



# Big Data Analytics

Lecture 4:

Advanced R Programming

Prof. Dr. Ulrich Matter

18/03/2021

Updates




# Schedule update

1. Introduction: Big Data, Data Economy. Walkowiak (2016): Chapter 1.
2. Computation and Memory in Applied Econometrics.
3. **Computation and Memory in Applied Econometrics II.**
4. **Advanced R Programming. Wickham (2019): Chapters 2, 3, 17,23, 24.**
5. Import, Cleaning and Transformation of Big Data. Walkowiak (2016): Chapter 3: p. 74-118.
6. Aggregation and Visualization. Walkowiak (2016): Chapter 3: p. 118-127; Wickham et al.(2015); Schwabish (2014).
7. Data Storage, Databases Interaction with R. Walkowiak (2016): Chapter 5.
8. **Cloud Computing: Introduction/Overview, Distributed Systems, Walkowiak (2016): Chapter 4.**
9. Applied Econometrics with Spark; Machine Learning and GPUs.
10. Project Presentations (7 May, 2020; 08:15-10:00; Room 23-103).
11. Project Presentations; Q&A.

# Canvas

- Use canvas discussion for issues with installation etc.
- Discussion forum is under 'Modules'.
  - Also: Stackoverflow & co.
- **Also, get ready for online lectures via Canvas/Zoom!**

# Project/Exercises teams

- GitHub classroom **Registration** assignment:  
- **Group Formation for Examination** assignment: 
  - If you you are in a team, please make sure to complete the **group assignment on GitHub Classroom**.
  - Students without team: please approach me in the break.

## Projects: More data

- <https://datasetsearch.research.google.com/>
- [Google Cloud Datasets](#)

# Goals for today

1. Understand the basics of R's memory management (with a practical big data focus).
2. Understand how data types and data structures of R objects are related to efficient memory allocation.
3. Know the basic tools and approaches to measuring and improving the performance of your R code.
4. (Review of/ideas about workflow with RStudio and GitHub for data projects.)

# Advanced R Programming



# 'Data projects' with RStudio and GitHub



*Image by [jonobacon](#) (CC BY 2.0)*

## Suggestion for set up

- Organize data analytics project as RStudio-project
- Rstudio project folder = GitHub repository
- (essentially what you will do in your group examination tasks)

# Version control with Git

- Keep track of your code.
- Develop in different branches.
- Safely go back to previous versions.

# Code repository on GitHub

- Work from different machines.
- Manage and document the project.
- Publish and collaborate.

# Names and Values

# Names and Values

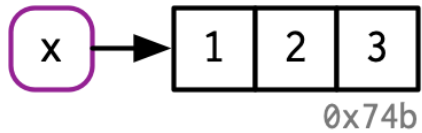
(Code examples and illustrations by ( ???), chapter 2, licensed under [CC BY-NC-SA 4.0](#))

- Prerequisites: `install.packages("lobstr")`
- Background: memory allocation and memory addresses
- 'Where' is an R object located in memory?
- How is a variable name associated with the object?
- What happens when we 'copy'/modify an object in R?

# Bindings basics

- Objects/values do not have names but **names have values!**
- Objects have a 'memory address'/identifiers.

```
x <- c(1, 2, 3)
```

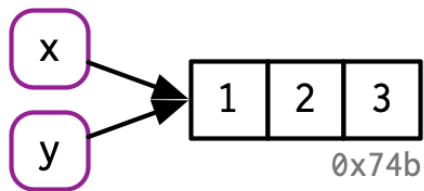




# Bindings basics

- We can 'bind' several different names to values.

```
y <- x
```



# Binding basics

- Understand the concept of names and values: check identifiers

```
obj_addr(x)
```

```
## [1] "0x7f05d01e6828"
```

```
obj_addr(y)
```

```
## [1] "0x7f05d01e6828"
```

# Copy-on-modify

- 'Copying' simply binds a new name to the same (existing) value.

```
x <- c(1, 2, 3)
```

```
y <- x
```

```
obj_addr(x)
```

```
## [1] "0x7f05d01a6ad8"
```

```
obj_addr(y)
```

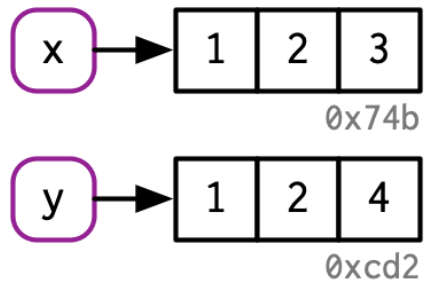
```
## [1] "0x7f05d01a6ad8"
```

# Copy-on-modify

- But if we modify values in a vector, actual 'copying' is necessary (depending on the data structure of the object...).

```
y[[3]] <- 4  
x
```

```
## [1] 1 2 3
```



# Copy-on-modify

- Understand the concept better with `tracemem()`: observe changes in identifiers.

```
x <- c(1, 2, 3)
cat(tracemem(x), "\n")
```

```
## <0x5608bffe6ee8>
```

# Copy-on-modify

- Only the first modification actually triggers the copying.

```
y <- x  
y[[3]] <- 4L
```

```
## tracemem[0x5608bffe6ee8 -> 0x5608c25234b8]: eval eval withVisible withCallingHandlers handle time
```

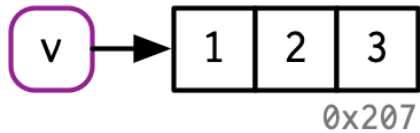
```
y[[3]] <- 5L
```

- Why?

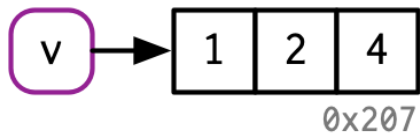
# Modify-in-place

- Objects with a single binding get modified in place (no copying needed).
- Enhances performance.

```
v <- c(1, 2, 3)
```



```
v[[3]] <- 4
```



# Modify-in-place

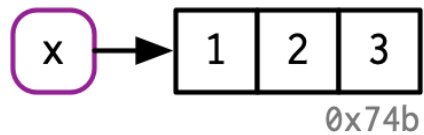
- In practice (more complex code) it is often hard to predict whether or not a copy will occur.
  - E.g., usual R functions vs. 'primitive' C functions.
- Use `tracemem()` to check your code for potential improvements (avoid unnecessary copying).



# Unbinding and the garbage collector

- What happens when we 'delete' (remove) an object?

