Chapter 24: Improving Performance

Advanced R Book Group, Cohort 3

Megan Stodel (@MeganStodel, www.meganstodel.com)

15 March, 2021

Some overarching points

- It can be inefficient to 'improve performance' if you save a tiny amount of time on a rarely used process and spend hours achieving it.
- It is useful to be exposed to as much R information as possible to achieve a more intuitive sense of measures you can take and to gain a vocabulary that will allow you to more effectively search for answers.
- Keep a record of things as you troubleshoot.

Code organisation

Where you have a few options, write them up as functions so they are easy to compare with benchmarking.

Perform benchmarking with a good number of cases so it can be representative.

Check for existing solutions

- CRAN task views: aim to provide some guidance which packages on CRAN are relevant for tasks related to a certain topic.
- Reverse dependencies of Rcpp, as listed on its CRAN page. Since these packages use C++, they're likely to be fast.

Exercises (24.3.1)

What are faster alternatives to `lm()`? Which are specifically designed to work with larger datasets?

```
set.seed(129)

n <- 7
p <- 2
X <- matrix(rnorm(n * p), n, p) # no intercept!
y <- rnorm(n)
w <- rnorm(n)^2
```

```
mb <- microbenchmark::microbenchmark(lm(y~X),
                                     lm.fit(X,y),
                                     .lm.fit(X,y),
                                     speedglm::speedlm(y~X))

boxplot(mb, notch=TRUE)
```

```
## Warning in bxp(list(stats = structure(c(1385.001, 1454.1005, 1581.201,
## 2379.6015, : some notches went outside hinges ('box'): maybe set notch=FAL
```

Exercises (24.3.1)

What package implements a version of match() that's faster for repeated lookups? How much faster is it?

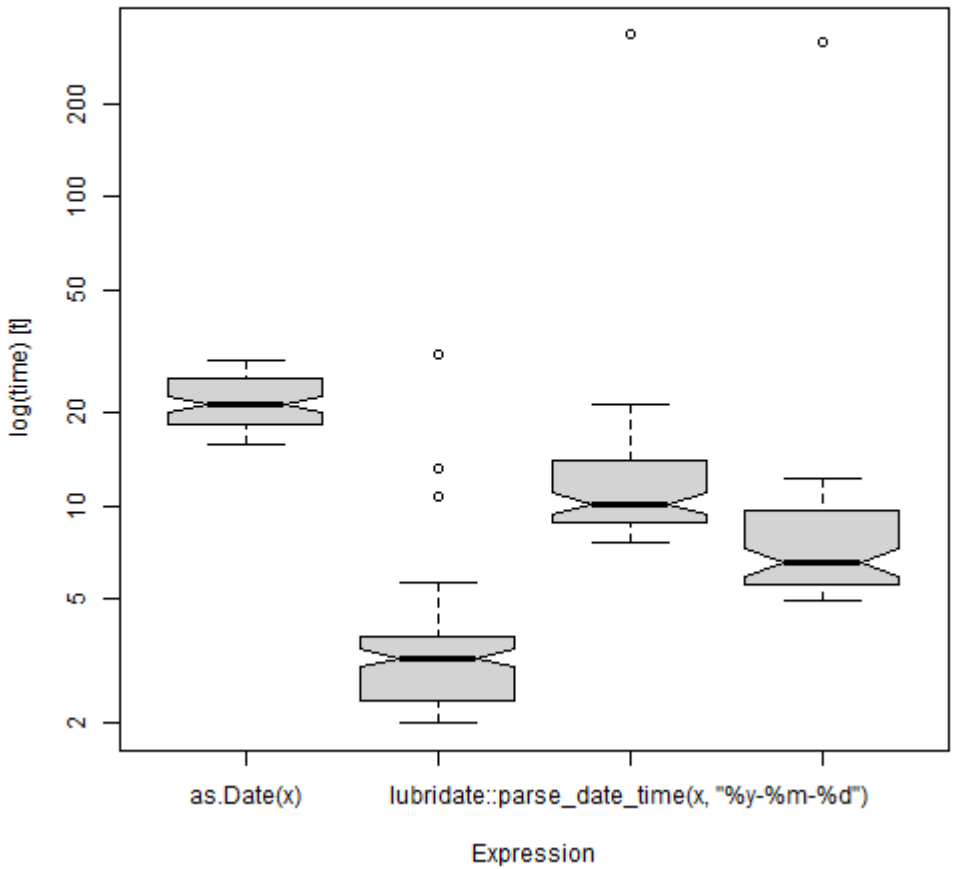
{fastmatch} - only slightly faster initially but much faster subsequently.

