

# (A bit of) Advanced R

Part 1 - towards better R-base programming

Julien Chiquet

<http://github/jchiquet/CourseAdvancedR>

Université Paris Dauphine, Juin 2018



# Outline I

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Outline II

- ⑧ Interface with lower-level languages

# References

- R Core Team (2017): A Language and Environment for Statistical Computing  
<https://www.R-project.org/>
- Wickham (2014): Advanced R, retrieved from <http://adv-r.had.co.nz/>
- Gillespie & Lovelace (2016): efficient R programming  
<https://bookdown.org/csgillespie/efficientR/>

# Prerequisites

## Data Structure in base R

- ① Atomic vector (integer, double, logical, character)
- ② Recursive vector (list)
- ③ Factors
- ④ Matrices and array
- ⑤ Data Frame

→ Creation, Basic Operation, Manipulation, Representation

## Resources

- Advanced R, chapters 1.2, 1.3 (Wickham, 2014, <http://adv-r.had.co.nz/>)
- An introduction to R programming  
[http://julien.cremeriefamily.info/teachings\\_L3BI\\_ISV51.html](http://julien.cremeriefamily.info/teachings_L3BI_ISV51.html)

# Outline

- 1 Function, Functionals
- 2 Good and bad practices in R
- 3 Benchmark your code
- 4 Use all your cores when needed
- 5 Tricks
- 6 Remember that R is object oriented
- 7 Mind your vocabulary: R fast built in function

# The [a-z]\*pply family I

## Example with factors (tapply)

```
data <- rnorm(100)
sexe <- factor( sample(c("H","F"), 100, replace = TRUE))
mean.1 <- tapply(data, sexe, mean) ## good
mean.2 <- c() ## complicated
for (l in levels(sexe))
  mean.2 <- c(mean.2, mean(data[sexe == l]))
```

## Example with list or data.frame (sapply/lapply)

# The [a-z]\*pply family II

```
data(oats, package = "MASS")  
oats[1:2, ]
```

```
##      B      V      N      Y  
## 1 I Victory 0.0cwt 111  
## 2 I Victory 0.2cwt 130
```

```
sapply(oats, is.factor) ## readable
```

```
##      B      V      N      Y  
## TRUE TRUE TRUE FALSE
```

```
for (c in 1:ncol(oats)) ## less readable (I think)  
  print(is.factor(oats[,c]))
```

```
## [1] TRUE  
## [1] TRUE  
## [1] TRUE  
## [1] FALSE
```



# The `do.call` function I

*constructs and executes a function call from a name or a function and a list of arguments to be passed to it*

Suppose you have the outputs of 100 simulations at your disposal, stored in a list like that

```
res <- replicate(100, rbind(data.frame(method="lasso", mse=runif(1,.75,1) , timing=runif(1)),
                             data.frame(method="ridge", mse=runif(1,.5,.75), timing=runif(1)),
                             data.frame(method="bayes", mse=runif(1,.85,1) , timing=runif(1,100),
                             ), simplify=FALSE)
```

```
class(res)
```

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

```
res[[1]]
```

```
##      method      mse      timing
## 1  lasso 0.7746281  0.7198101
## 2  ridge 0.5630359  0.6188649
## 3  bayes 0.8553590 173.4643823
```

```
length(res)
```

```
## [1] 100
```

# The `do.call` function II

How would you store them in a single data frame?

```
all.res <- do.call(rbind, res)
dim(all.res)
```

```
## [1] 300  3
```

# The Reduce function

*'Reduce' uses a binary function to successively combine the elements of a given vector*

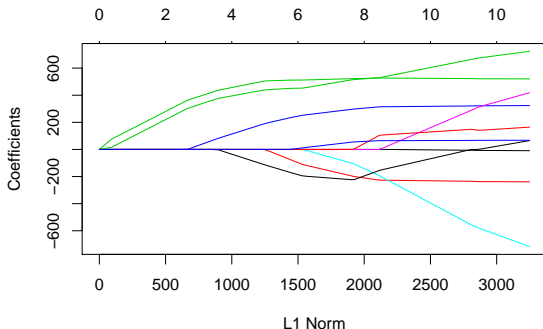
↪ can be use to post-process your list of simulations obtained via `mclapply` just like `do.call`

Say more... (map, Reduce)

# A Reduce example: “jackknifing” a lasso solution path

A single Lasso fit of the diabetes data set

```
library(glmnet)
library(lars) # the diabetes data set (part of the lars package)
data(diabetes)
y <- diabetes$y
x <- diabetes$x
n <- length(y)
lasso <- glmnet(x,y)
plot(lasso)
```



# A Reduce example: “jackknifing” a lasso solution path II I

Compute the regularization paths for all subsets, removing one individual at once

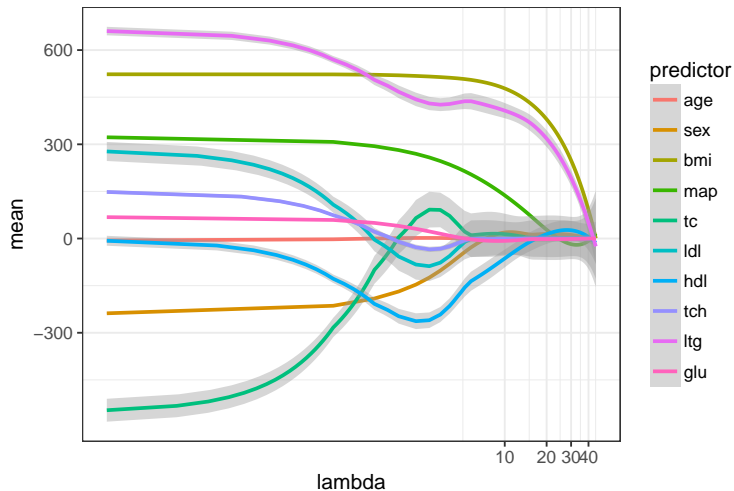
```
paths <- parallel::mclapply(1:n, function(i) {  
  glmnet(x[-i, ], y[-i], lambda = lasso$lambda)$beta  
}, mc.cores = 4)
```

Computing the envelop around the average regularization path with Reduce

```
mean.path <- Reduce("+", paths)/n  
sdev.path <- sqrt(Reduce("+", lapply(paths, function(path) path**2))/n -  
  mean.path**2)
```

```
library(ggplot2)  
library(reshape2)  
mean.path <- t(as.matrix(mean.path)); rownames(mean.path) <- lasso$lambda  
sdev.path <- t(as.matrix(sdev.path)); rownames(sdev.path) <- lasso$lambda  
dplot <- cbind(reshape2::melt(mean.path), reshape2::melt(sdev.path)[, 3])  
colnames(dplot) <- c("lambda", "predictor", "mean", "sdev")  
ggobj <- ggplot(dplot, aes(x=lambda, y=mean, group=predictor, color=predictor)) + geom_smooth(a  
print(ggobj + coord_trans(x="log10"))
```

