Applied Statistics: R

Semester 2018-I

Francisco Rosales Marticorena

Academic Department of Finance

Table of Contents:

1 Introduction

2 R Basics

3 Advanced Graphics

4 Numerics

5 Simulations

6 Data Management

7 Functions, Debugging & Condition Handling

8 Object Oriented Programming

9 Profiling, Performance & Parallelization

10 R & Other Languages

11 Building R Packages

Introduction

Objectives

- new graphical methods
- data mangement (import/export)
- functions, debugging, condition handling
- object-oriented programing: S3 and S4 classes
- parallelization of R code
- integration of other programing languages, e.g. C++
- building R packages

Coordination & Examination

- No Blackboard.
- For lecture notes, code, references, etc. go here: `https://github.com/franciscorosales-marticorena/ApplStatsR`
- One exam on mid-term week. 20% of your grade.

Reading Material

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

Who are you?

- operating system: Linux, Apple, Windows?
- text editor for R: Rstudio, emacs, Vim?
- other programing languages: C, C++, Java, Python, Fortran, Julia?
- compiled already an R package?
- LaTeX?

R Basics

Really Basic

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

case sensitive: usage of CAPITAL and small letters matters!

Basic Data Structures

integer:	`1,2,301L`
double:	`1.0, .141, 1.23e-3, NaN, Inf, -Inf`
logical:	`TRUE, FALSE`
character:	`"hello", "I'm a string"`
numeric	`general class for 'numberliness'` (integer, double)
complex	`1+0i, 1i, 3+5i`
missings:	`NA`

```
mode(1:10)              ## [1] "numeric"
storage.mode(1:10)      ## [1] "integer"
mode(pi)                ## [1] "numeric"
storage.mode(pi)        ## [1] "double"
mode(TRUE)              ## [1] "logical"
mode("Hello")           ## [1] "character"
### also useful:
str(1:10)               ## int [1:10] 1 2 3 4 5 6 7 8 9 10
str(LETTERS)            ## chr [1:26] "A" "B" "C" "D" "E" ...
```

Construction & Coercion

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

```
integer(5)           ## [1] 0 0 0 0 0
double(0)            numeric(0)
str(c("a",1))        ##chr [1:2] "a" "1"
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)        ## [1] 0 0 1
# Total number of TRUEs
sum(x)               ## [1] 1
```

`NA`, `NaN` and `NULL`

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

`NA`	missing sample values, impossible coercion, . . .
`NaN`	undefined results in mathematical operation (e.g. `log(-1)`, `1/0`)
`NULL`	null pointer, i.e. pointer in empty/undefined memory

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

Inifinity

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

```
max(3, Inf)
## [1] Inf
```

```
min(3, Inf)
## [1] 3
```

```
c(Inf + Inf, (-Inf) * Inf, Inf - Inf)
## [1] Inf -Inf NaN
```

Complex Data Types

- complex data structures in R can be organized by their dimensionality and if all their contents are of the same type, or not:
- a data.frame is a matrix, in which not every column has to have the same data type, and a list, in which each element has the same length

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

Attributes

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

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

Attributes

Example 2.1

```
x <- matrix(1:10, ncol = 5)
attributes(x)            ## $dim
                         ## [1] 2 5
rownames(x) <- c("Eins", "Zwei")
attributes(x)            ## $dim
                         ## [1] 2 5
                         ## $dimnames
                         ## $dimnames[[1]]
                         ## [1] "Eins" "Zwei"
                         ## $dimnames[[2]]
                         ## NULL
as.character(x)          ## [1] "1"  "2"  "3"  "4"  "5"  "6"...
attributes(as.character(x))  ## NULL
```

Subsetting

- There are three subsetting operators: `[`, `[[`, `and` `$`
- the three types of subsetting:
  - Positive integers return elements at the specified positions.
  - Logical vectors select elements where the corresponding logical value is TRUE; application of logical expressions.
  - character vectors to return elements with matching names.
- important differences in behavior of different objects (e.g., vectors, lists, factors, matrices, and data frames).
- More advanced subsetting, in particular in combination with complex logical expressions, can be done using the functions subset() and which().
- There are also two additional subsetting operators that are needed for S4 objects: `@` (equivalent to `$`), and `slot()` (equivalent to `[[`).
- The default drop=TRUE simplifies the data type of the result.

