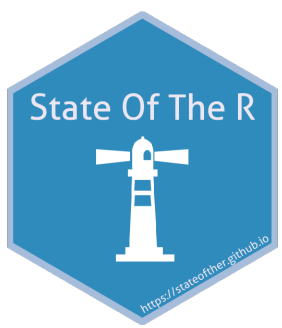
(A bit of) Advanced R

Part 2 - faster R programming

Julien Chiquet

https://github.com/jchiquet/CourseAdvancedR

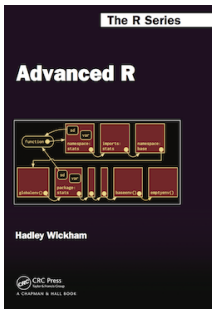
Université Paris Dauphine, Juin 2018



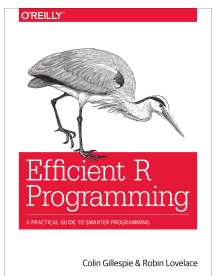
Outline

References I

Advanced R (Wickham, 2014), http://adv-r.had.co.nz/

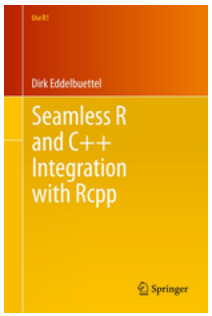


Efficient R programming (Gillespie & Lovelace, 2016), https://bookdown.org/csgillespie/efficientR/

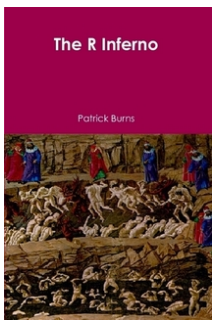


References II

Seamless R and C++ integration with Rcpp (Eddelbuettel, 2013), for sale but see http://dirk.eddelbuettel.com



The R inferno (Burns, 2012), http://www.burns-stat.com/documents/books/the-r-inferno/



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

$\rightsquigarrow$ Creation, Basic Operation, Manipulation, Representation

Basic `R` programming

1. Control Statements
2. Functions
3. Basics on Functionals

$\rightsquigarrow$ Advanced R, Chapters I.6, II.10, II.11

Outline

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)))
```

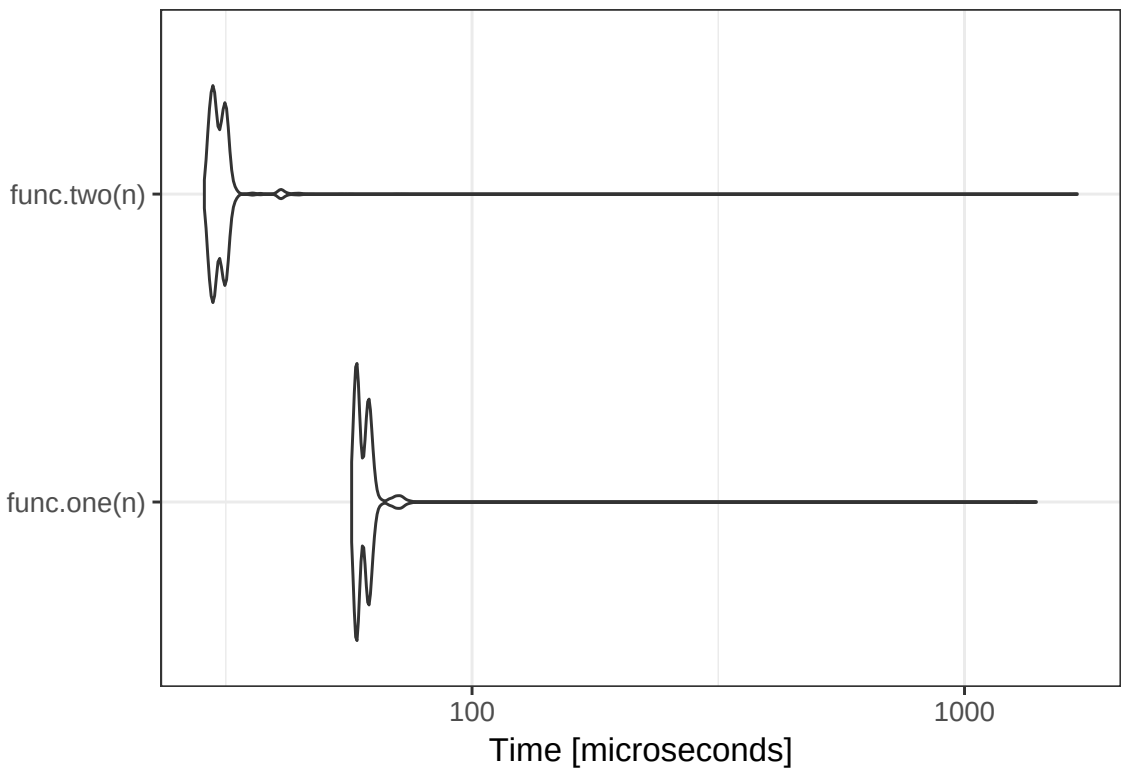
```
##    user  system elapsed
##   0.006   0.000   0.006
```

Exercice

Write functions to compute the variance of a real vector, with an without loops. Benchmark them.

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 I

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)
```

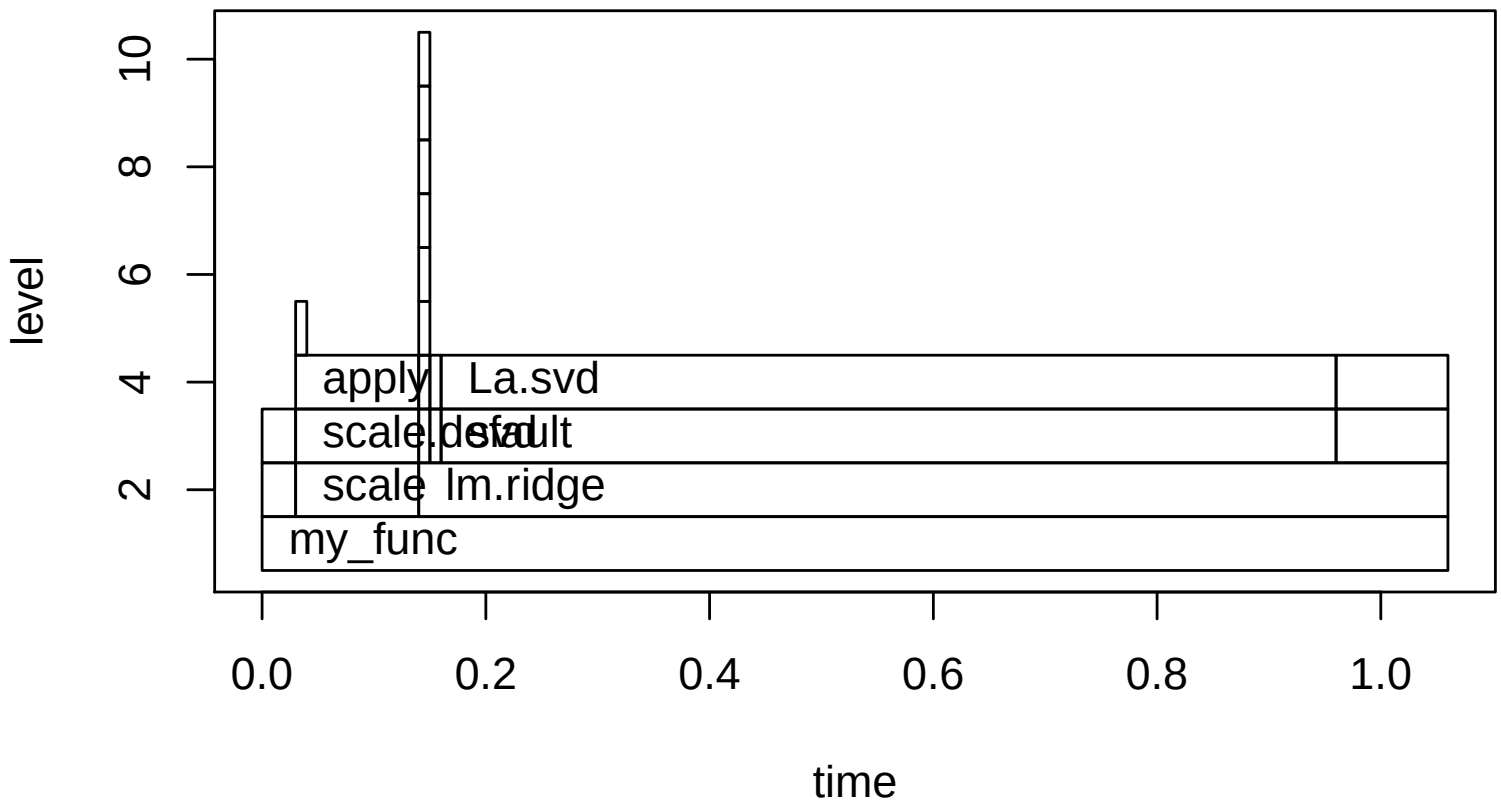
```
summaryRprof("profiling.out")$by.self
```

```
##                   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
```