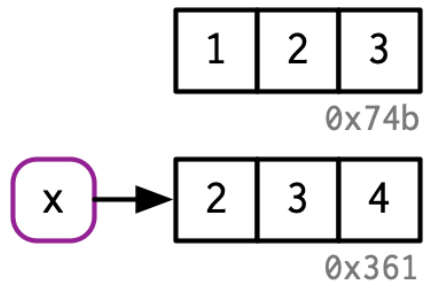
```
x <- 1:3
```



# Unbinding and the garbage collector

- What happens when we 'delete' (remove) an object?

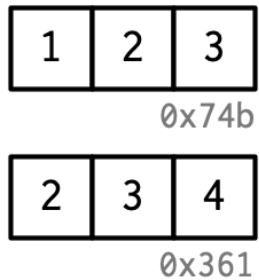
```
x <- 2:4
```



# Unbinding and the garbage collector

- What happens when we 'delete' (remove) an object?

`rm(x)`



# Unbinding and the garbage collector

- R collects the garbage automatically (but only cares about R, not other programs).
- Force garbage collection with `gc()` (OS has more memory available!).

`gc()`

##		used	(Mb)	gc trigger	(Mb)	max used	(Mb)
##	Ncells	1163080	62.2	2149366	114.8	1757805	93.9
##	Vcells	115026334	877.6	213343868	1627.7	211038278	1610.1

# Data Structures and Data Types

# R-tools to investigate structures and types

package	function	purpose
<b>utils</b>	<code>str()</code>	Compactly display the structure of an arbitrary R object.
<b>base</b>	<code>class()</code>	Prints the class(es) of an R object.
<b>base</b>	<code>typeof()</code>	Determines the (R-internal) type or storage mode of an object.

---

# Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types: integers; real numbers (floating point numbers); text ('string', 'character values').

# Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types: integers; real numbers (floating point numbers); text ('string', 'character values').
2. Basic data structures in RAM:
  - (Atomic) vectors
  - Factors
  - Arrays/Matrices
  - Lists
  - Data frames et al. (very R-specific)



# Data types: numeric

```
a <- 1.5
```

```
b <- 3
```

```
a + b
```

```
## [1] 4.5
```

# Data types: numeric

R interprets this data as type `double` (class 'numeric'):

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
object.size(a)
```

```
## 56 bytes
```

# Data types: character

```
a <- "1.5"
```

```
b <- "3"
```

```
a + b
```

# Data types: character

```
typeof(a)
```

```
## [1] "character"
```

```
class(a)
```

```
## [1] "character"
```

```
object.size(a)
```

```
## 112 bytes
```

# Data structures: vectors

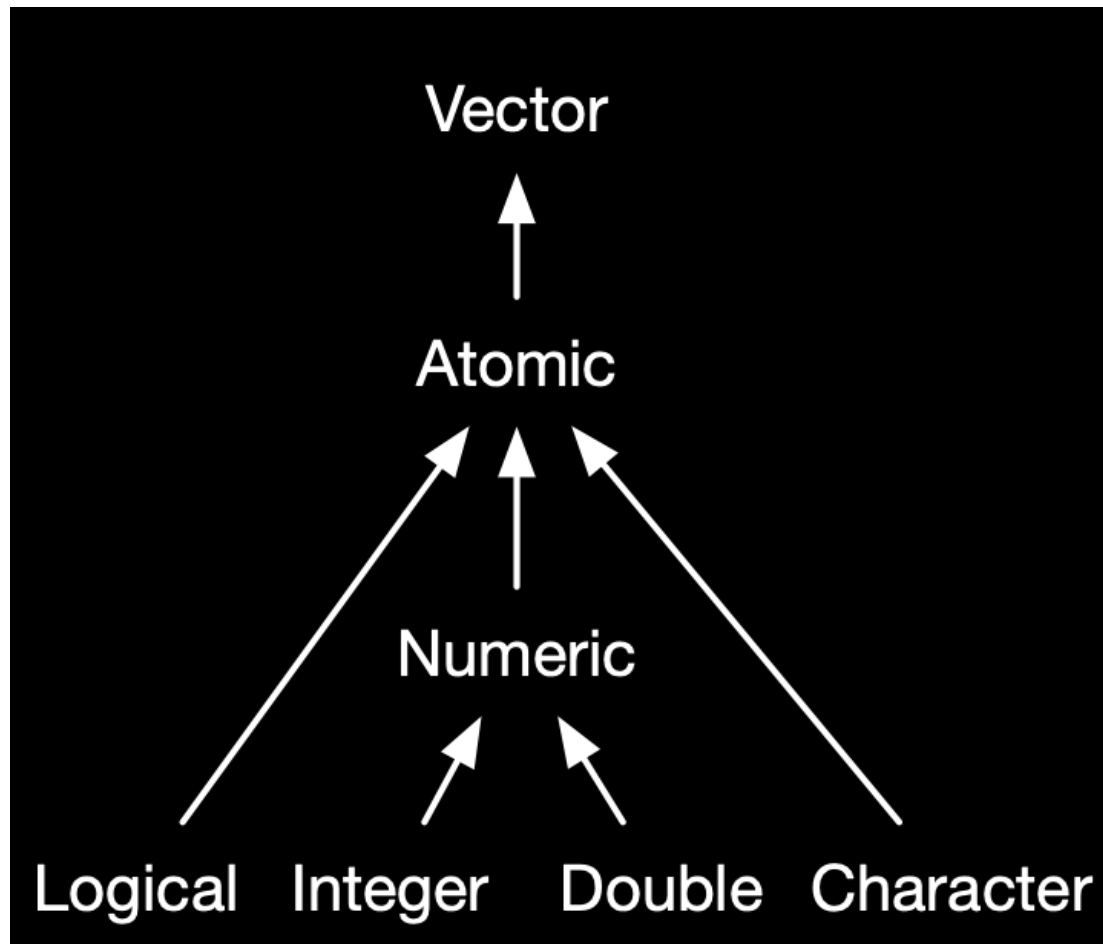


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Data structures: vectors

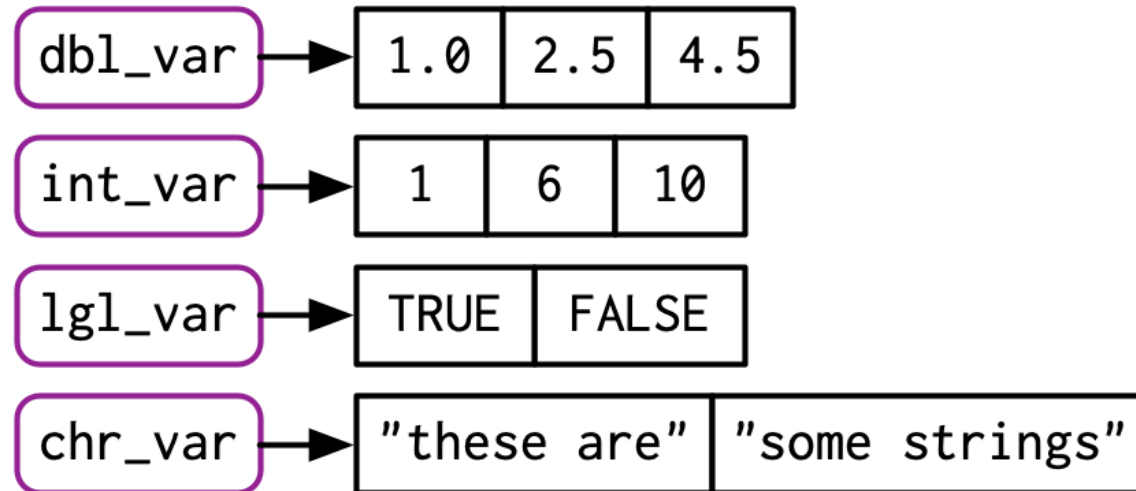


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Data structures: vectors

## Example:

```
hometown <- c("St.Gallen", "Basel", "St.Gallen")
```

```
hometown
```

```
## [1] "St.Gallen" "Basel"      "St.Gallen"
```

```
object.size(hometown)
```

```
## 200 bytes
```

# Character vectors and memory

```
x <- c("a", "a", "abc", "d")
```

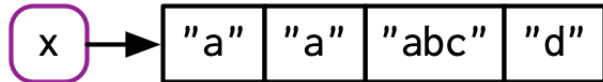


Figure by (???) (licensed under [CC BY-NC-SA 4.0](#)).



# Character vectors and memory

- R uses a global string pool where each element of a character vector is a pointer to a unique string in the pool.

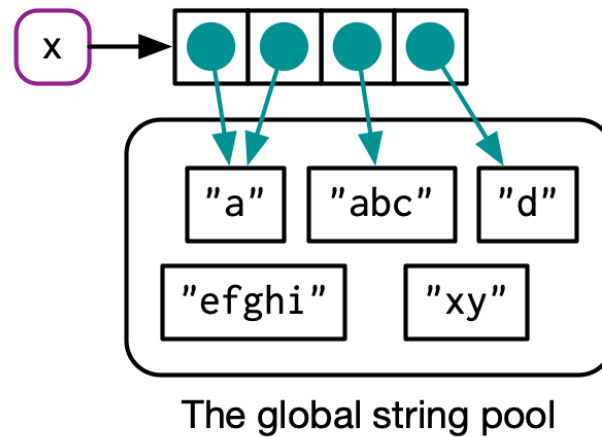


Figure by (???) (licensed under [CC BY-NC-SA 4.0](#)).

# Character vectors and memory

```
ref(x, character = TRUE)
```

```
## ■ [1:0x5608bdd2fcd8] <chr>  
## └─[2:0x5608b93fc2d0] <string: "a">  
## └─[2:0x5608b93fc2d0]  
## └─[3:0x5608c0fac888] <string: "abc">  
## └─[4:0x5608b958a090] <string: "d">
```

# Character vectors and memory

- The global string pool saves memory if a string vector is large!

```
obj_size(x)
```

```
## 248 B
```

```
obj_size(rep(x, 100))
```

```
## 3,416 B
```

# Data structures: factors

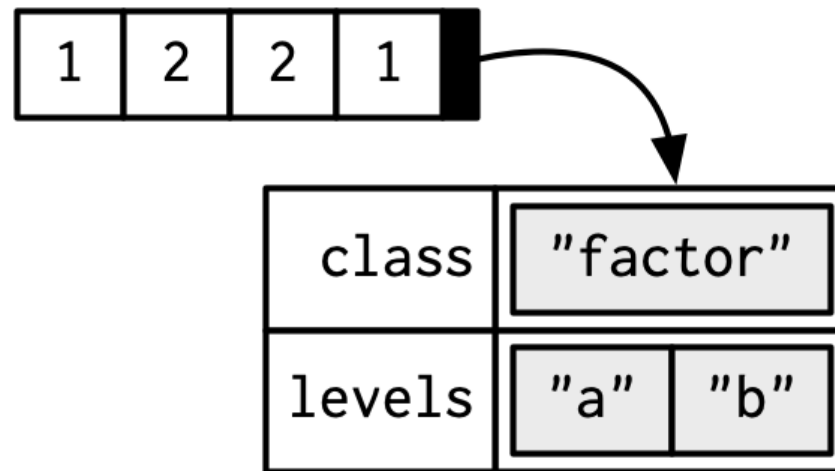


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Data structures: factors

## Example:

```
hometown_f <- factor(c("St.Gallen", "Basel", "St.Gallen"))
```

```
hometown_f
```

```
## [1] St.Gallen Basel      St.Gallen
```

```
## Levels: Basel St.Gallen
```

```
object.size(hometown_f)
```

```
## 584 bytes
```

# Data structures: Factors

- Certain 'overhead' costs: the structure stored in a factor object is also information (takes up memory)
- Similar as in previous examples: 'overhead' diminishes (relatively) with larger datasets

```
# create a large character vector  
hometown_large <- rep(hometown, times = 1000)  
# and the same content as factor  
hometown_large_f <- factor(hometown_large)  
# compare size  
object.size(hometown_large)
```

```
## 24168 bytes
```

```
object.size(hometown_large_f)
```

```
## 12568 bytes
```

# Data structures: matrices/arrays

- Like (atomic) vectors, but in 2 or more dimensions.