Atomic Vectors

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

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

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

Matrices & Arrays

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

```
x <- matrix(1:10, ncol=2); colnames(x) = c("Eins", "Zwei")
x[1:2,]                  # results a matrix
x[,"Zwei"]               # results a vector
x[,"Zwei", drop=FALSE] # results a matrix
x[-3,]                   # everything, but not third row
```

Lists

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

```
(x <- list(a = "Hallo", b = 1:10, pi = pi))
x$a       # first element of the list
x[['a']]
x[[1]]
x[1]      # list with one element
x[2:3]    # list with two elements
x[[2:3]]  # wrong result
```

Data Frames

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

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

Flow Control

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

basic vocabulary:

```
if, &&, || (short circuiting)
for, while, repeat
next, break
switch
ifelse
```

if/else Conditions

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

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

for Loop

In each interation, <var> is set to the next element of <vector> and <expression> is evaluated.

```
for (<var> in <vector>) {<expression>}
sum <- 0
for(i in 1 : length(x)) {
sum <- sum + x[i]
}
sum <- 0
for(x_value in x) {   ## more efficient
sum <- sum + x_value
}
```

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

```
x <- numeric(0)
1 : length(x)       ## [1] 1 0
seq_along(x)        ## integer(0)
```

while Loop

As long <test> is TRUE the <expression> is evaluated.

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

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

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

Be aware of infinite loops!

`next` & `break`

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

```
 x <- c(1, 1, 1, NA, 2)
sum <- 0
for(val in x) {
  if(is.na(val)) break
  sum <- sum + val
}
sum     ## [1] 3
x <- c(1, 1, 1, NA, 2)
sum <- 0
for(val in x) {
  if(is.na(val)) next
  sum <- sum + val
}
sum     ## [1] 5
```

Style Example: Bad

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

Style Example: Good

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

Notation & Names

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

  ```
  0-download.R
  1-parse.R
  2-explore.R
  ```

- avoid uppercase
- use an underscore to separate words within a name.
- generally, variable names should be nouns and function names should be verbs.

Notation & Names

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

```
# Good      # Bad
day_one     first_day_of_the_month
day_1       dayone
            djm1
```

- avoid using names of existing functions and variables.

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

Formatting

- strive to limit your code to 80 characters per line.
- when indenting your code, use two spaces. Never use tabs or mix tabs and spaces.
- use <-, not =, for assignment
- place spaces around all infix operators (=, +, -, <-, etc.), before parentheses, and after comma (just like in regular English)

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)
# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
# Good
if (debug) do(x)
plot(x, y)
# Bad
if(debug)do(x)
plot (x, y)
```

Formatting

- an opening curly brace should always be followed by a new line, while a closing curly brace should always go on its own line, unless it?s followed by else.

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

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

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

- `formatR::tidy.source()` cleans up and does some automatic formatting such as consistent indent

Advanced Graphics

Motivation

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

Graphical Devices

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

Graphical Devices

The following graphics devices are currently available:

- `pdf()`: Write PDF graphics commands to a file; can be handy for distribution to cooperation partners, integration in PDFLaTeX, or viewing many graphics
- `postscript()`: Writes PostScript graphics commands to a file
- `bitmap()`: bitmap pseudo-device via 'Ghostscript' (if available).

Interactive plotting with GUI:

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

For further information see `?Devices`.

this section is incomplete

Numerics

Basics

Representation of numbers

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

Representation of integers

The sign-and-magnitude scheme

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

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

where each b_i is 0 or 1 and which is translated to

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

Example 4.1
For $k = 8, -1001101$ represents the number

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

Representation of integers

Properties and extensions

- Integers range symmetrically from $-(2^{k-1}-1)$ to $2^{k-1}-1$
- Two representations of 0 ($\pm$)
- The biased scheme1 avoids -0, but addition of integers is complex and slow
- The two's complement scheme uses the binary representation for positive integers and represents $-1, -2, \dots, -2^{k-1} via 2^k - 1, 2^k - 2, ..., 2^k - 2^{k-1}$
- Efficient implementation of addition (equiv to addition modulo 2^k)

in R

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

Representation of real numbers

Properties:

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

- use 8 bytes (i.e. 64 bits) in total
- 1 bit for the sign
- 52 bits for the mantissa
- 11 bits for the exponent (representation via biased scheme, ranges from -1022 to 1024)

in R

