Computing in 

Amsterdam UMC Doctoral School
June 10-13, 2025

Aldo Jongejan
Michel Hof and Perry Moerland
Dept. of Epidemiology and Data Science
Amsterdam UMC, Meibergdreef 9, Amsterdam, the Netherlands

Amsterdam UMC
University Medical Centers

Part I

Day 1 and 2

Outline

Course setup

- Course aim: become familiar with the basics of R
- Four days, one morning session per day: 9:30-12:30
- Mix of interactive lectures and computer exercises
- Course website: `https://bioinformaticslaboratory.eu/gs-computing-in-r/`
- Comments and suggestions for improvement are most welcome

Stages in statistical analysis

1. Importing data into statistical program
2. Inspection of data
   - finding errors, cleaning
   - recoding and transforming
   - description and summarizing of the data

   using spreadsheets, tables and graphics
3. Analysis: estimation, uncertainty (confidence intervals, p-value), predictive value
4. Model validation
   Check the assumptions of the model
5. Reporting of results
   summary, tables, graphics
   export

Characteristics of a statistical program

1. Two ways to perform the task
   - Via the menu, graphical user interface (GUI)
   - Writing code in a script (syntax) window

   Actions performed via the menu can also be saved in a script

Characteristics of a statistical program

1. Two ways to perform the task
   - Via the menu, graphical user interface (GUI)
   - Writing code in a script (syntax) window

   Actions performed via the menu can also be saved in a script
2. At least five windows
   - Script (syntax).
     A good editor is really helpful
   - Results (output).
     Often in *structured markup language* (html, Word, ODF (open document format), LaTeX)
   - Graphics.
     Can be saved in various formats (pdf, wmf, png)
     Sometimes combined with results window (SPSS)
   - Spreadsheet. To see the complete data set.
   - Help. In program or via web browser.

R: What is it?

- On `http://www.r-project.org/about.html`: "a language and environment for statistical computing and graphics"
- Free statistical package: no money and open source
- Runs on all major operating systems

R: What is it?

- On `http://www.r-project.org/about.html`: "a language and environment for statistical computing and graphics"
- Free statistical package: no money and open source
- Runs on all major operating systems
- Standard distribution with basic statistical procedures
- Extensions via packages
  - Recommended; come installed together with R
  - Thousands more; can be installed from the R website
- Hard to learn(?)
- Very powerful language; has become very popular over the past 10-15 years

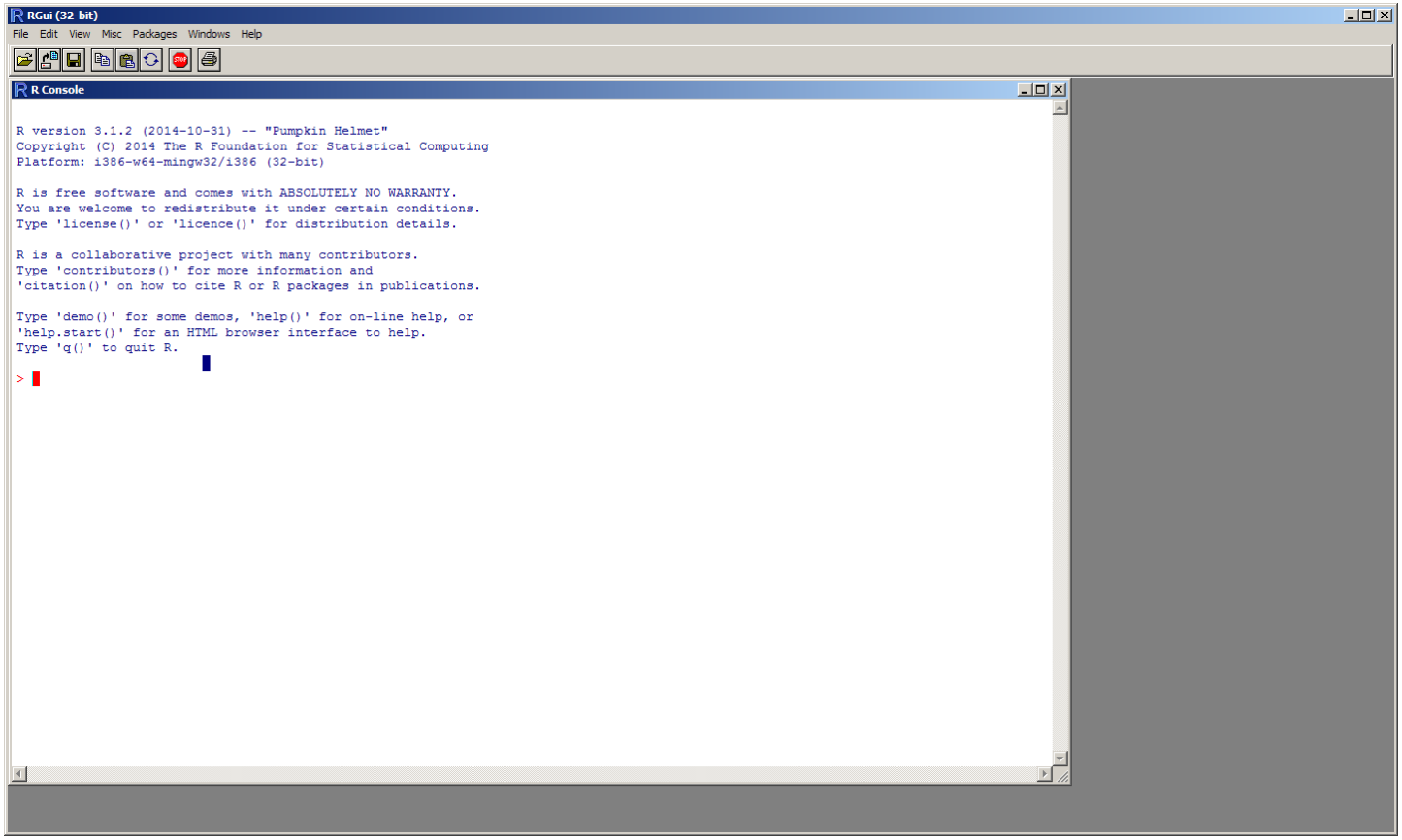
Characteristics of a statistical program: R

1. Two ways to perform the task
   - Via the menu (GUI)
     - Standard R: very few options
     - GUI: Rcmdr, jamovi and others (see links at the end of the handouts).
   - Via scripts. Saved in file with ".R" extension

Characteristics of a statistical program: R

1. Two ways to perform the task
   - Via the menu (GUI)
     - Standard R: very few options
     - GUI: Rcmdr, jamovi and others (see links at the end of the handouts).
   - Via scripts. Saved in file with ".R" extension
2. Windows in R
   - Standard R: opens with "Console"
     Can be used for simple calculations; input and output in same window

