Applied Data Analysis — R Introduction

Lea Kaufmann, Thomas van Bentum, Alina Müllenmeister

Institute of Statistics

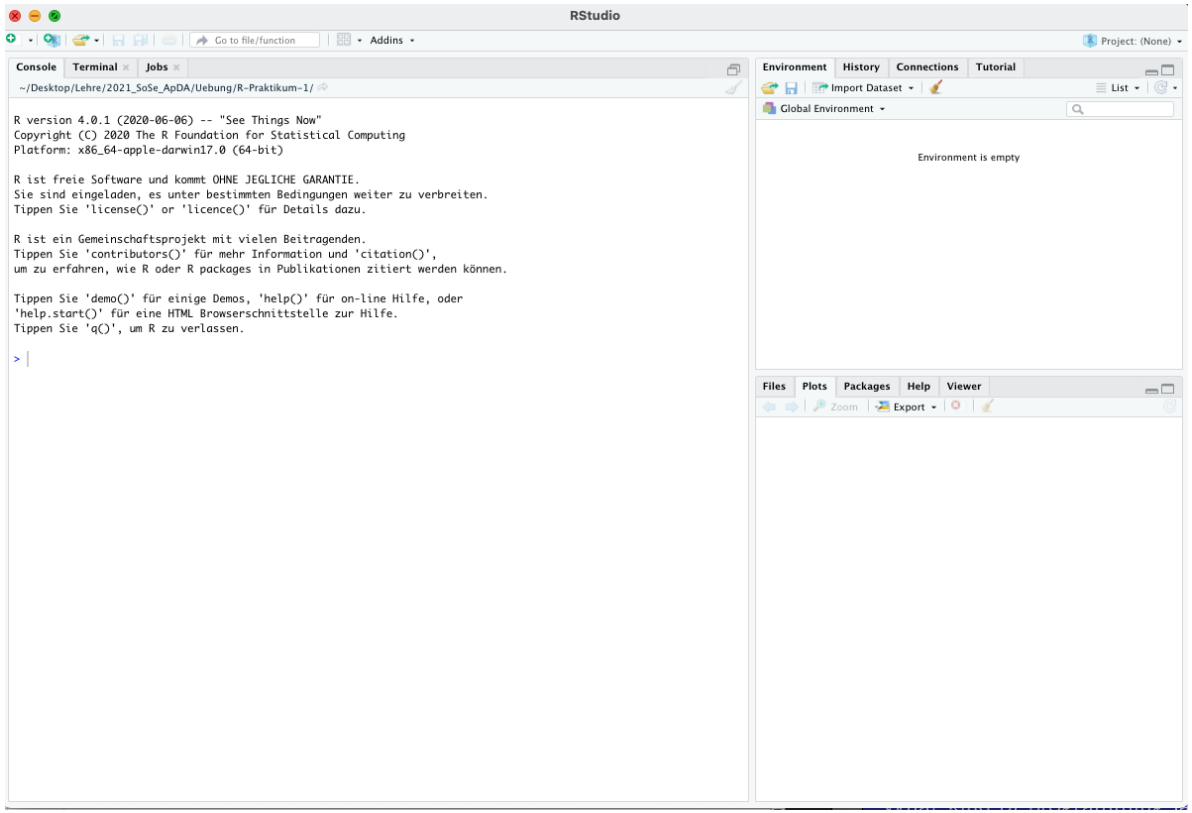
RWTH Aachen University, Aachen, Germany

Aachen, 15.04.2021

RWTH AACHEN UNIVERSITY

Preparations

- Download and install the newest R version from CRAN.
- It is advisable to use R not directly from the shell but by some editor (e.g., RStudio).
- Additional packages can be installed in RStudio via "Tools -> Install Packages..." and are loaded by `library(package_name)`.



What Kind of Programming Language is R?

R is a **scripting** language. Especially note:

- The core object of R are functions. They have equal rights to all other objects. Especially, functions can be both input and output objects of functions and can be defined independently of class-structures (see also classes in R).
- R programs are not compiled, but each program is executed instruction after instruction. As consequence, plain R is awfully slow when too many instructions/calculations have to be performed (see improving speed of R). However, R is easier to debug than compiled programs.

Important Note!

This will be only a very short and biased introduction. Only those facts about R are shown which are assumed to be relevant in our context.

