Chapters 23-24: Optimization

Ryan Heslin

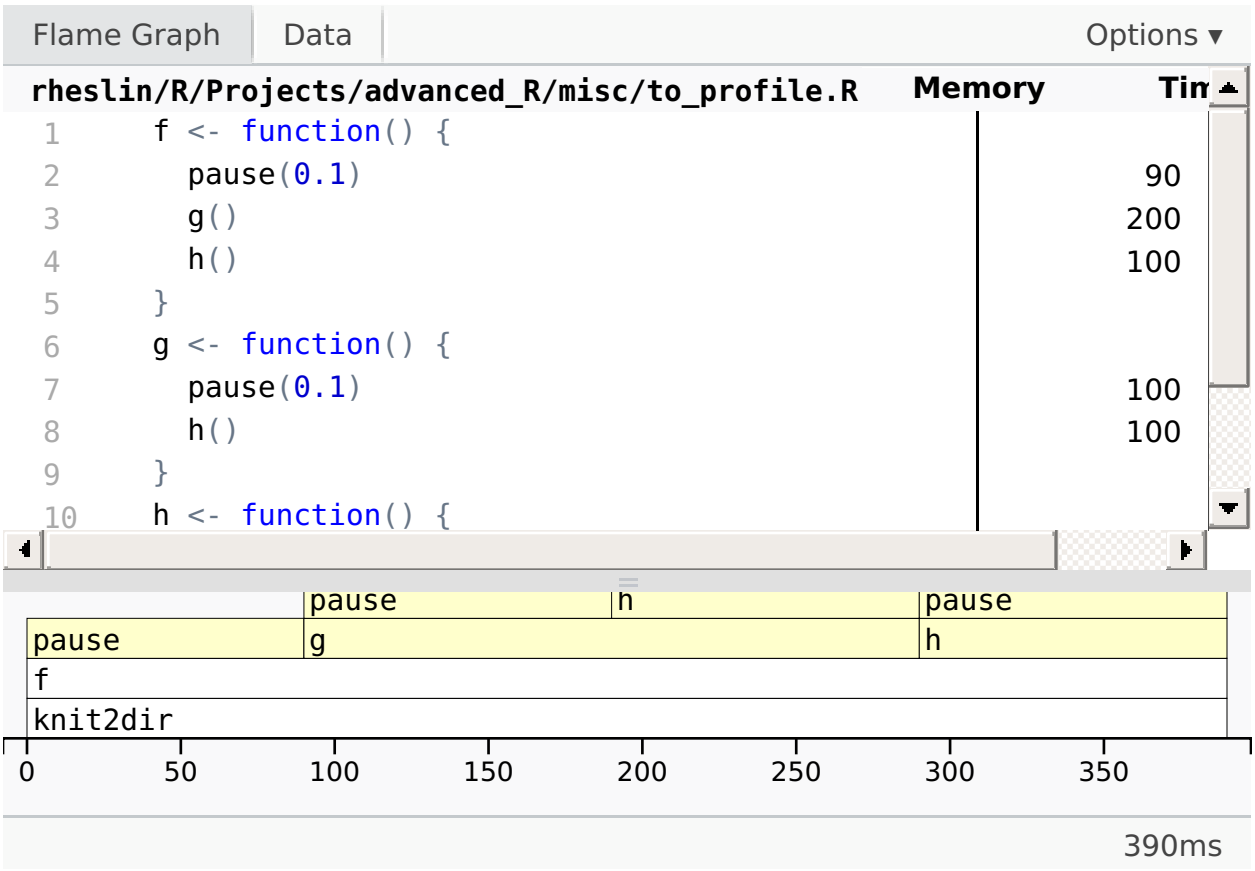
May 30, 2022

Profilers

Profilers run code repeatedly, pausing every few seconds to record the active call stack. This is imprecise, so results are stochastic.

```
library(bench)
library(profvis)

source(here::here("misc/to_profile.R"))
profvis(f())
```



1.

Lazy evaluation often baffles the profiler.

`torture` invokes the garbage collector for almost all memory allocations (making it torturous for the poor beast). Naturally, it makes this code run veeeeery slowly.

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
profvis(f(), torture = TRUE)
```

Microbenchmarks

Microbenchmarks measure the performance of small code snippets. They can help pinpoint bottlenecks.

`rm` calls `vapply` and `c`, and torture slowed them down drastically.

