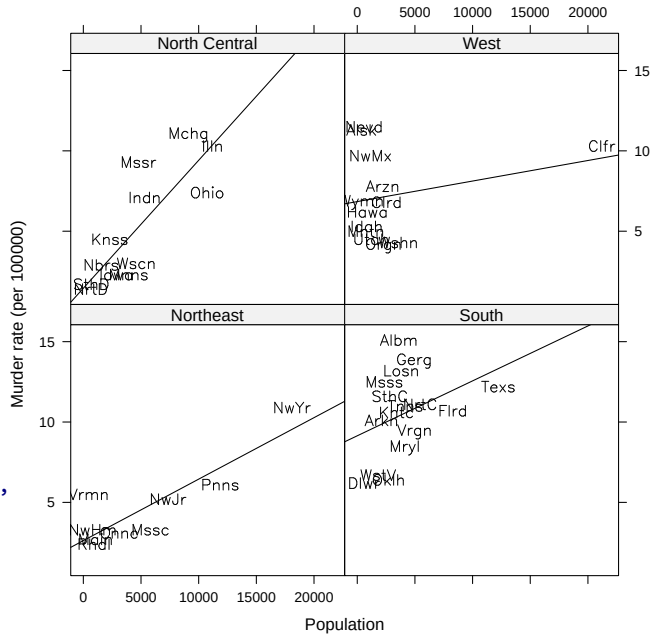
The lattice package is loaded into R by

```
library(lattice)
```

Lattice graphics: example

Murder rate in the USA

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

## Conditional scatter plot
xyplot(
  Murder ~ Population | region,
  dat
)
```



Lattice graphics functions

High-level functions

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

High level functions in lattice take an extended formula argument to indicate the conditioning variable.

```
xyplot(y ~ x | z1 + z2 + ..., data)
xyplot(Murder ~ Population | region, dat)
```

Panel functions and strip functions

Panel functions control what is drawn in each panel. A panel function may be passed via the panel argument.

```
panel.xyplot()    # scatter plot
panel.abline()    # draw (regression) lines
panel.grid()      # draw grid lines
lpoints()         # draw points
llines()          # draw lines
ltext()           # 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.

Fine control using panel and strip functions

```
xyplot(Murder ~ Population | region, dat,
  groups = name,
  xlim = c(-5000, 25000),
  ylab = "Murder rate (per 100000)",

  # panel function
  panel = function(x, y, subscripts, groups) {
    panel.grid(-1, -1)                                  # align with ticks
    panel.abline(lm(y ~ x))                             # regression line
    ltext(x, y, labels = groups[subscripts], cex = 1)   # add state names
  },

  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 Programming in Python and PsychoPy
- Python
- PsychoPy

What is Python?

- Programming language created by Guido van Rossum, developed by the Python Software Foundation

  Current version: Python 3
- Many add-on packages or modules, one of them is PsychoPy

  PsychoPy (>= `1.90`) supports Python 3.
- Download and documentation: `https://www.python.org/`
  – Tutorial
  – FAQs
  – HOWTOs

Working with Python

Interactive mode: working from the Python console

- Entering `python` into the shell starts the Python console.
- Within the Python console, commands and functions can be executed; `quit()` closes the console.
- Documentation of Python functions can be displayed via `help()`

```
help(input)
```

Script-based mode: working from the shell

- Write Python script in your text editor and save it (e. g., `favSubject.py`).
- Execute entire script from the shell

```
python favSubject.py
```

Example script: What is your favorite subject?