```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
my_array <- array(c(1,2,3,4,5,6), dim = 3)
my_array
```

```
## [1] 1 2 3
```

# Data structures: lists

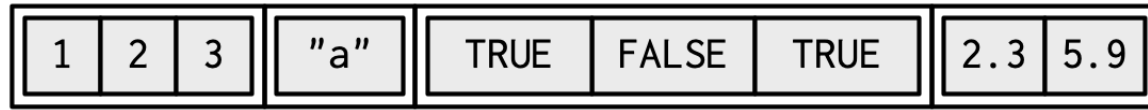


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).



# Data structures: lists

```
l1 <- list(  
  1:3,  
  "a",  
  c(TRUE, FALSE, TRUE),  
  c(2.3, 5.9)  
)
```

```
typeof(l1)
```

```
## [1] "list"
```

```
str(l1)
```

```
## List of 4  
## $ : int [1:3] 1 2 3  
## $ : chr "a"  
## $ : logi [1:3] TRUE FALSE TRUE  
## $ : num [1:2] 2.3 5.9
```

# Lists and memory

```
l1 <- list(1, 2, 3)
```

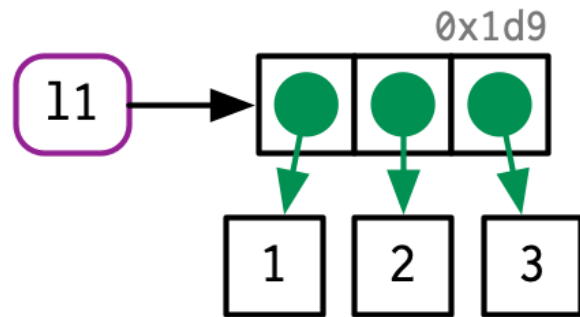


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Lists and memory

```
l2 <- l1
```

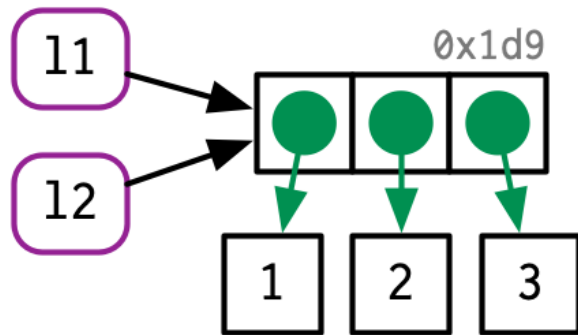


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Lists and memory

```
l2[[3]] <- 4
```

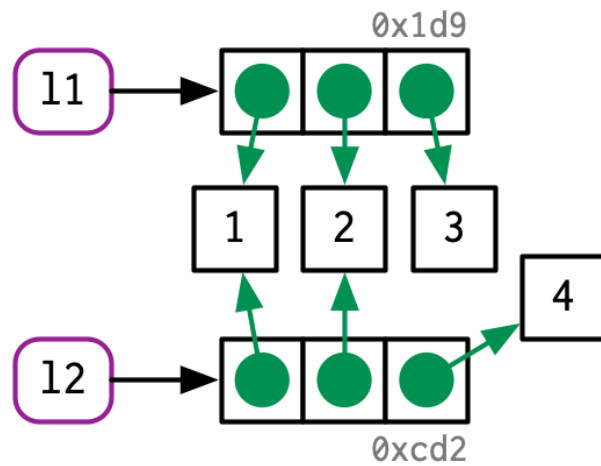


Figure by (???) (licensed under [CC BY-NC-SA 4.0](#)).

# Lists and memory

- **Shallow copy**: list object and its bindings are copied, values pointed to by the bindings not.
- Opposite of a shallow copy is a **deep copy**: contents of every reference are copied.
- Prior to R 3.1.0, copies were always deep copies!
  - 🤔🤔🤔

# Data frames, tibbles, and data tables

x	y
1	"a"
2	"b"
3	"c"

Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Data frames, tibbles, and data tables

- Classic `data.frame`

```
df <- data.frame(person = c("Alice", "Ben"),  
                 age = c(50, 30),  
                 gender = c("f", "m"))
```

df

```
##   person age gender  
## 1  Alice  50      f  
## 2   Ben  30      m
```

# Data frames, tibbles, and data tables

- `data.table`

```
library(data.table)
dt <- data.table(person = c("Alice", "Ben"),
                  age = c(50, 30),
                  gender = c("f", "m"))
```

dt

```
##      person age gender
## 1:   Alice  50      f
## 2:    Ben  30      m
```



# Data frames, tibbles, and data tables

- tibble

```
library(tibble)
tib <- tibble(person = c("Alice", "Ben"),
              age = c(50, 30),
              gender = c("f", "m"))

tib
```

```
## # A tibble: 2 x 3
##   person    age gender
##   <chr>  <dbl> <chr>
## 1 Alice     50    f
## 2 Ben      30    m
```

# Data frames and memory

```
d1 <- data.frame(x = c(1, 5, 6), y = c(2, 4, 3))
```

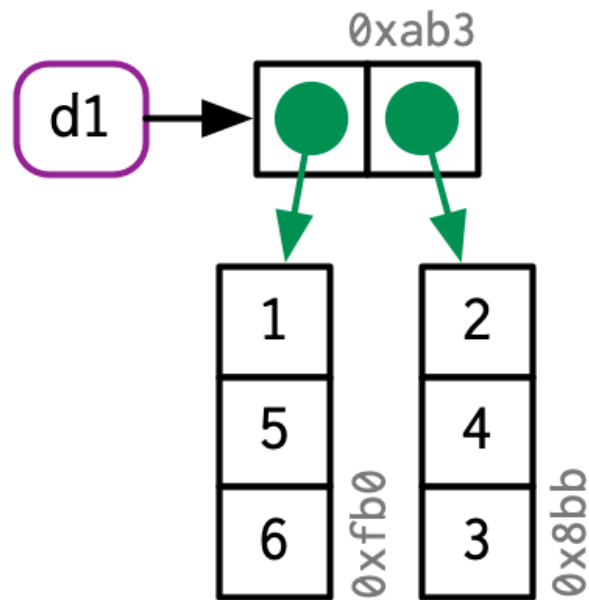


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Data frames and memory

- Modify one column: only one column needs to be copied.

