Vectorisation in R

L. Gatto

January 24, 2013

Outline

Vectorisation

A **vectorised computation** is one that, when applied to a vector (of length greater than 1), automatically operates directly on all elements of the input vector.

```
(x <- 1:5)
(y <- 5:1)
x + y
```

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

"Many operations in R are vectorized, and understanding and using vectorization is an essential component of becoming a proficient programmer." R Gentleman in *R Programming for Bioinformatics*

Recycling rule

What is `x` and `y` are of different length: the shorter vector is replicate so that its length matches the longer ones.

```
(x <- 1:6)
(y <- 1:2)
x+y
```

```
 [1] 1 2 3 4 5 6
 [1] 1 2
 [1] 2 4 4 6 6 8
```

If the shorter vector is not an even multiple of the longer, a warning is issued.

With matrices (1)

Matrices must be conformable.

```
m <- matrix(1:9, 3)
m
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
n <- matrix(9:1, 3)
n
```

```
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
[3,]    7    4    1
```

With matrices (2)

```
m * n
```

```
     [,1] [,2] [,3]
[1,]    9   24   21
[2,]   16   25   16
[3,]   21   24    9
```

```
m %*%n
```

```
     [,1] [,2] [,3]
[1,]   90   54   18
[2,]  114   69   24
[3,]  138   84   30
```

diff example (1)

Compute difference between times of events, `e`. Given `n` events, there will be `n-1` inter-event times. `interval[i] <- e[i+1] - e[i]`

Procedural implementation:

```
diff1 <- function(e) {
  n <- length(e)
  interval <- rep(0, n - 1)
  for (i in 1:(n - 1)) {
    interval[i] <-
      e[i + 1] - e[i]
  }
  interval
}
e <- c(2, 5, 10.2, 12, 19)
diff1(e)
```

```
 [1] 3.0 5.2 1.8 7.0
```

diff example (2)

Vectorised implementation

```
diff2 <- function(e) {
  n <- length(e)
  e[-1] - e[-n]
}
e <- c(2, 5, 10.2, 12, 19)
diff2(e)
```

```
 [1] 3.0 5.2 1.8 7.0
```

```
all.equal(diff1(e), diff2(e))
```

```
 [1] TRUE
```

Outline

When using `for` loops

Initialising the result variable before iteration to avoid unnecessary copies at each iteration substantially increases performance.

```
n <- 5e3
f1 <- function(n) {
  a <- NULL; for (i in 1:n) a <- c(a, sqrt(i)); a }
f2 <- function(n) {
  a <- numeric(n); for (i in 1:n) a[i] <- sqrt(i); a }
system.time(f1(n))
system.time(f2(n))
```

```
   user  system elapsed
  0.076   0.004   0.079
   user  system elapsed
  0.012   0.000   0.014
```

*apply functions

How to apply a function, iteratively, on a set of elements?

`apply(X, MARGIN, FUN, ...)`

- `MARGIN` = 1 for row, 2 for cols.
- `FUN` = function to apply
- `...` = extra args to function.
- `simplify` = should the result be simplified if possible.

*apply functions are (generally) **NOT** faster than loops, but more succint and thus clearer.

Usage (1)

```
v <- rnorm(1000) ## or a list
res <- numeric(length(v))

for (i in 1:length(v))
  res[i] <- f(v[i])

res <- sapply(v, f)

## if f is vectorised
f(v)
```

Usage (2)

```
## M is a matrix/data.frame/array
rowResults <- numeric(nrow(M))
colResults <- numeric(ncol(M))

for (i in 1:nrow(M))
  rowResults <- f(M[i, ])

for (j in 1:ncol(M))
  colResults <- f(M[, j])

rowResults <- apply(M, 1, f)
colResults <- apply(M, 2, f)
```

*apply functions

apply	matrices, arrays, data.frames
lapply	lists, vectors
sapply	lists, vectors
vapply	with a pre-specified type of return value
tapply	atomic objects, typically vectors
by	similar to tapply
eapply	environments
mapply	multiple values
rapply	recursive version of lapply
esApply	`ExpressionSet`, defined in `Biobase`

See also the `BiocGenerics` package for `[l|m|s|t]apply` S4 generics, as well as parallel versions in the `parallel` package.

See also the `plyr` package, that offers its own flavour of **apply** functions.