## A Reduce example: “jackknifing” a lasso solution path II II



# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Vectorize any algebraic operation I

**Example:** compute  $\exp(x) = \sum_{k=0}^n \frac{x^k}{k!}$

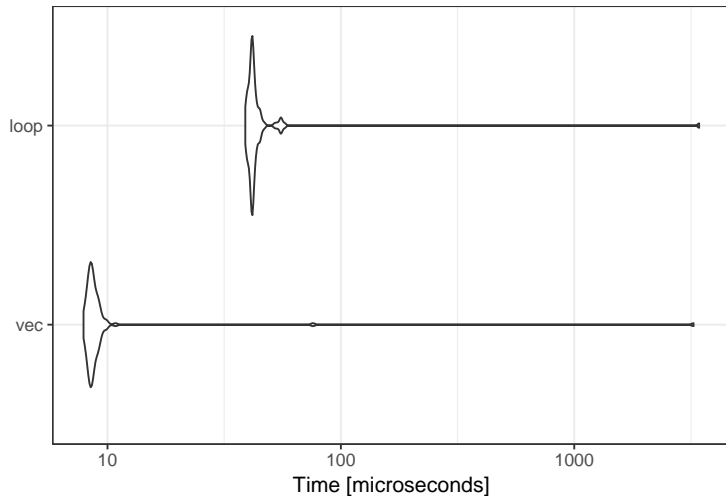
```
## the good way
exp_vec <- function(x, n){
  res <- sum(x^(0:n)/c(1,cumprod(1:n)))
  res
}

## the sad/bad/less readable way
exp_loop <- function(x, n){
  res <- 1
  for (k in 1:n) res <- res + 2^k/factorial(k)
  res
}

autoplot(microbenchmark(vec = exp_vec(2, 100), loop = exp_loop(2, 100)))
```



## Vectorize any algebraic operation II



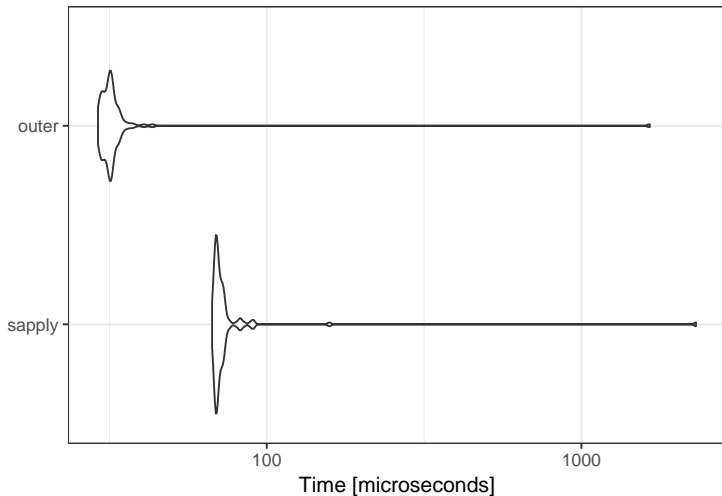
# Vectorize, even for non-algebraic operation I

```
month_year_apply <- function(year) {  
  sapply(month.name, function(month) paste(month, year, sep = "_"))  
}  
  
month_year_outer <- function(year) {  
  outer(month.name, year, FUN = paste, sep = '_')  
}  
head(month_year_outer(c(2010, 2013)), 3)
```

```
##      [,1]      [,2]  
## [1,] "January_2010" "January_2013"  
## [2,] "February_2010" "February_2013"  
## [3,] "March_2010"    "March_2013"
```

```
autoplot(microbenchmark(  
  sapply = month_year_apply(c(2011, 2013)),  
  outer  = month_year_outer(c(2011, 2013)),  
  times = 100))
```

## Vectorize, even for non-algebraic operation II

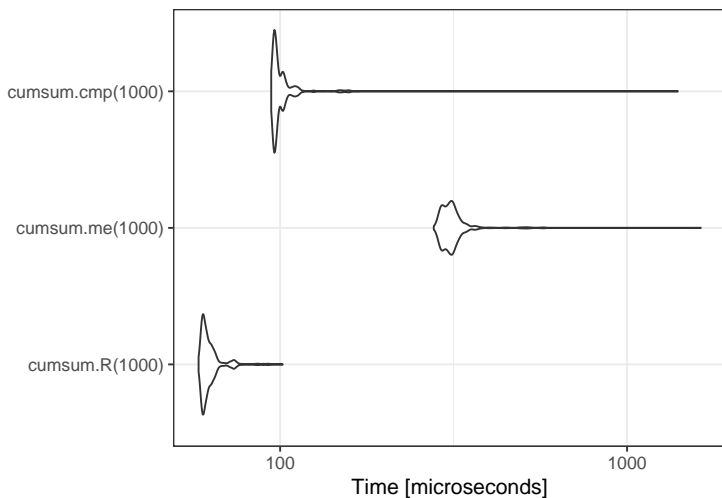


# Compile your functions with `base::compiler` I

If you cannot avoid a loop, you will save some time

```
cumsum.R <- function(n) {  
  x <- rnorm(n)  
  cumsum(x)  
}  
  
cumsum.me <- function(n) {  
  x <- rnorm(n)  
  res <- 0  
  for (i in 1:length(x))  
    res <- res + x[i]  
  res  
}  
  
cumsum.cmp <- compiler::cmpfun(cumsum.me)  
  
autoplot(  
  microbenchmark(  
    cumsum.R(1000),  
    cumsum.me(1000),  
    cumsum.cmp(1000),  
    times=1000)  
)
```

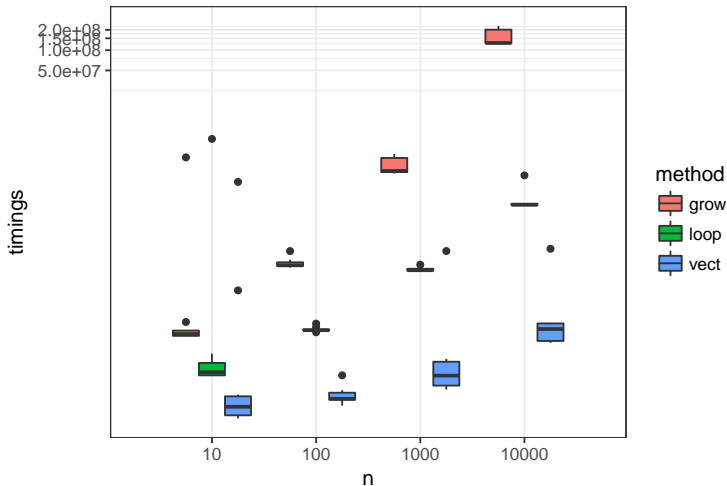
# Compile your functions with `base::compiler` II



~> Can be set automatically with `compiler::enableJIT(3)`

# Preallocate whenever it is possible

```
grow <- function(n) {vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)}  
loop <- function(n) {vec <- numeric(n); for (i in 1:n) vec[i] <- i}  
vect <- function(n) {1:n}
```



# Do not stack objects I

Even if it is tempting when the final size is unknown.

