Some R tricks

basically for more speed

Rochebrune – March 2016

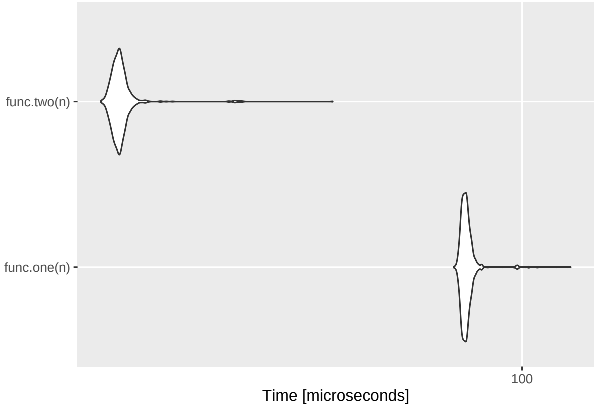
- The R inferno, Patrick Burns
  http://www.burns-stat.com/documents/books/the-r-inferno/
- FasteR! HigheR! StrongeR!, Noam Ross
  http://www.noamross.net/blog/2013/4/25/faster-talk.html
- Seamless R and C++ integration with Rcpp, Dirk EddelBuettel
  http://dirk.eddelbuettel.com
- Hadley Wickham, ggplot2, an implementation of the grammar of graphics
  http://had.co.nz/, http://ggplot2.org/, http://yihui.name/knitr/

Part I

Benchmark your code

How to quickly benchmark your code

```
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)
autoplot(res)
```



How to profile your code I

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

```r
## 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)
}
```

How to profile your code II

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

```r
Rprof(file="profiling.out", interval=0.05)
res <- my.func(1000,500)

## Loading required package:  MASS

Rprof(NULL)
```

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

##                      self.time self.pct total.time total.pct
## "La.svd"                  1.00    74.07       1.00     74.07
## ".External2"              0.05     3.70       0.10      7.41
## "matrix"                  0.05     3.70       0.10      7.41
## "aperm.default"           0.05     3.70       0.05      3.70
## "apply"                   0.05     3.70       0.05      3.70
## "is.finite"               0.05     3.70       0.05      3.70
## "na.omit.data.frame"      0.05     3.70       0.05      3.70
## "rnorm"                   0.05     3.70       0.05      3.70
```

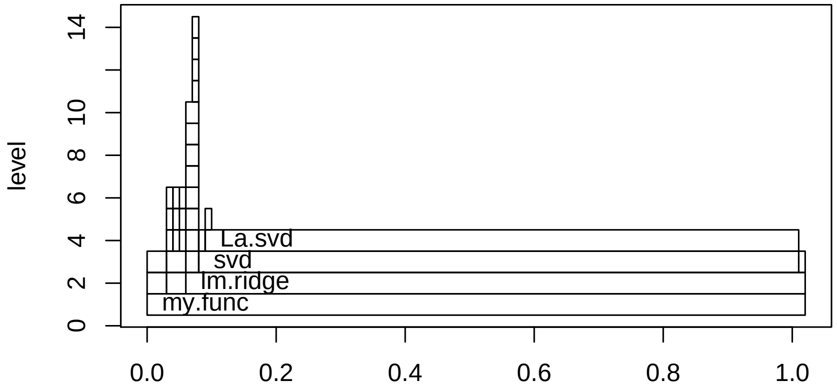
How to profile your code III

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


##                         total.time total.pct self.time self.pct
## "<Anonymous>"                 1.35    100.00      0.00     0.00
## "block_exec"                  1.35    100.00      0.00     0.00
## "call_block"                  1.35    100.00      0.00     0.00
## "doTryCatch"                  1.35    100.00      0.00     0.00
## "eval"                        1.35    100.00      0.00     0.00
## "evaluate_call"               1.35    100.00      0.00     0.00
## "FUN"                         1.35    100.00      0.00     0.00
## "handle"                      1.35    100.00      0.00     0.00
## "in_dir"                      1.35    100.00      0.00     0.00
## "knit"                        1.35    100.00      0.00     0.00
## "lapply"                      1.35    100.00      0.00     0.00
## "my.func"                     1.35    100.00      0.00     0.00
## "process_file"                1.35    100.00      0.00     0.00
## "process_group"               1.35    100.00      0.00     0.00
## "process_group.block"         1.35    100.00      0.00     0.00
## "try"                         1.35    100.00      0.00     0.00
## "tryCatch"                    1.35    100.00      0.00     0.00
## "tryCatchList"                1.35    100.00      0.00     0.00
## "tryCatchOne"                 1.35    100.00      0.00     0.00
## "withCallingHandlers"         1.35    100.00      0.00     0.00
## "withVisible"                 1.35    100.00      0.00     0.00
## "lm.ridge"                    1.15     85.19      0.00     0.00
## "svd"                         1.05     77.78      0.00     0.00
## "La.svd"                      1.00     74.07      1.00    74.07
## ".External2"                  0.10      7.41      0.05     3.70
## "matrix"                      0.10      7.41      0.05     3.70
## "scale"                       0.10      7.41      0.00     0.00
## "scale.default"               0.10      7.41      0.00     0.00
## "aperm.default"               0.05      3.70      0.05     3.70
## "apply"                       0.05      3.70      0.05     3.70
## "is.finite"                   0.05      3.70      0.05     3.70
```