Other functions

- `replicate` – repeated evaluation of an expression
- `aggregate` – compute summary statistics of data subsets
- `ave` – group averages over level combinations of factors
- `sweep` – sweep out array summaries

Anonymous functions

A function defined/called without being assigned to an identifier and generally passed as argument to other functions (and in particular `apply` functions).

```
M <- matrix(rnorm(100), 10)
apply(M, 1, function(Mrow) 'do something with Mrow')
apply(M, 2, function(Mcol) 'do something with Mcol')
```

Example - extract (1)

Extracting the i^{th} column of elements in a list:

```
A <- matrix(1:4, 2)
B <- matrix(1:6, 2)
L <- list(A, B)
sapply(L, function(x) x[,2])
```

```
     [,1] [,2]
[1,]    3    3
[2,]    4    4
```

Example - extract (2)

Extracting the i^{th} column of elements in a list:

```
A <- matrix(1:4, 2)
B <- matrix(1:6, 2)
L <- list(A, B)
lapply(L, "[", , 2)
```

```
 [[1]]
 [1] 3 4

 [[2]]
 [1] 3 4
```

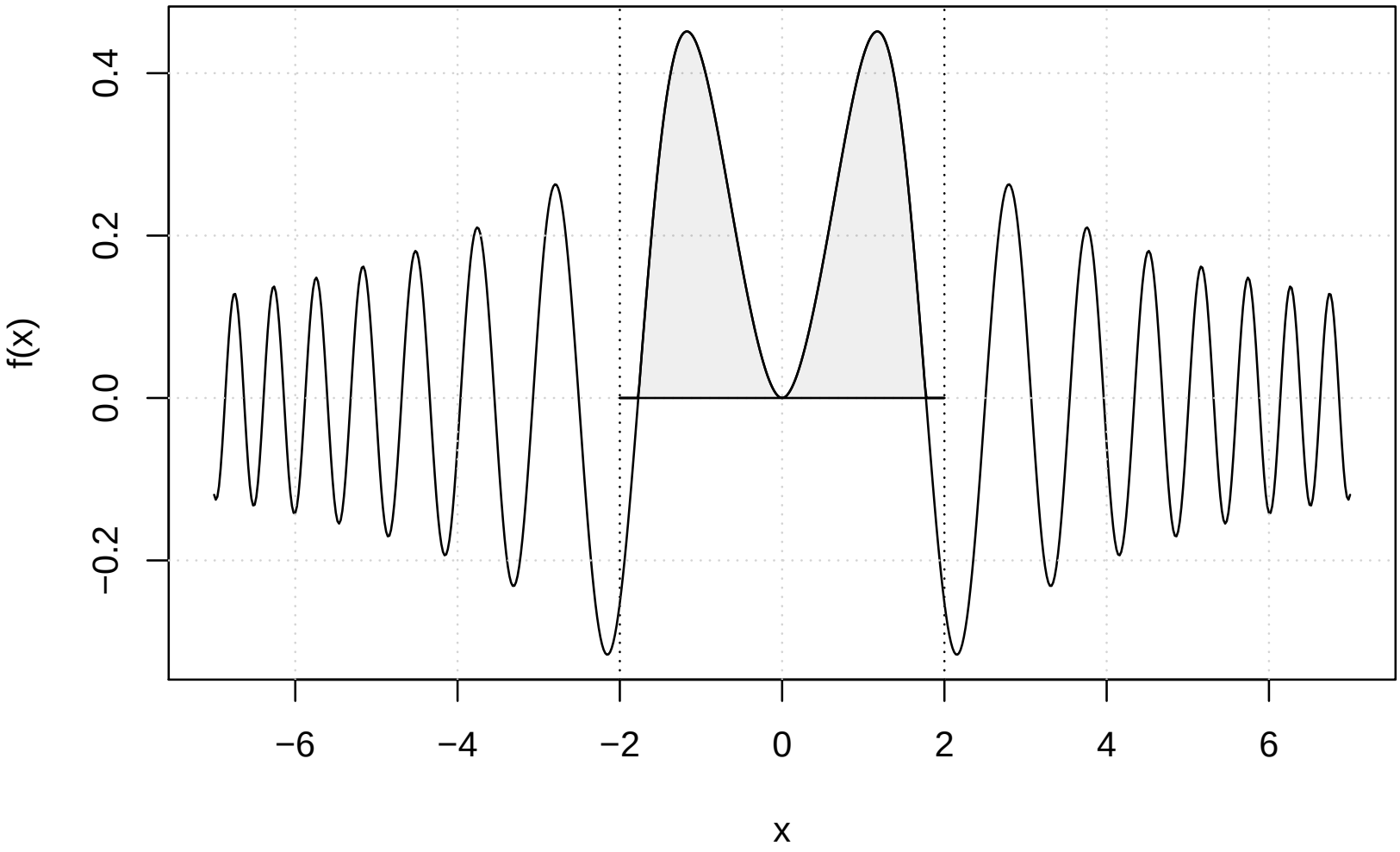
(See `help("[")` if the syntax is unexpected.)