```
subscript <- c("expression", "min", "median", "itr/sec", "n_gc")
```

1.

I expect the first to be slower. `bench::mark` takes longer because it runs the snippet repeatedly to obtain an accurate summary, while `system.time` just runs once.

```
n <- 1e6
x <- runif(100)

system.time(for (i in 1:n) sqrt(x)) / n
system.time(for (i in 1:n) x^0.5) / n

bench::mark(
  for (i in 1:n) sqrt(x),
  for (i in 1:n) x^0.5,
  iterations = 1
)[subscript]
```

2.

I expect the second to be slower. I'm wrong.

```
bench::mark(
  x^(1 / 2),
  exp(log(x) / 2)
)
```

```
# A tibble: 2 x 6
  expression        min median ‘itr/sec‘ mem_alloc
  <bch:expr>    <bch:t> <bch:>     <dbl> <bch:byt>
1 x^(1/2)        2.44us  2.5us   313738.      848B
2 exp(log(x)/2)   1.6us 1.67us   491635.      848B
# ... with 1 more variable: ‘gc/sec‘ <dbl>
```

Optimization

The worst pitfalls are writing fast but incorrect code, and writing code you only *think* is faster.

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
n <- runif(1e5)
```

Those aren't the same as Hadley's results.

```
bench::mark(
  mean1(x),
  mean2(x)
)[subscript]
```

```
# A tibble: 2 x 4
  expression      min   median ‘itr/sec‘
  <bch:expr> <bch:tm> <bch:tm>     <dbl>
1 mean1(x)     2.56us   3.51us   253334.
2 mean2(x)   521.08ns 564.96ns  1230446.
```

Checking for Existing Solutions

1.

Use `lm.fit` instead of `lm`, or one of the implementations of `fastLm` in several Rcpp packages. Or just do matrix multiplication!

2.

`fmatch`. It uses hashing, so initial lookups aren't much faster, but subsequent ones take constant time.

3.

`as.Date` helpfully offers a format specification and the option to set the epoch. `lubridate` has high-level specialized functions (e.g., `mdy`). `strptime` is simple but usable. `lubridate`'s `parse_date_time` offers a cleaner interface to format specification (no nasty `%` escapes).

4.

`zoo` and `data.table` come to mind.

5.

In base, one can try `nlm` or use `optim` with a faster method. The CRAN task view reveals the `optimx` package and many packages implementing solvers. See [https://stackoverflow.com/questions/3757321/moving-beyond-rs-optim-function].

Doing Less

Rewriting a function to use only a particular kind of input is dangerous but potentially effective.

1.

The dotted versions are "bare-bones" implementations that expect numeric matrices and do not name outputs.

3.

```
library(testthat)
table2 <- function(x, y) {
  x2 <- factor(x, levels = seq(from = min(x), to = max(x)))
  y2 <- factor(y, levels = seq(from = min(y), to = max(y)))
  dims <- c(nlevels(x2), nlevels(y2))
  bin <- (as.integer(x2) - 1 + dims[[2]] * (as.integer(y2) - 1L)) + 1
  pd <- dims[[1]] * dims[[2]]
  out <- array(tabulate(bin, pd), dims, dimnames = list(levels(x2), levels(y2)))
  class(out) <- "table"
  out
}
set.seed(1)
x <- sample(30, 100, replace = TRUE)
y <- sample(30, 100, replace = TRUE)
expect_equivalent(table2(x, y), table(x, y))
```

2.