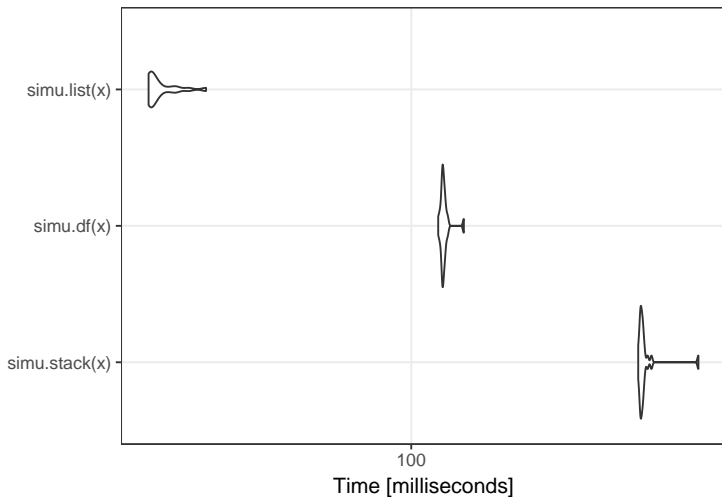
```
simu.stack <- function(x) { ## x is a n x p matrix
  out <- data.frame(mean = numeric(0), sd = numeric(0))
  for (i in 1:n)
    out <- rbind(out, data.frame(mean = mean(x[i,]), sd = sd(x[i, ])) )
  return(out)
}
```

```
simu.df <- function(x) {
  out <- data.frame(mean = numeric(n), sd = numeric(n))
  for (i in 1:n)
    out[i, ] <- c(mean = mean(x[i,]), sd = sd(x[i, ]))
  return(out)
}
```

```
simu.list <- function(x) {
  my.list <- lapply(1:n, function(i) c(mean(x[i,]), sd(x[i, ])))
  out <- data.frame(do.call(rbind, my.list))
  colnames(out) <- c("mean", "sd")
  return(out)
}
```

```
n <- 1000; p <- 10; x <- matrix(rnorm(n*p), n, p)
autoplot(microbenchmark(simu.stack(x), simu.df(x), simu.list(x), times=20))
```

# Do not stack objects II





# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ **Benchmark your code**
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Quick (and dirty) benchmarking with `system.time()`

One usually relies on the command `system.time(expr)` to evaluate the timings:

```
func.one <- function(n) {return(rnorm(n,0,1))}  
func.two <- function(n) {return(rpois(n,1))}  
  
n <- 1000  
system.time(replicate(100, func.one(n)))
```

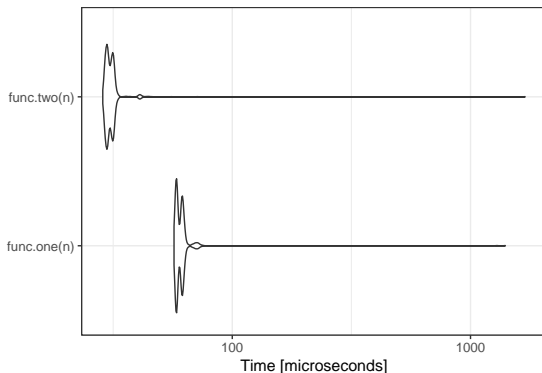
```
##      user  system elapsed  
## 0.009   0.000   0.009
```

```
system.time(replicate(100, func.two(n)))
```

```
##      user  system elapsed  
## 0.006   0.000   0.006
```

# Quick benchmarking with microbenchmark

```
func.one <- function(n) {return(rnorm(n,0,1))}  
func.two <- function(n) {return(rpois(n,1))}  
  
library(microbenchmark)  
  
n <- 1000  
res <- microbenchmark(func.one(n), func.two(n), times=1000)  
ggplot2::autoplot(res)
```



# Profile your code

Suppose you want to evaluate which part of the following function is hot:

```
## generate data, center/scale and perform ridge regression
my_func <- function(n,p) {

  require(MASS)

  ## draw data
  x <- matrix(rnorm(n*p),n,p)
  y <- rnorm(n)

  ## center/scale
  xs <- scale(x)
  ys <- y - mean(y)

  ## return ridge's coefficients
  ridge <- lm.ridge(ys~xs+0,lambda=1)

  return(ridge$coef)
}
```

# Profile your code with base Rprof |

One can rely on the default Rprof function, with somewhat technical outputs

```
Rprof(file="profiling.out", interval=0.05)
res <- my_func(1000,500)
Rprof(NULL)
```

```
summaryRprof("profiling.out")$by.self
```

##	self.time	self.pct	total.time	total.pct
## "La.svd"	0.90	69.23	0.90	69.23
## "[.data.frame"	0.10	7.69	0.10	7.69
## "aperm.default"	0.10	7.69	0.10	7.69
## "is.na"	0.10	7.69	0.10	7.69
## "as.matrix"	0.05	3.85	0.05	3.85
## "lazyLoadDBfetch"	0.05	3.85	0.05	3.85

```
summaryRprof("profiling.out")$by.total
```

# Profile your code with base Rprof II

##	total.time	total.pct	self.time	self.pct
## "block_exec"	1.30	100.00	0.00	0.00
## "call_block"	1.30	100.00	0.00	0.00
## "eval"	1.30	100.00	0.00	0.00
## "evaluate_call"	1.30	100.00	0.00	0.00
## "evaluate::evaluate"	1.30	100.00	0.00	0.00
## "evaluate"	1.30	100.00	0.00	0.00
## "FUN"	1.30	100.00	0.00	0.00
## "handle"	1.30	100.00	0.00	0.00
## "in_dir"	1.30	100.00	0.00	0.00
## "knit"	1.30	100.00	0.00	0.00
## "knitr::knit"	1.30	100.00	0.00	0.00
## "lapply"	1.30	100.00	0.00	0.00
## "my_func"	1.30	100.00	0.00	0.00
## "process_file"	1.30	100.00	0.00	0.00
## "process_group.block"	1.30	100.00	0.00	0.00
## "process_group"	1.30	100.00	0.00	0.00
## "rmarkdown::render"	1.30	100.00	0.00	0.00
## "timing_fn"	1.30	100.00	0.00	0.00
## "withCallingHandlers"	1.30	100.00	0.00	0.00
## "withVisible"	1.30	100.00	0.00	0.00
## "lm.ridge"	1.05	80.77	0.00	0.00
## "La.svd"	0.90	69.23	0.90	69.23
## "svd"	0.90	69.23	0.00	0.00
## "scale.default"	0.25	19.23	0.00	0.00
## "scale"	0.25	19.23	0.00	0.00
## "eval.parent"	0.15	11.54	0.00	0.00