Example - replicate

```
f <- function(d) {
  M <- matrix(runif(d^2), nrow=d)
  solve(M)
}
system.time(f(100))
(res <- replicate(10, system.time(f(100))[["elapsed"]]))

    user  system elapsed
   0.000   0.004   0.004
  [1] 0.002 0.003 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.0

summary(res)

    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.0020  0.0020  0.0020  0.0021  0.0020  0.0030
```

Example - integration (1)

$$f(x) = \frac{\sin(x^2)}{(a + |x|)}$$



Example - integration (2)

The `integrate` function approximates definite integrals by adaptive quadrature.

```
f <- function(x, a = 1) sin(x^2)/ (a + abs(x))
integrate(f, lower = -2, upper = 2)
```

```
 0.8077645 with absolute error < 1.5e-13
```

It is not vectorised.

```
lo <- c(-2, 0)
hi <- c(0, 2)
integrate(f, lower = lo, upper = hi)
```

```
 0.4038823 with absolute error < 7.4e-14
```

Example - integration (3)

To vectorise a function, we can explicitly wrap it inside a helper function that will take care of argument recycling (via `rep`), then loop over the inputs and call the non-vectorised function.

Example - integration (4)

To vectorise a function, we can explicitate the vectorised calculation using `mapply`

```
mapply(function(lo, hi) integrate(f, lo, hi)$value,
       lo, hi)
```

```
 [1] 0.4038823 0.4038823
```

Example - integration (5)

Create a vectorised form using `Vectorize`. It takes a function (here, an anonymous function) as input and returns a function.

```
Integrate <- Vectorize(
  function(fn, lower, upper)
  integrate(fn, lower, upper)$value,
  vectorize.args=c("lower", "upper")
  )
Integrate(f, lower=lo, upper=hi)
```

```
 [1] 0.4038823 0.4038823
```

Example - **tapply**

```
dfr <- data.frame(A = sample(letters[1:5], 100,
                             replace = TRUE),
                  B = rnorm(100))
tapply(dfr$B, dfr$A, mean)

          a           b           c           d           e
-0.12619087 -0.03547276  0.03428698 -0.22115673  0.01079840

tapply(dfr$B, dfr$A, summary)[1:2]

 $a
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.9970 -0.7942 -0.2892 -0.1262  0.7186  1.3430

 $b
     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-1.13100 -0.62530 -0.06107 -0.03547  0.46390  1.11800
```