This doesn't work for cases where the expected count is 0, but in that case the exact chi-square test is undefined anyway.

```
set.seed(1)
chisq2 <- function(x, y) {
  O <- table2(x, y)
  E <- outer(rowSums(O), colSums(O)) / (sum(O))
  print(E)
  sum((O - E)^2 / E)
}
x <- sample(30, 100, replace = TRUE)
y <- sample(30, 100, replace = TRUE)
chisq2(x, y)
```

```
      1    2    3    4    5    6    7    8    9
1  0.20 0.16 0.16 0.04 0.04 0.08 0.24 0.04 0.16
2  0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
3  0.05 0.04 0.04 0.01 0.01 0.02 0.06 0.01 0.04
4  0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
5  0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
6  0.40 0.32 0.32 0.08 0.08 0.16 0.48 0.08 0.32
7  0.25 0.20 0.20 0.05 0.05 0.10 0.30 0.05 0.20
8  0.20 0.16 0.16 0.04 0.04 0.08 0.24 0.04 0.16
9  0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
10 0.30 0.24 0.24 0.06 0.06 0.12 0.36 0.06 0.24
11 0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
12 0.25 0.20 0.20 0.05 0.05 0.10 0.30 0.05 0.20
13 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
14 0.30 0.24 0.24 0.06 0.06 0.12 0.36 0.06 0.24
15 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
16 0.05 0.04 0.04 0.01 0.01 0.02 0.06 0.01 0.04
17 0.05 0.04 0.04 0.01 0.01 0.02 0.06 0.01 0.04
18 0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
19 0.20 0.16 0.16 0.04 0.04 0.08 0.24 0.04 0.16
20 0.20 0.16 0.16 0.04 0.04 0.08 0.24 0.04 0.16
21 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
22 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
23 0.25 0.20 0.20 0.05 0.05 0.10 0.30 0.05 0.20
24 0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
25 0.40 0.32 0.32 0.08 0.08 0.16 0.48 0.08 0.32
26 0.10 0.08 0.08 0.02 0.02 0.04 0.12 0.02 0.08
27 0.05 0.04 0.04 0.01 0.01 0.02 0.06 0.01 0.04
28 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
29 0.15 0.12 0.12 0.03 0.03 0.06 0.18 0.03 0.12
30 0.05 0.04 0.04 0.01 0.01 0.02 0.06 0.01 0.04
     10   11   12   13   14   15   16   17   18
1  0.12 0.12 0.12 0.08 0.12 0.12 0.20 0.16 0.12
2  0.09 0.09 0.09 0.06 0.09 0.09 0.15 0.12 0.09
3  0.03 0.03 0.03 0.02 0.03 0.03 0.05 0.04 0.03
```

4	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
5	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
6	0.24	0.24	0.24	0.16	0.24	0.24	0.40	0.32	0.24
7	0.15	0.15	0.15	0.10	0.15	0.15	0.25	0.20	0.15
8	0.12	0.12	0.12	0.08	0.12	0.12	0.20	0.16	0.12
9	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
10	0.18	0.18	0.18	0.12	0.18	0.18	0.30	0.24	0.18
11	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
12	0.15	0.15	0.15	0.10	0.15	0.15	0.25	0.20	0.15
13	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
14	0.18	0.18	0.18	0.12	0.18	0.18	0.30	0.24	0.18
15	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
16	0.03	0.03	0.03	0.02	0.03	0.03	0.05	0.04	0.03
17	0.03	0.03	0.03	0.02	0.03	0.03	0.05	0.04	0.03
18	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
19	0.12	0.12	0.12	0.08	0.12	0.12	0.20	0.16	0.12
20	0.12	0.12	0.12	0.08	0.12	0.12	0.20	0.16	0.12
21	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
22	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
23	0.15	0.15	0.15	0.10	0.15	0.15	0.25	0.20	0.15
24	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
25	0.24	0.24	0.24	0.16	0.24	0.24	0.40	0.32	0.24
26	0.06	0.06	0.06	0.04	0.06	0.06	0.10	0.08	0.06
27	0.03	0.03	0.03	0.02	0.03	0.03	0.05	0.04	0.03
28	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
29	0.09	0.09	0.09	0.06	0.09	0.09	0.15	0.12	0.09
30	0.03	0.03	0.03	0.02	0.03	0.03	0.05	0.04	0.03