```
> .Machine$double.xmin
[1] 2.225074e-308
> .Machine$double.xmax
[1] 1.797693e+308
```

Representation of real numbers

Further examples:

- Underflow/ overflow

  ```
  > 2^1023 + 2^1022 + 2^1021
  [1] 1.572981e+308
  > 2^1023 + 2^1022 + 2^1022
  [1] Inf
  ```

- “Asymmetry”

  ```
  > 2^(-1074) == 0
  [1] FALSE
  > 2^(-1075) == 0
  [1] TRUE
  > 1 / 2^(-1074)
  [1] Inf
  ```

Representation of real numbers

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

```
> x <- 1 + 2 ^ -52
> x - 1 == 0
[1] FALSE
> y <- 1 + 2 ^ -53
> y - 1 == 0
[1] TRUE
```

Representation of real numbers

Numerical errors:
Numerical errors occur all the time. E.g. there is no finite binary representation of 0.1. Denote by $\tilde{x}$ an approximation of x.

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

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

Example

Computation of $\sin(x) - x$ to 0

- Standard computation

$$\lim_{x \to 0} \frac{\sin(x)}{x} = 1 \Rightarrow \sin(x) \approx x \quad \text{near} \quad 0$$

- Taylor expansion of order 2

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

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

- Relative errors: $\approx 10^{-11}$, 10^{-4}, 1
- Catastrophic cancellation at $x = 2^{-20}$ and $x = 2^{-30}$!

Numerical Algorithms

Generalities

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

Numerical Algorithms

Some important functions for numerical mathematics in R:

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

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

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

Example

Find the minimum of the function $f(x) = x^2$. What is the value of $\int_{-2}^{2} x^2 dx$?

```
> my.square = function(x) {
+ x^2
+}
> optimize(f = my.square, interval = c(-2, 2)) $minimum
[1] -5.551115e-17
$objective
[1] 3.081488e-33
> optimize(f = my.square, interval = c(2, 3))
$minimum
[1] 2.000066
$objective
[1] 4.000264
> integrate(f = my.square, lower = -2, upper = 2)
5.333333 with absolute error < 5.9e-14
```

Simulations

Generalities

Definition 5.1
A (Monte-Carlo)-Simulation is a numerical technique for conducting experiments on a computer. The term Monte-Carlo refers to the involvement of random experiments.

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

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

Generalities

Rationale

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

Usual setup

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

Stochastic distributions in R

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

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

Stochastic distributions in R

Further notes

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

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

Case study

Stochastic Frontier type data:
Stochastic Frontier Analysis (SFA) belongs to the field of productivity analysis

- Aim: quantify inefficiency and determine a production function
- Assumptions: deviations from the production function (the errors) are a combination of stochastic noise and inefficiency, formally $\epsilon = v - u$.
- Comparison: the model formulation deviates from the classical linear model only in terms of inefficiency

Case Study

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

$$y_i = \beta x_i + \epsilon_i, \quad i = 1, ..., n, \tag{1}$$

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

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

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

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

Further Reading

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

Data Management

Motivation

Data Import and Export

- can be a frustrating task and can take far more time than the statistical analysis itself
- majority of the data will be given in the format of spreadsheet–like data
- it is often easier to import badly formatted data than explain what a “good” format is ⇒ importing from other systems
- R is not particularly well suited to manipulations of large-scale data ⇒ relational Databases
- in practice, you will be often faced with data corrections and updates ⇒ project management

Data Import Vocabulary

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

Data Import Vocabulary

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

# low-level interface to the computer's file system:
file.copy, file.create, file.remove, file.rename, dir.create,
file.exists, file.info
```

Import Spreadsheet-like Data

To read spreadsheet-like data you probably have already used function `read.table()` and variations like `read.csv()`, . . .