How to profile your code III

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

```r
library(profr)
profiling <- profr({my.func(1000,500)}, interval=0.01)
plot(profiling)
```



Part II

Use multiple cores for your simulation

The `do.call` function

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 disposable, stored in a list like that

```
res[[1]]

##   method       mse      timing
## 1  lasso 0.7862968   0.9399695
## 2  ridge 0.5057219   0.7958627
## 3  bayes 0.9310022 115.8219670

length(res)

## [1] 100
```

How would you store them in a single data frame?

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

## [1] 300   3
```

The do.call function

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 disposable, stored in a list like that

```
res[[1]]


##   method       mse      timing
## 1  lasso 0.7862968   0.9399695
## 2  ridge 0.5057219   0.7958627
## 3  bayes 0.9310022 115.8219670


length(res)


## [1] 100
```

How would you store them in a single data frame?

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


## [1] 300   3
```

Parallelizing is very easy I

Do some parallel computation as soon as you do simulations (this should happen sometimes)

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

## [1] 4
```

My simulation study estimates the test error from ridge regression

```r
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 <- 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)))
}
```

Parallelizing is very easy II

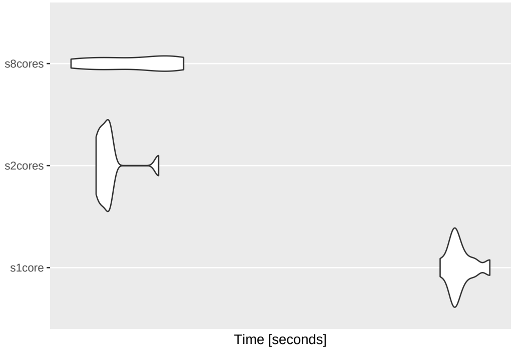
```r
out <- mclapply(1:8, one.simu, mc.cores=cores)
head(do.call(rbind, out))
```

```
##      err      sd
## [1,] 13.72301 18.94939
## [2,] 10.66215 16.23548
## [3,] 9.22876  13.05196
## [4,] 9.438628 12.84848
## [5,] 10.57839 14.97044
## [6,] 12.95024 17.24075
```

Be careful though...

- Parallelize piece of code complex enough
- Do not choose stupidly the number of cores

```
res <- microbenchmark(s1core = mclapply(1:8, one.simu, mc.cores=1),
                      s2cores = mclapply(1:8, one.simu, mc.cores=2),
                      s8cores = mclapply(1:8, one.simu, mc.cores=8), times=10)
```



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`

Example

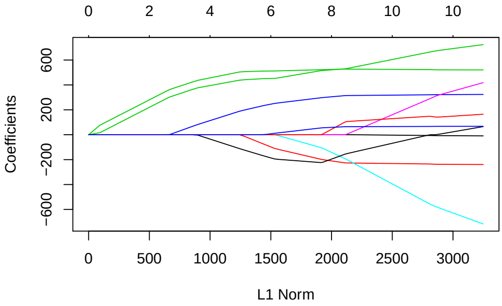
Work in progress with Avner for "jacknifing" a lasso solution path