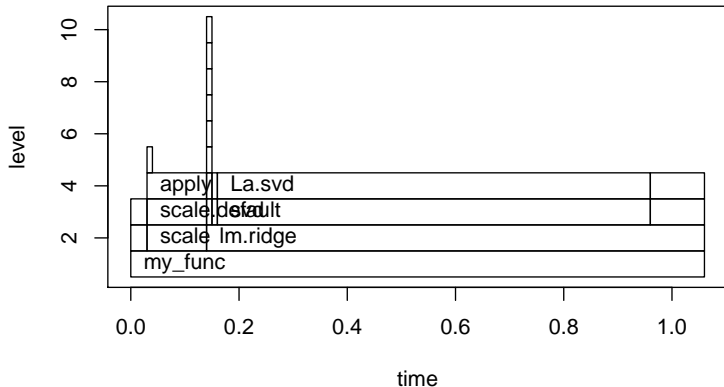
# Profile your code with base Rprof III

## "model.frame.default"	0.15	11.54	0.00	0.00
## "stats::model.frame"	0.15	11.54	0.00	0.00
## "[.data.frame"	0.10	7.69	0.10	7.69
## "aperm.default"	0.10	7.69	0.10	7.69
## "is.na"	0.10	7.69	0.10	7.69
## ".External2"	0.10	7.69	0.00	0.00
## "["	0.10	7.69	0.00	0.00
## "aperm"	0.10	7.69	0.00	0.00
## "apply"	0.10	7.69	0.00	0.00
## "na.omit.data.frame"	0.10	7.69	0.00	0.00
## "na.omit"	0.10	7.69	0.00	0.00
## "sweep"	0.10	7.69	0.00	0.00
## "as.matrix"	0.05	3.85	0.05	3.85
## "lazyLoadDBfetch"	0.05	3.85	0.05	3.85

# Profile your code with profr

The *profr* package is maybe a little easier to understand...

```
library(profr)
profiling <- profr({my_func(1000,500)}, interval = 0.01)
plot(profiling)
```

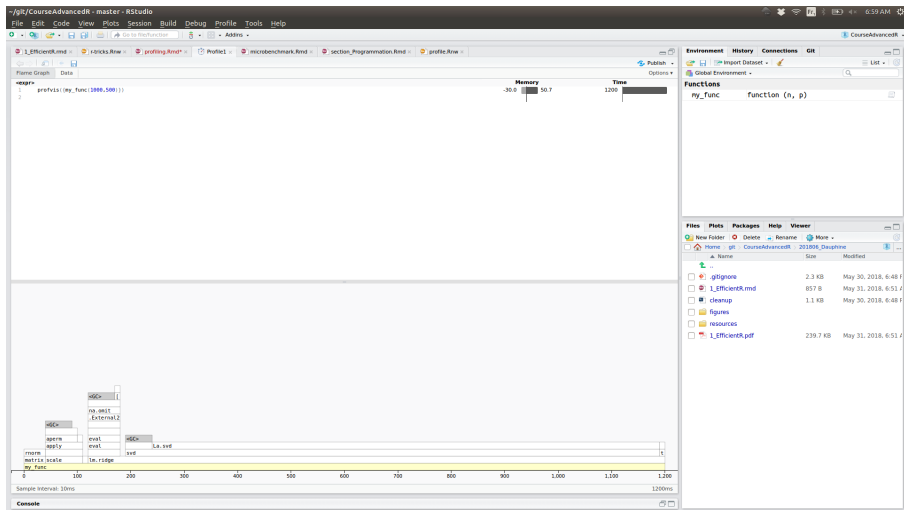




# Profile your code within R Studio with profvis

Profvis integrates the profiling to the Rstudio API

```
library(profvis)
profvis({my_func(1000,500)})
```



# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Parallel computing

## Usual Roadmap

- ① Start up and initialize  $M$  'worker' processes
- ② Send data required for each task to the workers
- ③ Split the task into  $M$  roughly equally-sized chunks and send them (including the R code needed) to the workers
- ④ Wait for all the workers to complete their tasks, and ask them for their results
- ⑤ Repeat steps (2–4) for any further tasks
- ⑥ Shut down the worker processes

# Socketing vs Forking

Two approaches achieving the same goal

## The socket approach

- launches a new version of R on each core
- connection is done via networking all happening on your own computer

## The forking approach

- copies the entire current version of R and moves it to a new core
- several processes achieve the same task resulting in different outputs

~> Forking is only possible on Unix systems (Linux, Mac OS)

# Parallel computing with parallel

## Package parallel

- merge of packages multicore and snow
- included in base R and maintained by the R Core team

## Check your computer

```
library(parallel) ## embedded with R since version 2.9 or something
cores <- detectCores() ## How many cores do I have?
print(cores)
```

```
## [1] 12
```

→ parallel features both socketing (parLapply) and forking (mclapply)

# Forking approach with `parallel::mclapply`

Very easy: use parallel features as soon as you do simulations !

Example: estimates the test error from ridge regression

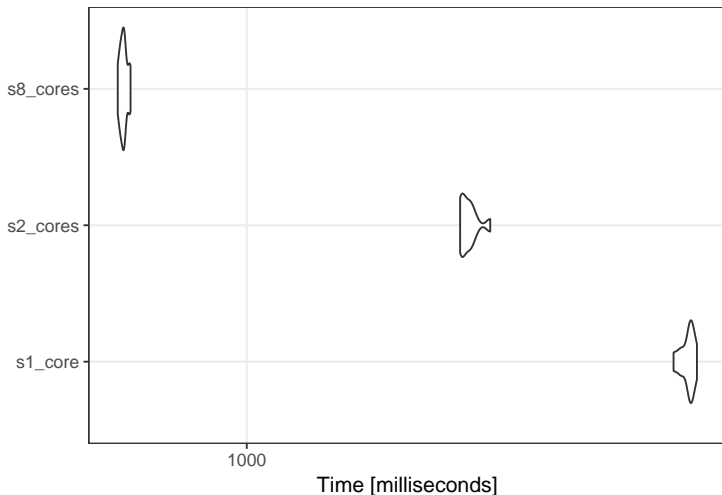
```
one.simu <- function(i) {  
  ## draw data  
  n <- 1000; p <- 500  
  x <- matrix(rnorm(n*p),n,p) ; y <- rnorm(n)  
  ## return ridge's coefficients  
  train <- 1:floor(n/2)  
  test  <- setdiff(1:n,train)  
  ridge <- MASS::lm.ridge(y~x+0,lambda=1,subset=train)  
  err <- (y[test] - x[test, ] %*% ridge$coef )^2  
  return(list(err = mean(err), sd = sd(err)))  
}
```

```
head(do.call(rbind, mclapply(1:8, one.simu, mc.cores = cores)), n = 3)
```

```
##      err      sd  
## [1,] 9.050608 13.04748  
## [2,] 13.99557 18.62884  
## [3,] 13.27724 21.60819
```