Efficient apply-like functions

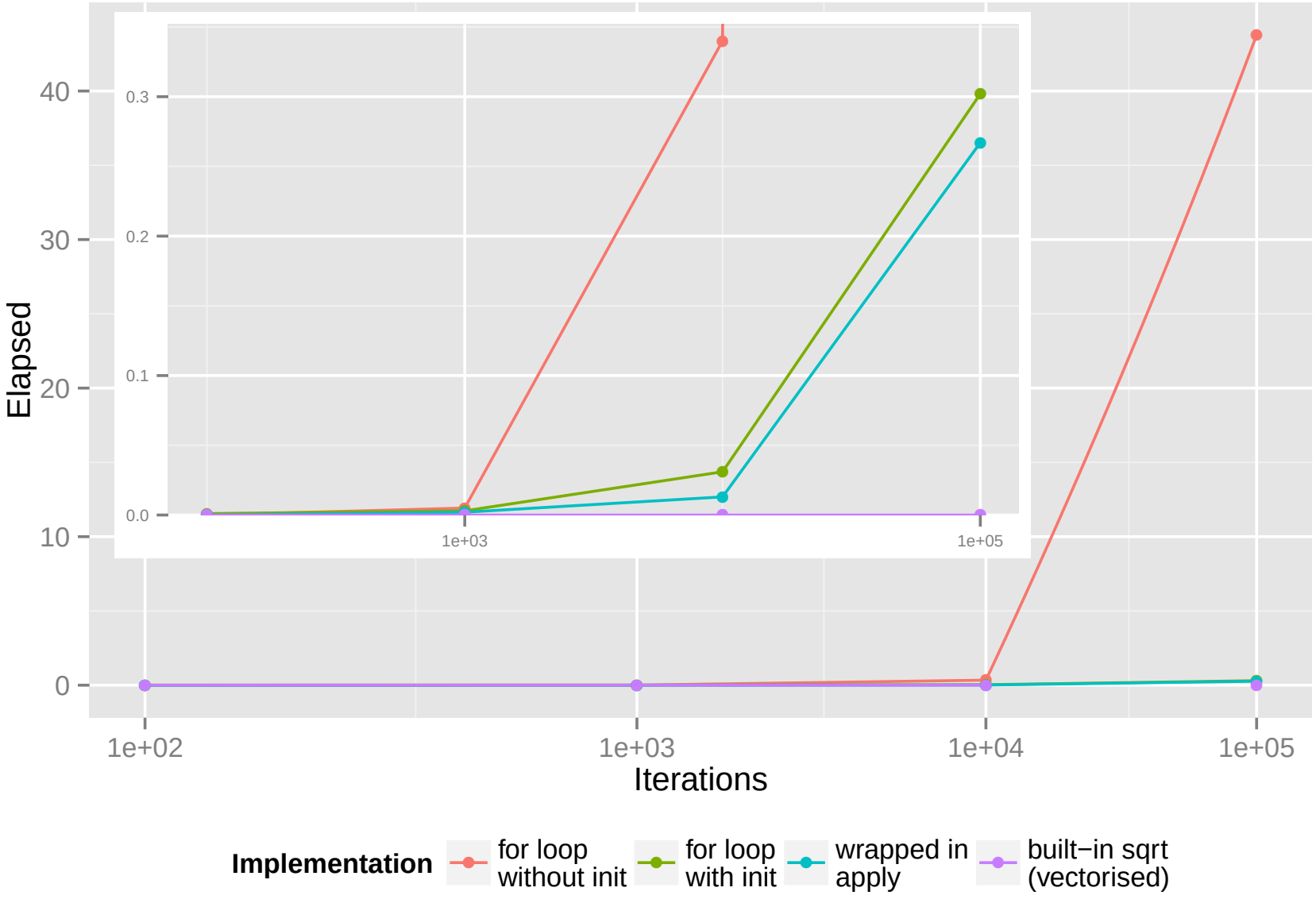
- In `base`: rowSums, rowMeans, colSums, colMeans
- In `Biobase`: rowQ, rowMax, rowMin, rowMedias, ...
- In `genefilter`: rowttests, rowFtests, rowSds, rowVars, ...

Generalisable on other data structures, like `ExpressionSet` instances.

Timings (1)

```
f1 <- function(n) {
  a <- NULL
  for (i in 1:n) a <- c(a, sqrt(i))
  a
}
f2 <- function(n) {
  a <- numeric(n)
  for (i in 1:n) a[i] <- sqrt(i)
  a
}

f3 <- function(n)
  sapply(seq_len(n), sqrt)

f4 <- function(n) sqrt(n)
```