Exercises (24.3.1)

List four functions (not just those in base R) that convert a string into a date time object. What are their strengths and weaknesses?

```
set.seed(100)
x <- as.character(Sys.Date()-sample(40000, 1000, TRUE))
mb <- microbenchmark::microbenchmark(as.Date(x),
                                     date::as.date(x,"ymd"),
                                     lubridate::parse_date_time(x,'%y
                                     anytime::anydate(x))
```

```
boxplot(mb, notch=TRUE)
```



Exercises (24.3.1)

Which packages provide the ability to compute a rolling mean?

{zoo}, {roll}, {data.table}, {slider}...

Exercises (24.3.1)

What are the alternatives to optim()?

There is a CRAN task view on this.

Doing as little as possible

Functions that are more tailored to the task are easier to optimize.

(Discussion point: but less broadly useful?)

Things that might be relevant:

- The function should work with your data as it is, rather than coercing it to a different type. For example, using `apply()` on a data.frame is not ideal because the input gets turned into a matrix.
- Some functions are more efficient if you provide more information, i.e. specify argument values. This is because they either will need to intuit the best approach if you don't provide it, or could use one that is inefficient for your specific use case. For example, `unlist(x, use.names = FALSE)` is much faster than `unlist(x)`.
- If you are calling a method repeatedly, sometimes you can avoid the costs of method dispatch by doing the method lookup once only. For example, `mean.default()` is faster than `mean()` for some vectors.
- The size of the input matters - for example, using `mean()` and `mean.default()` does not have a real different for large vectors because the majority of the time is spent computing the mean and not finding the underlying implementation . So, if you are benchmarking and optimising, use realistic data.

NB: Sometimes speed is at the cost of safety because checks are being removed, so results are more likely to fail in unanticipated ways or handle unexpected (missing) data weirdly.

Exercises (24.4.3) Q1

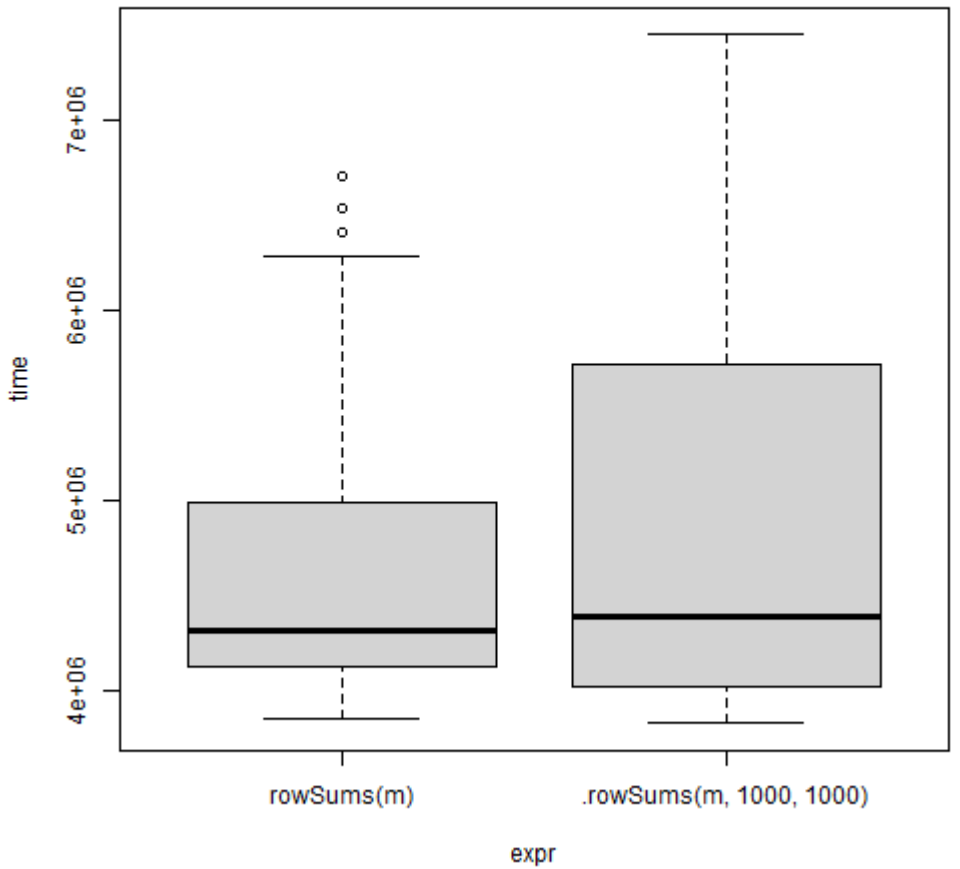
What's the difference between rowSums() and .rowSums()?