	19	20	21	22	23	24	25	26	27
1	0.28	0.08	0.16	0.20	0.08	0.16	0.04	0.24	0.08
2	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
3	0.07	0.02	0.04	0.05	0.02	0.04	0.01	0.06	0.02
4	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04
5	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
6	0.56	0.16	0.32	0.40	0.16	0.32	0.08	0.48	0.16
7	0.35	0.10	0.20	0.25	0.10	0.20	0.05	0.30	0.10
8	0.28	0.08	0.16	0.20	0.08	0.16	0.04	0.24	0.08
9	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04
10	0.42	0.12	0.24	0.30	0.12	0.24	0.06	0.36	0.12
11	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04
12	0.35	0.10	0.20	0.25	0.10	0.20	0.05	0.30	0.10
13	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
14	0.42	0.12	0.24	0.30	0.12	0.24	0.06	0.36	0.12
15	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
16	0.07	0.02	0.04	0.05	0.02	0.04	0.01	0.06	0.02
17	0.07	0.02	0.04	0.05	0.02	0.04	0.01	0.06	0.02
18	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04
19	0.28	0.08	0.16	0.20	0.08	0.16	0.04	0.24	0.08
20	0.28	0.08	0.16	0.20	0.08	0.16	0.04	0.24	0.08
21	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
22	0.21	0.06	0.12	0.15	0.06	0.12	0.03	0.18	0.06
23	0.35	0.10	0.20	0.25	0.10	0.20	0.05	0.30	0.10
24	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04
25	0.56	0.16	0.32	0.40	0.16	0.32	0.08	0.48	0.16
26	0.14	0.04	0.08	0.10	0.04	0.08	0.02	0.12	0.04

```
27 0.07 0.02 0.04 0.05 0.02 0.04 0.01 0.06 0.02
28 0.21 0.06 0.12 0.15 0.06 0.12 0.03 0.18 0.06
29 0.21 0.06 0.12 0.15 0.06 0.12 0.03 0.18 0.06
30 0.07 0.02 0.04 0.05 0.02 0.04 0.01 0.06 0.02
     28   29   30
1  0.20 0.16 0.04
2  0.15 0.12 0.03
3  0.05 0.04 0.01
4  0.10 0.08 0.02
5  0.15 0.12 0.03
6  0.40 0.32 0.08
7  0.25 0.20 0.05
8  0.20 0.16 0.04
9  0.10 0.08 0.02
10 0.30 0.24 0.06
11 0.10 0.08 0.02
12 0.25 0.20 0.05
13 0.15 0.12 0.03
14 0.30 0.24 0.06
15 0.15 0.12 0.03
16 0.05 0.04 0.01
17 0.05 0.04 0.01
18 0.10 0.08 0.02
19 0.20 0.16 0.04
20 0.20 0.16 0.04
21 0.15 0.12 0.03
22 0.15 0.12 0.03
23 0.25 0.20 0.05
24 0.10 0.08 0.02
25 0.40 0.32 0.08
26 0.10 0.08 0.02
27 0.05 0.04 0.01
28 0.15 0.12 0.03
29 0.15 0.12 0.03
30 0.05 0.04 0.01


[1] 793.0516
```

```
chisq.test(table(x, y))
```

```
    Pearson's Chi-squared test

data:  table(x, y)
X-squared = 793.05, df = 841, p-value =
0.8802
```

Vectorization

This usually just means delegating to (or writing) the appropriate vectorized C function.

1.

It creates a vector of 10 random normal variables, with the means given by `10:1` The `mean` and `sd` arguments are vectorized, so a vector of random variables with different parameters can be created in a single call.

```
rnorm(10, mean = 10:1)
```

```
 [1] 8.869614 9.576719 6.719251 8.625447 5.499303
 [6] 6.678297 3.587480 2.027713 2.025383 1.027475
```

2.

The performance penalty of using `apply` grows nonlinearly with input size.

```
sizes <- c(100, 1000, 10000)
data <- as.data.frame(matrix(rnorm(60000), ncol = 6))
(lapply(sizes, \(x) bench::mark(rowSums(data[seq_len(x), ]), apply(data[seq_len(x), ],
  MARGIN = 1, sum
))) |>
  do.call(what = rbind))[subscript]
```

```
# A tibble: 6 x 4
  expression
  <bch:expr>
1 rowSums(data[seq_len(x), ])
2 apply(data[seq_len(x), ], MARGIN = 1, sum)
3 rowSums(data[seq_len(x), ])
4 apply(data[seq_len(x), ], MARGIN = 1, sum)
5 rowSums(data[seq_len(x), ])
6 apply(data[seq_len(x), ], MARGIN = 1, sum)
# ... with 3 more variables: min <bch:tm>,
#   median <bch:tm>, `itr/sec` <dbl>
```

3.

Actually, it's slower.

```r
x <- rnorm(10e6)
weights <- sample(10, 10e6, replace = TRUE)
bench::mark(
  sum(x * weights),
  c(crossprod(x, weights))
)[subscript]
```

```
# A tibble: 2 x 4
  expression                 min median `itr/sec`
  <bch:expr>               <bch:> <bch:>     <dbl>
1 sum(x * weights)         41.6ms 41.9ms      23.9
2 c(crossprod(x, weights)) 48.1ms 49.8ms      20.2
```

A last piece of advice is to avoid for loops that create unnecessary copies of objects.