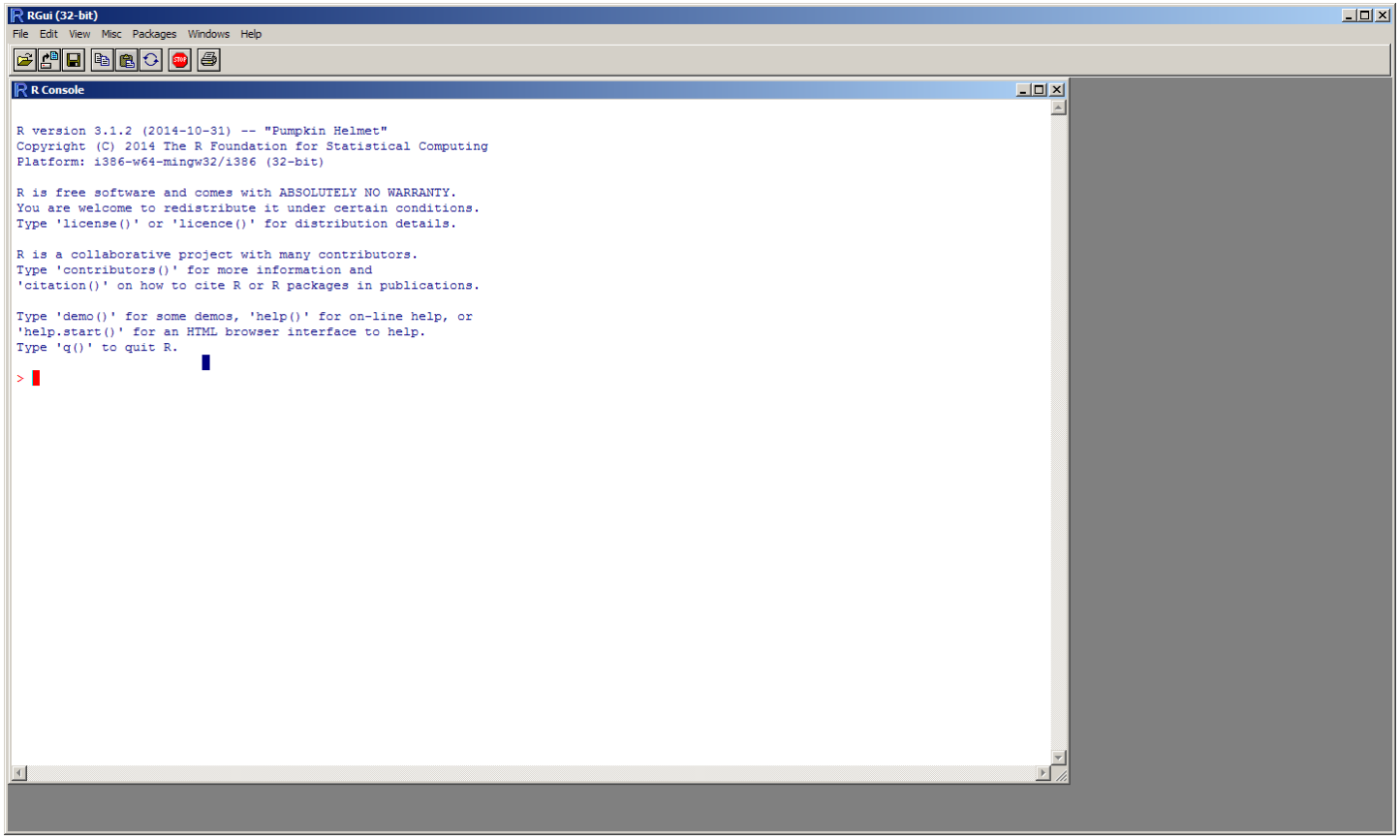
Don't be afraid of the console



Characteristics of a statistical program: R

1. Two ways to perform the task
   - Via the menu (GUI)
     - Standard R: very few options
     - GUI: Rcmdr, jamovi and others (see links at the end of the handouts).
   - Via scripts. Saved in file with ".R" extension
2. Windows in R
   - Standard R: opens with "Console"
     Can be used for simple calculations; input and output in same window
     Script window can be opened; results still in Console

Don't be afraid of the console



- Try it yourself: start R version 4.4.3 via Starten - Alle programma's - R - R 4.4.3 . . . This opens the R Console

Outline

R as a pocket calculator

- First of all, R can be used as a pocket calculator
- Many mathematical operations are pre-defined in R

```
> 2+7
[1] 9

> sqrt(2)
[1] 1.414214

> cos(pi)
[1] -1

> log10(10^3)
[1] 3
```

A simple R session

- Now we are ready to type some R code

```
> x <- 2
> x
[1] 2
```

- The left arrow <- denotes an `assignment` statement. This stores a value in object `x`, that can then be used later on.
- Remember: without assignment, it's lost

```
> x^2
[1] 4
> x
[1] 2
```

Interacting with the R Console

- Use up/down keys to go back/forth on the command history.

```
y < - x
```

Interacting with the R Console

- Use up/down keys to go back/forth on the command history.

  ```
  y < - x
  ```

  Can easily be corrected using the up key:

  ```
  y <- x
  ```

Interacting with the R Console

- Use up/down keys to go back/forth on the command history.

  ```
  y < - x
  ```

  Can easily be corrected using the up key:

  ```
  y <- x
  ```

- Use CTRL+A or HOME to go to the start of a line
- Use CTRL+E or END to go to the end of a line
- Use TAB to complete pre-defined words and filenames
- If for some reason R gets stuck try ESC (Windows) or CTRL+C (Mac, Linux)

Help (I)

If you want to know more about an operator or function just use `help` (or `?`)

```
> help(sqrt)

MathFun package:base
Description:
sqrt(x) computes the (principal) square root of x.
Usage:
sqrt(x)
```

Functions

- `help` is another example of a function
- The basic R distribution consists of a large collection of functions
- Functions generate some output given some input
- The inputs are specified via arguments of the function between parentheses `( )`:
  `name_of_function(argument_1)`
- `help(sqrt)`: `sqrt` is argument of function `help`
- The output of a function can be a value written to the Console or assigned to an object, a figure, a help page, ...

Packages

- Functions in R are in general part of a package, such as the **base** package for `sqrt`
- Only the standard packages are loaded when you start R: **base**, **graphics**, **stats**, **utils** ...
- Other packages are loaded by the `library` command
- `library()` shows the packages installed on your computer
- `help(package=stats)` gives help on all functions defined in **stats**
- Running `help.start()` launches a web browser that allows all (installed) help pages to be browsed with hyperlinks

Help (II)

```
> help(mean)

Description: Generic function for the (trimmed) arithmetic mean.

Usage: mean(x, ...)
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x: An R object. Currently there are methods for numeric/log-
ical vectors and date, date-time and time interval objects.
Complex vectors are allowed for trim = 0, only ...

Value
If trim is zero (the default), the arithmetic mean of the val-
ues in x is computed, as a numeric or complex vector ...
```

Help (III)

Outline of a help page is always the same:

- Description: what does the function do
- Usage: what arguments does the function expect
- Arguments: description of the individual arguments
- Value: what is the result of a function call
- Details, references, See Also
- Example: `example(mean)`

Outline

Vectors (I)

A vector is one of the basic data structures in R:

```
> x <- c(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
> x
 [1] 10  9  8  7  6  5  4  3  2  1
```

Vectors (I)

A vector is one of the basic data structures in R:

```
> x <- c(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
> x
 [1] 10  9  8  7  6  5  4  3  2  1
```

These commands also give a vector of the numbers 10 to 1:

```
> x <- seq(from = 10, to = 1, by = -1)
> x <- seq(10, 1)
> x <- 10:1
```

`c` (short for concatenate) and `seq` are functions as well

Vectors (II)

- Vectors can be indexed using square brackets `[ ]`:

```
> x[5] + x[10]
[1] 7
```

- Negative indices exclude elements from a vector:

```
> c(-5, -10)
[1] -5 -10

> x[c(-5, -10)]
[1] 10  9  8  7  5  4  3  2
```

- Indices can be used to replace an element of a vector

```
> x[4] <- 12
> x
[1] 10  9  8 12  6  5  4  3  2  1
```

Vectors (III)

- Functions can be applied to vectors:

```
> mean(x)
[1] 6
```

- Many calculations are vectorized:

```
> x + 1
[1] 11 10  9 13  7  6  5  4  3  2

> 2*x
[1] 20 18 16 24 12 10  8  6  4  2
```

Matrices (I)

- From one to two dimensions:

```
> help(matrix)
matrix                  package:base
...
Usage:
     matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
     dimnames = NULL)
```

- Note: arguments to a function can be supplied by name or by position

Matrices (II)

- Matrices store data in a table-like structure, with rows and columns:

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

- Indexing is simple (elements):

```
> A[2, 3]
[1] 6
```

- Indices can be used to replace an element of a matrix

```
A[2,3] <- 12
```

Matrices (III)

- Selecting entire row(s)

```
> A[1, ] # Same as A[1,1:5]
[1] 1 3 5 7 9
```

- Selecting entire column(s)

```
> A[, c(1, 5)] # Same as A[1:2, c(1,5)]
     [,1] [,2]
[1,]    1    9
[2,]    2   10
```

- Functions can be applied to matrices:

```
> dim(A[, c(1, 5)])
[1] 2 2
```

- The generalization to any number of dimensions is an `array`

Objects (I)

- Scalars, vectors, matrices are examples of objects. You can get an overview of all objects you created until now via `ls` (short for list)

```
> ls()
[1] "A" "x"
```