`fileEncoding`:	use `latin1` for data produced in Windows, and utf-8 for data from Unix
`header`:	names of variables (columns) and observations (rows)
`sep`:	field separator vs. record separator
`quote`:	protecting separators appearing in strings
`na.strings`:	which string (or number) represents missing values?
`fill=TRUE`:	are lines with missing trailing fields OK? be careful!
`colClasses`:	which type of variable is contained in which column?
`skip`:	number of lines skipped before beginning to read data

Importing from other statistical systems

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

read.epiinfo: will read fixed-width text format .REC files into an R data frame

read.mtp: imports 'Minitab Portable Worksheet'

read.xport: reads a file in SAS Transport (XPORT) format

read.ssd: generates a SAS program to convert the ssd contents to SAS transport format

read.spss: can read files created by the 'save' and 'export' commands in SPSS

read.dta: imports binary files created by Stata

read.S: reads binary data files or 'data.dump' files that were produced in S

Data Export: Basic Vocabulary

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

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

Data Export

- exporting results from R is usually a less contentious task than importing
- normally a text or csv file will be the most convenient interchange vehicle with `write.table` or `write.csv`
- `MASS:write.matrix` provides a specialized interface for writing matrices, with the option of writing them in blocks and thereby reducing memory usage
- sink diverts the standard R output to a file, and thereby captures the output of (possibly implicit) print statements
- `foreign::write.foreign` writes a code file that will write this text file into another statistical package
- the precision is governed by the current setting of `options(digits)`: Export for report writing or further analysis?

Relational Database

Why use a database?

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

R Database Interface Packages

R Database Interface Packages:

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

Example for SQL queries:

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

Memory Management

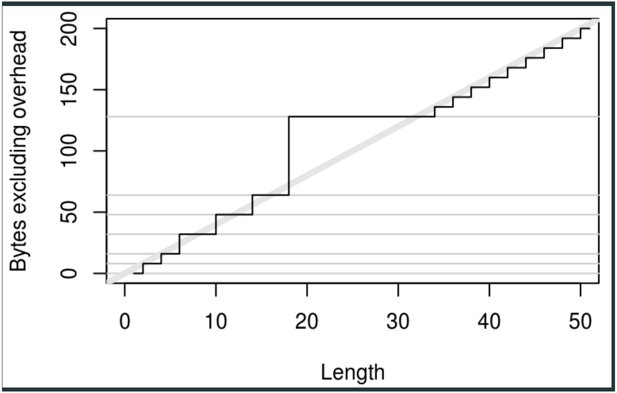
- some understanding of R's memory management will help you predict how much memory you will need
- `pryr::object size()` tells you how much memory an object occupies (including environments)

```
> require(pryr)
> object_size(1:10)      # 88 B
> object_size(mtcars)    # 6.74 kB
> object_size(numeric())  # 40 B
>
> sizes <- sapply(0:50, function(n) object_size(seq_len(n)))
> plot(0:50, sizes,
+      xlab = "Length", ylab = "Size (bytes)", type = "s")
```

- for memory profiling use the function `utils::Rprof()` or `lineprof::lineprof()`
- for more information see also `?Memory`

Memory Management

```
 > sizes <- sapply(0:50, function(n) object_size(seq_len(n)))
> plot(0:50, sizes,
+      xlab = "Length", ylab = "Size (bytes)", type = "s")
```



Memory Management

- `pryr::mem used()` tells you the total size of all objects in memory, while `pryr::mem_change()` tells you how memory changes during code execution
- R has automatic garbage collection, but it is lazy, so that it won?t reclaim memory until it is actually needed, otherwise use `gc()`
- be aware of possible memory leaks due to deleted objects

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

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

Project Management

Object Storage

Workspace: is your current R working environment and includes any user–defined objects (vectors, matrices, data frames, lists, functions). When R is started, a `.RData` file is loaded, which can be saved when R is closed (intermediate `save.image()`) - Be careful!