Profile your code with `profr`

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

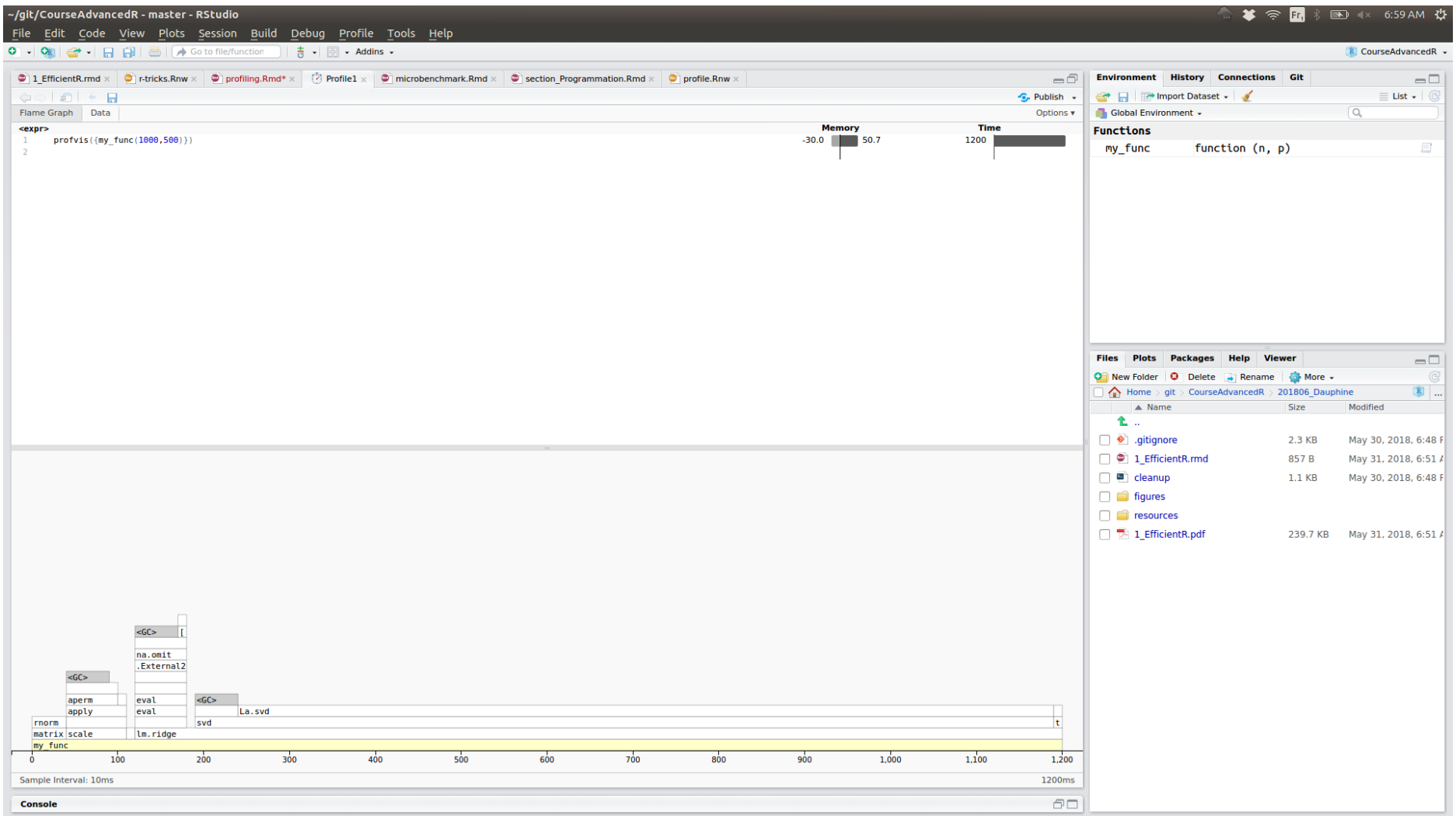
```
library(profr)
profiling <- profr({my_func(1000,500)}, interval = 0.01)
plot(profiling)
```



Profile your code within Rstudio with `profvis`

Profvis integrates the profiling to the Rstudio API: try it!