```
d2 <- d1  
d2[, 2] <- d2[, 2] * 2
```

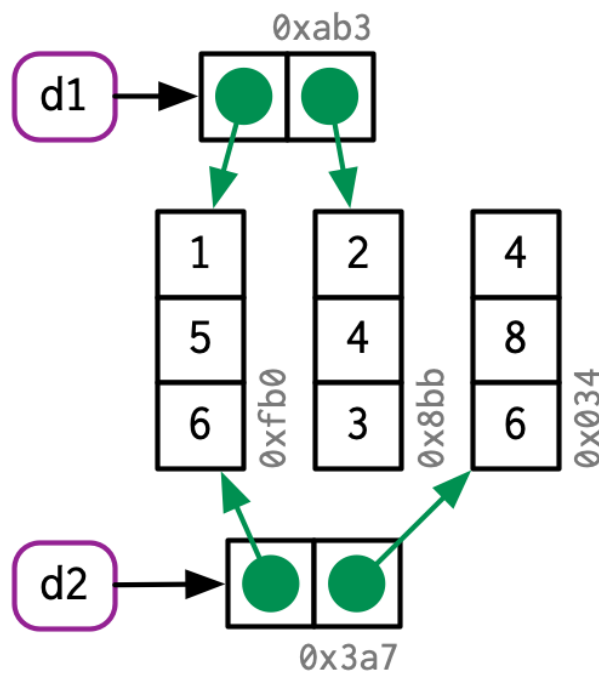


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/)).

# Data frames and memory

- Modify one row: **all** columns need to be copied.

```
d3 <- d1  
d3[1, ] <- d3[1, ] * 3
```

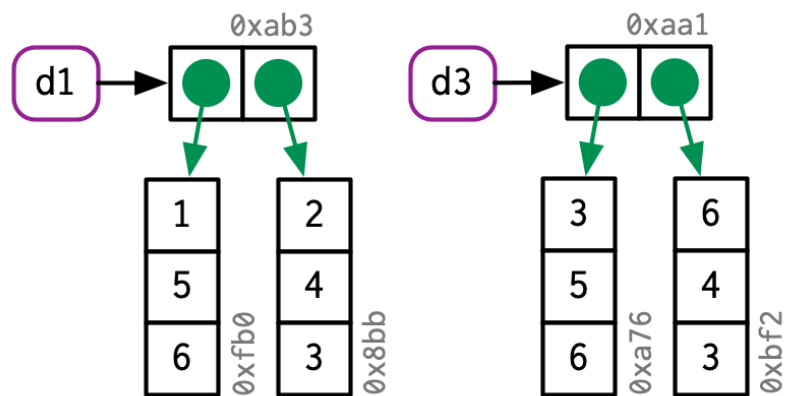


Figure by ( ??? ) (licensed under [CC BY-NC-SA 4.0](#)).

# Programming with (Big) Data in R

# Typical Programming Tasks

- Procedures to import/export data.
- Procedures to clean and filter data.
- Implement functions for statistical analysis.

# R-tools to investigate performance/resource allocation

package	function	purpose
<b>utils</b>	<code>object.size()</code>	Provides an estimate of the memory that is being used to store an R object.
<b>pryr</b>	<code>object_size()</code>	Works similarly to <code>object.size()</code> , but counts more accurately and includes the size of environments.
<b>pryr</b>	<code>compare_size()</code>	Makes it easy to compare the output of <code>object_size</code> and <code>object.size</code> .
<b>pryr</b>	<code>mem_used()</code>	Returns the total amount of memory (in megabytes) currently used by R.
<b>pryr</b>	<code>mem_change()</code>	Shows the change in memory (in megabytes) before and after running code.
<b>base</b>	<code>system.time()</code>	Returns CPU (and other) times that an R

# Building blocks for programming with big data

- Several basic functions and packages: Which one to use?
- Example: Data import.
  - `utils::read.csv()`
  - `data.table::fread()`



# Building blocks for programming with big data

```
# read a CSV-file the 'traditional way'  
flights <- read.csv("../data/flights.csv")  
class(flights)
```

```
## [1] "data.frame"
```

```
# alternative (needs the data.table package)  
library(data.table)  
flights <- fread("../data/flights.csv")  
class(flights)
```

```
## [1] "data.table" "data.frame"
```

# Building blocks for programming with big data

```
system.time(flights <- read.csv("../data/flights.csv"))
```

```
##      user  system elapsed  
##    1.214    0.000    1.215
```

```
system.time(flights <- fread("../data/flights.csv"))
```

```
##      user  system elapsed  
##    0.252    0.006    0.049
```

# Writing efficient code

- Memory allocation (before looping)
- Vectorization (different approaches)
- Beyond R

# Loops: Memory allocation before looping

*# naïve implementation*

```
sqrt_vector <-  
  function(x) {  
    output <- c()  
    for (i in 1:length(x)) {  
      output <- c(output, x[i]^(1/2))  
    }  
  
    return(output)  
  }
```

# Loops: Memory allocation before looping

```
# implementation with pre-allocation of memory
sqrt_vector_faster <-
  function(x) {
    output <- rep(NA, length(x))
    for (i in 1:length(x)) {
      output[i] <- x[i]^(1/2)
    }