```r
rm(list=ls())
library(lars)
library(glmnet)
## the diabetes data set (part of the lars package)
data(diabetes)
y <- diabetes$y
x <- diabetes$x
n <- length(y)
```

The Reduce function II

A single lasso fit

```
## recover a grid of lambda on the complete data set
lasso <- glmnet(x,y)
```



The Reduce function III

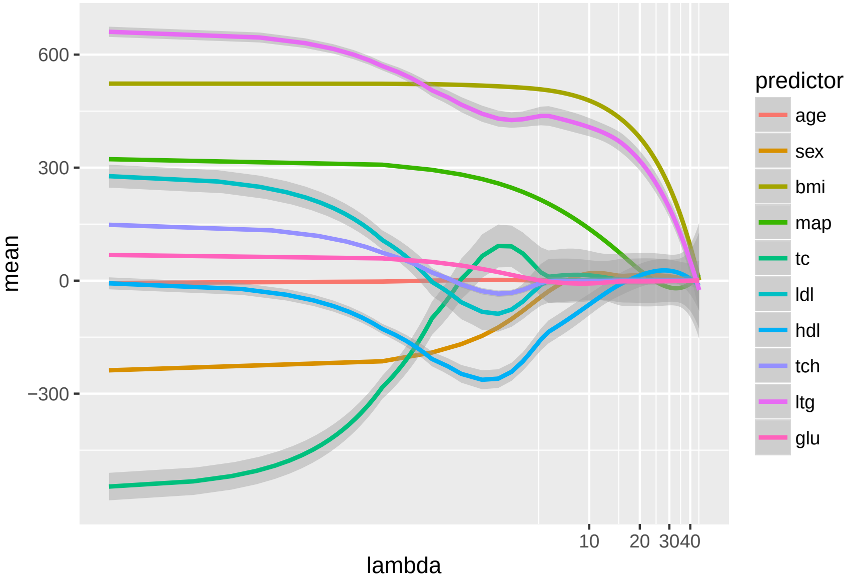
Jacknifing the path

```
library(parallel)
## compute the regularization paths for all subsets,
## removing one individual at once
paths <- 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)
```

The Reduce function IV



Part III

Be aware of what R is good (and bad) for

Use the vector capabilities of R

Any algebraic operation should be thought in a "vectorized" way

```r
exp2.1 <- sum(2^(0:10)/c(1,cumprod(1:10))) ## good
exp2.2 <- 1
for(k in 1:10) ## bad
  exp2.2 <- exp2.2 + 2^k/factorial(k)
```

Even non-algebraic operation should be thought as algebraic:

```r
outer(1:4,c("A","B","C","D"),FUN=paste,sep="-")
```

```
##      [,1]  [,2]  [,3]  [,4] 
## [1,] "1-A" "1-B" "1-C" "1-D"
## [2,] "2-A" "2-B" "2-C" "2-D"
## [3,] "3-A" "3-B" "3-C" "3-D"
## [4,] "4-A" "4-B" "4-C" "4-D"
```

Use the vector capabilities of R

Any algebraic operation should be thought in a “vectorized” way

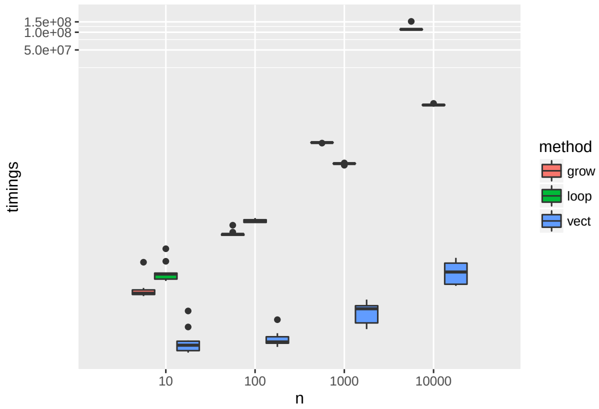
```
exp2.1 <- sum(2^(0:10)/c(1,cumprod(1:10))) ## good
exp2.2 <- 1
for(k in 1:10) ## bad
  exp2.2 <- exp2.2 + 2^k/factorial(k)
```

Even non-algebraic operation should be thought as algebraic:

```
outer(1:4,c("A","B","C","D"),FUN=paste,sep="-")

##      [,1]  [,2]  [,3]  [,4] 
## [1,] "1-A" "1-B" "1-C" "1-D"
## [2,] "2-A" "2-B" "2-C" "2-D"
## [3,] "3-A" "3-B" "3-C" "3-D"
## [4,] "4-A" "4-B" "4-C" "4-D"
```

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}
```



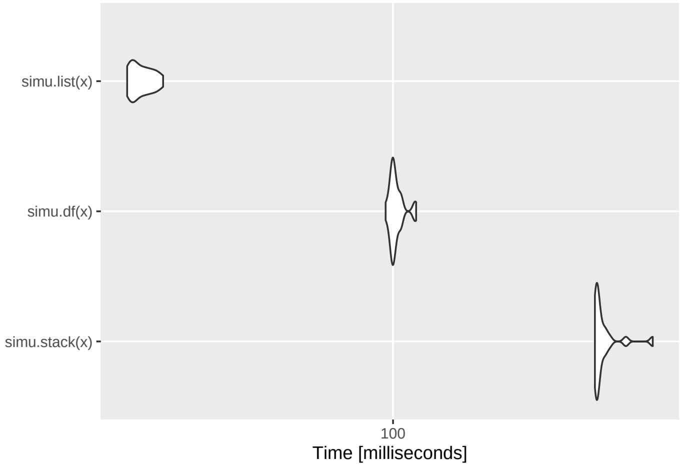
Do not stack objects I

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

```r
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)
}
```

Do not stack objects II

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



Use the [a-z]*pply family

Example with factors (tapply)

```
data <- rnorm(100)
sexe <- factor(sample(c("H","F"),100,rep=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)

```
data(oats)
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]))
```

Use the [a-z]*pply family

Example with factors (tapply)

```
data <- rnorm(100)
sexe <- factor(sample(c("H","F"),100,rep=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]))
```

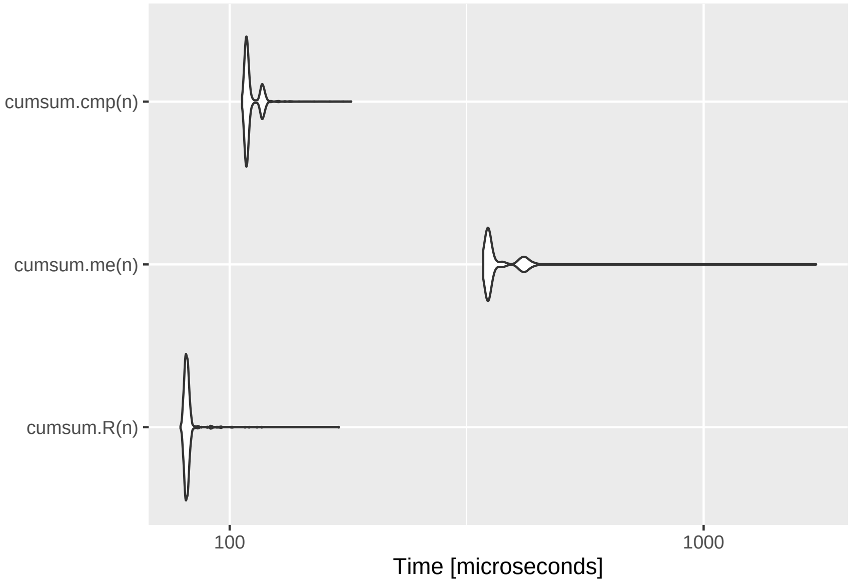
Example with list or data.frame (sapply/lapply)