If you want to learn more about the presented objects/functions write

```
?object/function_of_interest
```

into R (beware, that the corresponding package must be loaded). If you are interested in a package write

```
help(package=package_name)
```

A great thing about R is, that it is actively used by a increasing community. Meaning that a great number of questions has already been asked and answered on the internet. Simply google...

Basic Calculations

Since R is a scripting language there is no need to define a main-function or to define classes. Simply start writing instructions, e.g.,

```
> 2*5
[1] 10
> 2/5
[1] 0.4
> 2+5; 2-5
[1] 7
[1] -3
```

There is no need to end the line with `;`, but this is possible. The `[1]` before the result indicates the first element of a vector. Here it is only one-dimensional, hence simply a *numeric* value in R.

```
> str(2*5)
num 10
```

At first, all numbers are numeric (i.e., double/real values). If you precisely want a number to be integer use `as.integer` or write a capital `L` behind a number.

```
> str(2L)
int 2
```

Assignments and Characters

Assignments of variables can be done by either `=` or `<-`.

```
> a = 5; b <- 10
> print(a); b
[1] 5
[1] 10
```

Note that the values of the assignments are not printed directly. Note further that there is no need to declare a variable, especially the data type is not fixed in advance and can change without R giving warnings! This is both good, since implementations are much shorter and clearer, and bad, since this implementation style is unstable and can have terrifying side effects!

```
> str(4L*2L); str(4L/2L)
int 8
num 2
>
> str(a <- "some character string")
chr "some character string"
> str(a <- 5)
num 5
```

So, when using R, you need always to be sure what R is actually doing!

Strings (in R called `characters`) are indicated by quotation marks.

```
> print(c("printing of character", "vectors"))
[1] "printing of character" "vectors"
```

Logicals and Operators

Logicals are very handy in R, although caution is necessary too.

```
> c(T, TRUE)
[1] TRUE TRUE
```

are regarded as true and

```
> c(F, FALSE)
[1] FALSE FALSE
```

as false. As we will later clarify, it is highly recommended only to use `TRUE` and `FALSE`. A nice feature is, that we can calculate with them. True equals 1 and false equals 0. (Remark: Any non-zero numeric value is also considered as TRUE) E.g.,

```
> x <- c(TRUE,FALSE,TRUE,TRUE,FALSE)
> sum(x); prod(x)
[1] 3
[1] 0
```

This is highly useful to test if at least one element of a logical vector is true/false. `NA` ("not available") is also regarded as a logical, but behaves differently. Use with care. In R, there also exists `NaN` ("not a number"). This is not a logical.

Logicals and Operators

Logical statements can also be expressed with the usual logical operators AND (&) and OR (|).

```
> (TRUE & FALSE) | (TRUE & TRUE)
[1] TRUE
```

Note that && and || are also possible, but only have an effect on the first entries if vectors are supplied and are evaluated successively!

```
> y <- !x
> y[1] <- TRUE
> x & y
[1]  TRUE FALSE FALSE FALSE FALSE
> x && y
[1] TRUE
> rm(y)
> x || y
[1] TRUE
> x | y
Error: object 'y' not found
```

Definition of Functions

Before we introduce more data types we must necessarily introduce functions! Everything relies on functions! For example, a function with two arguments is defined by

```
test.function <- function(name.arg1, name.arg2) {
  #... instructions
  return('your result')
}
```

and called by

```
test.function(arg1, arg2)
```

The order of the arguments need not be retained, if we call the arguments by their names, e.g.

```
test.function(name.arg2=arg2, name.arg1=arg1)
```

Example:

```
> pythagoras <- function(a,b=5) {
+     c <- sqrt(a^2 + b**2)
+     return(c)
+ }
> pythagoras(3,4); pythagoras(3)
[1] 5
[1] 5.830952
```

`b=5` in the definition means, that 5 is the default value, if `b` is not specified.

Some Important Functions I

Basic important functions are

- `sum()`: calculating the sum of a vector
- `prod()`: calculating the product of a vector
- `str()`: showing the structure of an object
- `print()`, `plot()`: printing/plotting an object
- `summary()`: summarises information about an object
- `stop()`: stops the execution of a function
- `stopifnot()`: stops, if a condition is satisfied
- `warning()`: does not stop the execution of a function, but gives a warning message
- `message()`: prints a message
- `traceback()`: highly useful function for basic debugging
- `tryCatch()`: error handling function

`print`, `plot` and `summary` are generic functions (see 'classes' in R).