    return(output)
  }
```

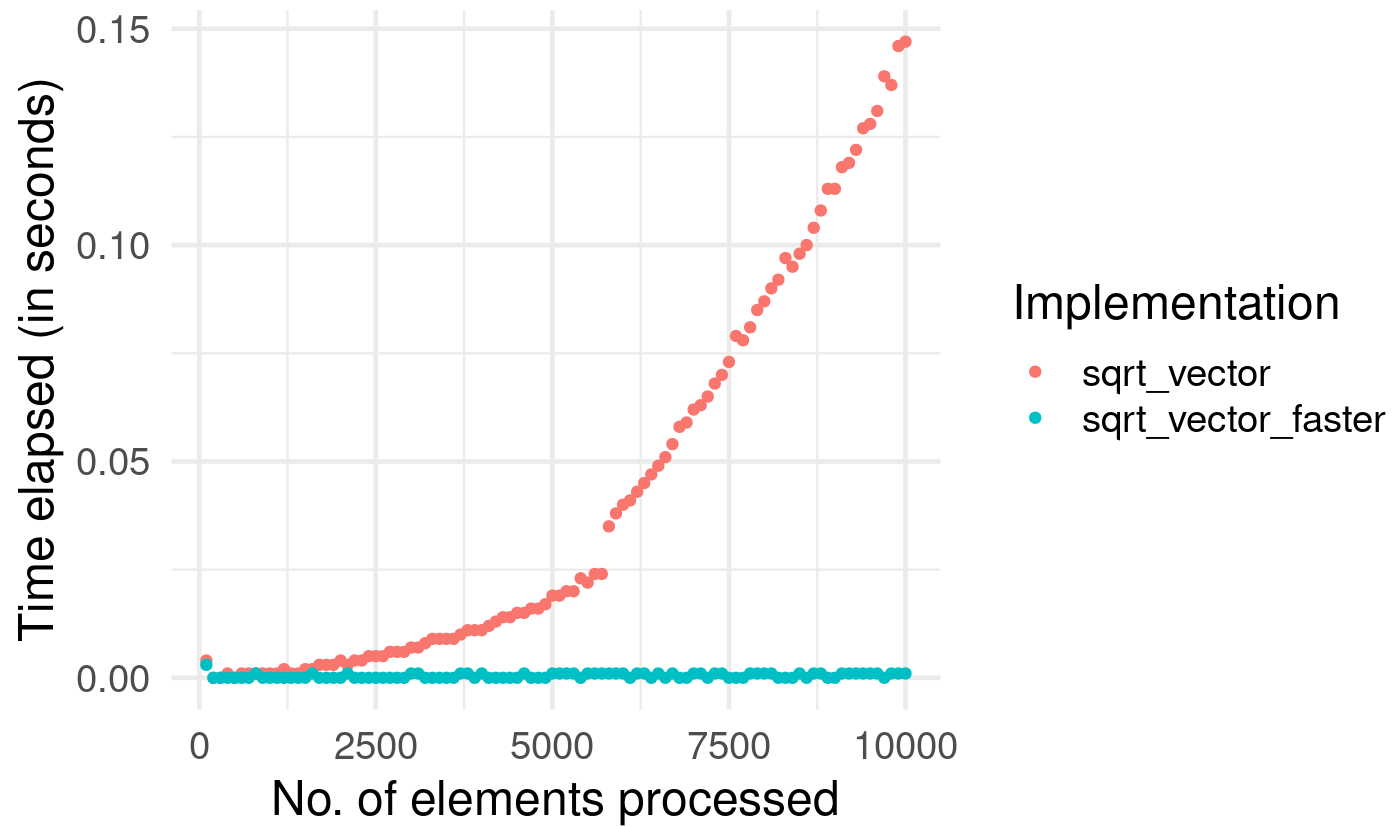
# Loops: Memory allocation before looping

## Test it!

```
# the different sizes of the vectors we will put into the two functions
input_sizes <- seq(from = 100, to = 10000, by = 100)
# create the input vectors
inputs <- sapply(input_sizes, rnorm)

# compute outputs for each of the functions
output_slower <-
  sapply(inputs,
    function(x){ system.time(sqrt_vector(x))["elapsed"]
    }
  )
output_faster <-
  sapply(inputs,
    function(x){ system.time(sqrt_vector_faster(x))["elapsed"]
    }
  )
```

# Loops: Memory allocation before looping



# Loops: Avoid unnecessary copying

- Subtract a number from each column of a large `data.frame`.
- Very slow...

```
x <- data.frame(matrix(runif(5 * 1e4), ncol = 5))  
numbers <- rnorm(5)
```

```
for (i in 1:5) {  
  x[[i]] <- x[[i]] - numbers[i]  
}
```



# Loops: Avoid unnecessary copying

- Problem: each iteration of the loop copies the `data.frame`.
- Copying means additional memory allocation.

```
cat(tracemem(x), "\n")
```

```
## <0x5608c1307d18>
```

```
for (i in 1:5) {  
  x[[i]] <- x[[i]] - numbers[i]  
}
```

```
## tracemem[0x5608c1307d18 -> 0x5608c08988e8]: eval eval withVisible withCallingHandlers handle time  
## tracemem[0x5608c08988e8 -> 0x5608c086cdb8]: [[<- .data.frame [[<- eval eval withVisible withCallir  
## tracemem[0x5608c086cdb8 -> 0x5608c086d288]: eval eval withVisible withCallingHandlers handle time  
## tracemem[0x5608c086d288 -> 0x5608c086d6e8]: [[<- .data.frame [[<- eval eval withVisible withCallir  
## tracemem[0x5608c086d6e8 -> 0x5608c086dc98]: eval eval withVisible withCallingHandlers handle time  
## tracemem[0x5608c086dc98 -> 0x5608c086e0f8]: [[<- .data.frame [[<- eval eval withVisible withCallir  
## tracemem[0x5608c086e0f8 -> 0x5608c0829d78]: eval eval withVisible withCallingHandlers handle time  
## tracemem[0x5608c0829d78 -> 0x5608c0829fa8]: [[<- .data.frame [[<- eval eval withVisible withCallir  
## tracemem[0x5608c0829fa8 -> 0x5608c082a478]: eval eval withVisible withCallingHandlers handle time  
## tracemem[0x5608c082a478 -> 0x5608c082a5c8]: [[<- .data.frame [[<- eval eval withVisible withCallir
```

# Loops: Avoid unnecessary copying

- Solution: store data (columns) in list.
- Uses internal C code and avoids additional copies.

```
y <- as.list(x)
```

```
## tracemem[0x5608c082a5c8 -> 0x5608c0ae7038]: as.list.data.frame as.list eval eval withVisible with
```