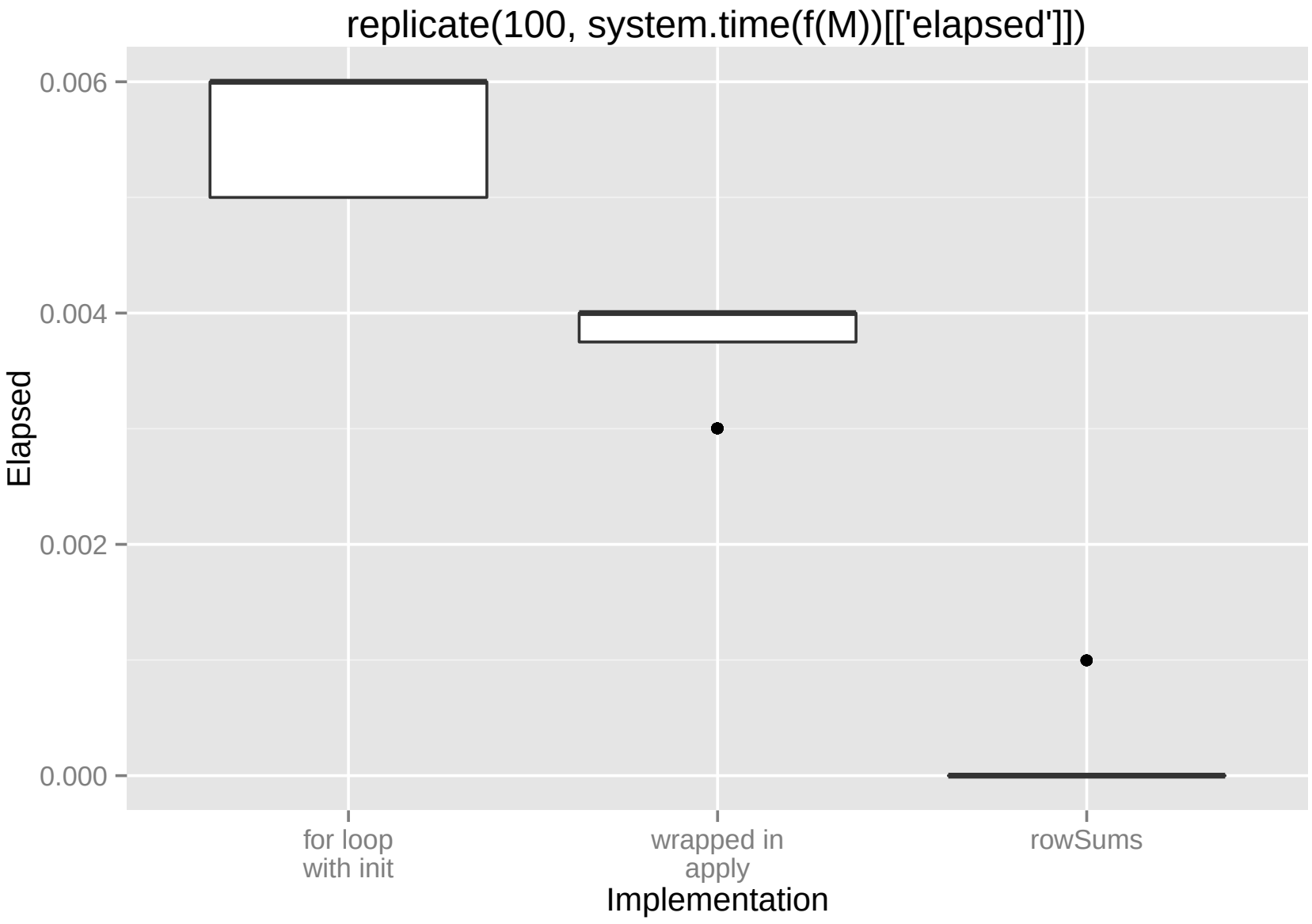
Timings (1)


Elapsed
40
30
20
10
0
0.3
0.2
0.1
0.0
1e+03
1e+05
1e+02
1e+03
1e+04
1e+05
Iterations
Implementation
for loop without init
for loop with init
wrapped in apply
built−in sqrt (vectorised)

Timings (2)

```
f1 <- function(M) {
  res <- numeric(nrow(M))
  for (i in 1:nrow(M))
    res[i] <- sum(M[i, ])
  res
}

f2 <- function(M)
  apply(M, 1, sum)

f3 <- function(M)
  rowSums(M)
```

Timings (2)


replicate(100, system.time(f(M))[['elapsed']])
0.006
0.004
0.002
0.000
Elapsed
for loop
with init
wrapped in
apply
rowSums
Implementation

Parallelisation

Vectorised operations are natural candidats for parallel execution.
See later, *Parallel computation* topic.

Outline

References

- R Gentleman, *R Programming for Bioinformatics*, CRC Press, 2008
- Ligges and Fox, *R Help Desk, How Can I Avoid This Loop or Make It Faster?* **R News**, Vol 8/1. May 2008.
- R Grouping functions: sapply vs. lapply vs. apply. vs. tapply vs. by vs. aggregate ... http://stackoverflow.com/questions/3505701/