`tryCatch()` is important for error handling in R. Using this function it is possible to tell R, what to do, when errors/warnings/... occur.

Vectors

Now, we introduce important data types in R. They are generated by calling functions. For example vectors.

```
> str(c(1,2,3,4))
num [1:4] 1 2 3 4
> str(seq(from=1,to=4,by=1))
num [1:4] 1 2 3 4
> str(1:4)
int [1:4] 1 2 3 4
> str(rep(1,times=5))
num [1:5] 1 1 1 1 1
```

Attention: `seq(from, to, by)` creates a numeric vector, while `from:to` creates an integer vector.

The access can be done either by name (if specified) or by the `[]`-operator.

```
> res <- c(a=3, 5, 6, 10)
> res[1]; res[c(2,4)]
a
3

5 10
> res["a"]
a
3
```

In contrast to C++, indices start at 1 and not at 0!

Matrices

Matrices are highly important, too. They can be generated by using the functions

```
> (res <- matrix(c(1,2,3,4),nrow=2,ncol=2))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> c(1,2)%*%t(c(1,2))
 [,1] [,2]
[1,]    1    2
[2,]    2    4
> str(c(1,2)%*%t(c(1,2)))
num [1:2, 1:2] 1 2 2 4
 > # numeric matrix
```

Obviously, R is column major, i.e., vectors and matrices are stored internally as columns (note that C++ is row major). If you want to fill a matrix row-wise use the option `byrow=TRUE`.
Analogously to vectors, the access of matrices can be done using names or the `[]`-operator.

```
> res[1,2]; res[1,]; res[,2]
[1] 3
[1] 1 3
[1] 3 4
```

Some Important Functions II

Apart from `sum()` and `prod()` there are some other important functions connected with vectors and matrices.

- `c()`: "combines" values into a vector or a list
- `mean()`: calculates the arithmetic mean
- `median()`: calculates the median
- `quantile()`: calculates quantiles; pay attention to the `type`-argument (see the help-page before usage)
- `colSums()`: calculates the column sums of a matrix
- `colMeans()`: calculates the column means of a matrix
- `t()`: transposes a matrix

If you want to apply a different function than `sum()` or `mean()` on the columns of a matrix, use

```
apply(matrix, 2, function)
```

For applying a function on the rows, use `rowSums()`/`rowMeans()` or

```
apply(matrix, 1, function)
```

Vectorisation I (Matrix-Vector Operations)

When calculating in plain R there is one crucial, basic, most important rule:

Rule

Try to vectorise calculations as far as possible.

This is because matrix-vector operations are C-compiled routines in R and freaking fast. Iterated evaluation of functions in R however is freaking slow. Hence, do not use loops (or equivalently something like `sapply()`) in R on large data sets if it is not absolutely necessary!

```
> A <- matrix(1:6, nrow=2); d <- 1:3; e <- 4:6; f <- 1:2
```

```
> 2*A
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
> A%*%d
     [,1]
[1,]   22
[2,]   28
> d*d
[1] 1 4 9
> t(d)%*%e
     [,1]
[1,]   32
```

```
> d%*%t(e)
     [,1] [,2] [,3]
[1,]    4    5    6
[2,]    8   10   12
[3,]   12   15   18
> d+e
[1] 5 7 9
> d/e
[1] 0.25 0.40 0.50
> A+f
     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    4    6    8
```

Remark: `+,-,*,/` are applied component-wise! Matrix-vector-multiplication is done using `%*%`.

Data Types (list, data.frame) and Access-Operators

Further, `list` and `data.frame` are useful data structures, especially if we want a function to return more than one value. Then we need to combine them by a `list` or `data.frame`.
A `list` can comprise all sorts of objects, whereas `data.frame` is a list of variables of the same number of rows with unique row names. (Or, a `data.frame` can be viewed as a `matrix` with possibly different variable types for each column)

```
> (llist <- list(obj1=c(1,2), obj2=list(c(2,3), bool=TRUE)))
$obj1
[1] 1 2

$obj2
$obj2[[1]]
[1] 2 3

$obj2$bool
[1] TRUE
> llist$obj1; llist[[1]]
[1] 1 2
[1] 1 2
```

Access can be gained with `$`, `[[]]` (only content is returned) or `[]` (content and names are returned).