```
data(oats)
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]))
```

Compile your functions I

```r
cumsum.R <- function(n) {
  x <- rnorm(n)
  return(cumsum(x))
}

cumsum.me <- function(n) {
  x <- rnorm(n)
  res <- 0
  for (i in 1:length(x)) {
    res <- res+x[i]
  }
  return(res)
}

library(compiler)
cumsum.cmp <- cmpfun(cumsum.me)

n <- 1000
res <- microbenchmark(cumsum.R(n), cumsum.me(n), cumsum.cmp(n), times=1000)
```

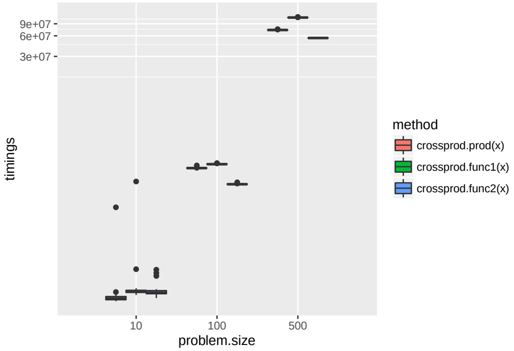
Compile your functions II



The `crossprod` function

As can be guessed, it computes the cross-product between two vector or matrices. . . and is generally fastest than % * % !

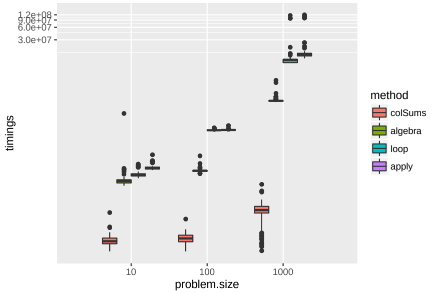
```
crossprod.prod <- function(x) return(t(x) %*% x)
crossprod.func1 <- function(x) return(crossprod(x,x))
crossprod.func2 <- function(x) return(crossprod(x))
```



The row/colSums family

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])
  return(res)
}
```



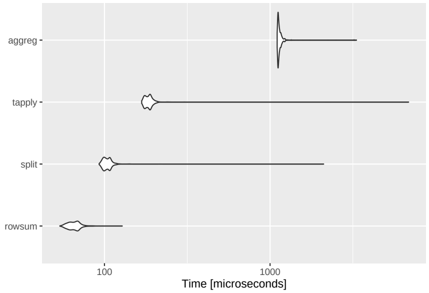
The secret function `rowsum`

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

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

```
## [1] 102.87880  99.03421  93.72513  97.05151  89.06845
```

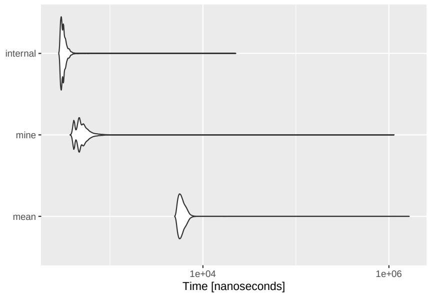
```r
res <- microbenchmark(
 rowsum = rowsum(vec, grp),
 split  = sapply(split(vec, grp), sum),
 tapply = tapply(vec, grp, sum),
 aggreg = aggregate(vec, list(grp), sum),
times=1000)
```



Internal function are faster

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

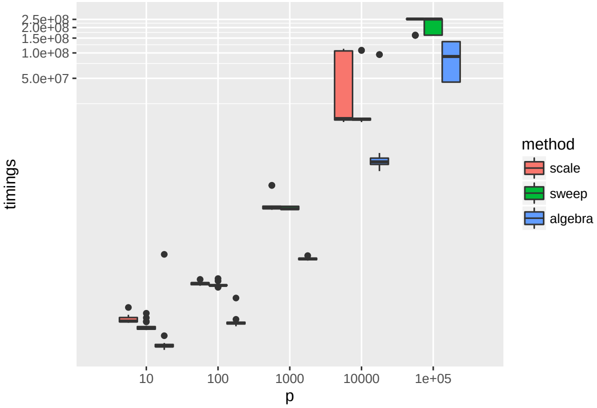
```r
x <- rnorm(100)
res <- microbenchmark(mean = mean(x),
                      mine = sum(x)/length(x),
                      internal = .Internal(mean(x)), times = 1e+05)
```



The sweep function

Fancy way to apply a statistic on a given dimension of an array.