```
cat(tracemem(y), "\n")
```

```
## <0x5608c0ae7038>
```

```
for (i in 1:5) {  
  y[[i]] <- y[[i]] - numbers[i]  
}
```

```
## tracemem[0x5608c0ae7038 -> 0x5608c0a19668]: eval eval withVisible withCallingHandlers handle timi
```

# Vectorization

- “In R, everything is a vector...”
- Directly operate on vectors, not elements.
- Avoid unnecessary repetition of ‘preparatory steps’.

# Vectorization: Example

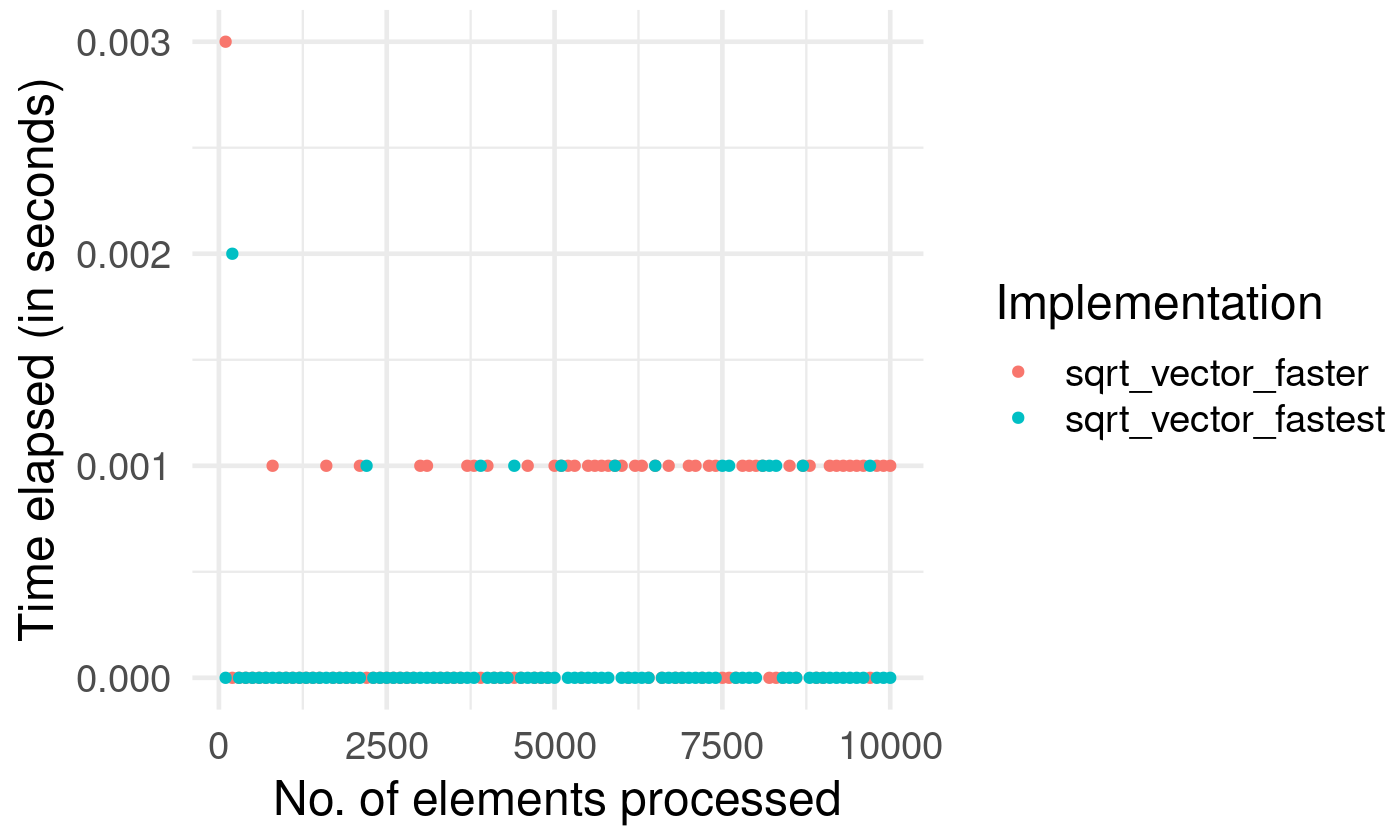
*# implementation with vectorization*

```
sqrt_vector_fastest <-  
  function(x) {  
    output <- x^(1/2)  
    return(output)  
  }
```

*# speed test*

```
output_fastest <-  
  sapply(inputs,  
    function(x){ system.time(sqrt_vector_fastest(x))["elapsed"]  
    },  
    )
```

# Vectorization: Example



## Vectorization: `apply`-type functions vs loops

- Apply a function to each element of a vector/list.
- For example, `lapply()`.

# Example

- Read several data files into R.
- Example data source: [Health News in Twitter Data Set](#) by Karami et al. (2017).
- Loop vs `lapply()`, vs `Vectorization()`

# Example: Preparations

```
# load packages  
library(data.table)  
  
# get a list of all file-paths  
textfiles <- list.files("../data/twitter_texts", full.names = TRUE)
```



## Example: for-loop approach

```
# prepare loop
all_texts <- list()
n_files <- length(textfiles)
length(all_texts) <- n_files
# read all files listed in textfiles
for (i in 1:n_files) {
  all_texts[[i]] <- fread(textfiles[i])
}
```

# Example: for-loop approach

## Check the results

```
# combine all in one data.table  
twitter_text <- rbindlist(all_texts)  
# check result  
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:  
## $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 58579  
## $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1  
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - E  
## - attr(*, ".internal.selfref")=<externalptr>
```

# Example: lapply approach

```
# prepare loop
```

```
all_texts <- lapply(textfiles, fread)
```

```
# combine all in one data.table
```

```
twitter_text <- rbindlist(all_texts)
```

```
# check result
```

```
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:
```

```
## $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 58579
```

```
## $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1
```

```
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - E
```

```
## - attr(*, ".internal.selfref")=<externalptr>
```

# Example: Vectorization approach

*# initiate the import function*

```
import_file <-  
  function(x) {  
    parsed_x <- fread(x)  
    return(parsed_x)  
  }
```

*# 'vectorize' it*

```
import_files <- Vectorize(import_file, SIMPLIFY = FALSE)
```

# Example: Vectorization approach

```
# Apply the vectorized function
```

```
all_texts <- import_files(textfiles)
```

```
twitter_text <- rbindlist(all_texts)
```

```
# check the result
```

```
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':  42422 obs. of  3 variables:
```

```
## $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 58579
```

```
## $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:1
```

```
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - E
```

```
## - attr(*, ".internal.selfref")=<externalptr>
```

# Profiling and Benchmarking

# Profiling

- Use a 'profiler' to understand code performance.
- Get an overview over which parts of a program need how much memory and how much execution time.

# Profiling with profvis

A simple nested function (with clearly defined execution time):

```
# implement function
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```



# Profiling with profvis

```
# load package with profiler  
library(profvis)  
# get performance profile of function  
profvis(f())
```

## Benchmarking with `bench::mark()`

- Alternative tool to measure execution time (see `microbenchmark` in previous lectures)
- Recall: execution time is not deterministic (it comes with statistical error).
- Benchmarking means running the code several times to get a distribution of execution times.