Data Types (list, data.frame) and Access-Operators

data.frames can be accessed like a matrix or a list.

```
> dat.fr <- data.frame(obj=c(10,15,4), factor=factor(c("x","y","x")))

> str(dat.fr)
'data.frame': 3 obs. of  2 variables:
$ obj    : num  10 15 4
$ factor: Factor w/ 2 levels "x","y": 1 2 1

> summary(dat.fr)
     obj         factor
Min.   : 4.000   x:2
1st Qu.: 7.000   y:1
Median :10.000
Mean   : 9.667
3rd Qu.:12.500
Max.   :15.000

> dat.fr$obj
[1] 10 15  4
> dat.fr[[1]]
[1] 10 15  4
> dat.fr[,1]; dat.fr[1,]
[1] 10 15  4
  obj factor
1  10      x
```

Control Flow Statements

R provides of course the usual control flow statements `for`, `while`, `repeat`, `if`. Mainly, `for` and `if` are used.

```
> a <-0
> for (i in 1:6) {
+     a <- a + i^2
+ }
> a
[1] 91

> if(91==a) {
+   print("Correct value")
+ } else {
+   print("Wrong value")
+ }
[1] "Correct value"
```

Note that `==` compares numeric values (be careful! See floating point trap) on equality.

Distributions in R

The `base`-package of R provides many frequently used distributions such as normal, exponential, Weibull, Gamma, log-normal, ...
But be careful, which parametrisation is implemented! Consult the help pages for details, e.g., `?dgamma`.

Access to density, distribution function, quantile function and random generation is given by the prefixes

- `d'distribution'`: density
- `p'distribution'`: distribution function
- `q'distribution'`: quantile function
- `r'distribution'`: (pseudo-)random number generation

```
> dnorm(1)
[1] 0.2419707
> (F.dist <- pnorm(1, mean=0, sd=2))
[1] 0.6914625
> qnorm(0.975,0,2)
[1] 3.919928
> rnorm(5)
[1] -0.1025304  1.3773074  0.9319042  0.1220497 -1.2832777
> 1 - F.dist # survival function
[1] 0.3085375
```

Creating Plots

R has strong tools for graphical outputs. We will focus on 2D outputs of functions with `plot`. Basically all we need to provide are the x- and y-coordinates or provide a proper univariate function.

```
x.axis <- seq(0,10,0.1)
plot(x.axis, dnorm(x.axis,0,2), type="l")
plot(function(x) dnorm(x,0,2), xlim=c(0,10))
```

We now can set additional options to make the plot look nicer. Some examples:

- `type`: changes the type of plotting (e.g., `"p"` for points and `"l"` for lines)
- `main`: sets overall title
- `xlab, ylab`: sets x- and y-labels
- `col`: colour of the points/line
- `lty`: line type (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash)
- `lwd`: line width
- `xaxs="i", yaxs="i"`: if we want to start at the origin
- `xlim, ylim`: plotting ranges
- `add`: if we want to add a plot to another one

Beware: For historical reasons, `add` is allowed as an argument to the "function" method of plot, but its behaviour may surprise you. It is recommended to use `add` only with `curve`.

Creating Plots

Instead of using the `add`-option in the plot-function, one can also show plots in the same graphic by using `points`, `lines` or `curve`, respectively. This is even recommended!

In order to plot multiple plots, set

```
par(mfrow=c('number of rows', 'number of columns'))
```