```
#!/usr/bin/env python

print('What is your favorite subject?\n\n' +
      '(1) Statistics\n(2) Programming in R\n' +
      '(3) Social Psychology\n')

resp = input('> ')  # record keyboard input

if resp[0] == '1':
    print('Statistics?\n\nGet a life.')
elif resp[0] == '2':
    print('Programming?\n\nOMG, you\'re such a nerd.')
elif resp[0] == '3':
    print('Social Psychology?\n\nOK ... weird.')
else:
    print('Think you\'re being extra smart, huh?\n\n' +
          'Next time, press 1, 2, or 3.')
```

What is PsychoPy?

- Python module for controlling psychological experiments
- Advantages: free and open source, platform independent, intuitive, flexible
- Created and maintained by Jon Peirce (2007, 2009)
- Download and documentation: `https://www.psychopy.org/`
  - Reference manual (API)
  - Resources (e. g., for teaching)
  - Mailing list
- Usage in Python scripts

```
import psychopy                    # import entire module
from psychopy import core, visual  # import submodules
```

PsychoPy submodules

PsychoPy organizes its functions into several submodules

visual	creating visual stimuli, like windows, rectangles, circles, dots, text, ...
hardware	keyboard, camera, ...
core	basic functions, quit
gui	dialogue boxes
sound	creating sound stimuli
...	

Call a function from submodule

```
visual.Window()  # open window
```

Example script: Hello world in PsychoPy

```
1  #!/usr/bin/env python
2
3  from psychopy import core, visual          # import PsychoPy submodules
4  from psychopy.hardware import keyboard
5
6  win = visual.Window()                      # open window
7  kb = keyboard.Keyboard()                   # create keyboard object
8
9  ## Create text object
10 ts = visual.TextStim(win, text='Hello World!', color=(-1.0, -1.0, 1.0))
11
12 ts.draw()                                  # draw text on back buffer
13 win.flip()                                 # flip buffers, i.e. show text
14
15 # core.wait(5)                             # wait for 5 s
16 kb.waitKeys()                             # wait for key press
17 core.quit()
```

Experiments with PsychoPy

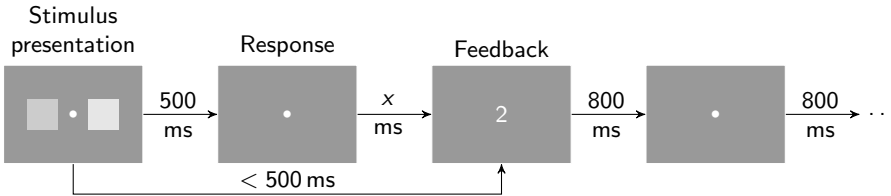
Example: lightness discrimination, method of constant stimuli

Organizing the experiment: recommended folder structure

```
lightness/                # top level: name of experiment
  data/                   # data files
    subj01.txt
    ...
  session/                # session files
    createSession.R
    ses01.txt
    ...
  runexp.py               # Python code for running the experiment
```

Main Python script (`runexp.py`) contains the trial function, defines the stimuli, opens a data file, and executes the session file.

Trial function (1)




```
1  ## 2AFC procedure + feedback
2  def trial(col_left, col_right, isi=0.8, presentation_duration=0.5):
3      rect_left.color  = col_left    # set new colors
4      rect_right.color = col_right
5      fixpoint.draw()
6      rect_left.draw()
7      rect_right.draw()
8      win.flip()                     # display stimuli
```

Trial function (2)

```
9  ## Collect response
10 kb.clock.reset()
11 response = kb.waitKeys(maxWait=presentation_duration,
12                        keyList=['left', 'right', 'q'])
13
14 ## Clear screen when presentation_duration has elapsed
15 if response is None:
16     fixpoint.draw()
17     win.flip()
18     response = kb.waitKeys(keyList=['left', 'right', 'q'])
19 if response[0] in ['q']:
20     core.quit()
```

Trial function (3)

```
21 ## Save data
22 datfile.write('%6.2f %6.2f %5s %6.3f\n' %
23               (col_left, col_right,
24                'l' if response[0].name == 'left' else 'r',
25                response[0].rt))
26
27 ## Feedback
28 if col_left > col_right:
29     fdbk.text = '1'
30 elif col_left < col_right:
31     fdbk.text = '2'
32 else:
33     fdbk.text = '='
34 fdbk.draw()
35 win.flip()
36 core.wait(isi)
```

Session files

- Control the sequence of trials within an experimental session.
- Contain calls to the trial function in order of presentation.