## Forking approach with `parallel::mclapply` (cont'd)

```
library(microbenchmark)
res <- microbenchmark(s1_core = mclapply(1:8, one.simu, mc.cores = 1),
                      s2_cores = mclapply(1:8, one.simu, mc.cores = 2),
                      s8_cores = mclapply(1:8, one.simu, mc.cores = 8), times = 10)
```



# Socket approach with `parallel::parLapply`

Windows users need a bit more code to make it work

A possible option: export from base workspace

```
cl <- makeCluster(4)
clusterExport(cl, "one.simu")
res <- parSapply(cl, 1:8, one.simu) # several parLapply call are possible
stopCluster(cl)
res
```

```
##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## err 10.34165 10.82015 13.71423 11.45783 11.55332 9.867533 11.52445
## sd  17.00782 14.57958 18.8415  17.23    15.2085  14.30755 15.6147
##      [,8]
## err 9.878061
## sd  14.74476
```



# Parallel computing with parallel: final remarks

- Parallelize pieces of code complex enough
- Do not choose stupidly the number of cores
- Screen outputs are lost in Rstudio: use `pbmccapply` (progress bar)

```
pbmccapply::pbmccapply(1:8, FUN = one.simu, mc.cores = 2)
```

```
##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## err 11.32937 11.6187  10.99148 17.58251 10.67018 12.98023 10.27076
## sd  15.62033 17.50672 14.50146 26.1142  15.03213 16.77578 14.89964
##      [,8]
## err 9.200708
## sd  13.43777
```

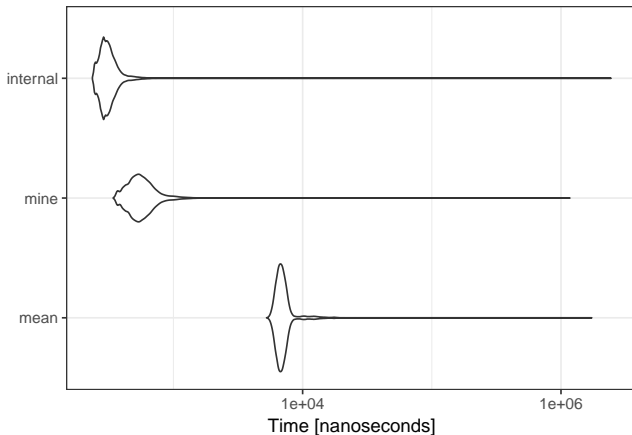
# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks**
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Internal function are faster

Function defined internally are sometimes incredibly faster (written in C), but cannot be called in packages submitted to CRAN.

```
x <- rnorm(100)
res <- microbenchmark(mean = mean(x),
  mine = sum(x)/length(x),
  internal = .Internal(mean(x)), times = 1e+05)
```

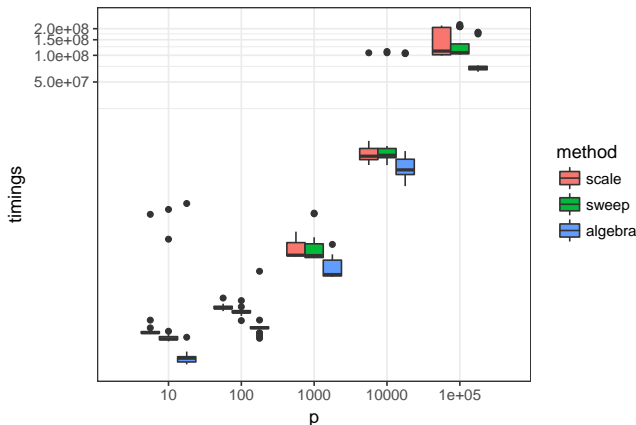


# Mind some algebra

Sweep is a general way to apply a statistic on a given dimension of an array.

```
center1 <- function(x) return(scale(x, colMeans(x), FALSE))
center2 <- function(x) return(sweep(x, 2, colMeans(x), "-", check.margin = FALSE))
center3 <- function(x) return(x - outer(rep(1, nrow(x)), colMeans(x)) )
```

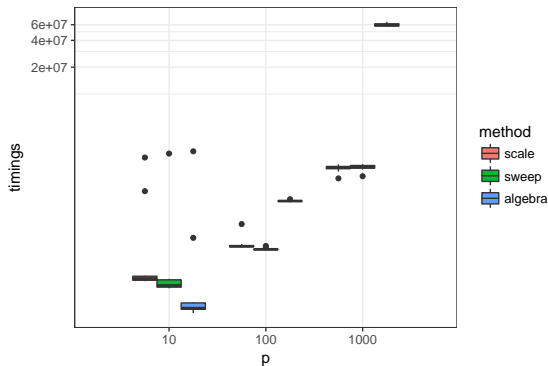
```
seq.p <- 10^(1:5); n <- 100; times <- 20
```



# Algebra does not always pay

## Example for scaling a matrix

```
scale1 <- function(x) return(scale(x, FALSE, colSums(x^2)))  
scale2 <- function(x) return(sweep(x, 2, colSums(x^2), "/", check.margin=FALSE))  
scale3 <- function(x) return(x %*% diag(1/colSums(x^2)) )  
  
seq.p <- 10^(1:3); n <- 100; times <- 20
```



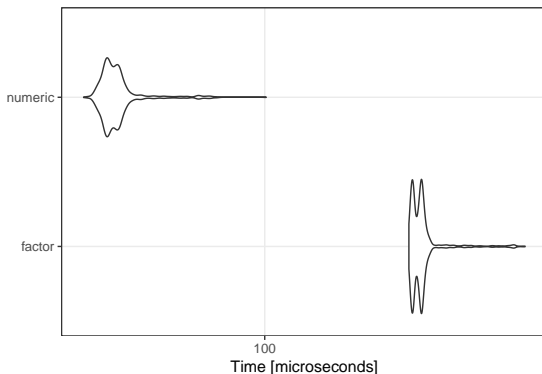
# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# Factor conversion are slow (nlevels)

Do not convert large vector to factor if you need to perform just one operation on it.