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



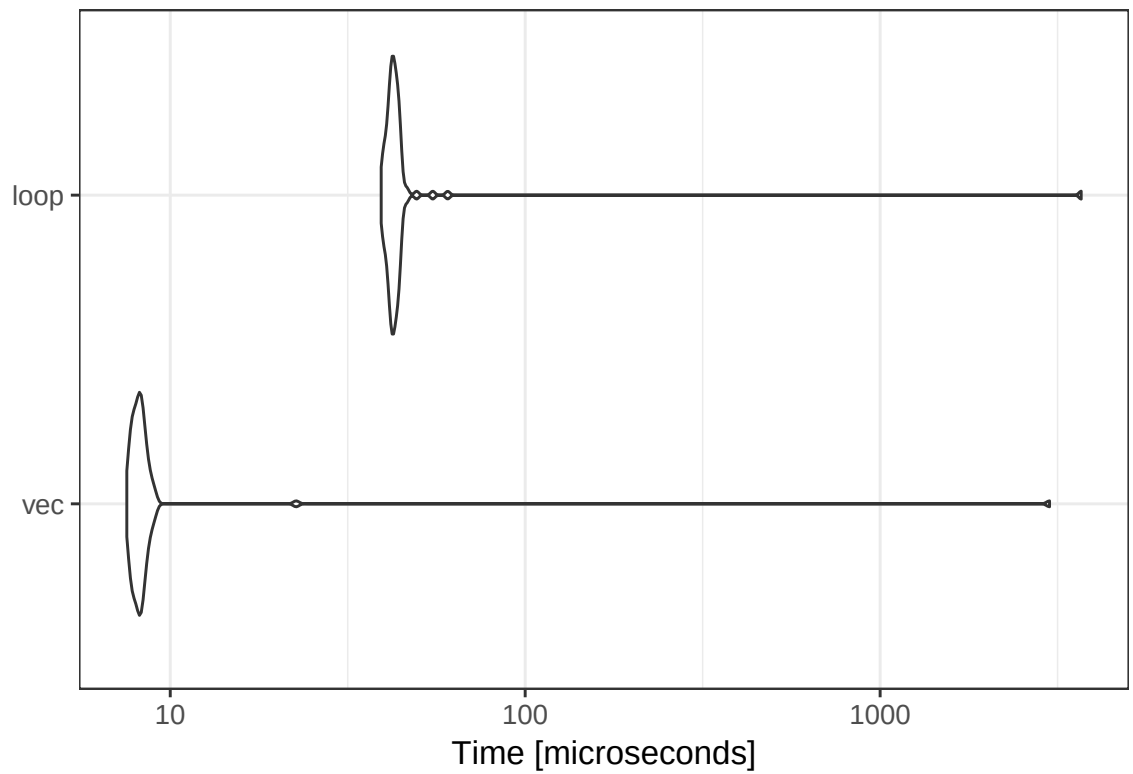
Outline

Vectorize any algebraic operation

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

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

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

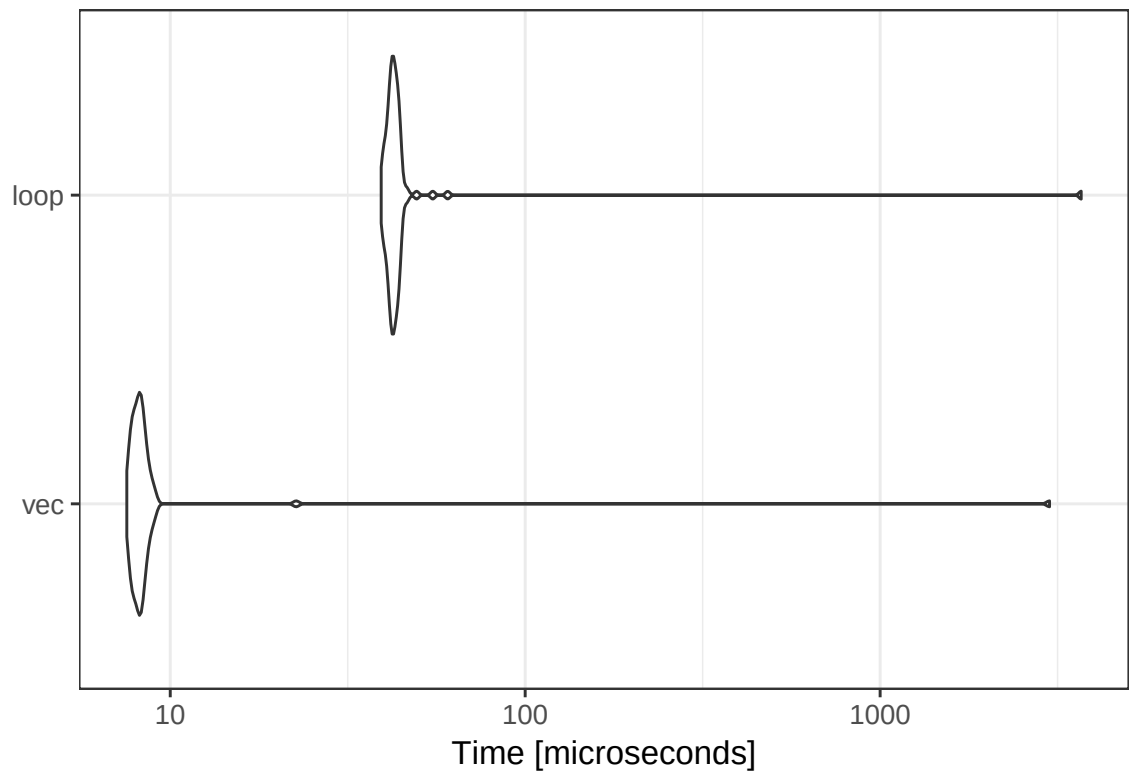


Vectorize any algebraic operation

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

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

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



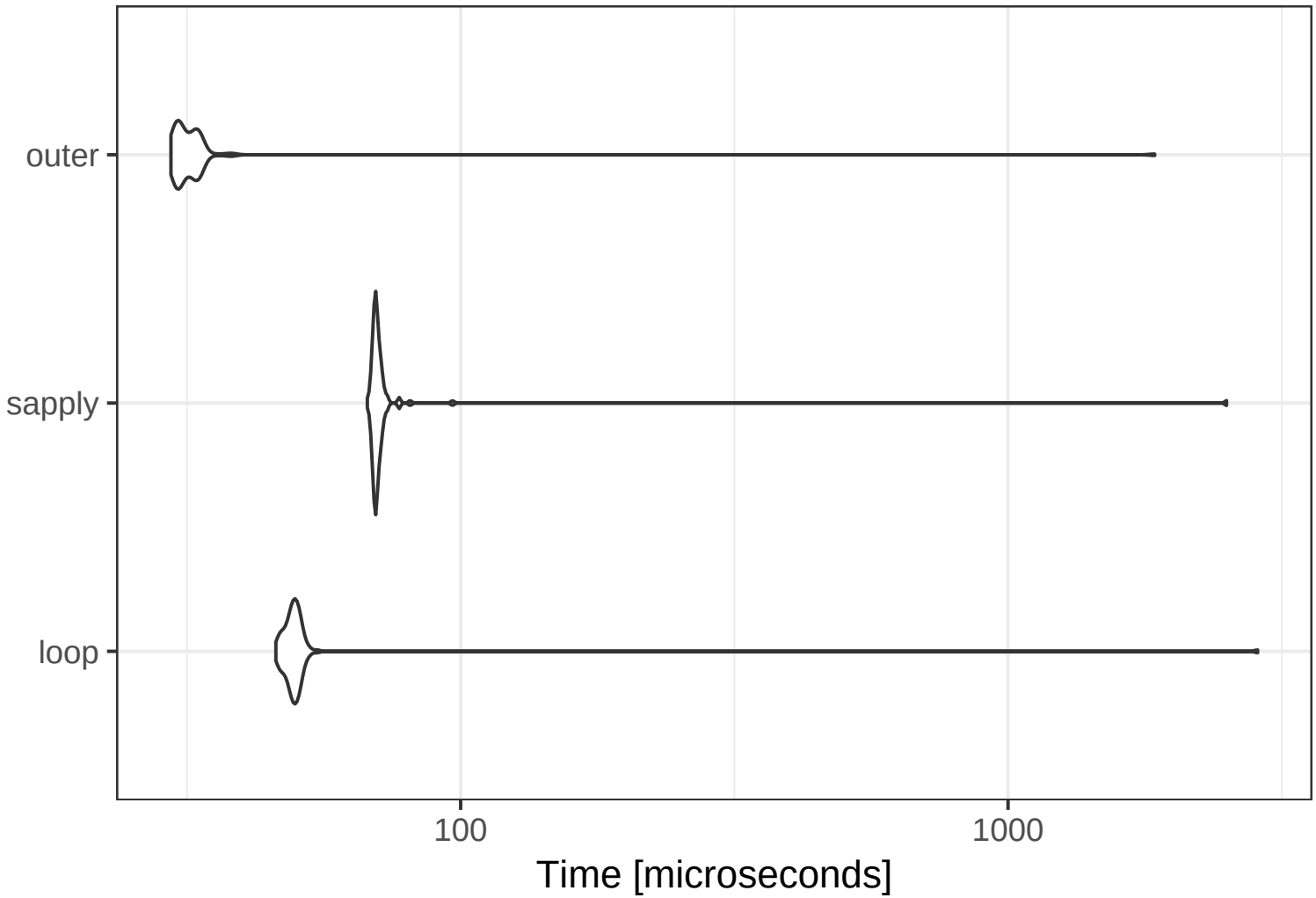
Vectorize, even for non-algebraic operation I

```
month_year_loop <- function(year) {
  res <- c()
  for (month in month.name)
    res <- c(res, paste(month, year, sep = "_"))
  res
}

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(
  loop   = month_year_loop(c(2011, 2013)),
  sapply = month_year_apply(c(2011, 2013)),
  outer  = month_year_outer(c(2011, 2013))
  )
)
```

Vectorize, even for non-algebraic operation II

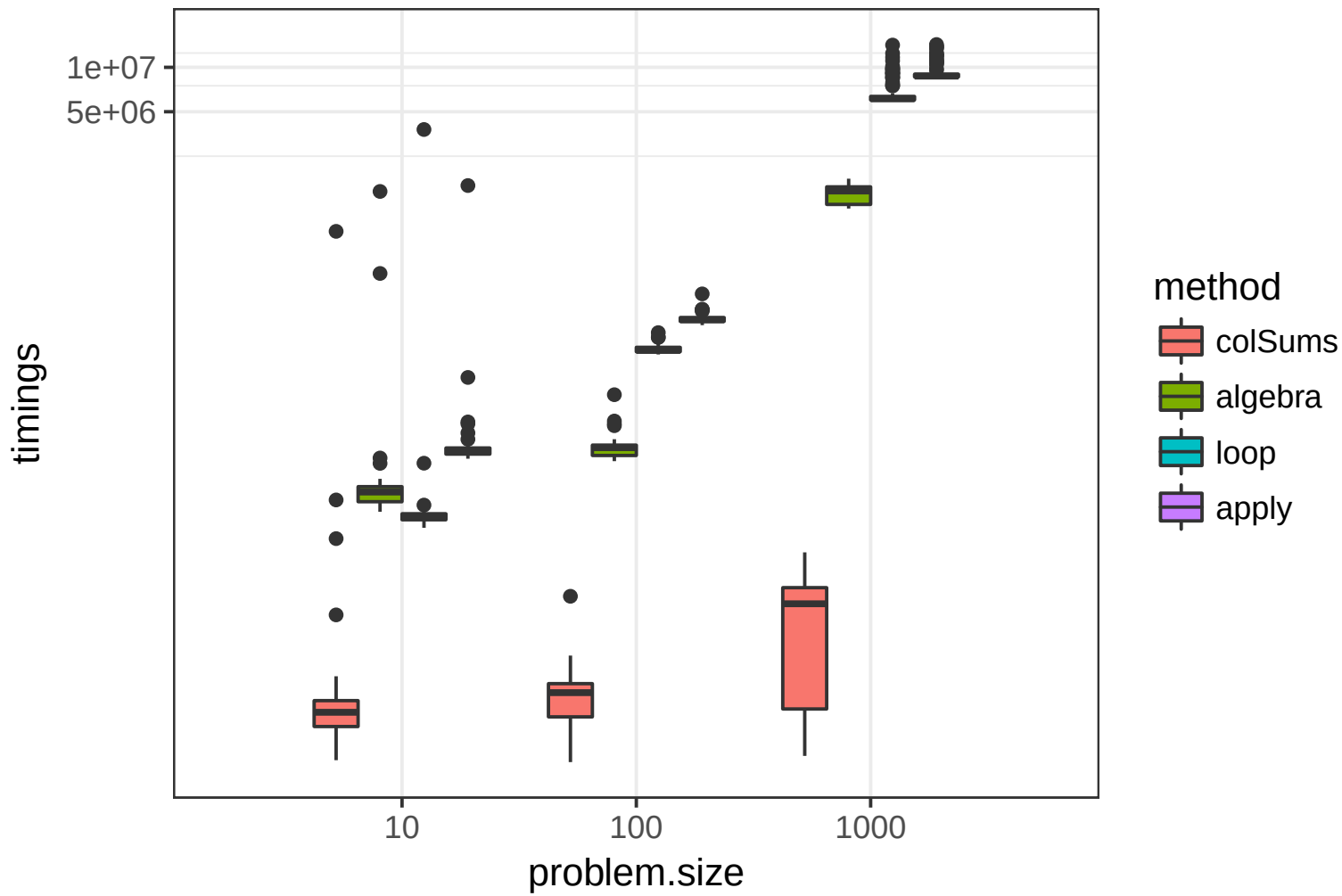


The row/colSums family I

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

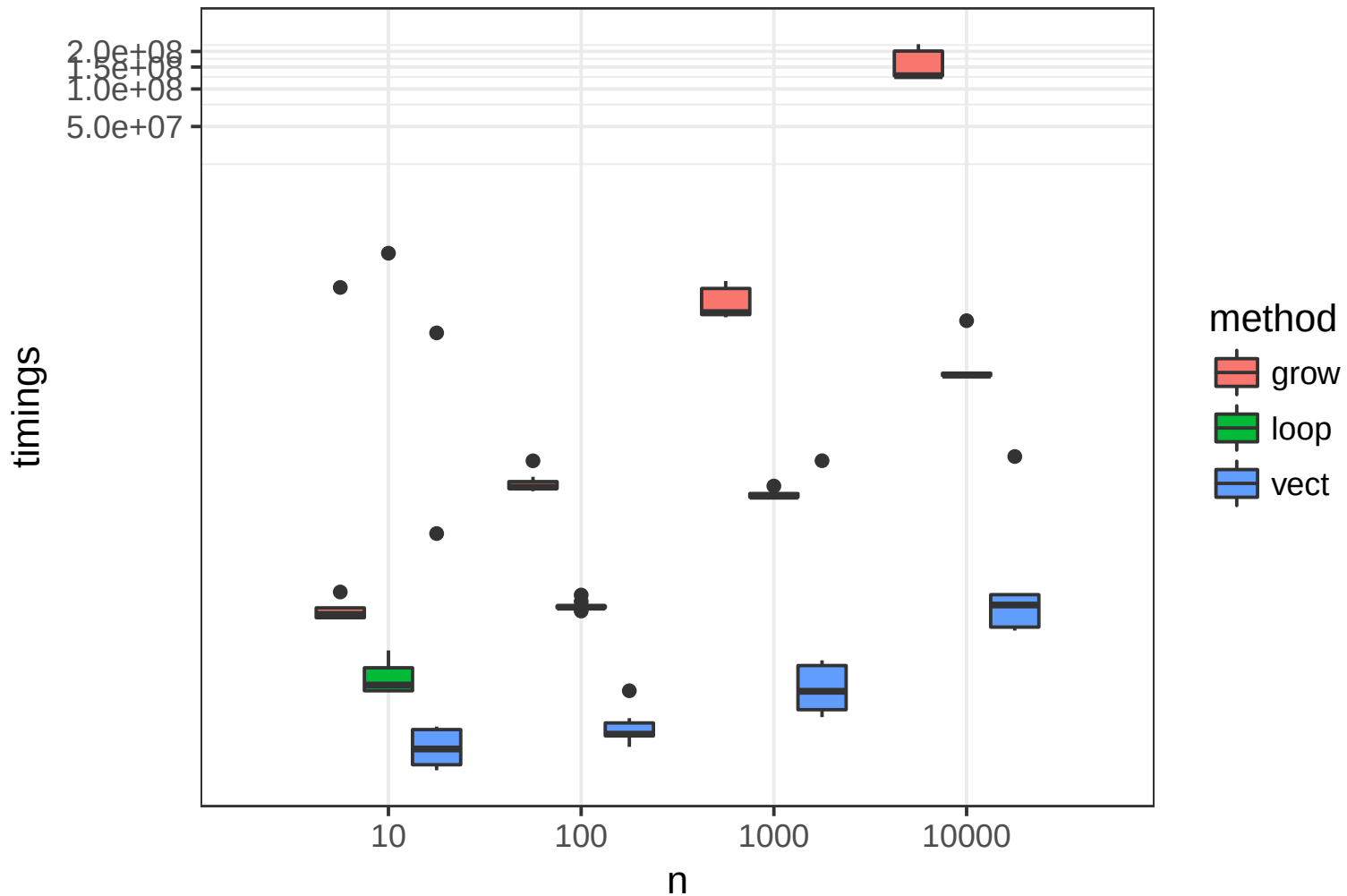
```
colSums.default <- function(x) return(colSums)
colSums.algebra <- function(x) return(crossprod(rep(1,nrow(x)), x))
colSums.apply   <- function(x) return(apply(x,2,sum))
colSums.loop    <- function(x) {
  res <- rep(0,ncol(x))
  for (i in 1:ncol(x)) {
    res[i] <- sum(x[,i])
  }
  res
}
```