```
trial(0.4, 0.3)  # 1st trial
trial(0.4, 0.6)  # 2nd trial
trial(0.4, 0.4)  # 3rd trial
...
```

- Can be created in R (`createSession.R`).
- Advantages
  - Experimental design (randomization, balancing) can be checked before the experiment is run.
  - No need for manipulating the trial sequence in main Python script.
- Execute session file from main Python script (`runexp.py`)

```
with open('session/ses02.txt', 'r') as sesfile:
    exec(sesfile.read())
```

9 Classical hypothesis tests
- Nominal variables
- Metric response variables
- Ordinal-continuous 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 (categorical) variables

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

Metric response variables

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

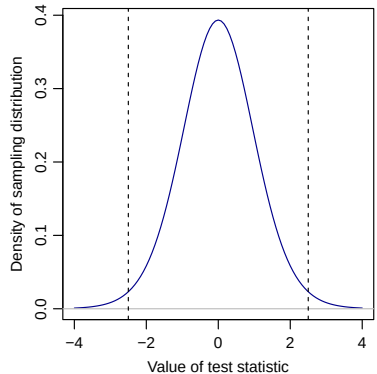
Ordinal-continuous response variables

- `wilcox.test()`: 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 (and its underlying model) 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

```
## Testing Mendel's law of independent assortment
y <- c(AB = 315, aB = 101, Ab = 108, ab = 38)
chisq.test(y, p = c(9, 3, 3, 1)/16)
```

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), $i = 1, \ldots, n$, where $X_i \in \{1, \ldots, m\}$ 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 $Y_1, \ldots, Y_m \sim N(\mu_y, \sigma_y^2)$; 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(len ~ supp, ToothGrowth, 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 $Y_1, \ldots, Y_m \sim N(\mu_y, \sigma_y^2)$, unknown σ_x^2, σ_y^2
- Hypotheses
  1. H_0: $\sigma_x^2 = \sigma_y^2$ H_1: $\sigma_x^2 \neq \sigma_y^2$
  2. H_0: $\sigma_x^2 \geq \sigma_y^2$ H_1: $\sigma_x^2 < \sigma_y^2$
  3. H_0: $\sigma_x^2 \leq \sigma_y^2$ 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(len ~ supp, ToothGrowth)
```

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(len ~ supp, ToothGrowth)
```

10 Generalized linear models
Formulae
Linear models
Generalized linear models
Maximum likelihood

Formulae

Statistical models are specified in a compact symbolic form that resembles the mathematical notation.

Formulae used in `lm()` and `glm()`, let x be continuous and 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

anova()     # Compare nested models and produce an analysis
            #   of variance table

resid()     # Extract the (matrix of) residuals

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

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

Extractor functions for lm objects

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

predict()   # A new data frame must 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 data frame

step()      # Select a suitable model by adding or dropping
            #   terms and preserving hierarchies. The model
            #   with the smallest value of AIC (Akaike's
            #   Information Criterion) discovered in the
            #   stepwise search is returned
```

Coding of factors: contrasts

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

Dummy coding (default in R)

- 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 globally 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 set globally by

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

or locally by

```
lm(len ~ supp, ToothGrowth, contrasts = list(supp = "contr.sum"))
```

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. Automatically available are:

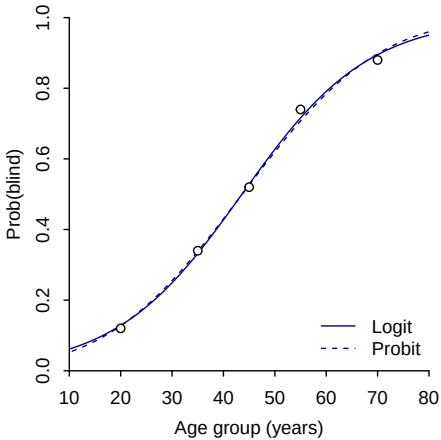
```
## Family name       Link functions
   binomial          logit, probit, log, cloglog
   gaussian          identity, log, inverse
   Gamma             identity, inverse, log
   inverse.gaussian  1/mu^2, identity, inverse, log
   poisson           log, identity, sqrt

   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 (from R Intro manual): 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),            # age group
                  n = rep(50, 5),                       # number tested
                  y = c(6, 17, 26, 37, 44))             # number blind
m1 <- glm(cbind(y, n - y) ~ x, binomial, dat)           # logit
m2 <- glm(cbind(y, n - y) ~ x, binomial(probit), dat)   # probit
```

Parameter estimates are obtained by

```
summary(m1)
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept)  -3.5378     0.5023   -7.04  1.9e-12
# x             0.0811     0.0108    7.50  6.5e-14
```

For the logit model, the parameter estimates may be interpreted as the log odds ratio: The odds of becoming 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, goodness of fit may be assessed using a likelihood ratio test.

```
## Compare to saturated model
m3 <- glm(cbind(y, n - y) ~ factor(x), binomial, dat)
anova(m1, m3, test = "LRT")  # 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     0.3171
# 2         0     0.0000  3   0.3171    0.957
```

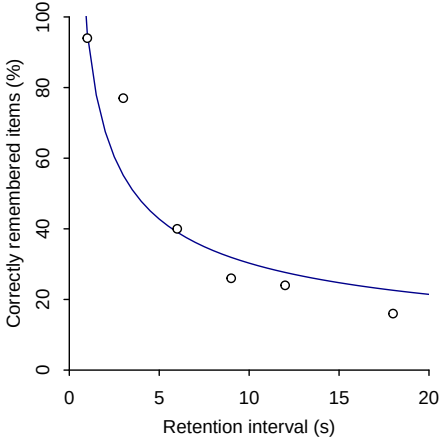
Predictions based on new observations (see `?predict.glm`)

```
newx <- 0:100
predict(m1, data.frame(x = newx), type = "response")
```

Maximum likelihood

Maximum likelihood estimation may be carried out directly using minimization functions: `nlm()`, `optim()`, `constrOptim()`, ...

Example in Myung (2003): modeling forgetting data



Power model

$$p(w,t) = w_1 t^{-w_2} \text{ for } w_1, w_2 > 0$$

Likelihood function

$$f(y_i|n,w) = p(w,t)^{y_i}(1-p(w,t))^{n-y_i}$$

Maximum likelihood

The `nlm()` function searches for the minimum of a multidimensional function using a Newton algorithm.

```
dat <- data.frame(y = c(94, 77, 40, 26, 24, 16),  # correct response
                  t = c(1, 3, 6, 9, 12, 18))      # retention interval

powmod <- function(w, y, n, t) {    # negative log-likelihood function
  p <- w[1] * t^-w[2]
  -sum( y*log(p) + (n - y)*log(1 - p) )
}

mle1 <- nlm(f = powmod,      # function to be minimized
            p = c(.5, .5),   # starting values
            y = dat$y,       # arguments passed to f
            n = 100,
            t = dat$t,
      hessian = TRUE)        # return negative hessian matrix
```

Maximum likelihood

Output of the `nlm()` function

```
mle1
# $minimum
# [1] 313.4             # value of neg. log-likelihood
# $estimate
# [1] 0.953 0.498       # ML estimate of w
# $hessian
#         [,1]   [,2]   # neg. hessian matrix -H
# [1,] 3150.3 -509.5
# [2,] -509.5 1036.9
```

Approximate covariance matrix $Cov((\hat{w}_1, \hat{w}_2)) = -H^{-1}$

```
solve(mle1$hessian)
#           [,1]     [,2]
# [1,] 0.000345 0.000169
# [2,] 0.000169 0.001048
```