Binary files: `save()` allows the explicit saving of functions and data in binary file, that can be loaded by `load()`

Source code: `source()` accepts its input from the named file or URL and runs the script in the current session

Project Management

Manage your Workspace

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

Publication Quality Output

Publication Quality Output and Documentation of Analysis:

knitr: enables integration of R code into LaTeX, LyX, HTML, Markdown, AsciiDoc, and reStructuredText documents

xtable: converts some R objects into LaTeXcode.

R2HTML: converts your output text, tables, and graphs in HTML format

odfWeave: has functions that allow you to imbedd R output in Open Document Format (ODF)

SWordInstaller: allows you to add R output to Microsoft Word documents

R2PPT: provides wrappers for adding R output to Microsoft PowerPoint presentations.

Sweave: Integration with LaTeX

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

Sweave Example

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

Sweave Chunk Options

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

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

The most important options for Sweave code chunks are:

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

Defaults can be adapted using `SweaveOpts()`

Further Reading

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

Functions, Debugging & Condition Handling

Motivation: Functions in R

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

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

  - the keyword function indicates the beginning of a function
  - arguments `<arglist>` are defined by comma-separated key-value-pairs
  - `<body>` is a block of R commands which are executed by the function

Function Components

All R functions are objects and consist of three parts:

`body()` the code inside the function.

`formals()` the list of arguments which controls how you can call the function.

`environment()` the "map" of the location of the function's variables. If the environment isn't displayed, it means that the function was created in the global environment.

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

Function Arguments

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

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

Default and Missing Arguments

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

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

match.arg()

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

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

- missing argument will be replaced with the first candidate

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

Lazy Evaluation

- By default, R function arguments are lazy – they're only evaluated if they're actually used.

```
myfun <- function(x, y){
    if(x < 0)
        return(NaN)
    else
        return( y * log(x))
}
myfun(-1)
myfun(2,3)
myfun(2)
```

- If you want to ensure that an argument is evaluated you can use `force()`
- More technically, an unevaluated argument is called a promise, or (less commonly) a thunk.
- Laziness is useful in if statements – the second statement below will be evaluated only if the first is true.

The Argument ...

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

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

Connecting the ...

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

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

  ## further calculations
}

myfun(x, fun2.args=list(arg1=value))
```

Return Values

- The last expression evaluated in a function becomes the return value, the result of invoking the function.
- Functions can return only a single object, so that you have to combine a number of objects in a list.
- Functions can return invisible values, which are not printed out by default when you call the function.
- The code in `on.exit()` is run regardless of how the function exits.

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

Basic Debugging Vocabulary

- While it's true that with a good technique, you can productively debug a problem with just print(), there are times when additional help would be welcome.
- Wickham [2015] provides an outline for a general procedure for debugging:
  1. realise that you have a bug: implement testing procedures!
  2. make it repeatable: create a minimal example
  3. figure out where it is: identify the line of code that?s causing the bug
  4. fix it and test it: ensure you fixed the bug and have not introduced new ones

Basic Debugging Vocabulary

Make the bug reproducible:

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

Bug Identification

`Binary search`: you repeatedly remove half of the code until you find the bug.

`traceback()`: determines the sequence of calls that lead up to an error (call stack).

`browser(), debug()`: starts an interactive console in the environment where the error occurred.

```
options(error = c(NULL, browser, recover))
```

Bug Identification: `traceback()`

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

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

f(10)
# Error in "a" + d (from #1) : non-numeric argument to binary
# operator

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

Bug Identification: `browser()`

`<RET>,n`: Go to the next statement if the function is being debugged. Continue execution if the browser was invoked.

`c`: Continue execution without single stepping.

`where`: Show the call stack.

`q`: Halt execution and jump to the top-level immediately.