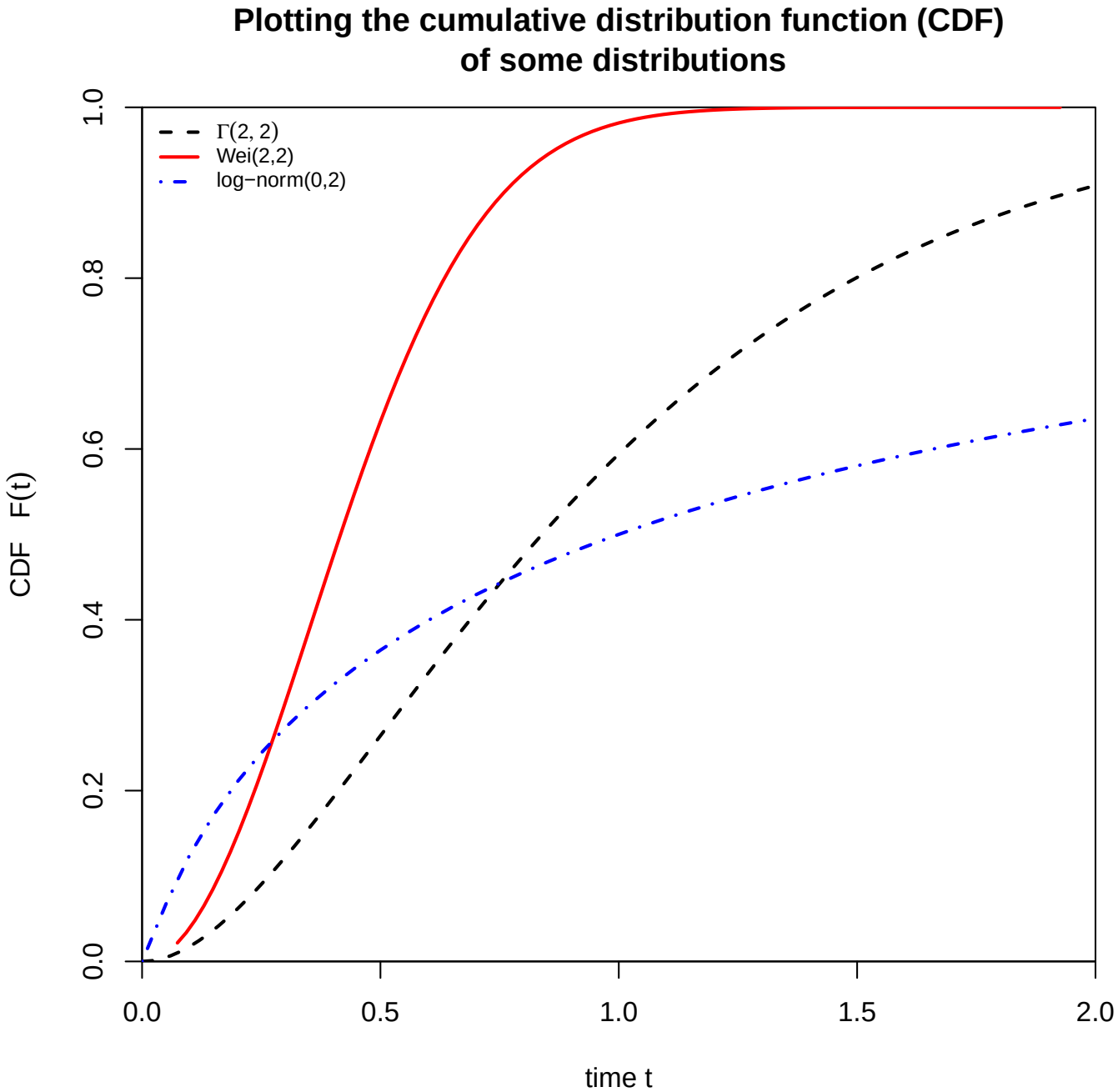
to the necessary values. Do not forget to set it back at the end (`par(mfrow=c(1,1))`).

```
plot(function(x) pgamma(x, shape = 2, rate = 2),
     main = "Plotting the cumulative distribution
             function (CDF)\n of some distributions",
     xlab = "time t", ylab = expression(paste("CDF ",phantom(0),F(t))),
     xlim = c(0,2), ylim = c(0,1), lwd = 2, lty = "dashed", xaxs = "i",
     yaxs = "i")

curve(pweibull(x, shape=2, scale=1/2), lwd = 2, col = "red", add = TRUE)

x.axis <- seq(0,2,0.05)
lines(x.axis, plnorm(x.axis, 0, 2), lwd = 2, col = "blue", lty = 4)

legend <- expression(Gamma(2,2), "Wei(2,2)", "log-norm(0,2)")
legend("topleft", legend, col = c("black", "red", "blue"),
       lwd = 2, lty = c(2,1,4), cex = 0.75, bty = "n")
# cex for scaling the legend, bty="n" for removing box around legend
```