11 Simulation based procedures
Random numbers generation
Simulated power
Resampling
Parametric bootstrap
Randomization test

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, lambda = 1)   # draw from Poisson distribution
```

Sampling with or without replacement from a vector

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

Simulated power is the proportion of significant tests for a specified effect size, variability, sample size, and α.

Pseudo code

```
Set sample size to n
replicate
{
  Draw sample from model with minimum relevant effect
  Test null hypothesis
}
Determine proportion of significant results
```

For planning a new data collection, adjust *n* until desired power (0.8, 0.95) is reached.

Example: power of the one-sample t-test

An experiment is conducted to determine the point of subjective equality to a 440-Hz standard tone.

On each of n trials, the participant adjusts the frequency of a comparison tone until the two tones sound equal in pitch.

What is the power of a one-sample t-test to detect a 3 Hz deviation from 440 Hz with $\sigma = 8$ Hz, $n = 30$, and $\alpha = 5\%$?

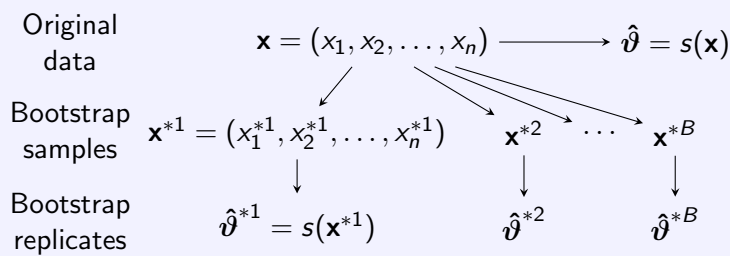
```
n <- 30                                      # adjust if power too low
pval <- replicate(2000, {
  x <- rnorm(n, mean = 440 + 3, sd = 8)      # draw from effect model
  t.test(x, mu = 440)$p.value                # test against H0
})
mean(pval < 0.05)                            # simulated power
```

```
[1] 0.503
```

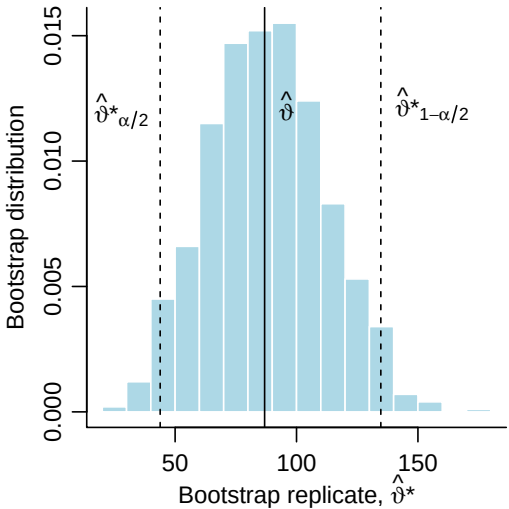
Bootstrap by resampling

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



Percentile intervals

The quantiles of the bootstrap distribution define the limits of $(1 - \alpha)$ confidence intervals 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
)
mean(mouse$surv[mouse$grp == "trt"])  # mean survival time
[1] 86.86

## Resampling
s.b <- replicate(1000, {      # 1000 bootstrap replicates
  x <- sample(mouse$surv[mouse$grp == "trt"], replace = TRUE)
  mean(x)
})
quantile(s.b, c(.025, .975))  # 95% bootstrap CI for the mean
#   2.5%  97.5%
#  42.71 136.86
```