`rowSums()` is a wrapper around `.rowSums()`, adding in some extra stuff like checking the input and dealing wiht complex numbers. So we would expect it to be slower but safer.

```
m <- matrix(rnorm(1e6), nrow = 1000)

mb <- microbenchmark::microbenchmark(
  rowSums(m),
  .rowSums(m, 1000, 1000)
)
```

```
plot(mb)
```



But it's not even that much slower...so probably worth using!

Exercises (24.4.3) Q2

Make a faster version of chisq.test() that only computes the chi-square test statistic when the input is two numeric vectors with no missing values.

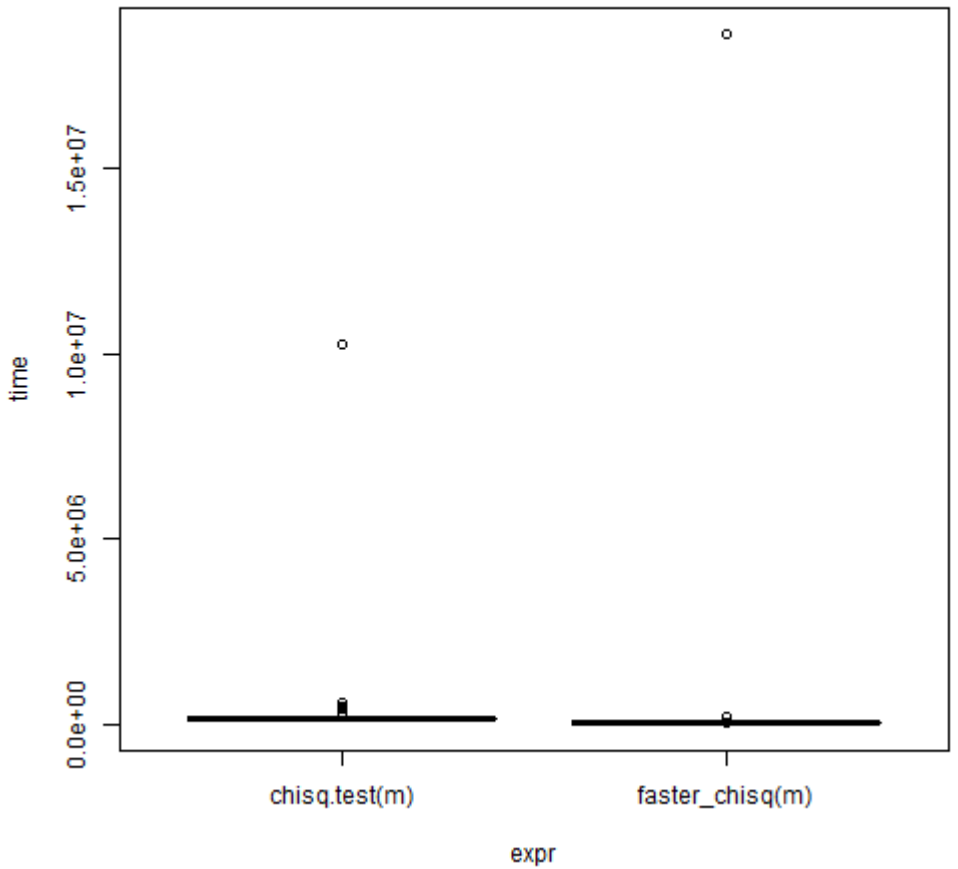
Looking at the source code for `chisq.test()` it includes a bunch of safety features like validating the input. Strip that all out...

```
faster_chisq <- function(xy_combo){
  margin1 <- rowSums(xy_combo)
  margin2 <- colSums(xy_combo)
  n <- sum(xy_combo)
  me <- tcrossprod(margin1, margin2) / n
  x_stat <- sum((xy_combo - me)^2 / me)
  df <- (length(margin1) - 1) * (length(margin2) - 1)
  p.value <- pchisq(x_stat, df = df, lower.tail = FALSE)
  list(x_stat = x_stat, df = df, p.value = p.value)
}
```

```
a <- 21:25
b <- seq(21, 29, 2)
m <- cbind(a, b)

mb <- microbenchmark::microbenchmark(
  chisq.test(m),
  faster_chisq(m)
)
```

```
plot(mb)
```


time
0.0e+00
5.0e+06
1.0e+07
1.5e+07
chisq.test(m)
faster_chisq(m)
expr

Vectorise

Makes problems simpler - think about entire vectors rather than the component parts. Loops are written in C instead of R, which are faster.