# Benchmarking with `bench::mark()`

```
# load package
```

```
library(bench)
```

```
# run squareroot example
```

```
# primitive (C) sqrt vs. 'own implementation'
```

```
x <- runif(100)
```

```
(lb <- bench::mark(
```

```
  sqrt(x),
```

```
  x ^ 0.5,
```

```
  memory = FALSE
```

```
))
```

```
## # A tibble: 2 x 6
```

```
##   expression      min    median `itr/sec` mem_alloc `gc/sec`
```

```
##   <bch:expr> <bch:tm> <bch:tm>      <dbl> <bch:byt>      <dbl>
```

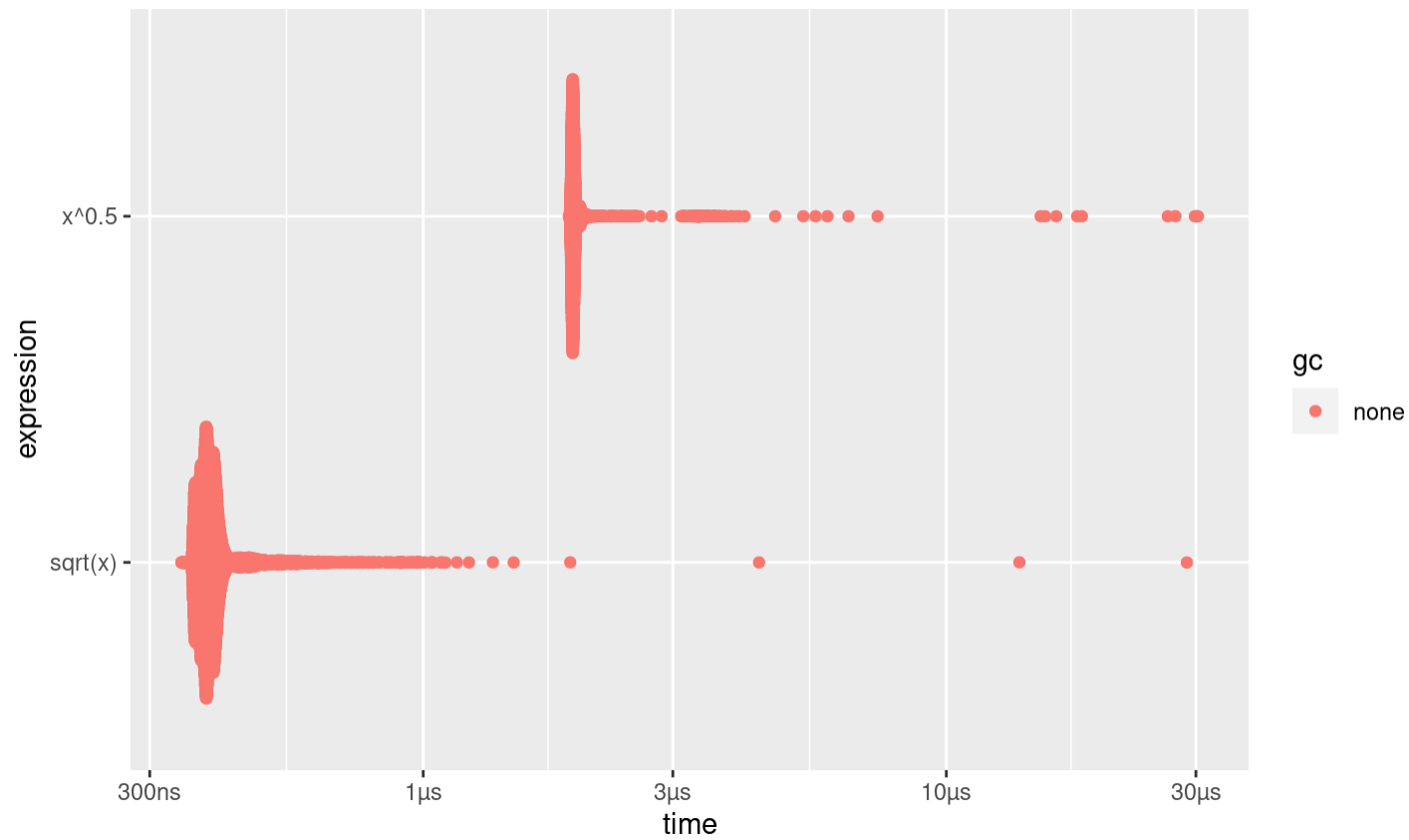
```
## 1 sqrt(x)    344.9ns  387.9ns  2473938.      NA          0
```

```
## 2 x^0.5      1.9µs   1.94µs   503260.      NA          0
```

# Benchmarking with `bench::mark()`

```
plot(lb)
```

```
## Loading required namespace: tidyr
```



Improving Performance

# Improving performance

- Bottleneck(s) identified, what now?
- See previous examples for typical problems in a data analytics context.
- Vast variety of potential bottlenecks. Hard to give general advice.

# Programming with Big Data

1. Which basic (already implemented) R functions are more or less suitable as building blocks for the program?
  2. How can we exploit/avoid some of R's lower-level characteristics in order to implement efficient functions?
  3. Is there a need to interface with a lower-level programming language in order to speed up the code? (advanced topic)
- Independent of **how** we write a statistical procedure in R (or in any other language, for that matter), is there an **alternative statistical procedure/algorithm** that is faster but delivers approximately the same result.

# Issues to keep in mind

- Vectorization.
- Memory: avoid copying, pre-allocate memory.
- Use built in primitive (C) functions (caution: not always faster, if aim is precision).
- Existing solutions: load additional packages (`read.csv()` vs. `data.table::fread()`).
  - Focus of what follows in this course (approach taken in Walkowiak (2016)).



# Procedural view and further reading

- Consider Hadley's advice: ( ???): Chapter 24
- Experienced coder? Have a look at [R Inferno](#)
- Further reading after this course: [The Art of R Programming](#)

## R, beyond R

- For advanced programmers, R offers various options to directly make use of compiled programs (for example, written in C, C++, or FORTRAN).
- Several of the core R functions are implemented in one of these lower-level programming languages.

# R, beyond R

## Have a look at a function's source code!

```
import_file
```

```
## function(x) {  
##     parsed_x <- fread(x)  
##     return(parsed_x)  
## }  
## <bytecode: 0x5608c098ea68>
```

# R, beyond R

Have a look at a function's source code!

```
sum
```

```
## function (... , na.rm = FALSE) .Primitive("sum")
```

# References

- Karami, Amir, Aryya Gangopadhyay, Bin Zhou, and Hadi Kharrazi. 2017. "Fuzzy Approach Topic Discovery in Health and Medical Corpora." *International Journal of Fuzzy Systems* 20 (4): 1334–45.
- Walkowiak, Simkon. 2016. *Big Data Analytics with R*. Birmingham, UK: PACKT Publishing.