The `row/colSums` family II



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}
```



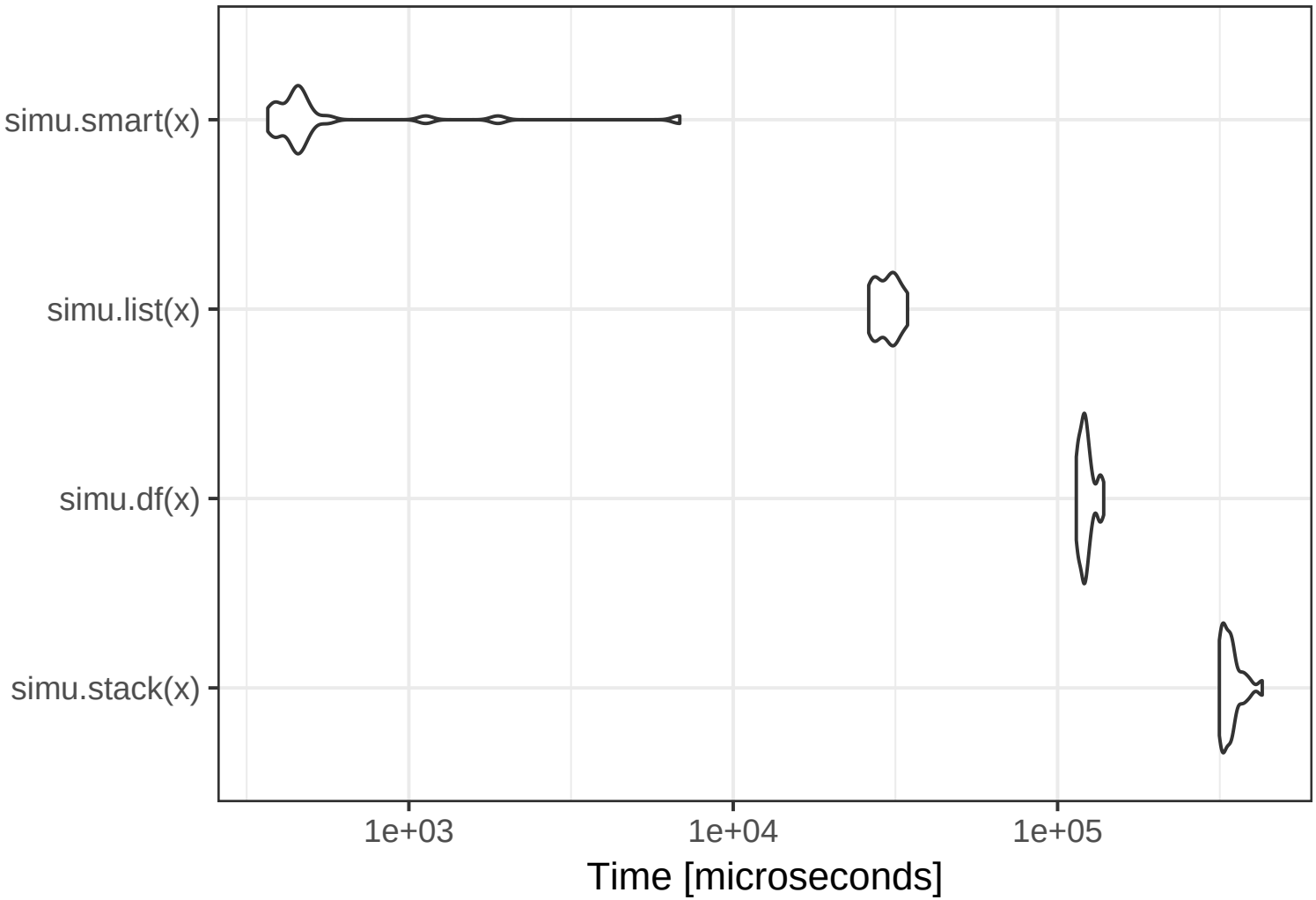
Do not stack objects I

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, ])) )
  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, ]))
  out
}

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

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

Do not stack objects II



Exercice: code the smart function (no loop)

Outline

Parallel computing

Usual Roadmap

1. Start up and intialize 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

$\rightsquigarrow$ 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

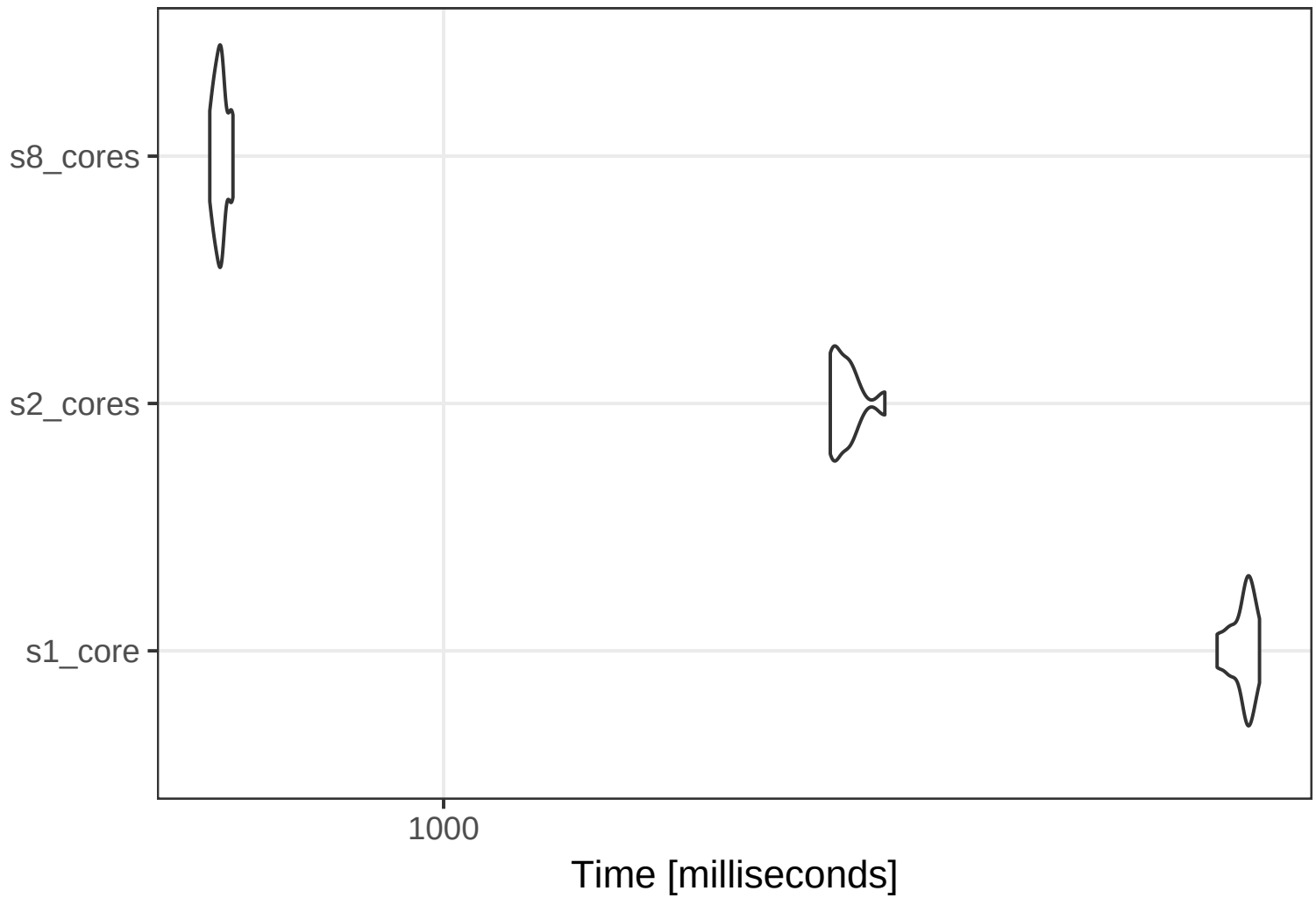
```
one.simu <- function(i) {
  ## draw data
  n <- 1000; p <- 500
  x <- matrix(rnorm(n*p),n,p) ; y <- rnorm(n)
  ## return ridge's coefficients
  train <- 1:floor(n/2)
  test  <- setdiff(1:n,train)
  ridge <- MASS::lm.ridge(y~x+0,lambda=1,subset=train)
  err <- (y[test] - x[test, ] %*% ridge$coef )^2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

```
pbmcapply::pbmcmapply(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
```

Parallel computing: exercise

Here are two function to bootstrap a table and to extract the R^2 from the output of `lm`, a linear model fit.

```
boot_df  <- function(x) x[sample(nrow(x), rep = T), ]
rsquared <- function(mod) summary(mod)$r.squared
summary(lm(mpg ~ wt + disp, data = mtcars))
```

```
##
## Call:
## lm(formula = mpg ~ wt + disp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4087 -2.3243 -0.7683  1.7721  6.3484
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.96055    2.16454  16.151 4.91e-16 ***
## wt          -3.35082    1.16413  -2.878  0.00743 **
## disp        -0.01773    0.00919  -1.929  0.06362 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.917 on 29 degrees of freedom
## Multiple R-squared:  0.7809, Adjusted R-squared:  0.7658
## F-statistic: 51.69 on 2 and 29 DF,  p-value: 2.744e-10
```

Bootstrap the R^2 with `lapply`, `mclapply` and `replicate`.