Some key applications:

- Vectorised matrix functions, like `rowSums()` and `colMeans()`.
- Vectorised subsetting, for example `x[is.na(x)] <- 0` will replace all missing values with 0.
- Good for extracting or replacing values in scattered locations.

But there is a downside: it is harder to predict how operations will scale.

You're often better off writing your vectorised function in C++ which is next chapter!

Excercises (24.5.1) Q1

The density functions, e.g., dnorm(), have a common interface. Which arguments are vectorised over? What does rnorm(10, mean = 10:1) do?

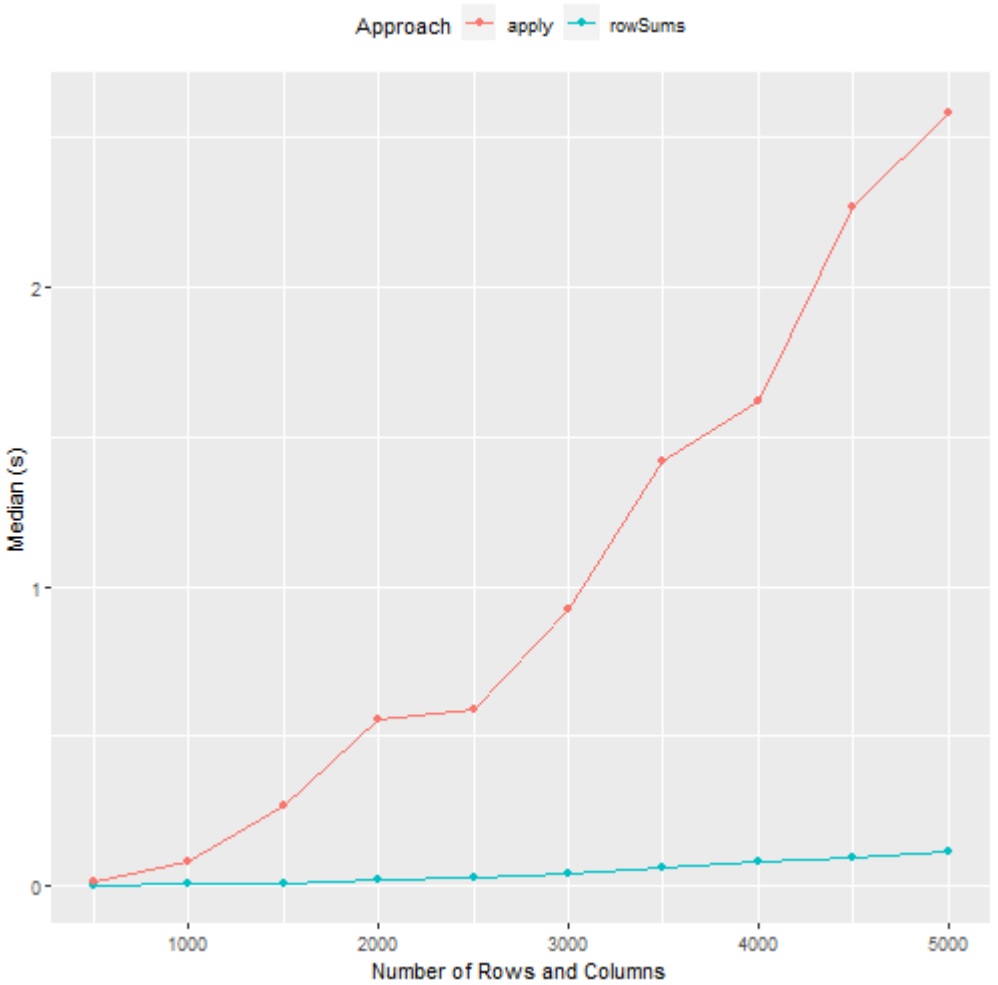
They are vectorised over their numeric arguments, which are the first argument, and then `mean` and `sd`.

The given function generates ten random numbers from different normal distributions, which differe in their means based on the sequence.