```
browser(mean(1:10))
Browse[2]> where
where 1: mean.default(1:10)
where 2: mean(1:10)
.
.
Browse[2]> n
debug: if (na.rm) x <- x[!is.na(x)]
Browse[2]> x
[1] 1 2 3 4 5 6 7 8 910
.
.
```

Additional Notes on Debugging

- prevent bugs: Instead of trying to write one big function all at once, work interactively on small pieces.
- a function may generate an unexpected warning. The easiest way to track down warnings is to convert them into errors with `options(warn = 2)`
- a function may generate an unexpected message. There is no built-in tool to help solve this problem, but it is possible to create one:

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

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

Exception Handling

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

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

  `warning()` Display potential problems

  `message()` Give informative output in a way that can easily be suppressed by the user (`suppressMessages()`)

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

stop()

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

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

As alternative command, you can also use stopifnot().

try-error()

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

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

tryCatch()

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

```
f <- function(x, silent = FALSE) {
  s <- tryCatch(sum(x),
         error = function(e) {
                  warning("x of wrong type, sum is NA.")
                  if (!silent) print(e)
                  return(NA)
               },
               finally = cat("buh bye!\n"))
 s }
f(1:10) # buh bye!
        # [1] 55
f("nonsense")
# Warning: x of wrong type, sum is NA.
# <simpleError in sum(x): invalid ’type’ (character) of argument>
# buh bye!
# [1] NA
```

Further References

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

Object Oriented Programming

Motivation

- in OOP, computer programs are designed by making them out of objects that interact with one another
- a class defines the behaviour of objects by describing their attributes and their relationship to other classes.
- the class is also used when selecting methods, functions that behave differently depending on the class of their input.
- R has three OO systems: S3, S4, Reference classes (not covered by this lecture), and the system of base types

OOP Essentials

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

Base Types

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

S3 Object System

- S3 is R's first and simplest OO system
- S3 is informal and ad hoc, but it has a certain elegance in its minimalism
- test if an object is an S3 object by using `pryr::otype()`
- test for a specific class by `class()` or `inherits(object,'class name')`

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

S3 Methods

- in S3, methods belong to functions, called generic functions, i.e. S3 methods do not belong to objects or classes.
- `pryr::ftype()` describes the object system, if any, associated with a function

  ```
  mean
  # function (x, ...)
  # UseMethod("mean")
  # <bytecode: 0x2bc14a0>
  # <environment: namespace:base>
  ftype(mean)  # [1] "s3"      "generic"
  ```

- in general, you can recognise S3 methods by their names, e.g. the Date method for the `mean()` generic is called `mean.Date()`
- `methods()` returns all methods that belong to a generic or lists all generics that have a method for a given class

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

Define S3 Classes and Create Objects

To make an object an instance of a class, you just take an existing base object and set the class attribute.

- S3 objects are usually built on top of lists, or atomic vectors with attributes
- class names are usually lower case, and you should avoid "."
- S3 has no checks for correctness

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

Create new Methods and Generics

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

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

S3 Problems

- S3 has no formal definition and therefore no checks for correctness, when construction objects, i.e. every object can get every class
- while you can change the type of an object, you never should. R doesn't protect you from yourself:

  ```
  mod <- lm(log(mpg) ~ log(disp), data = mtcars)
  class(mod)
  print(mod)
  class(mod) <- "data.frame"
  print(mod)
  # [1] "lm"
  # a reasonable model summary
  # not very helpful
  ```

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

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

S4 Object System

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

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

Define S4 Classes and Create Objects

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

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

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

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

Create new Methods and Generics

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

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

Picking a System

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

S3/S4 Object System Comparison

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

Conventions

For S3 and S4, there are the following conventions

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

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

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

Further Reading

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

Profiling, Performance & Parallelization

Motivation

- for time-critical application, it can be useful to measure the runtime of code
- optimising code to make it run faster is an iterative process:
  1. find the biggest bottleneck (the slowest part of your code).
  2. try to eliminate it (you may not succeed but that?s ok).
  3. repeat until your code is ?fast enough.?
- parallelization does not reduce the computing time, but it saves your time because you are using more of your computer?s resources

Performance Measurment

- `system.time()` quick and dirty performance accessment
- `microbenchmark`: performance of a very small piece of code
- `Rprof` detailled analysis, where how much computing time is required
- `lineprof` and `shine()`: interactive profiling with information about memory usage and graphical interface (see memory session)

system.time()