Outline

R is a typed language

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 is a typed language

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 is a typed language

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

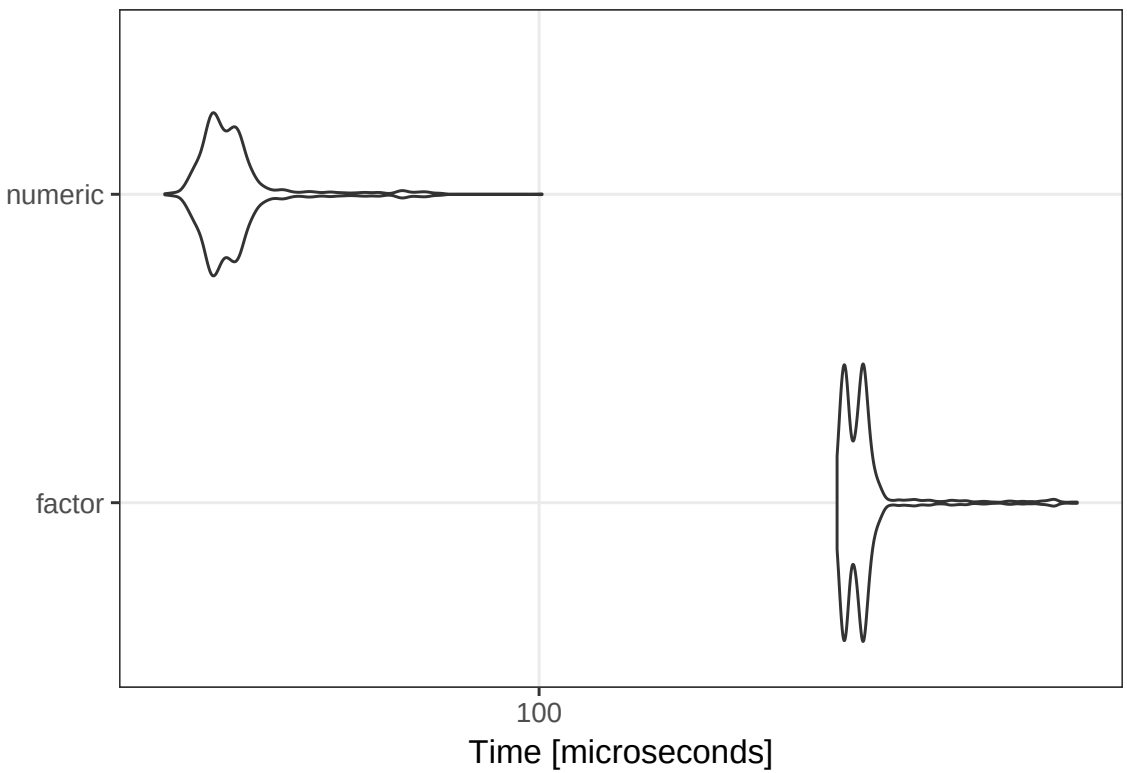
```
all.equal(.1, .3/3)
```

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

Factor conversion are slow (`nlevels`)

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

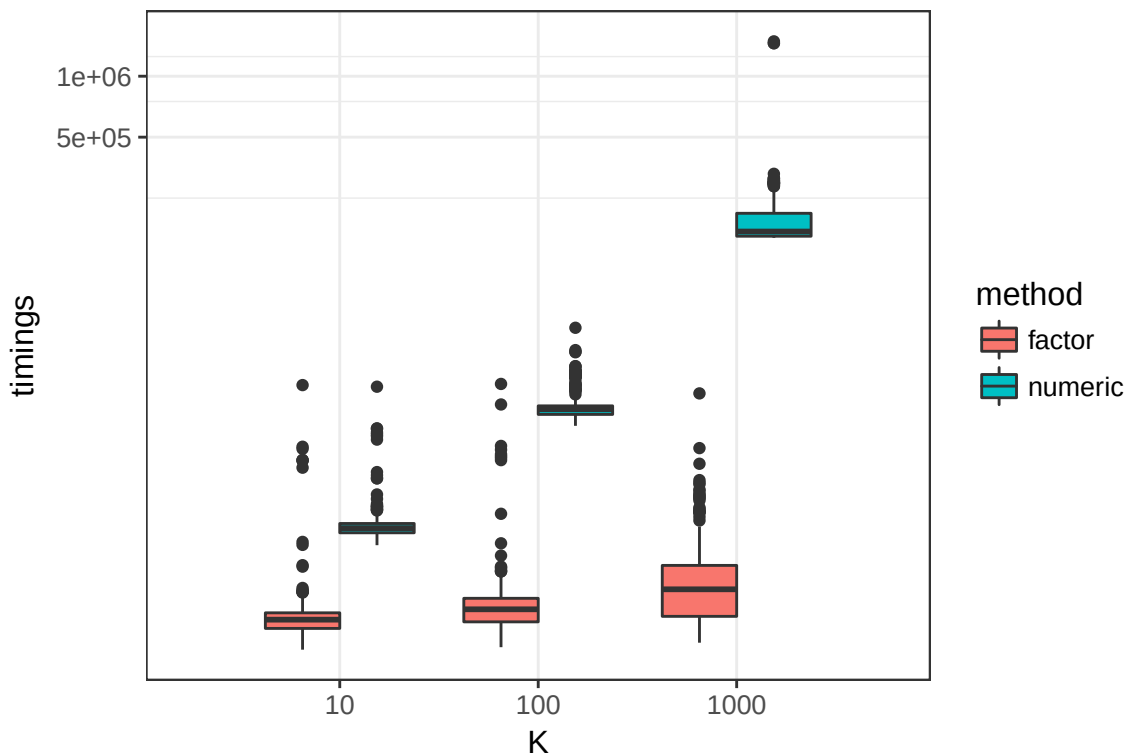
```
n <- 1000; K <- 10
autoplot(microbenchmark(
  factor = nlevels(factor(sample(1:K, n, rep=TRUE))),
  numeric = length(unique(sample(1:K, n, rep=TRUE))), times=1000)
)
```



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

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

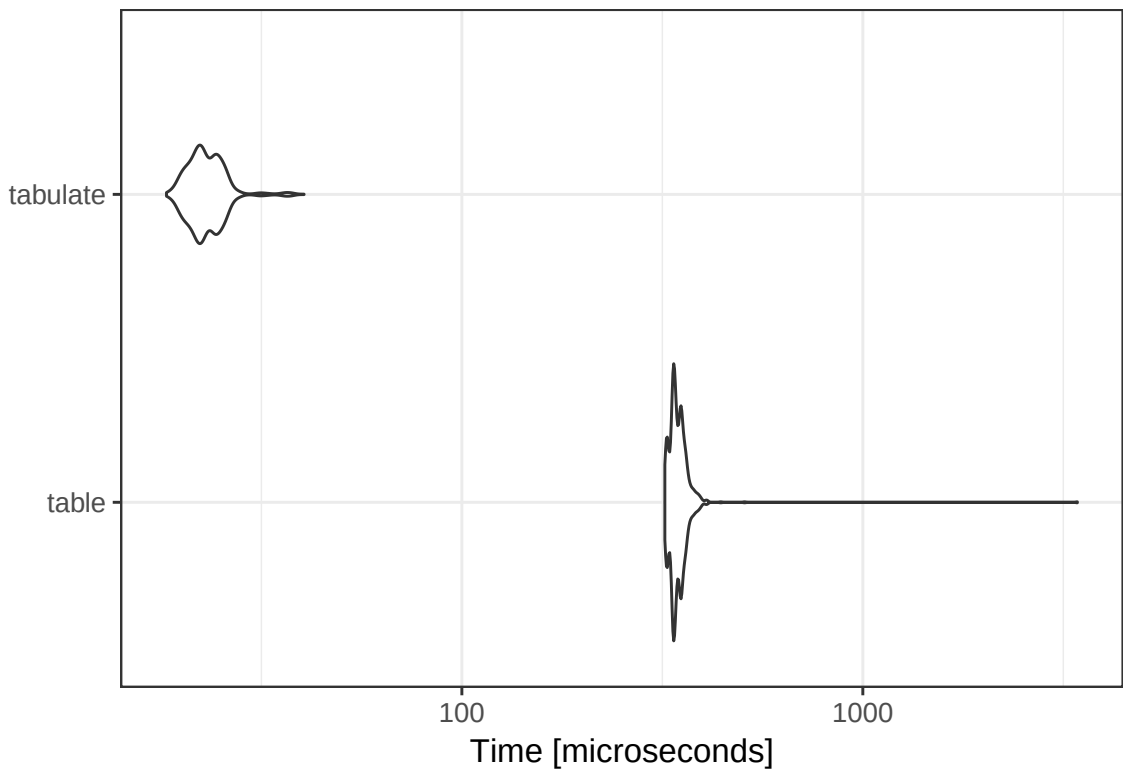
```
nk <- 20
seq.K <- c(10,100,1000)
res <- do.call(rbind, lapply(seq.K, function(K) {
  x1 <- rep(1:K,nk)
  x2 <- factor(x1)
  out <- microbenchmark(factor  = nlevels(x2),
                        numeric = length(unique(x1)), times=1000)
  return(data.frame(method = out$expr, timings = out$time, K =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

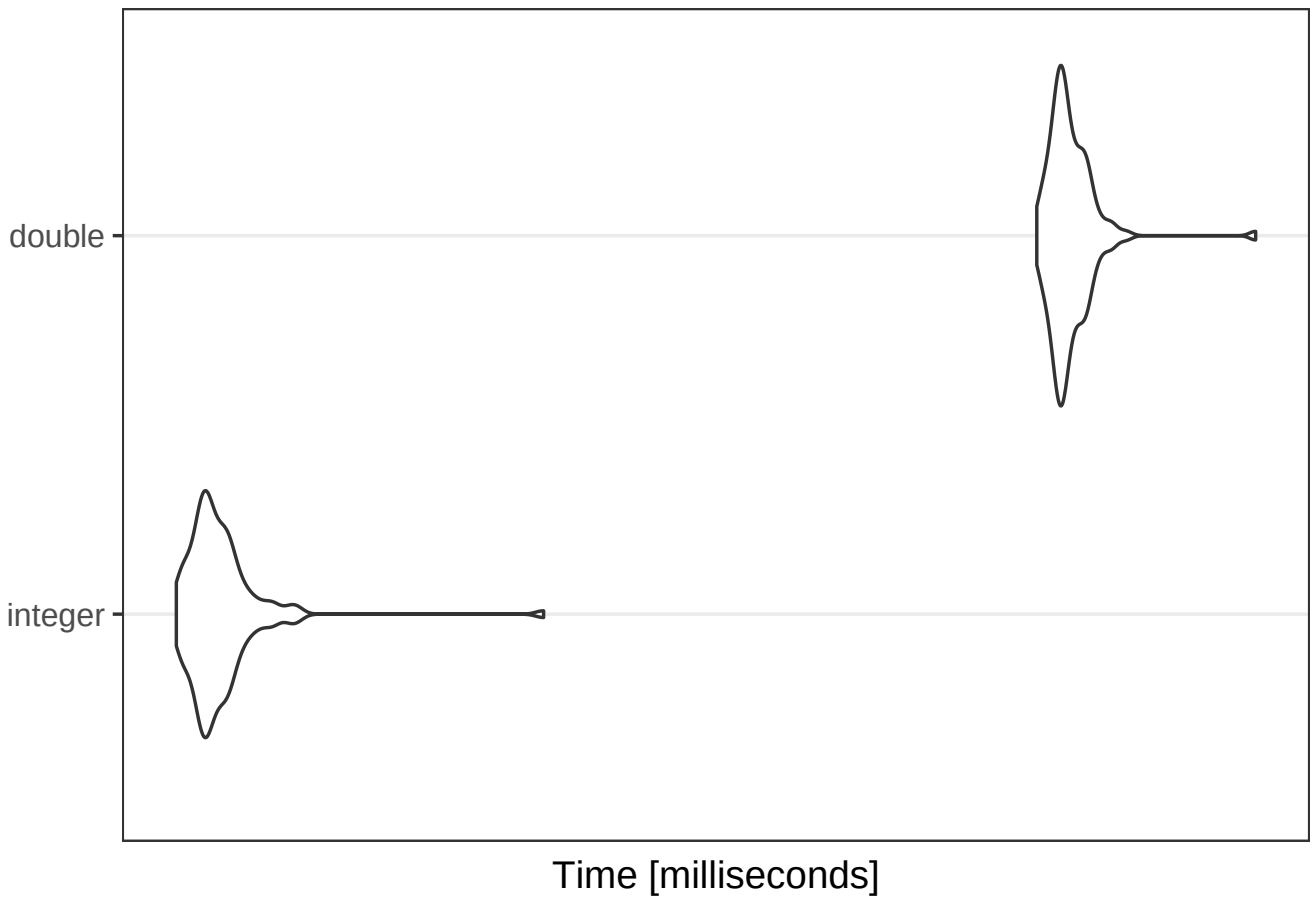
```
n <- 1000; K <- 10
autoplot(
  microbenchmark(
    table    = table   (sample(1:K, n, rep=TRUE)),
    tabulate = tabulate(sample(1:K, n, rep=TRUE)),
  times=1000)
)
```



Variable type matters

Sorting a vector of integers is much faster than a vector of double, but R is so permissive that you might loss 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

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

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

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

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

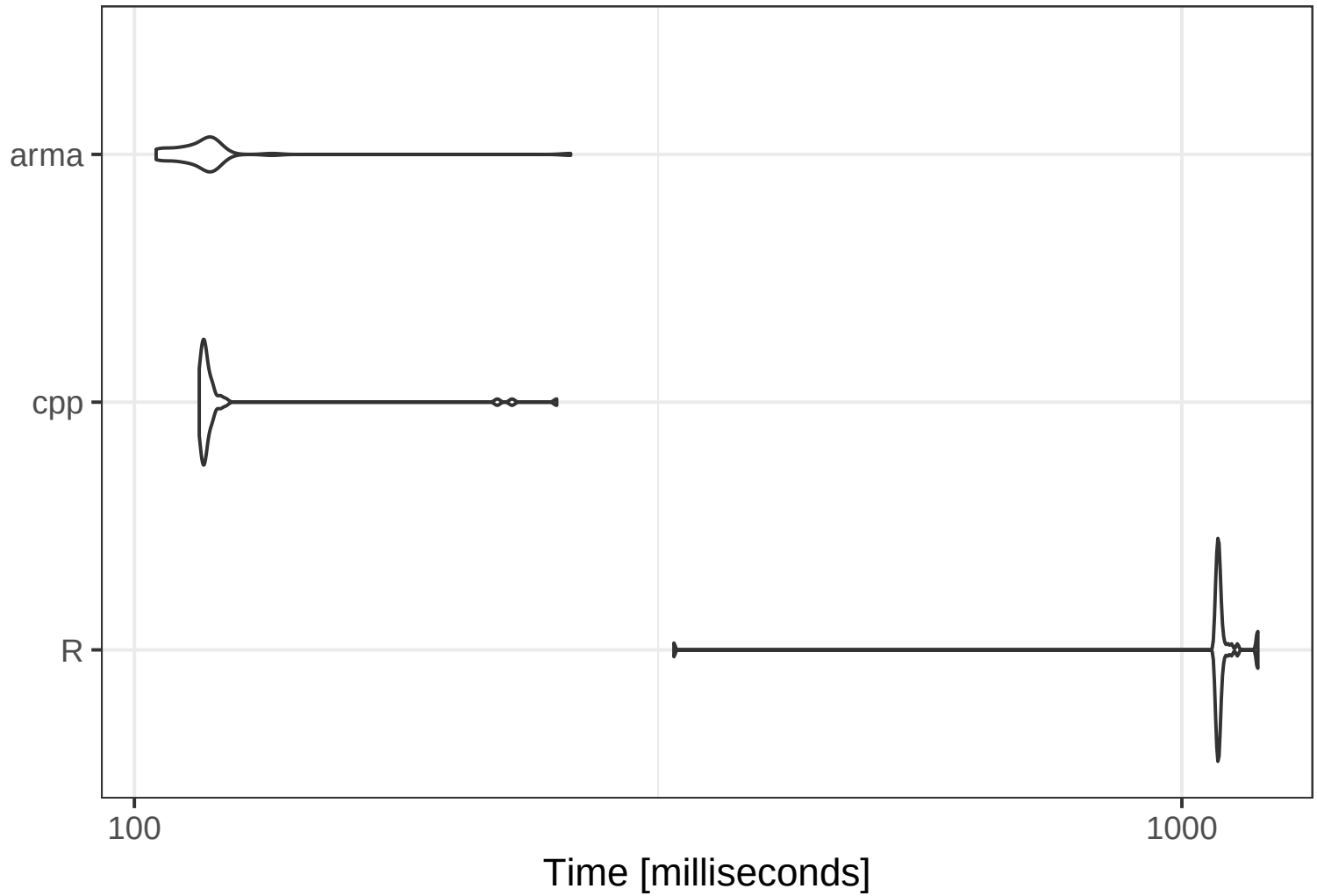
```
library(Rcpp)
rcpp <- cppFunction('NumericVector rcpp(NumericVector x) {
  using namespace Rcpp;