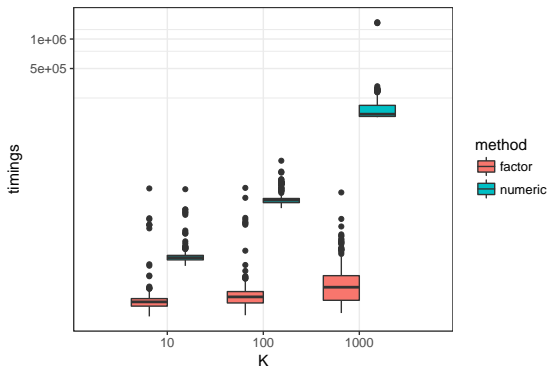
```
n <- 1000; K <- 10
autoplot(microbenchmark(
  factor = nlevels(factor(sample(1:K, n, rep=TRUE))),
  numeric = length(unique(sample(1:K, n, rep=TRUE))), times=1000)
)
```



# Operations on factors are fast (e.g. nlevels)

Use factor if you need repeated operations on the same vector.

```
nk <- 20
seq.K <- c(10,100,1000)
res <- do.call(rbind, lapply(seq.K, function(K) {
  x1 <- rep(1:K,nk)
  x2 <- factor(x1)
  out <- microbenchmark(factor = nlevels(x2),
                        numeric = length(unique(x1)), times=1000)
  return(data.frame(method = out$expr, timings = out$time, K = factor(K)))
})))
```

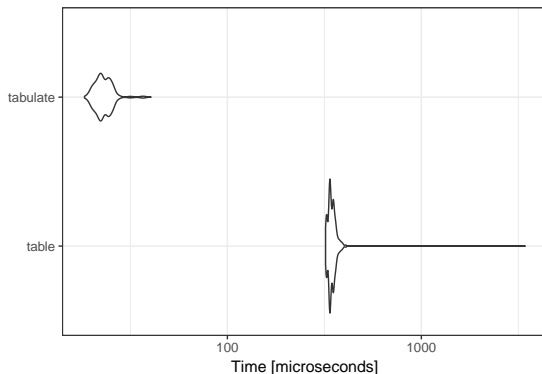




# Prefer tabulate to table whenever you can

table is a complex function that should not be use for simple operations like counting the occurrences of integers in a vector.

```
n <- 1000; K <- 10
autoplot(
  microbenchmark(
    table      = table  (sample(1:K, n, rep=TRUE)),
    tabulate   = tabulate(sample(1:K, n, rep=TRUE)),
    times=1000)
)
```



# R masks the numerical errors

by printing a *convenient* summary of objects

```
7/13
```

```
## [1] 0.5384615
```

```
print(7/13, digits=16)
```

```
## [1] 0.5384615384615384
```

So do not use binary operator to compare floats because

```
.1 == .3/3
```

```
## [1] FALSE
```

```
print(.3/3, digits=16)
```

```
## [1] 0.09999999999999999
```

Try

```
all.equal(.1, .3/3)
```

```
## [1] TRUE
```

# R masks the numerical errors

by printing a *convenient* summary of objects

```
7/13
```

```
## [1] 0.5384615
```

```
print(7/13, digits=16)
```

```
## [1] 0.5384615384615384
```

So do not use binary operator to compare floats because

```
.1 == .3/3
```

```
## [1] FALSE
```

```
print(.3/3, digits=16)
```

```
## [1] 0.09999999999999999
```

Try

```
all.equal(.1, .3/3)
```

```
## [1] TRUE
```

# R masks the numerical errors

by printing a *convenient* summary of objects

```
7/13
```

```
## [1] 0.5384615
```

```
print(7/13, digits=16)
```

```
## [1] 0.5384615384615384
```

So do not use binary operator to compare floats because

```
.1 == .3/3
```

```
## [1] FALSE
```

```
print(.3/3, digits=16)
```

```
## [1] 0.09999999999999999
```

Try

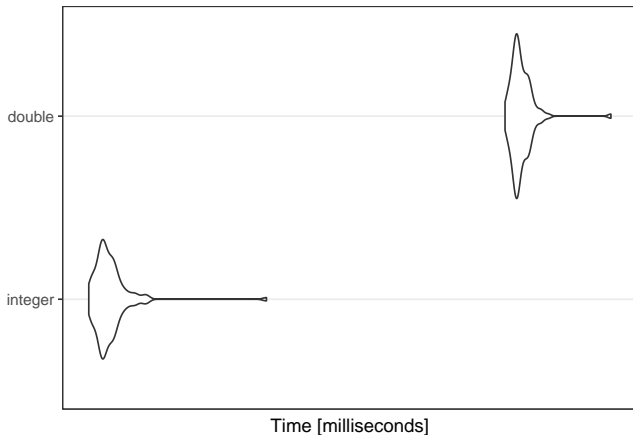
```
all.equal(.1, .3/3)
```

```
## [1] TRUE
```

# Variable type matters

Sorting a vector of integers is much faster than a vector of double, but R is so permissive that you might lose the gain if you do not take care:

```
x_int <- sample.int(1e7, 1e7)
x_dbl <- as.numeric(x_int)
res <- microbenchmark(integer = order(x_int),
                      double = order(x_dbl))
```



# Outline

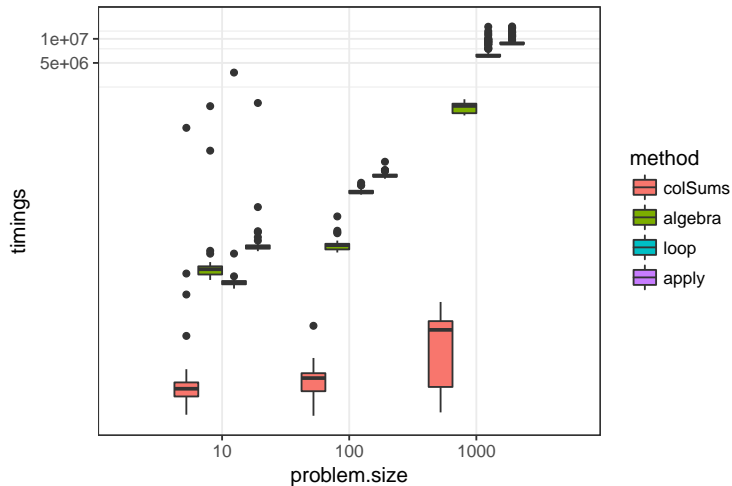
- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function

# The row/colSums family I

col/rowSums, col/rowMeans and their extensions in the matrixStats package (rank,max,min, etc.) are very efficient.

```
colSums.default <- function(x) return(colSums)
colSums.algebra <- function(x) return(crossprod(rep(1,nrow(x)), x))
colSums.apply   <- function(x) return(apply(x,2,sum))
colSums.loop    <- function(x) {
  res <- rep(0,ncol(x))
  for (i in 1:ncol(x)) {
    res[i] <- sum(x[,i])
  }
  res
}
```

# The row/colSums family II





# The secret function rowsum I

`rowsum` (not to be confused with `rowSums`) computes sums in a vector split according a grouping variable (work for matrices).