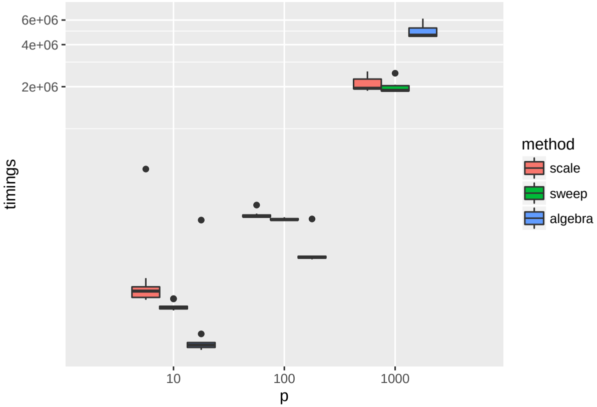
```r
center1 <- function(x) return(scale(x, colMeans(x), FALSE))
center2 <- function(x) return(sweep(x, 2, colMeans(x), "-", check.margin=FALSE))
center3 <- function(x) return(x - outer(rep(1, nrow(x)), colMeans(x)) )

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



Basic algebra does not always pay

Example for scaling a matrix

```r
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
```



Mind some algebra I

Example by inverting a positive definite matrices

```r
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)))
}
```

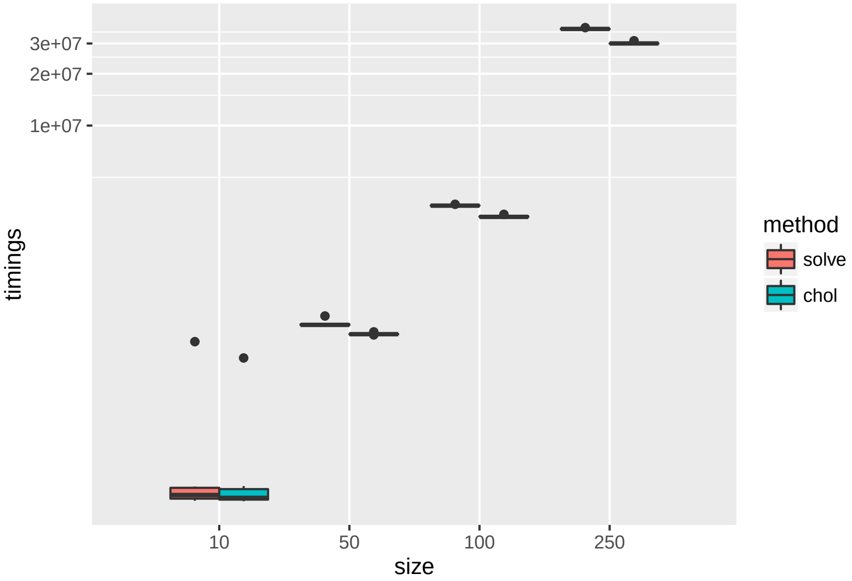
Mind some algebra II

```r
out <- do.call(rbind,
               lapply(c(10,50,100,250),
                      bench.p.fixed, times=10)
               )

head(out)
```

```
##   method timings size
## 1   chol  239439   10
## 2  solve  786924   10
## 3  solve   68059   10
## 4  solve   52322   10
## 5  solve   49488   10
## 6   chol   45483   10
```

```r
p <- ggplot(out, aes(x=size, y=timings, fill=method)) +
  geom_boxplot() +  coord_trans(y="log10")
```

Mind some algebra III



Part IV

Remind that R is object oriented

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 <u>convenient</u> 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
```