  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

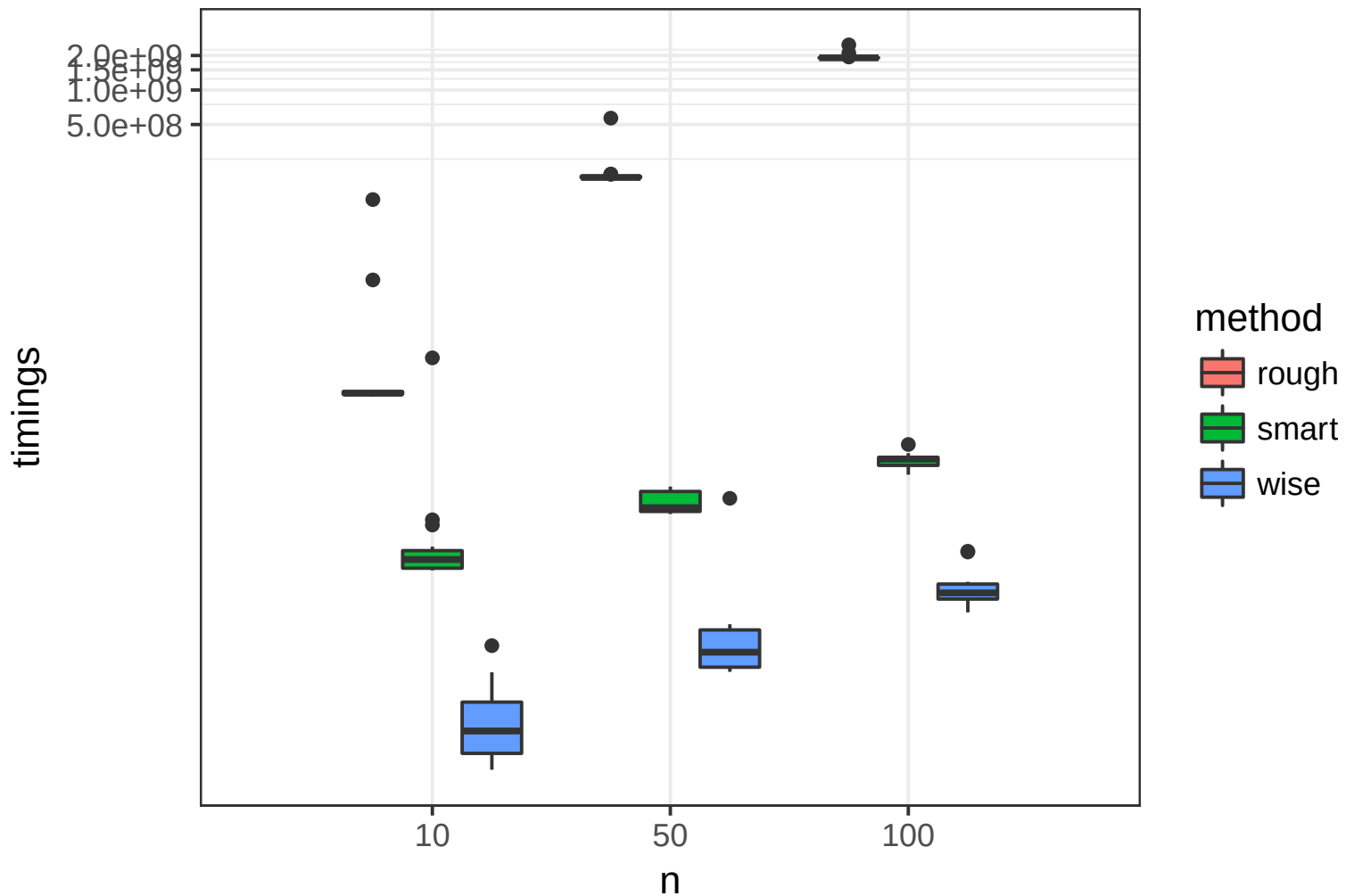
Example that couples C++ + algebraic tricks

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



Outline

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).