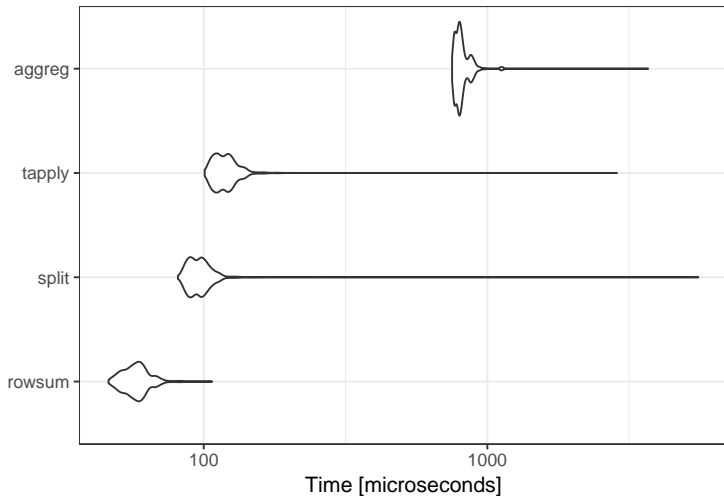
```
vec <- runif(1000)
grp <- sample(1:5, 1000, TRUE)
print(c(rowsum(vec, grp)))
```

```
## [1] 96.40567 98.36409 100.21287 94.20546 108.60807
```

There are many possibilities to perform the required task:

```
res <- microbenchmark(
  rowsum = rowsum(vec, grp),
  split  = sapply(split(vec, grp), sum),
  tapply = tapply(vec, grp, sum),
  aggreg = aggregate(vec, list(grp), sum),
  times = 1000)
```

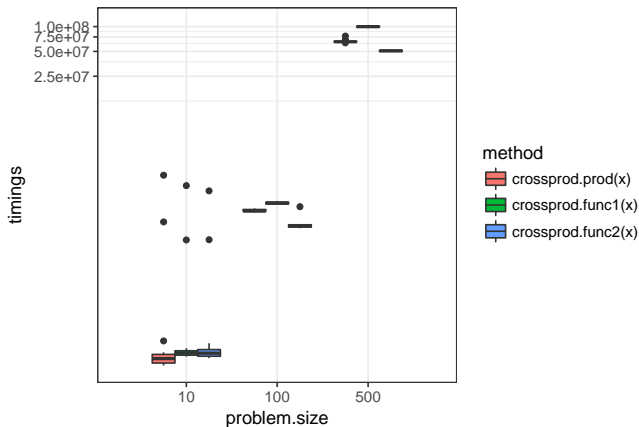
# The secret function rowsum II



# Dedicated function: cross-product

Generally (a bit) faster than `\% * \%` !

```
crossprod.prod <- function(x) return(t(x) %*% x)
crossprod.func1 <- function(x) return(crossprod(x,x))
crossprod.func2 <- function(x) return(crossprod(x))
```

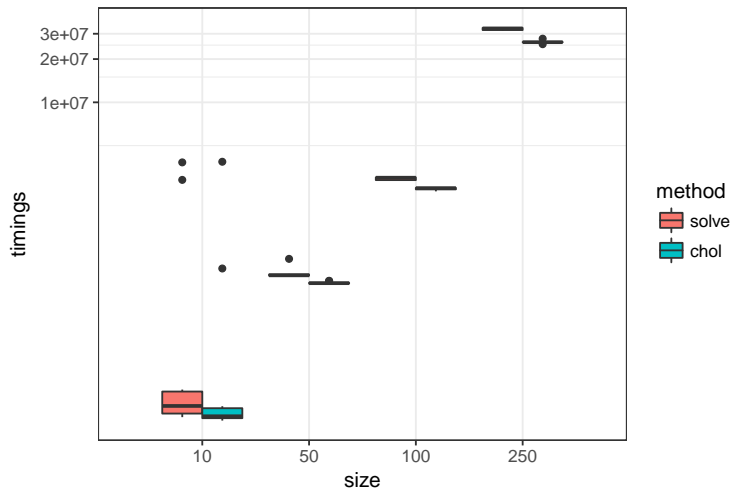


# Dedicated function: inverting a PD matrices I

## Use a Cholesky factorization

```
use.chol <- function(n,p) {  
  x <- matrix(rnorm(n*p),n,p)  
  xtx <- crossprod(x)  
  return(chol2inv(chol(xtx)))  
}  
  
use.solve <- function(n,p) {  
  x <- matrix(rnorm(n*p),n,p)  
  xtx <- crossprod(x)  
  return(solve(xtx))  
}  
  
bench.p.fixed <- function(p, times) {  
  res <- microbenchmark(solve = use.solve(2*p,p),  
                        chol = use.chol (2*p,p), times=times)  
  return(data.frame(method = res$expr,  
                    timings = res$time,  
                    size = rep(as.character(p),times)))  
}
```

# Dedicated function: inverting a PD matrices II

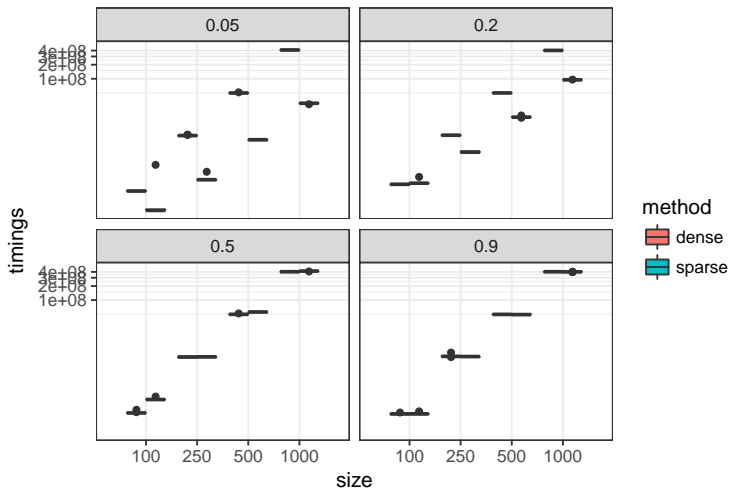


# The Matrix package I

Propose a collection of functions for of matrix algebra adapted to the type of matrix at hand (sparse, diagonal, triangular, block diagonal, etc.)

```
library(Matrix)
bench.par.fixed <- function(par) {
  n <- par$n; density <- par$density
  data <- sample(c(0,1),n**2,rep=TRUE,prob=c(1-density,density))
  x.dense <- matrix(data,n,n)
  x.sparse <- Matrix(data,n,n)
  res <- microbenchmark(dense = crossprod(x.dense) ,
                        sparse = crossprod(x.sparse), times=10)
  return(data.frame(method = res$expr,
                    timings = res$time,
                    size = n ,
                    density = density ))
}
```