Bootstrap based on a parametric model

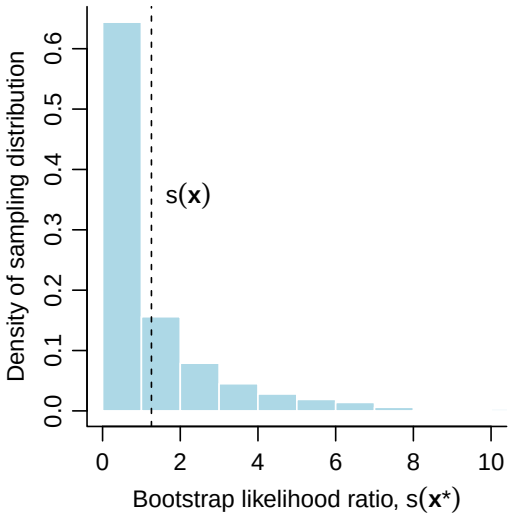
Parametric bootstrap algorithm for the likelihood ratio test:

- 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
Forig <- anova(lm0, lm1)$F[2]       # original likelihood ratio
#     F
# 1.258

## Parametric bootstrap
LR.b <- replicate(1000, {
  surv0 <- simulate(lm0)$sim_1      # simulate from null model
  m0    <- lm(surv0 ~ 1, mouse)     # fit null model
  m1    <- lm(surv0 ~ grp, mouse)   # fit alternative model
  anova(m0, m1)$F[2]                # bootstrap likelihood ratio
})
```

Example: survival time of mice revisited



Bootstrap p-value

Proportion of bootstrap replicates that is at least as extreme as the original likelihood ratio.

```
12  mean(LR.b >= Forig)
    [1] 0.298
```

Randomization test

Goal: By randomizing (permuting) the observations, approximate the sampling distribution under the null hypothesis of no effect.

What happens to the originally observed statistic if the response variable is randomly permuted?

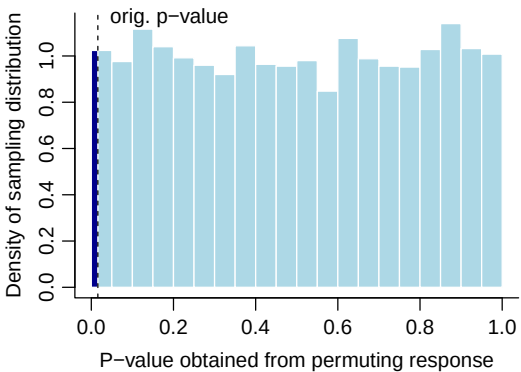
If the permuted statistics are

- similar to the original, then the null hypothesis is retained
  $\rightarrow$ not much is destroyed by permuting.
- much less extreme, then the null hypothesis is rejected
  $\rightarrow$ the original statistic is special.

Randomization test p-value

Proportion of p-values generated by permuting that is at least as small as the original p-value. (How many small p-values remain?)

Example: dried weight of plants




```
## Original p-value
p.orig <- anova(lm(weight ~ group,
           PlantGrowth))$"Pr(>F)"[1]

## Permute response variable
pval <- replicate(5000,
  anova(lm(sample(weight) ~ group,
           PlantGrowth))$"Pr(>F)"[1]
)

## Randomization test p-value
mean(pval <= p.orig)
[1] 0.0184
```

Summary

	Testing sampling distribution under H_0	**Estimation** sampling distribution of estimator
Model as RNG	**Parametric bootstrap** simulate from null model	**Parametric bootstrap** simulate from alternative model
Sample as RNG	**Randomization test** permute response	**Resampling** sample units with replacement

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.

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*(3), 299–314. https://doi.org/10.1080/10618600.1996.10474713

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

Myung, I. J. (2003). Tutorial on maximum likelihood estimation. *Journal of Mathematical Psychology*, *47*(1), 90–100. https://doi.org/10.1016/S0022-2496(02)00028-7

Peirce, J. W. (2007). PsychoPy – Psychophysics software in Python. *Journal of Neuroscience Methods*, *162*(1–2), 8–13. https://doi.org/10.1016/j.jneumeth.2006.11.017

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics*, *2*(10), 1–8. https://doi.org/10.3389/neuro.11.010.2008

R Core Team. (2025). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. (https://www.R-project.org/)

References II

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

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. (http://adv-r.had.co.nz/)