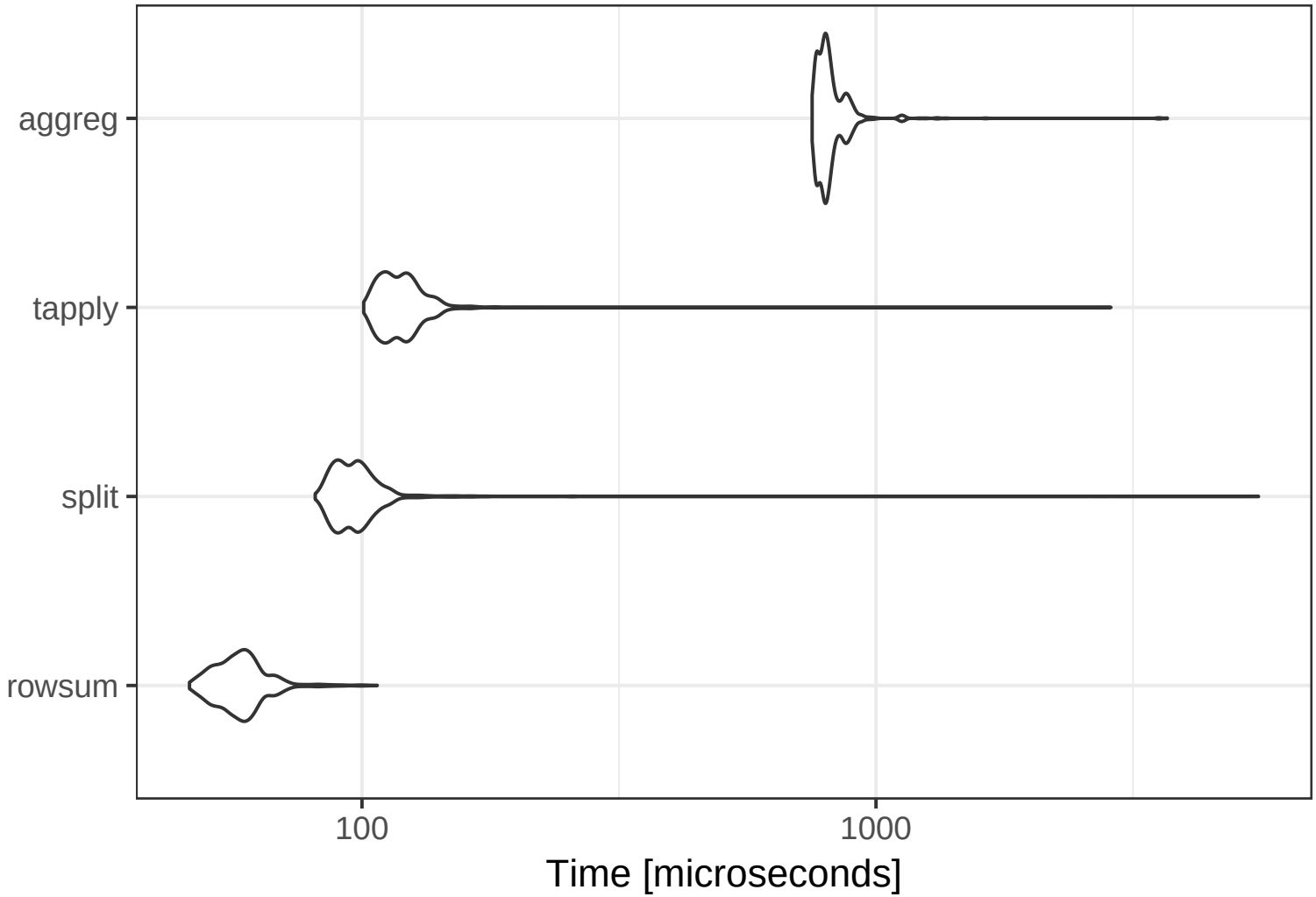
```
vec <- runif(1000)
grp <- sample(1:5, 1000, TRUE)
print(c(rowsum(vec, grp)))
```

```
## [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)
```

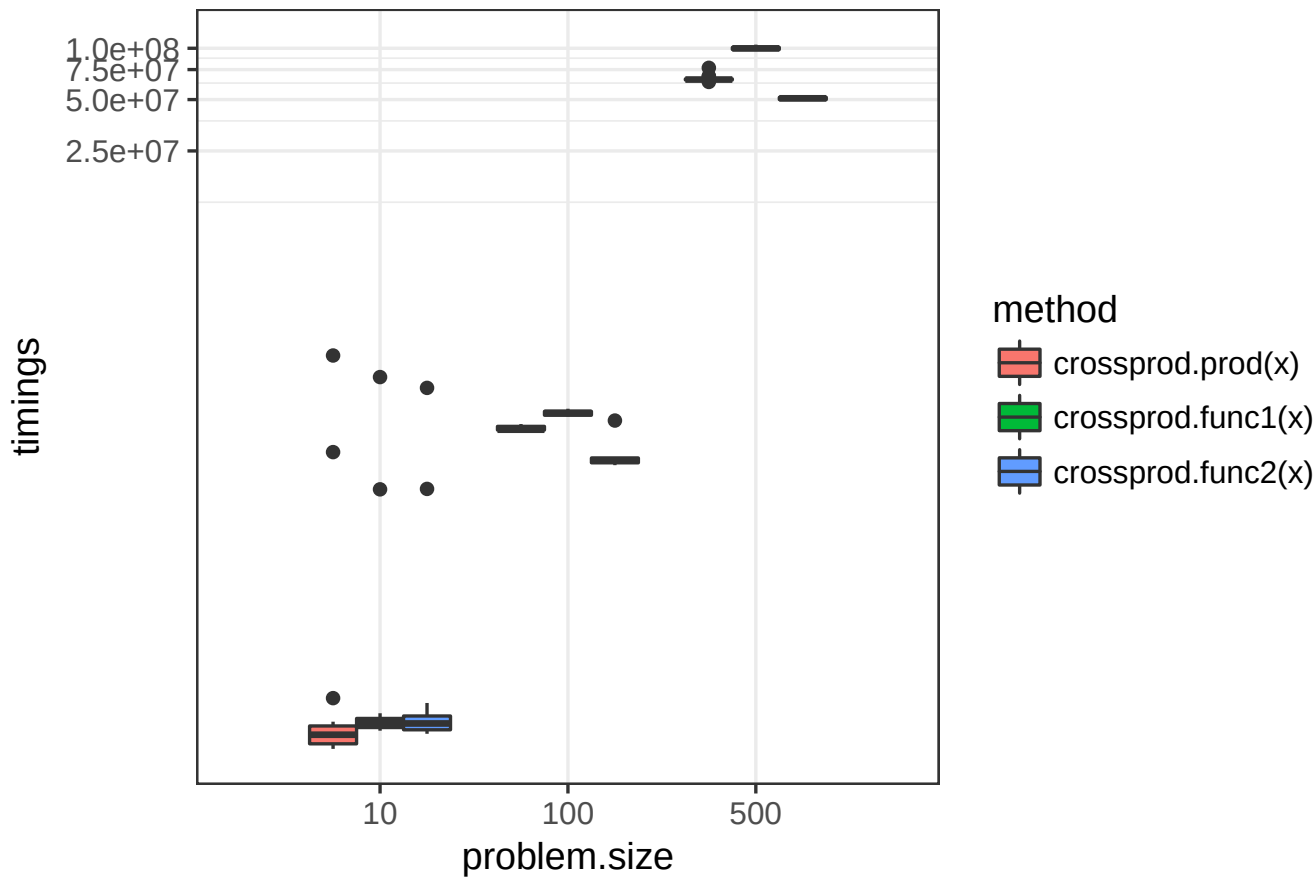
The secret function `rowsum` II



Dedicated function: cross-product

Generally (a bit) fastest than \% * \% !