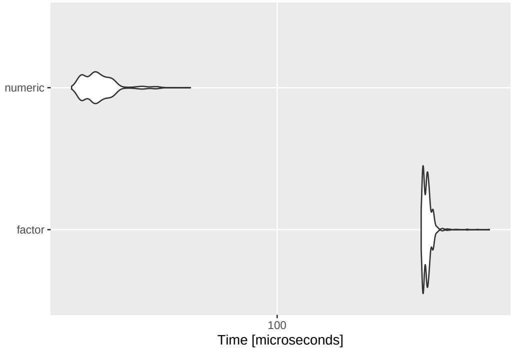
Factor conversion are slow (`nlevels`)

Do not use `factor` if you need to perform just one operation on it.

```r
nlevels.factor <- function(n,K) {
  x <- sample(1:K, n, rep=TRUE)
  return(nlevels(factor(x)))
}
```

```r
nlevels.numeric <- function(n,K) {
  x <- sample(1:K, n, rep=TRUE)
  return(length(unique(x)))
}
```

```r
res <- microbenchmark(factor = nlevels.factor (1000,10),
                      numeric = nlevels.numeric(1000,10), times=1000)
```



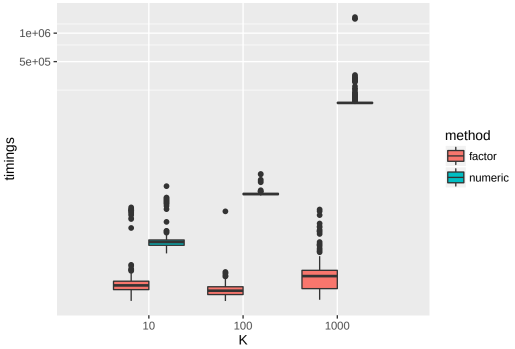
Operations on factors are fast (`nlevels`)

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

```r
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)))
}))
```

```r
head(res)
```

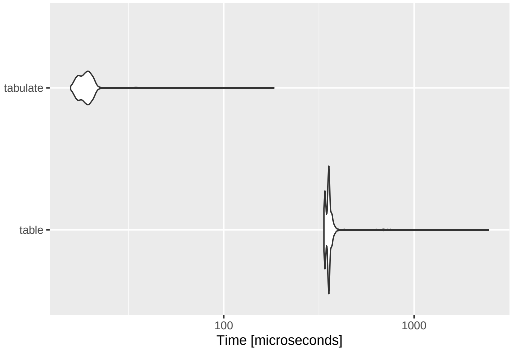
```
##    method timings  K
## 1  factor   15505 10
## 2 numeric   20432 10
## 3 numeric    7436 10
## 4 numeric    6203 10
## 5  factor    3353 10
## 6  factor    2521 10
```



Avoid `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
res <- microbenchmark(table    = table   (sample(1:K, n, rep=TRUE)),
                      tabulate = tabulate(sample(1:K, n, rep=TRUE)),
                      times=1000)
```



Use the Matrix package I

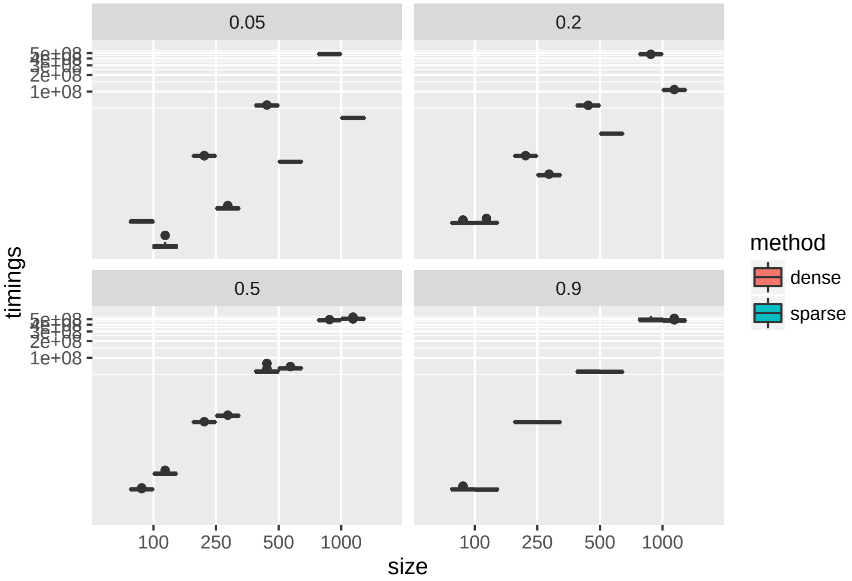
Propose a collection of functions for of matrix algebra adapted to the type of matrix at hand (sparse, diagonal, triangular, block diagonal, etc.)