# The Matrix package II



# Outline

- ① Function, Functionals
- ② Good and bad practices in R
- ③ Benchmark your code
- ④ Use all your cores when needed
- ⑤ Tricks
- ⑥ Remember that R is object oriented
- ⑦ Mind your vocabulary: R fast built in function



# Interfacing C++ with R is *really* easy I

For a vector  $\mathbf{x} = (x_1, \dots, x_n)$ , consider the simple task of computing

$$y_k = \sum_{i=1}^k \log(x_i), \quad k = 1, \dots, n.$$

One can easily integrate some C++ version of this code with Rcpp.

```
library(Rcpp)
rcpp <- cppFunction('NumericVector rcpp(NumericVector x) {
    using namespace Rcpp;

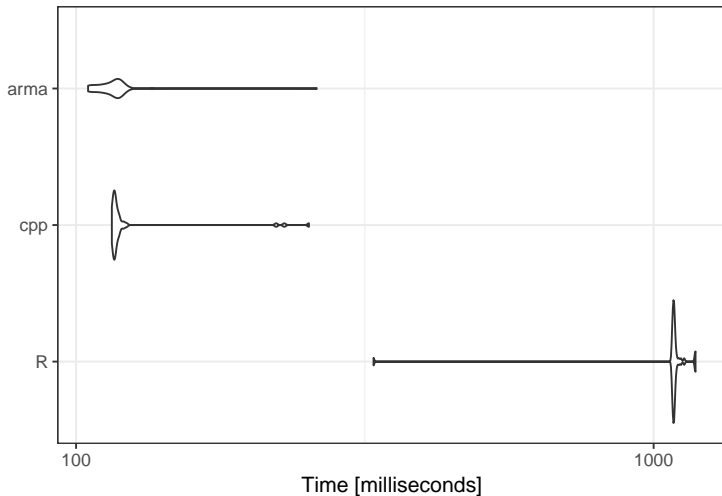
    int n = x.size() ;
    NumericVector res(x) ;
    res(0) = log(x(0));
    for (int i=1; i<n; i++) {
        res(i) = res(i-1) + log(x(i)) ;
    }
    return(wrap(res)) ;
}')
```

# Interfacing C++ with R is *really* easy II

```
library(RcppArmadillo)
Arma <- cppFunction(depends = "RcppArmadillo", 'NumericVector Arma(NumericVector x) {
  using namespace Rcpp;
  using namespace arma;
  return(wrap(cumsum(log(as<vec>(x))))) ;
}')

x <- runif(1e7, 1,2)
res <- microbenchmark(R = cumsum(log(x)), cpp = rcpp(x), arma = Arma(x), times = 40)
print(autoplot(res))
```

# Interfacing C++ with R is *really* easy III



# Interfacing C++ with R is *really* easy I

**Example** that couples C++ + algebraic tricks

Let  $\mathbf{T}$  be an  $n \times n$  lower triangular matrix with nonzero elements equal to one. We need fast computation of

$$\text{vec}(\mathbf{T}\mathbf{B}\mathbf{T}^\top) = (\mathbf{T} \otimes \mathbf{T}) \times \text{vec}(\mathbf{B}).$$

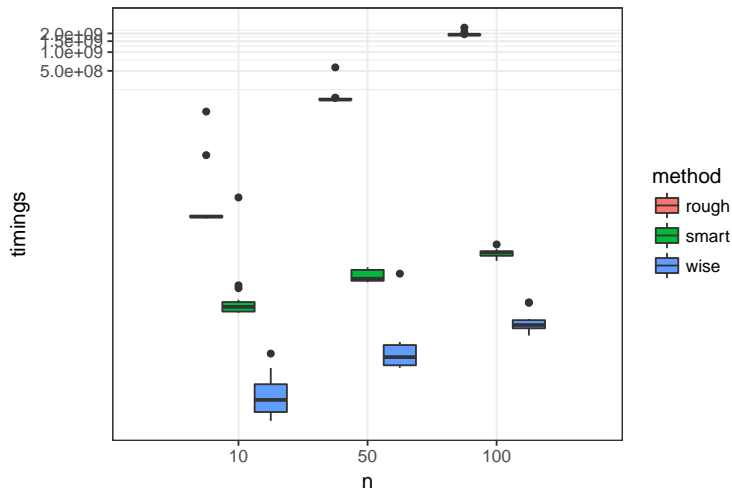
```
library(Matrix); library(inline); library(RcppArmadillo)

prod.rough <- function(B) {
  n <- ncol(B); T <- bandSparse(n,k=(-n+1):0)
  return(kronecker(T,T) %*% as.vector(B))}

prod.smart <- function(B) {
  return(as.vector(apply(apply(B,1,cumsum),1,cumsum))))}

prod.wise <- cxxfunction(signature(B="matrix"), '
  using namespace Rcpp;
  using namespace arma;
  return(wrap(vectorise(cumsum(cumsum(as<mat>(B),0),1)))) ;
  ' , plugin="RcppArmadillo")
```

# Interfacing C++ with R is *really* easy II



# References

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., ... Chang, W. (2018). *Rmarkdown: Dynamic documents for r*. Retrieved from <https://CRAN.R-project.org/package=rmarkdown>

Burns, P. (2012). *The r inferno*. Lulu. com. Retrieved from <http://www.burns-stat.com/documents/books/the-r-inferno/>

Chang, W. (2012). *R graphics cookbook: Practical recipes for visualizing data*. "O'Reilly Media, Inc." Retrieved from <http://www.cookbook-r.com/Graphs/>

Eddelbuettel, D. (2013). *Seamless r and c++ integration with rcpp*. Springer. Retrieved from <http://dirk.eddelbuettel.com>

Gandrud, C. (2016). *Reproducible research with r and r studio*. Chapman; Hall/CRC. Retrieved from <https://github.com/christophergandrud/Rep-Res-Book>

Gillespie, C., & Lovelace, R. (2016). *Efficient r programming*. "O'Reilly Media, Inc." Retrieved from <https://bookdown.org/csgillespie/efficientR/>

R Core Team. (2017). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from <https://www.R-project.org/>

Wickham, H. (2014). *Advanced r*. CRC Press. Retrieved from