- Many R functions are defined on any type of data. Examples are:

```
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    6.00    8.75   12.00
```

- Try `summary(A)`

Object names

- An object can have almost any name you choose: `patients`, `Data`, `abc`, `sorted.results_file`
- No space
- No special characters such as `@`,`$`,`+` etc.
- `_` and `.` are allowed
- Numbers allowed but not as first character
- Avoid names that are functions in R: `sort`, `c`, `mean`, `t`, `data`, `q`
- Some names are not allowed (reserved for programming constructs): `for`, `if`, `while` ...
- Names are case-sensitive: `Data` is not the same as `data`

Modes

- R has several atomic modes, the most important ones are:
  - *numeric*:

```
> c(1, 2, 3, 4)
```

  - *logical*: Boolean values: `TRUE`, `FALSE`

```
> -2 < 2
[1]  TRUE
```

  - *character*:

```
> letters[1:3]
[1] "a" "b" "c"
```

  - You can change the mode of an object

```
> as.character(x)
 [1] "10" "9"  "8"  "12" "6"  "5"  "4"  "3"  "2"
[10] "1"
```

- Modes can be mixed in lists, we'll come back to that later

Modes: logical (I)

- Booleans (`TRUE`, `FALSE`) can also be used as an index:

```
> x
[1] 10  9  8 12  6  5  4  3  2  1

> x[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE,
    FALSE, TRUE, FALSE)]
[1] 10  8  6  4  2
```

- Making Booleans by comparing numbers:
  Less/greater: <, >, <=, >=
  Exact equality: ==
  Not equal to: !=

```
> x[x>5]
[1] 10  9  8 12  6
```

- `%in%`: to test which values are part of a set of specified values
- Booleans are converted to integers if a numeric value is required: `TRUE` equals 1, `FALSE` equals 0

Modes: logical (II)

You can calculate with Booleans. Main operators are:

- &: AND - all must be true
- |: OR - at least one must be true
- !: NOT - negation

```
> TRUE & FALSE
 [1] FALSE
> TRUE | FALSE
 [1] TRUE
> x>5 & x<8
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[9]  FALSE FALSE
```

Naming (I)

- A useful concept in R is access by names:

```
> m <- c(1,2,3,4)
> names(m) <- c("gene1","gene2","gene3","gene4")
> m
gene1  gene2  gene3  gene4
    1      2      3      4
```

- We can also give names to rows and columns of matrix `A`:

```
> rownames(A) <- c("gene1", "gene2")
> colnames(A) <- c("sample1", "sample2", "sample3",
     "sample4", "sample5")
```

Naming (II)

- We can now index by name instead of by number or Boolean:

```
> A
      sample1 sample2  sample3 sample4 sample5
gene1       1       3        5       7       9
gene2       2       4       12       8      10

> A["gene1", ]
sample1 sample2 sample3 sample4 sample5
      1       3       5       7       9
```

- Indexing by name rather than by number makes code more readable: `Data["BRCA1",]` instead of `Data[4137,]`

RStudio

- Open **RStudio** via Starten - Alle programma's - R - RStudio
- A so-called integrated development environment (IDE)
- Editor, Console, Environment, History, Plots, etc in one environment
- Download the script file `CourseMain.R` from `https://bioinformaticslaboratory.eu/gs-computing-in-r/` to execute the R code used during the lecture

Lists (I)

- Something is needed for mixing different modes, for example character and numeric:

```
> c("gene1", 5)
[1] "gene1" "5"
```

- This can be done by lists:

```
> list(gene = "gene1", number = 5)
$gene
[1] "gene1"

$number
[1] 5
```

- `gene` and `number` are called components

Lists (II)

- Lists can be indexed in various ways:
  - As vectors, with square brackets. This returns a list:

```
> x <-  list(gene = "gene1", number = 5)
> x[1]
$gene
[1] "gene1"
```

  - With double square brackets. This extracts a component:

```
> x[[1]]
[1] "gene1"
```

  - Or equivalently, by name using the $ operator (if the list is named):

```
> x$gene
[1] "gene1"
```

Data frames (I)

- A special kind of list is a matrix with mixed modes, *e.g.*, rows correspond to individuals and columns to variables of different modes.
- All elements within a column should be of the same mode
- In R, this is dealt with by a `data.frame`
- External data (of the tab-delimited type, for example) imported via `read.table` is of class `data.frame`:

```
read.table                package:base
Description:
     Reads a file in table format and creates a data frame
     from it, with cases corresponding to lines and
     variables to fields in the file.
```

Constructing a data frame

```
> pclass <- c("1st","2nd","1st")
> survived <- c(1,1,0)
> name <- c("Elisabeth Walton","Hudson Trevor","Helen Loraine")
> age <- c(29.0,0.9167,2.0)
> titanic <- data.frame(pclass,survived,name,age)
> titanic
  pclass survived              name     age
1    1st        1 Elisabeth Walton 29.0000
2    2nd        1    Hudson Trevor  0.9167
3    1st        0    Helen Loraine  2.0000
```

Data frames (II)

- Data frames can be indexed like a matrix

```
> titanic[c(2,3),c("name","age")]
            name    age
2 Hudson Trevor 0.9167
3 Helen Loraine 2.0000
```

- Columns of a data frame can be indexed like a list, with `$` and `[[ ]]`

```
titanic$age  # titanic[["age"]] gives the same result
[1] 29.0000  0.9167  2.0000
```

- `$` and `[[ ]]` do not work for rows, use `subset` instead (see later)

Data frames (III)

For large data frames, several useful functions exist to get a more compact overview

- `dim` gives the number of rows and columns
- `head` shows the first six rows of a data frame

```
> dim(titanic3)
[1] 1309   17
> head(titanic3[,1:4])
  pclass survived                                 name    sex
1    1st        1   Allen, Miss. Elisabeth Walton female
2    1st        1  Allison, Master. Hudson Trevor   male
3    1st        0    Allison, Miss. Helen Loraine female
4    1st        0 Allison, Mr. Hudson Joshua Crei   male
5    1st        0 Allison, Mrs. Hudson J C (Bessi female
6    1st        1             Anderson, Mr. Harry   male
```

Data frames (IV)

- `tail`: similar to `head` but shows the last 6 rows
- `str`: compact display of the internal structure of an R object

```
> str(titanic3[,1:4])
'data.frame':   1309 obs. of  4 variables:
 $ pclass   : Factor w/ 3 levels "1st","2nd","3rd": 1 1 1 1 1 1 1 1 1 1 ...
 $ survived: num  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : chr  "Allen, Miss. Elisabeth Walton" "Allison, Master. Hud-
son Trevor" "Allison, Miss. Helen Loraine" "Allison, Mr. Hudson Joshua Cre
 $ sex      : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 2 ...
```

- `summary`
- `View`: opens a spreadsheet-style data viewer. In RStudio click on the name of an object in the Environment tab.
- `fix`: opens a spreadsheet-style data editor

Recapitulation: objects

You have seen the most important data objects in R:

- *vectors*
- *matrices* are a two-dimensional extension of vectors
- *lists* are a general form of vectors in which the various elements need not be of the same mode
- *data frames* are matrix-like structures, in which the columns can be of different modes
- Indexing of these objects can be done by number, by name, and using Booleans.