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

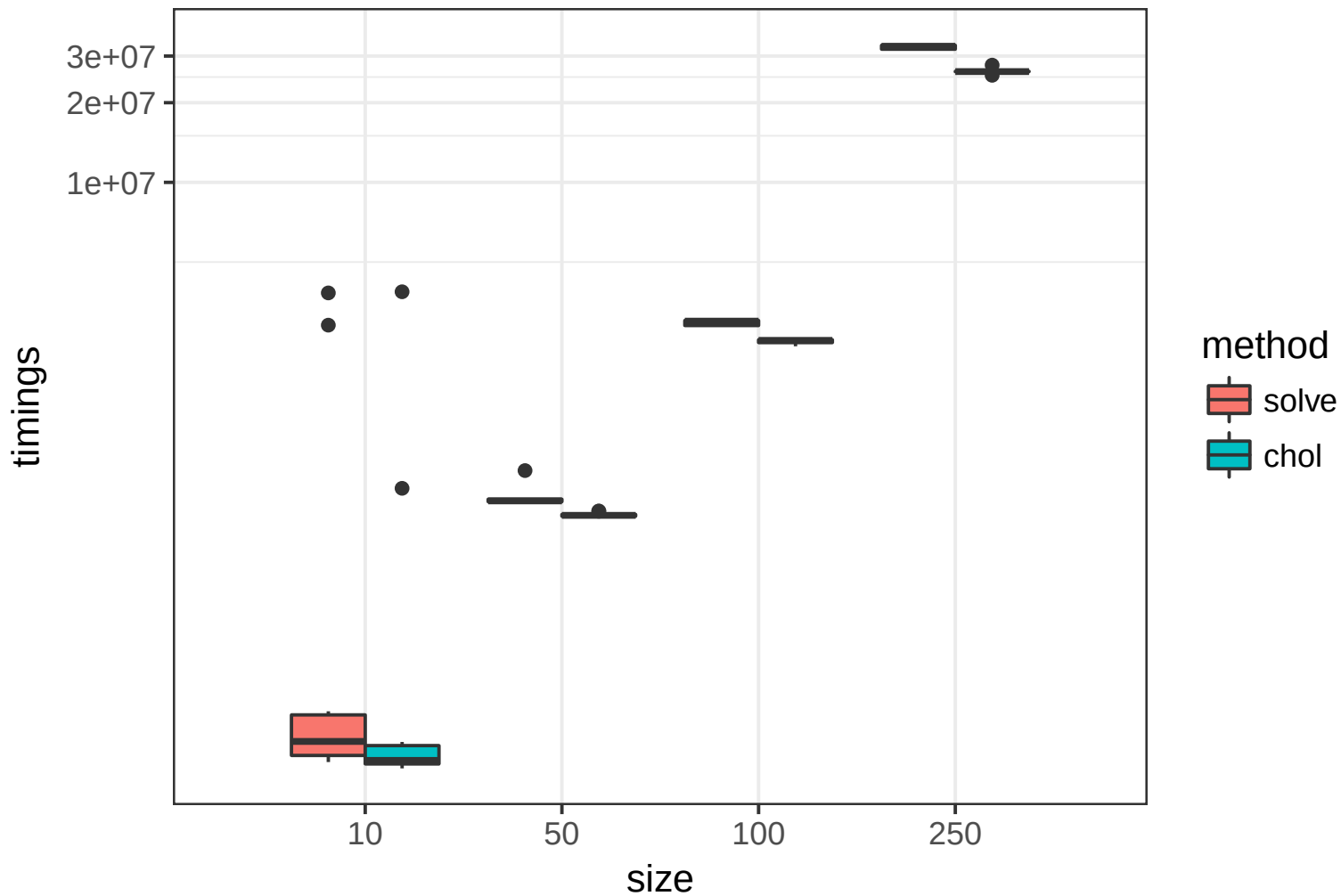


Dedicated function: inverting a PD matrices I

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

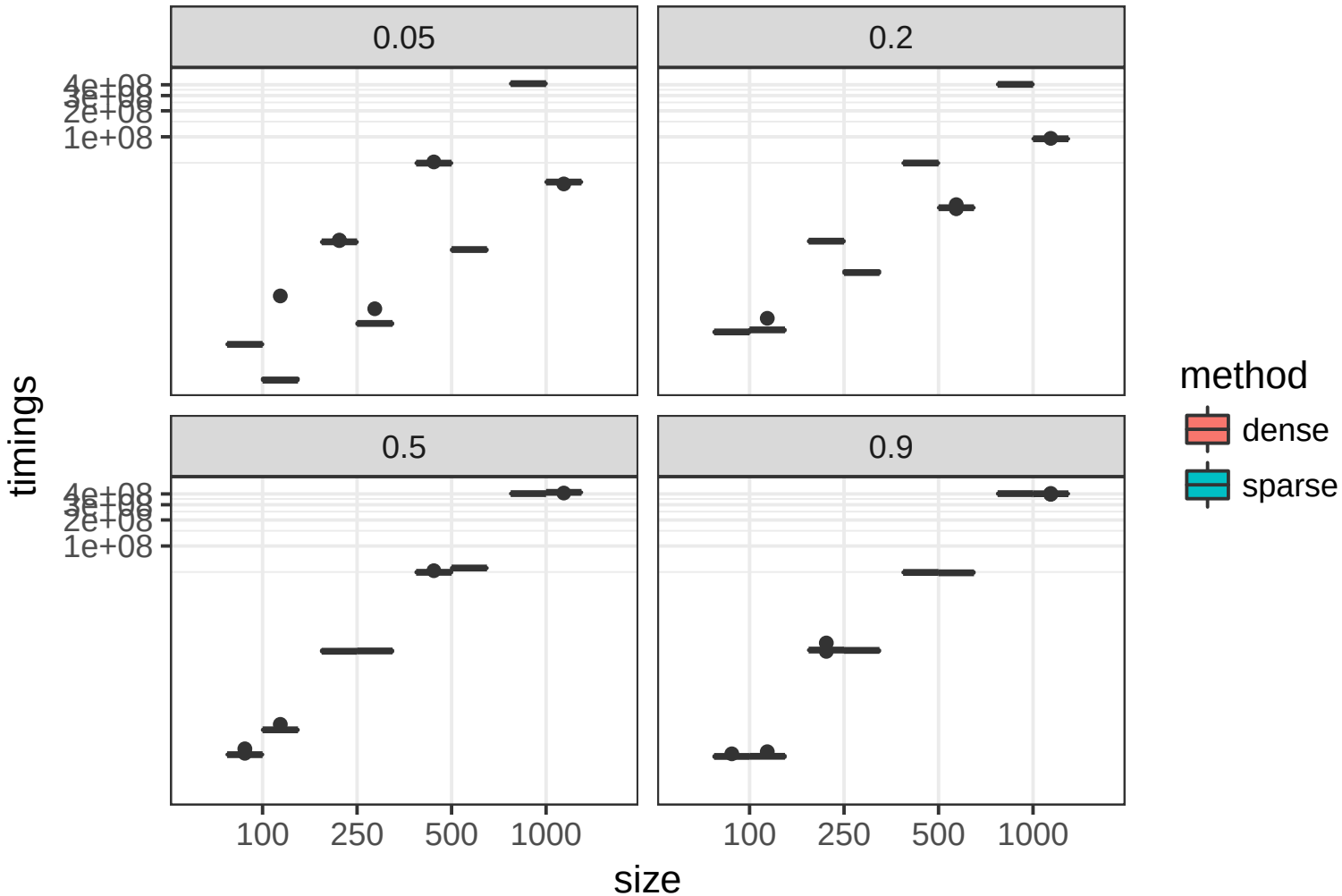


The Matrix package I

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 ))
}
```

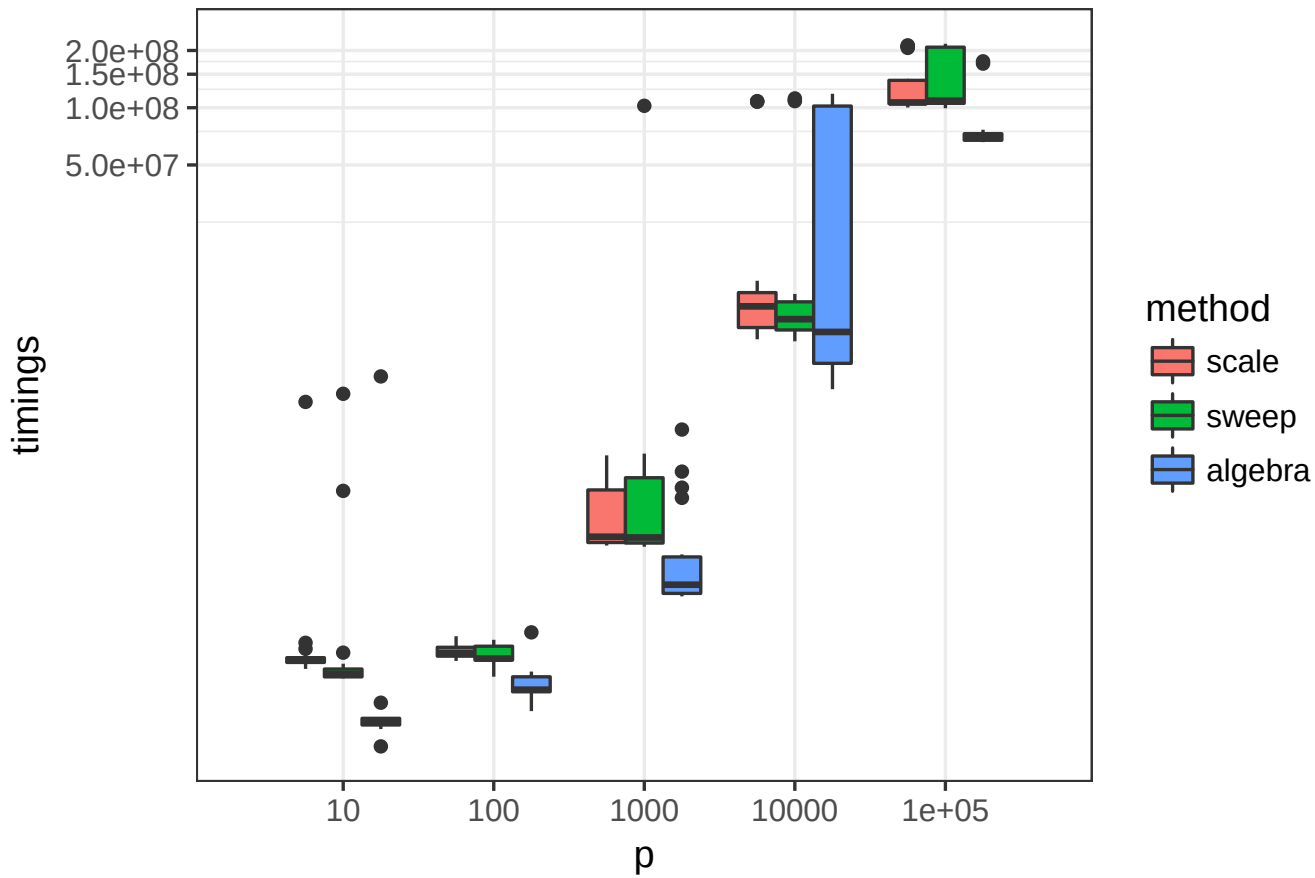
The Matrix package II



Mind some algebra

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

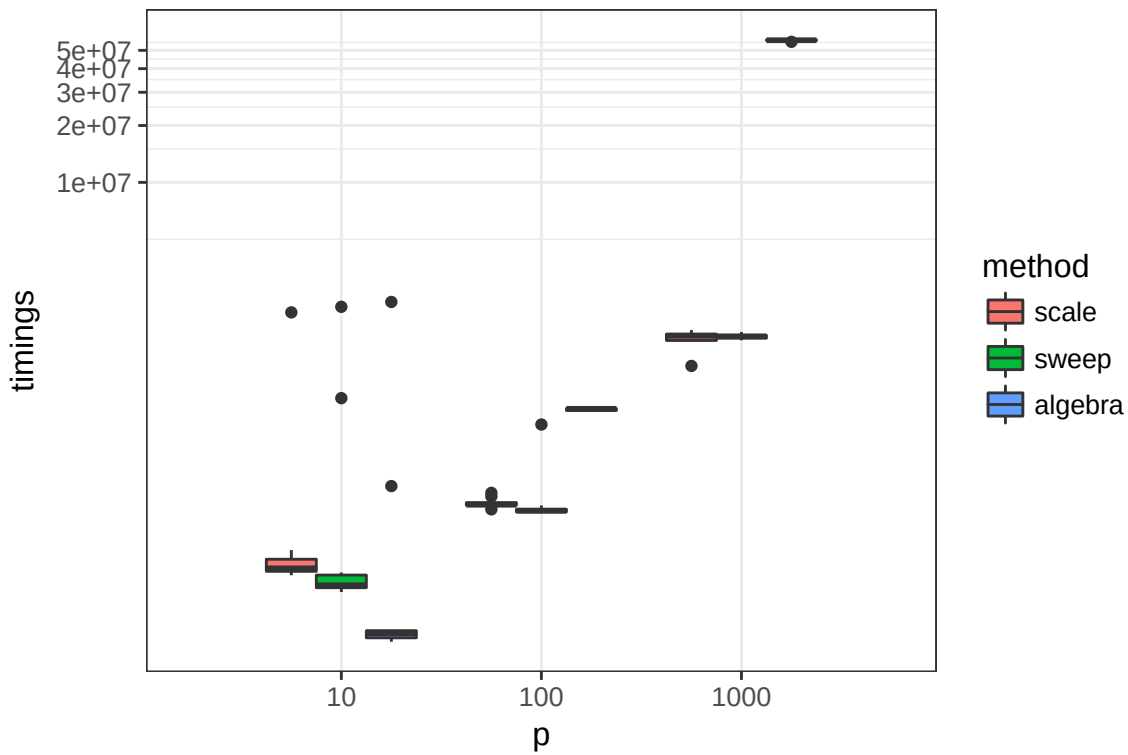
```
center1 <- function(x) return(scale(x, colMeans(x), FALSE))
center2 <- function(x) return(sweep(x, 2, colMeans(x), "-", check.margin = FALSE))
center3 <- function(x) return(x - outer(rep(1, nrow(x)), colMeans(x)) )

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
```



References

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

Eddelbuettel, D. (2013). *Seamless R and C++ integration with Rcpp*. Springer. Retrieved from http://dirk.eddelbuettel.com

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 http://adv-r.had.co.nz/