```
x <- rnorm(100000)
system.time({s <- 0
             for(i in 1:length(x))
s <- s + x[i] })
#   user  system elapsed
#   0.12    0.00    0.23
system.time({s <- 0
             for(v in x)
s <- s + v })
#   user  system elapsed
#  0.089   0.000   0.076
system.time(sum(x))
#   user  system elapsed
#  0.000   0.000   0.001
```

Microbenchmarking

- a microbenchmark is a measurement of the performance of a very small piece of code
- best tool for microbenchmarking in R is the microbenchmark package

```
require(microbenchmark)
x <- runif(100)
microbenchmark(
sqrt(x),
x ^ 0.5 )
# Unit: microseconds
#    expr    min      lq     mean  median     uq    max neval
# sqrt(x)  1.840  1.9395  2.17828  2.0150  2.076 15.885   100
#   x^0.5 11.519 11.7245 12.50866 11.7965 11.902 40.353   100
```

Rprof()

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

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

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

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

- check results with:

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

lineprof and shine()

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

Why is R slow?

- a lot of R code is poorly written, because a few R users have any formal training in programming or software development
- R language is informal and incomplete
- extreme dynamism, so that almost everything can be modified after it is created
- name lookup with mutable environments: due to the combination of lexical scoping and extreme dynamism, it is difficult to find the value associated with a name in the R-language
- lazy evaluation of function arguments: each additional argument to a function decreases its speed a little

Optimizing Code

Strategies to achieve better performance:

1. look for existing solutions (search online, talk to your colleagues, reverse Rccp dependencies, ...)
2. do less work
3. vectorize
4. parallelize
5. avoid copies (whenever you use `c()`, `append()`, ... to create a bigger object, R must first allocate space for the new object and then copy the old object to its new home)
6. byte-code compile (if code consists of many elemental operations, and is not calling often high-level functions)
7. rewrite key functions in C++ (next session)

Optimize code by doing less work

- use a function tailored to a more specific type of input or ouput, or a more specific problem, e.g.
  - rowSums(), colSums(), rowMeans(), and colMeans() are faster than equivalent invocations that use apply() because they are vectorised
- avoid functions which coerce their inputs into a specific type, if your input is not the right type
- functions will do less work if you give them more information about the problem: carefully read the documentation and experiment with different arguments, e.g.
  - `read.csv()`: specify known column types with `colClasses`
  - `cut()`: do not generate labels with labels = `FALSE` if you do not need them, or, even better, use `findInterval()`

Vectorize

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

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

```
sqrt2 <- function(x){
    out <- vector(?list?, length(x))
    for(i in seq_along(x)){
        out[[i]] <- sqrt(i)
    }return(out)}
microbenchmark(
    unlist(sqrt2(1:10)),
    unlist(lapply(1:10, sqrt)))
```

Functionals

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

	list	data frame	array
list	llply()	ldply()	laply
data frame	dlply()	ddply()	daply
array	alply()	adply()	aaply

Multicore CPU

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

```
system.time(lapply(1:10, pause(0.25)))
#   user  system elapsed
#  0.000   0.004   2.504
system.time(mclapply(1:10, pause(0.25), mc.cores = 4))
#   user  system elapsed
#  0.011   0.021   0.758
```

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

Further Reading

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

R & Other Languages

Other Languages

Several programming languages can be included in R:

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

Why R?

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

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

Application

Determination of the reasons for chronic undernutrition in Peru.

Estimation of the model

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

for $\tau = 5\%$ takes

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

Why R?

Why R?

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

Rcpp

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

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

To learn C++:

- `http://www.learncpp.com/`
- `http://www.cplusplus.com/`

Compiler

To run Rcpp a C++ compiler is required:

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

Take care when installing Rtools:

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

General Structure

General Structure:

```
cppFunction(?variable-type-output function-name(variable-type-input var
... ;
... ;
return output;
}’)
```

Example:

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

C++ Basics

- Specifyvariabletype(double,int,...)
- End each statement with ;
- Use = for assignment not ¡-
- Return value with return ...;
- Comment with

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

C++ Variable Types

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

NumericVector and for-loop

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

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

Sugar

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

- Basic operations:

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

- Standard functions:

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

- Basic statistics:

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

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

Sugar

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

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