The return of the help file

```
> ?mean

Description: Generic function for the (trimmed) arithmetic mean.

Usage: mean(x, ...)
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x: An R object. Currently there are methods for numeric/log-
ical vectors and date, date-time and time interval objects. Com-
plex vectors are  allowed for trim = 0, only.

Value
If trim is zero (the default), the arithmetic mean of the val-
ues in x is computed, as a numeric or complex vector ...
```

Data import and export: text format

- Data frames in ASCII text format (of the tab-delimited type, for example) can be imported via `read.table`:
- Many arguments (see `help(read.table)`)

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
dec = ".",row.names, col.names, as.is = !stringsAsFactors,
na.strings = "NA", colClasses = NA, nrows = -1, skip = 0,
check.names = TRUE, fill = !blank.lines.skip, ...)
```

- `read.csv` and `read.delim` are identical to `read.table` apart from other defaults: they are intended for comma-separated and tab-delimited files, respectively.
- Export to ASCII file: `write.table`

Data import of ASCII text format: common problems

- Common problems when reading in tabular data are (especially when you use "Save as - tab-delimited file" from Excel):
  - Additional tabs: between columns or at the end of a row
  - Extra carriage returns at the end of the file
  - Unusual characters such as the # symbol (see option `comment.char`) and " quotes (see option `quote`)
  - Presence of blank fields
  - Regional settings problems: decimal separator
  - Invisible spaces
- Use `dim`, `head` etc to compare the imported data with the original data file
- Be careful when using Excel as an intermediate in manipulating files:
  `https://www.bbc.com/news/technology-54423988`

Basic data import/export from other formats

- Data formats: sav (SPSS), xls, xlsx (Excel), mdb (Access), dta (STATA), txt, csv
- sav, xls, dta, txt, csv: Imported via a function "`read.`". E.g. a STATA file `titanic3.dta` can be imported via the commands

```
> library(foreign)
> titanic3 <- read.dta("Exercises/titanic3.dta")
```

- xlsx files: packages **openxlsx** and **readxl** (also xls files)
- SPSS, Stata, and SAS files: package **haven**
- In RStudio via the menu **Import Dataset**. See `https://support.posit.co/hc/en-us/articles/218611977-Importing-Data-with-RStudio`
- Export to other formats via a function "`write.`" : `write.dta`, `write.foreign`
- See R Data Import/Export Manual under **Help or Help - R Help (RStudio)**
- See `http://r4stats.com/examples/data-import/`

Outline

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt, mean, help, library`
  - Functions for analysis: `t.test`, `lm`, `plot`

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt`, `mean`, `help`, `library`
  - Functions for analysis: `t.test`, `lm`, `plot`
- Input: *required* and *optional* arguments; within parentheses (`sqrt(2)`, `help(seq)`), separated by comma
  - required: need to be supplied
  - optional: have default values

  Beware of sequence of arguments; required ones come first
  e.g. `log(x, base = exp(1))`, `x` required, `base` optional.
  Argument names can be abbreviated if no risk of ambiguity

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt`, `mean`, `help`, `library`
  - Functions for analysis: `t.test`, `lm`, `plot`
- Input: *required* and *optional* arguments; within parentheses (`sqrt(2)`, `help(seq)`), separated by comma
  - required: need to be supplied
  - optional: have default values

  Beware of sequence of arguments; required ones come first e.g. `log(x, base = exp(1))`, `x` required, `base` optional. Argument names can be abbreviated if no risk of ambiguity
- Special "argument" `...`: anything that makes sense, e.g. in `c` and `paste` function

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt`, `mean`, `help`, `library`
  - Functions for analysis: `t.test`, `lm`, `plot`
- Input: *required* and *optional* arguments; within parentheses (`sqrt(2)`, `help(seq)`), separated by comma
  - required: need to be supplied
  - optional: have default values

  Beware of sequence of arguments; required ones come first
  e.g. `log(x, base = exp(1))`, `x` required, `base` optional.
  Argument names can be abbreviated if no risk of ambiguity
- Special "argument" `...`: anything that makes sense, e.g. in `c` and `paste` function
- Output: result of calculations (typically assigned to R object), graphics, help window, ...

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt`, `mean`, `help`, `library`
  - Functions for analysis: `t.test`, `lm`, `plot`
- Input: *required* and *optional* arguments; within parentheses (`sqrt(2)`, `help(seq)`), separated by comma
  - required: need to be supplied
  - optional: have default values

  Beware of sequence of arguments; required ones come first
  e.g. `log(x, base = exp(1))`, `x` required, `base` optional.
  Argument names can be abbreviated if no risk of ambiguity
- Special "argument" `...`: anything that makes sense, e.g. in `c` and `paste` function
- Output: result of calculations (typically assigned to R object), graphics, help window, ...
- You can use functions within other functions, e.g. `mean(c(3,6,8))`

Functions: the inside

- Function code can be seen by leaving out the parentheses `( )`
- General structure: `function(args) SOME R CODE`
  with `SOME R CODE` a collection of other functions as compound expression

Functions: the inside

- Function code can be seen by leaving out the parentheses `( )`
- General structure: `function(args) SOME R CODE`
  with `SOME R CODE` a collection of other functions as compound expression
- Compound expressions are placed within "{ " and " }":

```
> z <- {
       x <- 2
       y <- x + 2
 }
> z
[1] 4
```

- A compound expression returns the last value

Functions and packages

- You can write your own functions:

```
> good.morning <- function(work){
    if(work==TRUE) cat("wake up") else
      cat("you can stay in bed")
}
```

Note: here the function is saved in the object `good.morning`

Functions and packages

- You can write your own functions:

```
> good.morning <- function(work){
    if(work==TRUE) cat("wake up") else
      cat("you can stay in bed")
}
```

  Note: here the function is saved in the object `good.morning`

- Can make it into a *package*, i.e. a collection of functions (and data):
  survival, ggplot2, Rcmdr
  sudoku, scuba, engsoccerdata
  See `http://cran.r-project.org/web/packages/`

Functions and packages

- You can write your own functions:

```
> good.morning <- function(work){
    if(work==TRUE) cat("wake up") else
      cat("you can stay in bed")
}
```

  Note: here the function is saved in the object `good.morning`

- Can make it into a *package*, i.e. a collection of functions (and data):
  survival, ggplot2, Rcmdr
  sudoku, scuba, engsoccerdata
  See `http://cran.r-project.org/web/packages/`
- R Reference Card 2.0 for overview of most important functions

Selection of rows and columns

- Index: `[ ]` (vector) or `[row, col]` (data frame)
  - By character: `titanic3[,"sex"]`, `titanic3[,c("age","sex")]`, `islands["Moluccas"]`
  - By number: `titanic3[,4]`, `titanic3[-1,]`
  - By logical: `titanic3[titanic3[,"sex"] != "male",]`

Selection of rows and columns

- Index: `[ ]` (vector) or `[row, col]` (data frame)
  - By character: `titanic3[,"sex"]`, `titanic3[,c("age","sex")]`, `islands["Moluccas"]`
  - By number: `titanic3[,4]`, `titanic3[-1,]`
  - By logical: `titanic3[titanic3[,"sex"] != "male",]`
- Columns in data frame can also be selected via `$`, e.g. `titanic3$sex`

Selection of rows and columns

- Index: `[ ]` (vector) or `[row, col]` (data frame)
  - By character: `titanic3[,"sex"]`, `titanic3[,c("age","sex")]`, `islands["Moluccas"]`
  - By number: `titanic3[,4]`, `titanic3[-1,]`
  - By logical: `titanic3[titanic3[,"sex"] != "male",]`
- Columns in data frame can also be selected via `$`, e.g. `titanic3$sex`
- We can assign values to selections or new columns

```
> titanic3[3,"age"] <- 23.4
> my.data$bmi <- my.data$weight/(my.data$height)^2
```

Selection of rows via functions

- Via special functions: `head`, `tail`, `subset`
  `subset(my.data, ...)` with `...` a logical condition

```
> subset(titanic3, pclass %in% c("1st","2nd"))
```

  (remember that `%in%`—"belongs to"—is a Boolean construct)

- Many functions have a *subset* argument
  Often combined with formula structure

```
> xtabs(~survived, data=titanic3, subset=(sex=="male"))
```

Selection of columns via functions

- Via `with` function:

```
> table(titanic3$sex, titanic3$survived)
> with(titanic3, table(sex, survived))
```

- Many functions have a *data* argument, combined with formula structure

```
> xtabs(~sex+survived, data=titanic3)
```

- Via *select* argument of `subset` function

Selection of columns via functions

- Via `with` function:

```
> table(titanic3$sex, titanic3$survived)
> with(titanic3, table(sex, survived))
```

- Many functions have a *data* argument, combined with formula structure

```
> xtabs(~sex+survived, data=titanic3)
```

- Via *select* argument of `subset` function
- Don't use "$" for column selection if function has a data argument
  Don't write:

```
> xtabs(~titanic3$sex+titanic3$survived, data=titanic3)
```

Missing data

- Special value: NA (short for "not available")
- The function `is.na` checks for missingness

Missing data

- Special value: NA (short for "not available")
- The function `is.na` checks for missingness