Excercises (24.5.1) Q1

Compare the speed of apply(x, 1, sum) with rowSums(x) for varying sizes of x.

```
rowsums <- bench::press(
  p = seq(500, 5000, length.out = 10),
  {
    mat <- tcrossprod(rnorm(p), rnorm(p))
    bench::mark(
      rowSums = rowSums(mat),
      apply = apply(mat, 1, sum)
    )
  }
)
```


Approach
apply
rowSums
Median (s)
0
1
2
1000
2000
3000
4000
5000
Number of Rows and Columns

Excercises (24.5.1) Q1

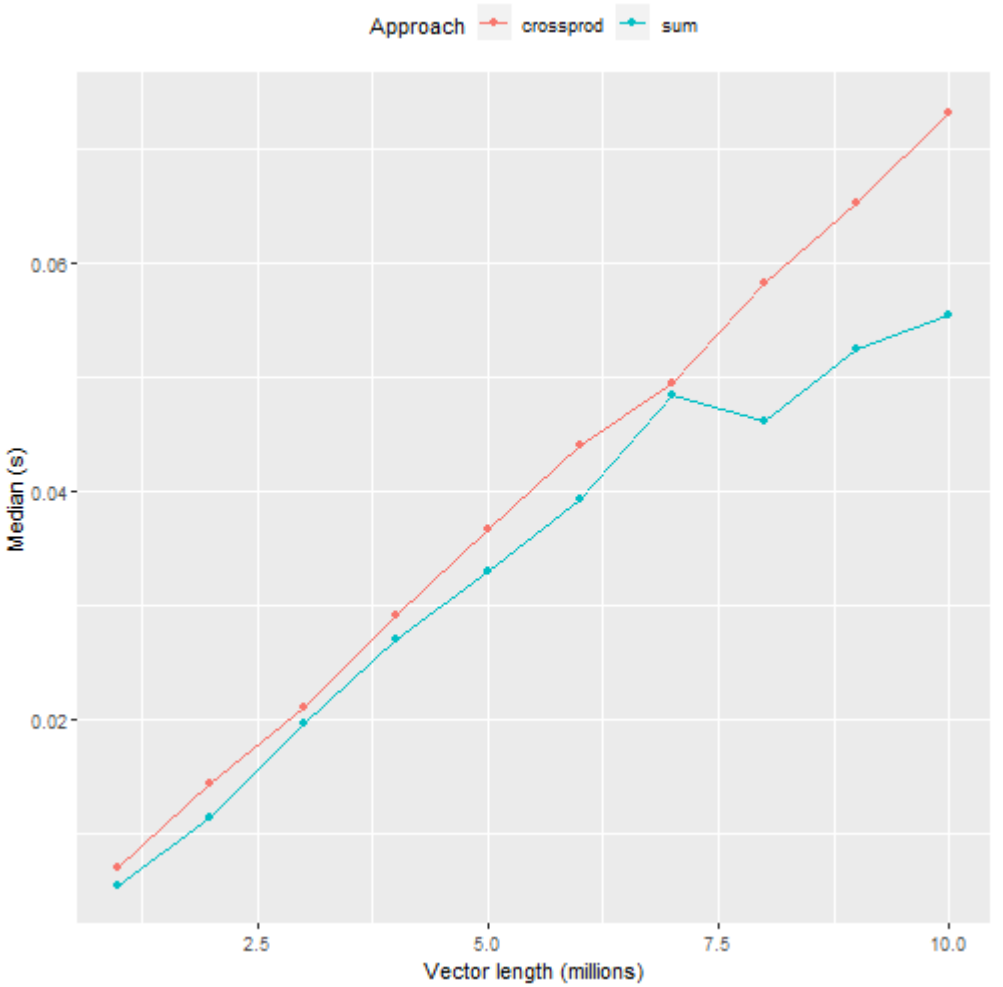
How can you use crossprod() to compute a weighted sum? How much faster is it than the naive sum(x * w)?

We can hand the vectors to crossprod(), which converts them to row- and column-vectors and then multiplies these. The result is the dot product, which corresponds to a weighted sum.

```
x <- rnorm(10)
w <- rnorm(10)
all.equal(sum(x * w), crossprod(x, w)[[1]])
```

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

```
weightedsum <- bench::press(
  n = 1:10,
  {
    x <- rnorm(n * 1e6)
    bench::mark(
      sum = sum(x * x),
      crossprod = crossprod(x, x)[[1]]
    )
  }
)
```


Approach
crossprod
sum
0.06
0.04
0.02
Median (s)
2.5
5.0
7.5
10.0
Vector length (millions)

Avoiding copies

Growing objects with loops is slow. Avoid.

Case study

Let's look through in the book.