```
 cppFunction('List eukldist1(NumericVector x, NumericVector y) {
    NumericVector dist = x - y;
    NumericVector dist2 = pow(dist, 2);
    double meandist = mean(dist2);
    double edist = sqrt(meandist);
    return List::create(Named("dist") = dist,
                        Named("dist2") = dist2,
                        Named("meandist") = meandist,
                        Named("edist") = edist);
}')
cppFunction('double eukldist2(NumericVector x, NumericVector y) {
    return sqrt(mean(pow((x - y), 2)));
    }')
```

Inline & Eigen

- With inline additional features like `RcppEigen` can easily be included
- `Eigen` is a powerful C++ library for matrix operations and advanced routines
  `http://eigen.tuxfamily.org/dox/index.html`
- Alternatively `RcppArmadillo` can be used
  `http://arma.sourceforge.net/`

Inline & Eigen

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

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

Inline & Eigen

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

For elementwise operations inside Eigen objects use array format

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

Pointer to R

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

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

Pointer to R

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

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

Further Reading

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

Building R Packages

Why Building R Packages?

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

Basic Structure

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

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

The standard structure can be obtained automatically using

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

Basic Structure

Naming R Packages:

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

R/ directory contains all R code:

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

Basic Structure

Additional (optional) files in R packages:

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

DESCRIPTION

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

DESCRIPTION

```
Package: mypackage
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("First", "Last",
                  email="first.last@example.com",
                  role=c("aut", "cre", "ctb"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
```

NAMESPACE

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

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

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

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

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

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

S3 NAMESPACE

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

```
# example samplingbook
export(Smean)
S3method(print, Smean)
export(htestimate)
S3method(print, htestimate)
.
.
# example ggplot2
S3method(autoplot,default)
export(autoplot)
import(plyr)
importFrom(MASS,cov.trob)
.
.
```

S4 NAMESPACE

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

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

.Rd manual file

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

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

Building R Packages

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

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

In windows, installation of `Rtools` is required:

```
# On windows:
R CMD INSTALL --build pkg
```

devtools

R functions from devtools that simplifies R packaging:

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

testthat

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

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

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

testthat

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

- within your workflow:

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

- within a package:

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

- using R CMD check:
  - DESCRIPTION needs Suggests testthat
  - test files in tests/testthat/ are automatically tested

Package `roxygen2`

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

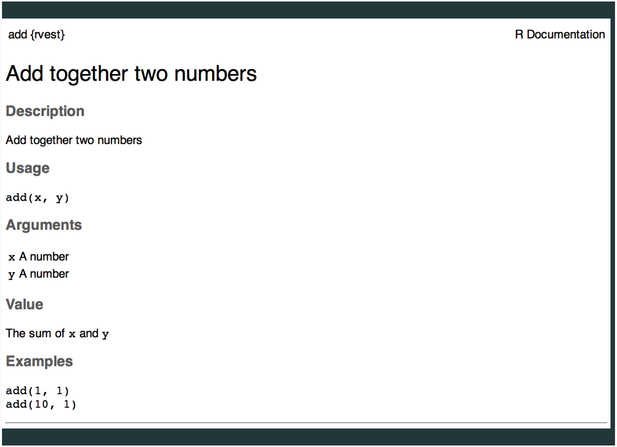
1. add roxygen comments to your source file
2. `roxygen2::roxygenise()` or `devtools::document()` converts roxygen comments to `.Rd` files
3. `R CMD` check converts `.Rd` files to human readable documentation

Package roxygen2

- roxygen comments start with #’
- tags like @param, @return, @author define parts in .Rd file
- tags like @includes, @export, @importFrom generate NAMESPACE und Collate
- tags like @method for OOP documentation

```
#’ Add together two numbers
#’
#’ @param x A number
#’ @param y A number
#’ @return The sum of \code{x} and \code{y}
#’ @examples
#’ add(1, 1)
#’ add(10, 1)
add <- function(x, y) { x+y
}
```

Package roxygen2



add {rvest}

R Documentation

Add together two numbers

Description

Add together two numbers

Usage

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

Arguments

`x` A number
`y` A number

Value

The sum of `x` and `y`

Examples

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

Further Reading

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