```
> table(is.na(titanic3$age))
FALSE  TRUE
 1046   263
```

- Within functions, missings are often excluded by default, but not always
  - `quantile`, `mean` give error if there are missings; specify argument `na.rm=TRUE`
  - `table` excludes missings, include them via argument `useNA="always"`

Factors: what are they?

- Categorical variable with "levels"

```
> DiseaseState <- factor(c("Cancer", "Cancer", "Normal"))
> DiseaseState
[1] Cancer Cancer Normal
Levels: Cancer Normal
> levels(DiseaseState)
[1] "Cancer" "Normal"
```

- Ordering: default is alphabetical/numeric
- Internally represented as integers $1, 2, \ldots$

```
> as.numeric(DiseaseState)
[1] 1 1 2
```

Factors: how to create?

- By default, character columns are converted into factor if data are read from other statistical programs. Numeric codings (e.g. 999) are not converted by default.
- Create or manipulate via `factor` function
  - Required argument `x`: vector with values
  - Optional argument `levels`: vector of unique values in `x`; sequence determines ordering. Compare

```
> table(factor(DiseaseState))
> table(factor(DiseaseState, levels=c("Normal","Cancer")))
```

  - Optional argument `labels`: labels given to levels. Default: same as levels

Factors: how to create?

- By default, character columns are converted into factor if data are read from other statistical programs. Numeric codings (e.g. 999) are not converted by default.
- Create or manipulate via `factor` function
  - Required argument `x`: vector with values
  - Optional argument `levels`: vector of unique values in `x`; sequence determines ordering. Compare

```
> table(factor(DiseaseState))
> table(factor(DiseaseState, levels=c("Normal","Cancer")))
```

  - Optional argument `labels`: labels given to levels. Default: same as levels
- Useful in statistical models.

  Standard in R: first group is reference group.
  Choice of reference group changed via `relevel`:

```
> relevel(DiseaseState, "Normal")
```

Dates

- Numeric value (units since time origin) with character representation
- Origin: SPSS: October 14, 1582 (seconds);
  R: January 1st, 1970 (days);
  STATA: January 1st, 1960 (days)
- SPSS files read into R via `read.spss` in **foreign** package need to be converted

```
> my.data$date <- as.Date(my.data$date+ISOdate(1582,10,14) )
```

  The **haven** package makes the conversion automatically
- R is very flexible in conversion between textual date representations
- `as.Date`: create date variable
  `format`: change display format

Part II

Day 3 and 4

Outline

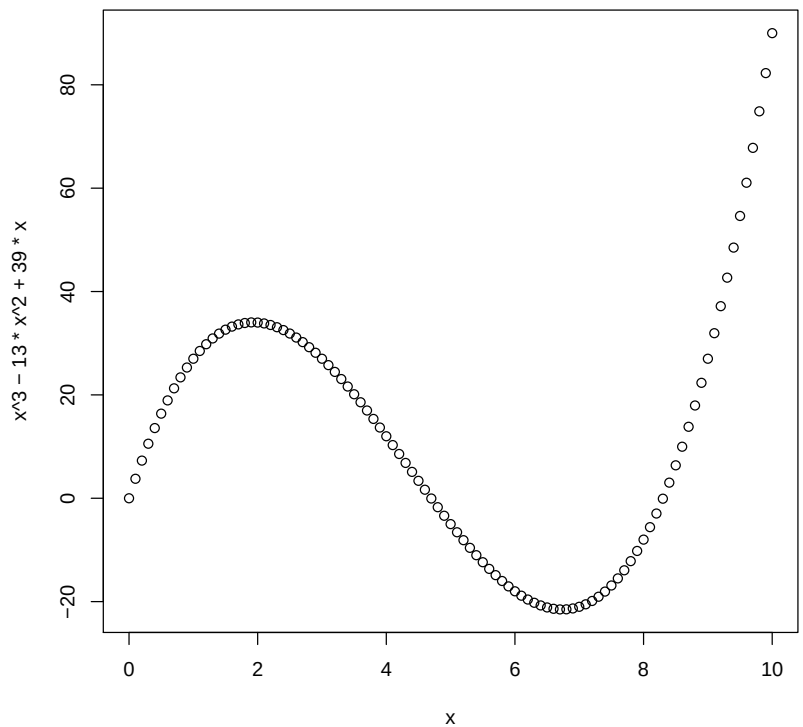
R has versatile tools for graphics. There are typically three steps to producing useful graphics:

1. Creating the basic plot
2. Enhancing the plot with labels, legends, colors etc.
3. Exporting the plot from R for use elsewhere

Basic plot (I)

It is straightforward to make a simple plot using functions from the **graphics** package (loaded by default):

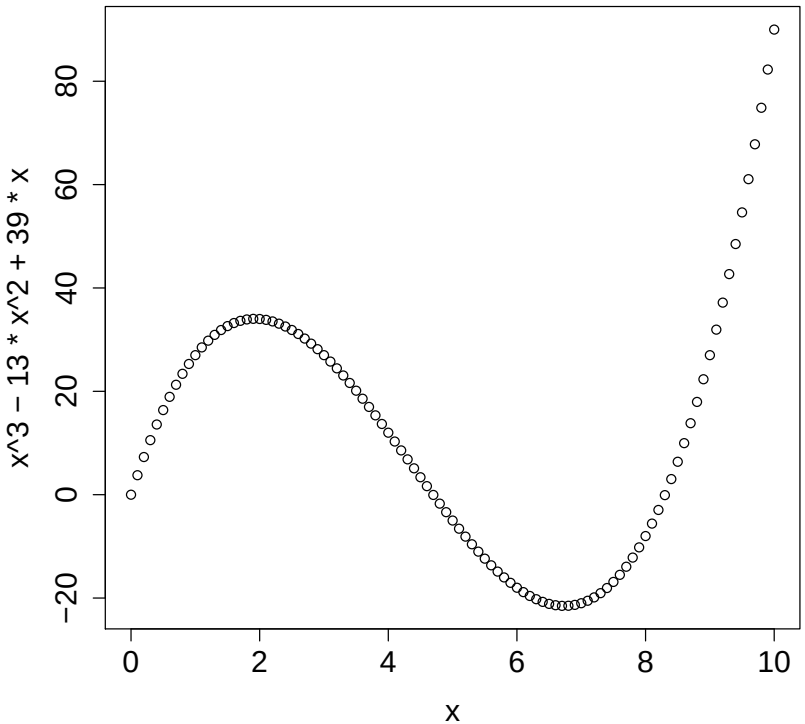
```
> x <- (0:100)/10
> plot(x, x^3 - 13 * x^2 + 39 * x)
```



Basic plot (II)

You can increase the size of the symbols on the axes and the axis labels (cex stands for character expansion factor):

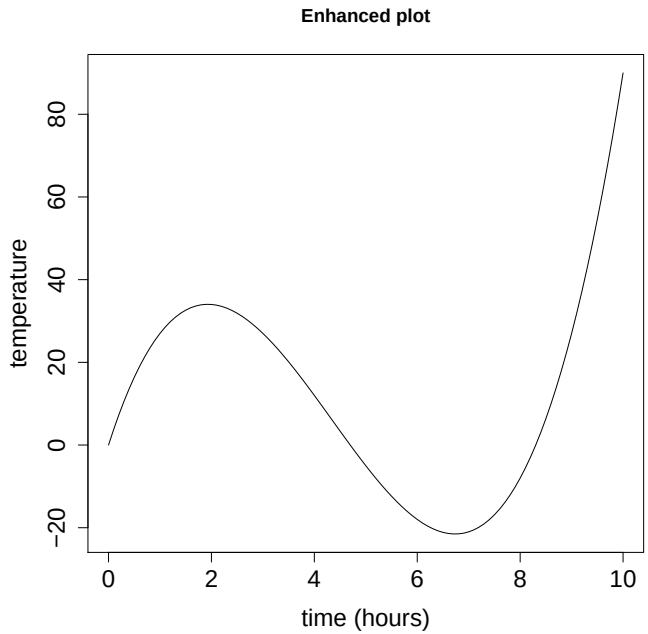
```
> plot(x, x^3 - 13 * x^2 + 39 * x,cex.axis=1.5,cex.lab=1.5)
```



Enhancing a plot (I)

- Change the type of plot via the argument `type`: `"p"` for **p**oints (is default), `"l"` for **l**ines, etc. See `?plot` for other options
- Change the titles for the axes via `xlab` and `ylab`
- Add an overall title for the plot via `main`

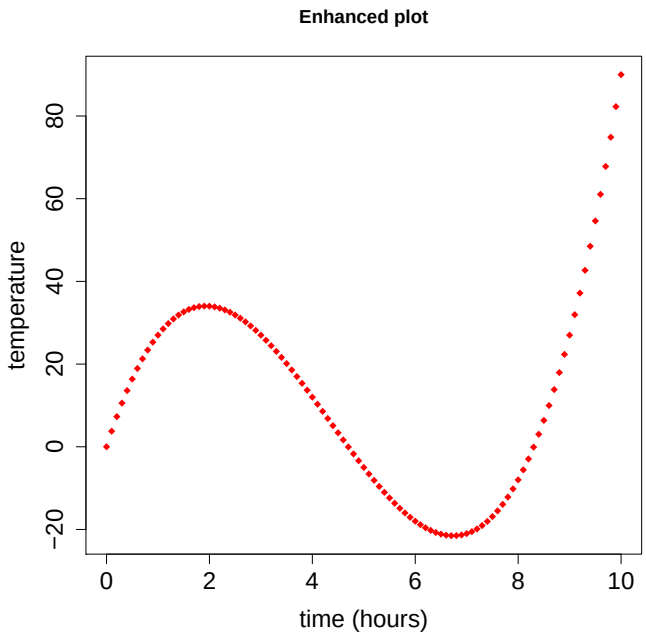
```
> plot(x,x^3-13*x^2+39*x,type="l",xlab="time (hours)",
  ylab="temperature",main="Enhanced plot",cex.axis=1.5,cex.lab=1.5)
```



Enhancing a plot (II)

- Change the plot symbol used from the default o via the argument `pch`
- Change the colour via the argument `col`. By name: see `colors()` for the 657 options. By number: see `palette()`

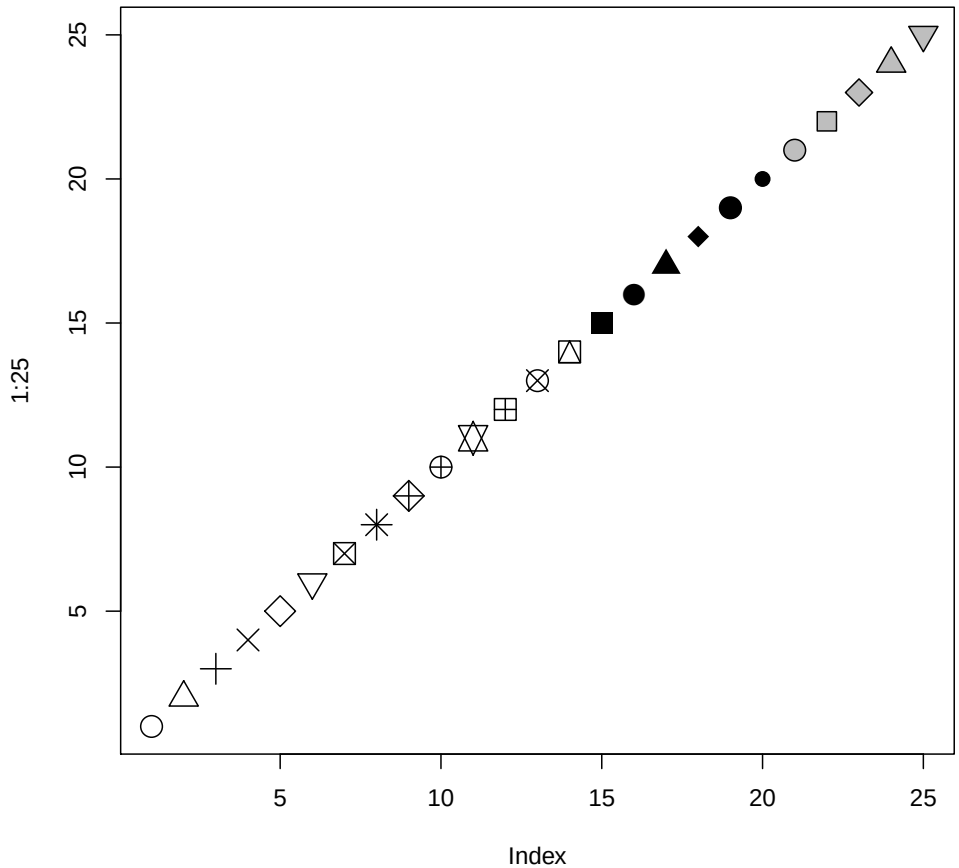
```
> plot(x,x^3-13*x^2+39*x,pch=18,xlab="time (hours)",
  ylab="temperature",col="red",main="Enhanced plot",
  cex.axis=1.5,cex.lab=1.5)
```



Plot symbols

There are 25 different plot symbols, see `?points`

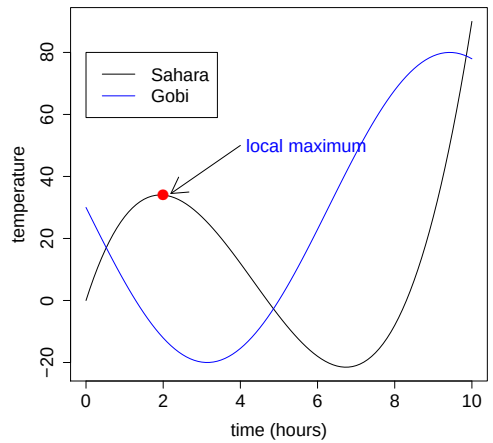
```
> plot(1:25, pch=1:25,cex=2,bg="grey")
# bg: background colors for open plot symbols
```



Enhancing a plot (III)

You can add points, arrows, text, lines, and a legend to an existing plot:

```
> x<-(0:100)/10
> plot(x,x^3-13*x^2+39*x,type="l",xlab=
  "time (hours)",ylab="temperature",cex.axis=1.5,cex.lab=1.5)
> points(2,34,col="red",pch=16,cex=2)
> arrows(4,50,2.2,34.5)
> text(4.15,50,"local maximum",adj=0,col="blue",cex=1.5)
> lines(x,30-50*sin(x/2),col="blue")
> legend(x=0,y=80,legend=c("Sahara","Gobi"),col=c("black","blue"),
  cex=1.5)
```

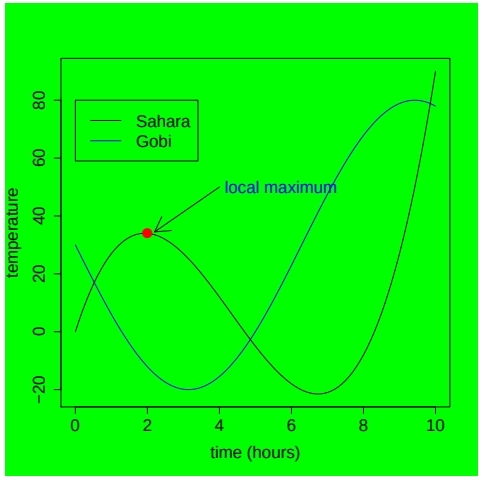


Graphical parameters (I)

You can change the default value of many graphical parameters via `par` (see `?par`). For example to reset the background of a plot to green:

```
> par(bg="green")
```

and then rerun the plot commands



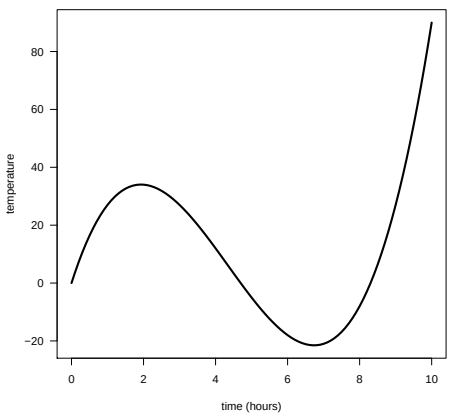
You can set a parameter back to its default value (white) by `par(bg="white")`

Graphical parameters (II)

Other often used options:

- `lwd` sets the line width
- `mfrow` and `mfcol` enable multiple plots in one figure
- `las` to rotate axis symbols
- `mar` to change the default margins of the figure

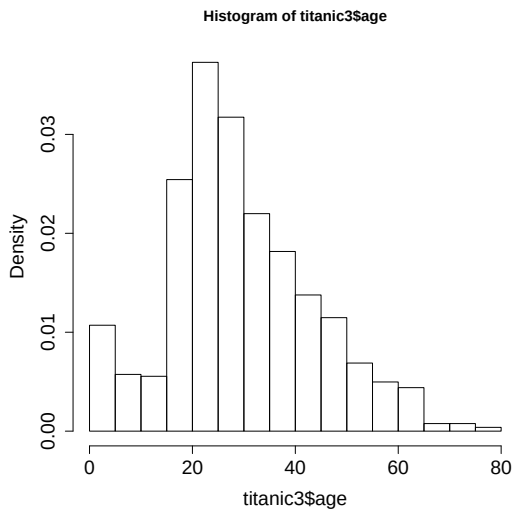
```
> x<-(0:100)/10
> plot(x,x^3-13*x^2+39*x,type="l",
  xlab= "time (hours)",ylab="temperature",
  lwd=3,las=1,cex.axis=1.5,cex.lab=1.5)
```



Histograms

Use `hist` for plotting histograms. As always, see `?hist` for the many arguments of this function

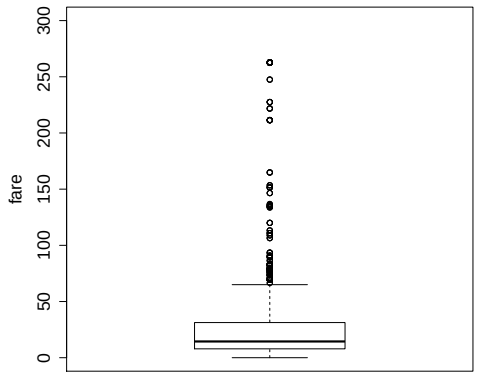
```
> hist(titanic3$age,breaks=15,freq=FALSE,
  cex.axis=1.5,cex.lab=1.5)
```



Boxplot

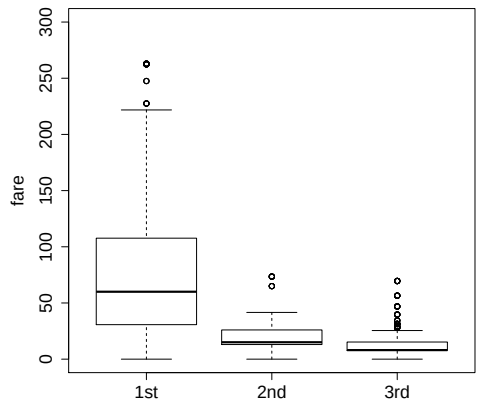
The function `boxplot` can be used on a vector

```
> boxplot(titanic3$fare,
ylim=c(0,300),ylab="fare",
cex.axis=1.5,cex.lab=1.5)
```



`boxplot` also has a formula interface

```
> boxplot(fare ~ pclass,
data=titanic3,ylim=c(0,300),ylab="fare",
cex.axis=1.5,cex.lab=1.5)
```



Advanced R graphics

- Ch 12 of "An Introduction to R" gives an introduction to base graphics
- **lattice**: very powerful for multipanel conditioning
  needs to be loaded first; `xyplot` is the main function
- **ggplot2**: based on "the grammar of graphics"

Advanced R graphics

- Ch 12 of "An Introduction to R" gives an introduction to base graphics
- **lattice**: very powerful for multipanel conditioning
  needs to be loaded first; `xyplot` is the main function
- **ggplot2**: based on "the grammar of graphics"
- **ggvis**, **plotly**, **rCharts**, **Shiny**: interactive visualizations
- and in many more packages (**gplots**, **plotrix**, . . .)

Export: two types of formats

- Vector format (pdf, eps, wmf, emf)
  - digital image consisting of independent geometric objects (segments, polygons, curves, etc.)
  - can be enlarged without losing resolution
- Raster (png, jpeg, tiff).
  - rectangular grid of pixels, possibly with color
  - Resolution impaired if image is enlarged
- Graphics can be saved via the menu in the graphics/plots window, or a specific graphics file type can be created directly (`pdf(...)`, `win.metafile(...)`, `png(...)` and ending with `dev.off()`)

Outline

Structure of R

- Objects: data, functions (statistical procedures), model output

Structure of R

- Objects: data, functions (statistical procedures), model output
- Environment: a collection of objects that is accessible in R session
  - Objects we create: in "Workspace" (RStudio: in Global Environment window)
  - Packages with existing functions: **base**, **stats**, **graphics**
  - When a package is loaded, a new environment is created
  - Some more environments, e.g. some tools in RStudio

Structure of R

- Objects: data, functions (statistical procedures), model output
- Environment: a collection of objects that is accessible in R session
  - Objects we create: in "Workspace" (RStudio: in Global Environment window)
  - Packages with existing functions: **base**, **stats**, **graphics**
  - When a package is loaded, a new environment is created
  - Some more environments, e.g. some tools in RStudio
- `search()` shows the environments in the search path
  `ls()` or `objects()` shows the objects in an environment

Structure of R

- Objects: data, functions (statistical procedures), model output
- Environment: a collection of objects that is accessible in R session
  - Objects we create: in "Workspace" (RStudio: in Global Environment window)
  - Packages with existing functions: **base**, **stats**, **graphics**
  - When a package is loaded, a new environment is created
  - Some more environments, e.g. some tools in RStudio
- `search()` shows the environments in the search path
  `ls()` or `objects()` shows the objects in an environment
- Hierarchical structure of environments; needed for dealing with duplicate names

R resembles operating system

R	OS
objects	files
Workspace	current folder
environments	folders in “path” variable
RStudio “Environment” window	Explorer window

Workspace management; connection with OS

- Save complete Workspace on disk
  - R: **File** → **Save Workspace** (or the `save.image` function)
  - RStudio: Floppy disk icon in the Global Environment window
  - Asked when you close the R session (e.g. via command: `q()`)
- Save specific objects: via `save` function
- Binary format file with extension: ".RData"

Workspace management; connection with OS

- Save complete Workspace on disk
  - R: **File** → **Save Workspace** (or the `save.image` function)
  - RStudio: Floppy disk icon in the Global Environment window
  - Asked when you close the R session (e.g. via command: `q()`)
- Save specific objects: via `save` function
- Binary format file with extension: ".RData"
- `load` can import R workspace or collection of R objects
- Delete objects from workspace within R via `rm` function

```
> rm(titanic3)
```

Remove all objects from workspace:

```
> rm(list=ls())
```

Project management

- Every project (analysis) in separate folder (*working directory*)
- Users can have several working directories with separate .RData files and script files

Project management

- Every project (analysis) in separate folder (*working directory*)
- Users can have several working directories with separate .RData files and script files
- R best started via double clicking on script file with ".R" extension. Working directory is that same folder
- Otherwise, use commands `getwd` and `setwd` or the GUI to get and set the working directory
  Note: R uses / or \\ instead of \ in path specification

Project management

- Every project (analysis) in separate folder (*working directory*)
- Users can have several working directories with separate .RData files and script files
- R best started via double clicking on script file with ".R" extension. Working directory is that same folder
- Otherwise, use commands `getwd` and `setwd` or the GUI to get and set the working directory
  Note: R uses / or \\ instead of \ in path specification
- RStudio has an elegant Project concept
  `https://support.posit.co/hc/en-us/articles/200526207-Using-Projects`

Export tables to other formats

- Copy and paste

Export tables to other formats

- Copy and paste
- Use `write.table`
- Function `write.xlsx` in package **xlsx** for Excel
- HTML output: packages **kableExtra**, **xtable**, **R2HTML** and **PrettyR**
- Many options for LaTeX users, e.g. **Hmisc, xtable**

Reproducible research

- See Task View at `http://cran.r-project.org/web/views/ReproducibleResearch.html`
- Most elegant approach: both R code and explanatory text in same file
- Compilation: run R code, and keep the surrounding text
- Recommended: use Markdown format in Rstudio Compilation via **knitr** package

Reproducible research

- See Task View at `http://cran.r-project.org/web/views/ReproducibleResearch.html`
- Most elegant approach: both R code and explanatory text in same file
- Compilation: run R code, and keep the surrounding text
- Recommended: use Markdown format in Rstudio Compilation via **knitr** package
- `https://statmodeling.stat.columbia.edu/2014/09/19/never-happened-r-markdown//`

Outline

Some functions for data management

- Sorting. Base R: `sort` and `order`. **dplyr**: `arrange`.
  Rstudio: sorting in spreadsheet window (not saved in object)

Some functions for data management

- Sorting. Base R: `sort` and `order`. **dplyr**: `arrange`.
  Rstudio: sorting in spreadsheet window (not saved in object)
- Merging. R: `merge`. **dplyr**: `left_join` (3 other options, see Data Transformation Cheat Sheet).

Some functions for data management

- Sorting. Base R: `sort` and `order`. **dplyr**: `arrange`.
  Rstudio: sorting in spreadsheet window (not saved in object)
- Merging. R: `merge`. **dplyr**: `left_join` (3 other options, see Data Transformation Cheat Sheet).
- Long to wide. R: `reshape`. **tidyr**: `pivot_wider`
  Wide to long. Base R: `reshape`. **tidyr**: `pivot_longer`

Creating transformed variables

- Arithmetic functions: `log` etc.
- `cut` to split continuous variable into groups
- Note: transformations not needed for model fitting

Creating transformed variables

- Arithmetic functions: `log` etc.
- `cut` to split continuous variable into groups
- Note: transformations not needed for model fitting
- Adding variables
  - Base R: via $
    Functions `within` and `transform` may be helpful
  - **dplyr**: `mutate`

Making basic summaries

- Data summary: `summary`

Making basic summaries

- Data summary: `summary`
- Contingency tables: `table`, `xtabs`
  `CrossTable` in **descr** package

Making basic summaries

- Data summary: `summary`
- Contingency tables: `table`, `xtabs`
  `CrossTable` in **descr** package
- Summary by subgroups
  - Base R: `aggregate`, `tapply`
  - Several functions in packages **doBy**, **Hmisc**, **compareGroups**, **dplyr**

Making basic summaries

- Data summary: `summary`
- Contingency tables: `table`, `xtabs`
  `CrossTable` in **descr** package
- Summary by subgroups
  - Base R: `aggregate`, `tapply`
  - Several functions in packages **doBy**, **Hmisc**, **compareGroups**, **dplyr**
- Graphical summary of data frames: `dfSummary` in package **summarytools**

Outline

Finding Information

- Function `help`. R is object oriented!

Finding Information

- Function `help`. R is object oriented!
- Function `help.search`
- Function `RSiteSearch`
  Opens web browser with all keyword specific info on functions from CRAN
- Package **sos**
- Manuals in R

Finding Information

- Function `help`. R is object oriented!
- Function `help.search`
- Function `RSiteSearch`
  Opens web browser with all keyword specific info on functions from CRAN
- Package **sos**
- Manuals in R
- CRAN (Task Views, Vignettes, list with packages)
- `http://stackoverflow.com/questions/tagged/r`
- And of course ChatGPT (or similar modern AI-based chatbots)
- Have a look at the links provided at the end of the handout or at `https://bioinformaticslaboratory.eu/gs-computing-in-r/`

Outline

Regression; Formulas

The regression equation is represented as a *formula*

General form `dependent` $\sim$ `independent`

Dependent Depends on type of model, check help file of modeling function

Independent Variable names separated by operators, without explicit reference to parameters

`fare` $\sim$ `age` $+$ `pclass` $+$ `sex` three main effects

Regression; Formulas

The regression equation is represented as a *formula*

General form `dependent` $\sim$ `independent`

Dependent Depends on type of model, check help file of modeling function

Independent Variable names separated by operators, without explicit reference to parameters

$$\texttt{fare} \sim \text{age} + \texttt{pclass} + \texttt{sex} \qquad \text{three main effects}$$

- interactions are denoted by ":"
  interaction and main effects by "$*$"

$$\text{age} * \texttt{sex} = \text{age} + \texttt{sex} + \text{age} : \texttt{sex}$$

Regression; Formulas

The regression equation is represented as a *formula*

General form `dependent` $\sim$ `independent`

Dependent Depends on type of model, check help file of modeling function

Independent Variable names separated by operators, without explicit reference to parameters

$$\texttt{fare} \sim \text{age} + \texttt{pclass} + \texttt{sex}$$ three main effects

- interactions are denoted by ":"
  interaction and main effects by "$*$"
  $$\text{age} * \texttt{sex} = \text{age} + \texttt{sex} + \text{age} : \texttt{sex}$$
- formulas may involve existing functions:
  `log(fare), I(age+dob), sqrt(age), cut(age,breaks=3)`

Model output

Output model stored in a list. Results observed via functions

print Short summary of model outcome; typing name is sufficient

summary Longer summary of model description

Model output

Output model stored in a list. Results observed via functions

print	Short summary of model outcome; typing name is sufficient
summary	Longer summary of model description
coef	Parameter values
confint	Confidence intervals
anova	Sequential anova table or compare two models
fitted	Calculates fitted values for records in model
predict	Calculates predicted values for certain values of covariates

Model output

Output model stored in a list. Results observed via functions

print	Short summary of model outcome; typing name is sufficient
summary	Longer summary of model description
coef	Parameter values
confint	Confidence intervals
anova	Sequential anova table or compare two models
fitted	Calculates fitted values for records in model
predict	Calculates predicted values for certain values of covariates
update	Used to refit the model with small changes

Formula structure

Same formula structure in other types of analysis

- graphics

```
> plot(age ~ fare, data=titanic3)
```

- summaries (`xtabs`)
- packages (**doBy**, **Hmisc**, **compareGroups**)
- and many many more

Outline

Statements: if-else

- R also has a conditional construct: depending on the outcome of a test, execute one or another statement

```
if (logical statement){
  do this
} else {
  do that
}
```

```
> x <- 10
> z <- if (x < 2) 4 else 3
> z
[1] 3
```

Statements: repetition (I)

- Let us look at a simple example using matrix `A`

```
        sample1 sample2 sample3 sample4 sample5
gene1         1       3       5       7       9
gene2         2       4       6       8      10
```

```
> results <- numeric(2)
> results
1] 0 0
> for (i in 1:2) {
      results[i] <- mean(A[i, ])
  }
> results
[1] 5 6
```

- We iteratively calculated the mean of each row

Statements: repetition (II)

Imagine that you have to repeat the same analysis for many files that are all in the same folder on your computer. A short solution using an iterative construct would be

```
> files <- dir()
> for (filename in files){
    infile <- read.table(filename, ...)
    do something with infile
  }
```

Apply

- Functions from the `apply` family are convenient shorthands for repetitions

```
apply(X, MARGIN, FUN, ...)
Arguments
X       an array, including a matrix
MARGIN  for a matrix 1 indicates rows, 2 indicates columns
FUN     the function to be applied
```

- Taking a row-wise mean can be handled using `apply`

```
> apply(A, 1, mean)
 gene1  gene2
     5      6
```

Other members of the apply family

- `lapply`: apply a function over a list or vector
- `sapply`: similar to `lapply` but more user-friendly if output can be coerced into a vector
- `tapply`: can be used to split a vector in subgroups and apply a function to each of the subgroups
- `replicate`: simpler version of `sapply` for the repeated evaluation of an expression. Often used for random number generation
- `aggregate`: extension of `tapply` for data frames that splits the data into subgroups and computes summary statistics for each of the subgroups.

tapply: example

- Let us again have a look at the *Titanic* data

```
> head(titanic3[,c("fare","pclass")])
      fare  pclass
1 211.3375     1st
2 151.5500     1st
3 151.5500     1st
4 151.5500     1st
5 151.5500     1st
6  26.5500     1st
```

- Now we can use `tapply` to calculate the mean fare per passenger class

```
> with(titanic3, tapply(fare, pclass, mean, na.rm=TRUE))
     1st      2nd      3rd
87.50899 21.17920 13.30289
```

- **dplyr**: `group_by`, `summarize`

THANKS!