Plotting the cumulative distribution function (CDF)
of some distributions
Γ(2, 2)
Wei(2,2)
log−norm(0,2)
CDF F(t)
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1.5
2.0
time t

Speeding up R – Vectorisation II

Vectorisation
Much of the ease in programming in R comes from *syntactic sugar* supplied. Key word: vectorisation! Practically all basic operations/functions in R are vectorised, i.e., they can take vectors as arguments! For example: `sum`, `prod`, `log`, `sin`, `cos`, `+`, `-`, `*`, `/`.
So, vectorisation often provides on the one hand faster computation (since C-loops are used instead of R-loops). On the other hand, even if there is little or no gain in computational speed, vectorisation is important for readability/clarity of the code.

Maxim of Uwe Ligges

Computers are cheap, and thinking hurts.

Speeding up R – Vectorisation II

Note: Only practically all basic statements in R are vectorized. An important exception is the `if`-condition, which can only take one-dimensional numeric values. Its vectorised version is `ifelse`.

```
> x <- 1:5
> if(x<2) TRUE else FALSE
[1] TRUE
Warnmeldung:
In if (x < 2) TRUE else FALSE :
Bedingung hat Länge > 1 und nur das erste Element wird benutzt
> ifelse(x<2,TRUE,FALSE)
[1]  TRUE FALSE FALSE FALSE FALSE
```

The latter is likely what you want!

This difference can especially be important when you want to plot a function, since R will not throw an error, but only a warning. Nevertheless you might get the wrong result!

Vectorisation II

```
> scal.prod1 <- function(a,b) {
+   stopifnot(length(a)==length(b))
+   a <- as.numeric(a); b <- as.numeric(b)
+   ret <- 0
+   for (i in seq_along(a)) {
+     ret <- ret + a[i]*b[i]
+   }
+   return(ret)
+ }
>
> scal.prod2 <- function(a,b) {
+   stopifnot(length(a)==length(b))
+   a <- as.numeric(a); b <- as.numeric(b)
+   sum(a*b)
+ }
>
> n <- 1e7
> system.time(scal.prod1(1:n,n:1))
User      System verstrichen
8.78        0.08        8.91
> system.time(scal.prod2(1:n,n:1))
User      System verstrichen
0.08        0.03        0.11
```

Vectorisation II – Subscripting

Assume you want to sum up all terms of a vector, which are positive. Easy to do with a loop!

```
> set.seed(1010)
> x <- rnorm(100)
> sum.vec <- 0
>
> for (i in seq_along(x)) {
+   sum.vec <- sum.vec + x[i]*(x[i]>0)
+ }
> sum.vec
[1] 34.06589
```

However, isn't it easier to write simply the following?

```
> sum(x[x>0])
[1] 34.06589
```

Note: It is good practice to set a seed (using `set.seed()`), in order to make calculation reproducible.

Testing Equality

Testing equality of integers is done by `==`. However, care is needed if we are testing equality of `NULL`, `NA`, `Inf` and `NaN`. This is not possible employing the above operator! Here, we have to use the functions `is.null()`, `is.na()`, `is.finite()`/`is.infinite()` and `is.nan()`.

```
> NA==NA
[1] NA
> NaN==NaN
[1] NA
> identical(NA,NA)
[1] TRUE
```

`identical()` is another option of testing. But be careful! It tests objects if they are really identical! This can lead to confusion.

```
> xi <- 1:10
> yi <- 1:10
> identical(xi, yi[1:10])
[1] TRUE
> yi[11] <- 11
> identical(xi, yi[1:10])
[1] FALSE
```

```
> zi <- 1:10
> zi[11] <- 11L
> identical(xi, zi[1:10])
[1] TRUE

# In contrast...
> 10L==10
[1] TRUE
```

The problem in the above example is, that `yi` is automatically converted from integer to numeric, when a numeric value is added (by using `yi[11] <- 11`).

Some Avoidable Mistakes...

There is a whole bunch of mistakes one could do when working with R (or when programming at all). And: Many of them are avoidable! A very nice collection of advises is The R Inferno. Many examples (and hence credits) in the following sections go to this tremendous script! READ IT!

We briefly want to mention some of the topics presented there (and also some others).