```r
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 ))
}
```

Use the Matrix package II

```
par <- expand.grid(n=c(100,250,500,1000), density=c(.05,.2,.5,.9))
out <- do.call(rbind,
         lapply(1:nrow(par),
             function(k) {
               return(bench.par.fixed(par[k, ]))
         }))
out$density <- factor(out$density)
out$method  <- factor(out$method)
out$size    <- factor(out$size)
p <- ggplot(out, aes(x=size, y=timings, fill=method)) +
    geom_boxplot() +  coord_trans(y="log10") + facet_wrap(~density, nrow=2)
```

Use the Matrix package III



Part V

Use (supposedly) lower-level languages

Interfacing C++ with R is really easy I

Example 1

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.

```r
library(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

Example 1

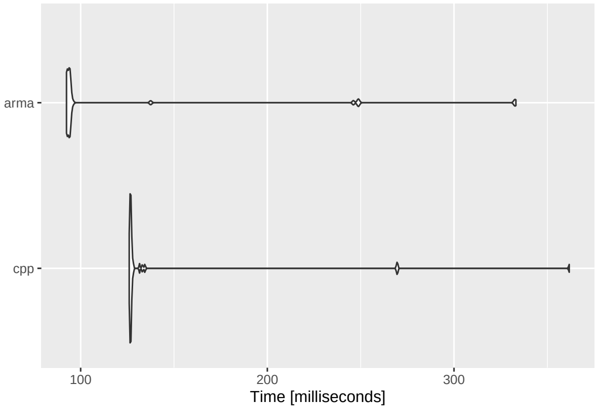
The Armadillo library for linear algebra facilitates even more the integration

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

Interfacing C++ with R is really easy III

Example 1

```r
x <-  runif(1e7, 1,2)
res <- microbenchmark(cpp = rcpp(x), arma = Arma(x), times=40)
```



Interfacing C++ with R is really easy

Example 2: from a work with C. Lévy-Leduc and V. Brault

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

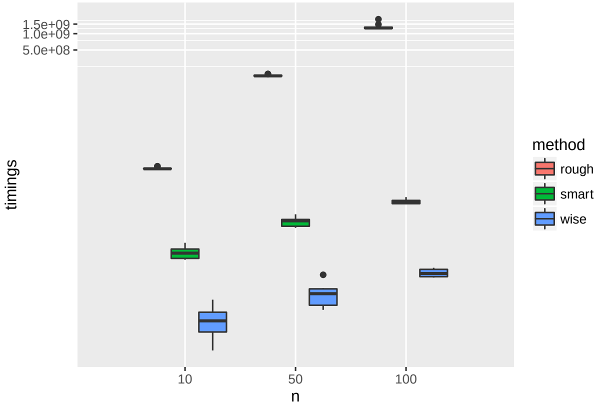
$$\mathrm{vec}(\mathbf{T}\mathbf{B}\mathbf{T}^\top) = (\mathbf{T} \otimes \mathbf{T}) \times \mathrm{vec}(\mathbf{B}).$$

```
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

Example 2

```r
B.ls <-  sapply(c(10, 50, 100), function(n) matrix(runif(n**2),n,n))
res <- do.call(rbind, lapply(B.ls, function(B) {
 out <- microbenchmark(rough = prod.rough(B),
                       smart = prod.smart(B),
                       wise  = prod.wise(B), times=20)
 return(data.frame(method = out$expr, timings = out$time, n = factor(ncol(B))))
}))
```


1.5e+09
1.0e+09
5.0e+08
timings
n
method
rough
smart
wise