First of all: **Comment and style your code properly!** Especially, this includes:

- Indent your code properly.
- Give the variables proper names (and not dummy1, dummy2, ...).
- Keep lines short and clear. Try to not let calculations go over several lines (see 'line breaks'-trap).

Some Avoidable Mistakes... (Floating point trap)

1. Floating point trap
Testing equality of numeric values can yield answers other than expected!

```
> seq(0,0.5,0.1)==0.3
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

or

```
> (uniq.vec <- unique(c(.3, .4 - .1, .5 - .2, .6 - .3, .7 - .4)))
[1] 0.3 0.3 0.3
```

since

```
> print(uniq.vec,digits=17)
[1] 0.29999999999999999 0.30000000000000004 0.29999999999999993
```

Consequence: Do not use `if(a==b)` (if `a` and `b` are numeric values), but ask for `b` to be in the neighbourhood of `a`.

```
> if(0.3 == 0.4 - 0.1) TRUE else FALSE
[1] FALSE
```

Some Avoidable Mistakes.... (Growing Objects)

2. Growing Objects

Never-ever let a data object grow dynamically, but instead allocate an object in advance of the right size! This holds for `c()`, `rbind()`, `cbind`, ...
The problem is, that R will always allocate a new object with increased size and copy all the available entries.

```
> library(tictoc)
> n <- 1e5
> tic(); vec <- numeric(0)
> for(i in 1:n) vec <- c(vec, i); toc()
18.3 sec elapsed
>
> tic(); vec <- numeric(n)
> for(i in 1:n) vec[i] <- i; toc()
0.1 sec elapsed
>
> tic(); vec <- 1:n; toc()
0 sec elapsed
```

Some Avoidable Mistakes.... (redefining existing objects)

Do not use variable/function-names which already exist in R for other purposes. E.g., it is not a good style to use `T` or `F` as variable names.

```
> TRUE <- 0
Error in TRUE <- 0 : invalid (do_set) left-hand side to assignment
> T <- 0
> T | T
[1] FALSE
```

Analogously, do not override functions like `c()` or `t()`.

In reverse, do not use `T` and `F` as logicals! `TRUE` and `FALSE` are reserved expression, which cannot be altered.

Some Avoidable Mistakes.... (line breaks)

Be careful with line breaks in R! When R calculates instructions it does it line by line - but this is only partially true! If an instruction is not completed at the end of a line, then the next line is added to the current instruction. And if an instruction is completed at the end of a line, the next line will be a new instruction. What seems to be reasonable may lead to almost impossible to find mistakes! Two scenarios (first the more harmless one, since it throws an error):

```
> if(1==1) {
+   print("Assertion is correct.")
+ }
[1] "Assertion is correct."
> else print("Assertion is incorrect.")
Error: unexpected 'else' in "else"
```

The second one can be really nasty:

```
> test.fnc <- function() {
+   some.complicated.calculation <- 1
+   which.is.too.long <- 0
+   for.one.line <- 50
+   ret <- some.complicated.calculation * which.is.too.long
+          + for.one.line
+   return(ret)
+ }
> test.fnc()
[1] 0
```

Some Avoidable Mistakes.... (line breaks)

Hopefully, you did not expect 50 to be the outcome of the function. The +-sign of course has to be in the line before. Otherwise

```
ret <- some.complicated.calculation * which.is.too.long
```

is a fully valid instruction as well as

```
+ for.one.line
```

The intended function is

```
> test.fnc <- function() {
+   some.complicated.calculation <- 1
+   which.is.too.long <- 0
+   for.one.line <- 50
+   ret <- some.complicated.calculation * which.is.too.long +
+          for.one.line
+   return(ret)
+ }
> test.fnc()
[1] 50
```

Note that we only get this problem with + and -, since these operators are also signs (* and / will throw errors)!

Some Advises Connected to attach()

Loading a package via `library()` is used to attach the namespace of the package to the R session. This is often not advisable since similar packages contain functions with the same name and hence, some are masked. In many cases it is enough to load only important packages and access functions from other packages directly via `::` (for public functions) or `:::` (for private functions).

```
# Be sure, that the package eha is installed.
> sessionInfo() # check, that eha is not attached
#  (output not shown)
> plot(dmakeham)
Fehler in plot(dmakeham) : Objekt 'dmakeham' nicht gefunden
> plot(eha::dmakeham)
```