Profiling

01-28-2020

Admin stuff

- Assignment 2 is due today
- Introduce `styler`
- About final project

```r
# we will need these packages
library(profvis)
library(bench)
library(tidyverse)
```

Optimization

Donald Knuth has famously said

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

Before you can make your code faster, you first need to figure out what's making it slow.

We consider this example of simple linear regression,

```r
x <- runif(1e7)
y <- runif(1e7)
coef(lm(y ~ x)) #  it is quite slow
```

```
## (Intercept)           x 
## 0.499860973 0.000101807
```

```r
system.time(coef(lm(y ~ x)))
```

```
##    user  system elapsed 
##   1.389   0.338   1.792
```

`Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function

```r
Rprof()
coef(lm(y ~ x))
```

```
## (Intercept)           x
## 0.499860973 0.000101807
```

```r
Rprof(NULL)
result <- summaryRprof()
result$by.self
```

```
##                         self.time self.pct total.time total.pct
## "lm.fit"                     0.82    46.59       0.86     48.86
## "anyDuplicated.default"      0.42    23.86       0.42     23.86
## "[.data.frame"               0.28    15.91       0.76     43.18
## "na.omit.data.frame"         0.08     4.55       0.86     48.86
## "!"                          0.06     3.41       0.06      3.41
## ".External2"                 0.04     2.27       0.90     51.14
## "c"                          0.04     2.27       0.04      2.27
## "any"                        0.02     1.14       0.02      1.14
```

```r
result$by.total
```

```
##                       total.time total.pct self.time self.pct
## "block_exec"                1.76    100.00      0.00     0.00
## "call_block"                1.76    100.00      0.00     0.00
## "coef"                      1.76    100.00      0.00     0.00
## "eval"                      1.76    100.00      0.00     0.00
## "evaluate_call"             1.76    100.00      0.00     0.00
## "evaluate::evaluate"        1.76    100.00      0.00     0.00
## "evaluate"                  1.76    100.00      0.00     0.00
## "handle"                    1.76    100.00      0.00     0.00
## "in_dir"                    1.76    100.00      0.00     0.00
## "knitr::knit"               1.76    100.00      0.00     0.00
## "lm"                        1.76    100.00      0.00     0.00
## "process_file"              1.76    100.00      0.00     0.00
## "process_group.block"       1.76    100.00      0.00     0.00
## "process_group"             1.76    100.00      0.00     0.00
## "rmarkdown::render"         1.76    100.00      0.00     0.00
## "timing_fn"                 1.76    100.00      0.00     0.00
## "withCallingHandlers"       1.76    100.00      0.00     0.00
## "withVisible"               1.76    100.00      0.00     0.00
## ".External2"                0.90     51.14      0.04     2.27
## "lm.fit"                    0.86     48.86      0.82    46.59
## "na.omit.data.frame"        0.86     48.86      0.08     4.55
## "model.frame.default"       0.86     48.86      0.00     0.00
## "na.omit"                   0.86     48.86      0.00     0.00
## "stats::model.frame"        0.86     48.86      0.00     0.00
## "[.data.frame"              0.76     43.18      0.28    15.91
##  [ reached 'max' / getOption("max.print") -- omitted 8 rows ]
```

`help(lm.fit)` gives us > These are the basic computing engines called by lm used to fit linear models. These should usually not be used directly unless by experienced users.

```r
system.time(lm.fit(cbind(1, x), y)$coefficients)
```

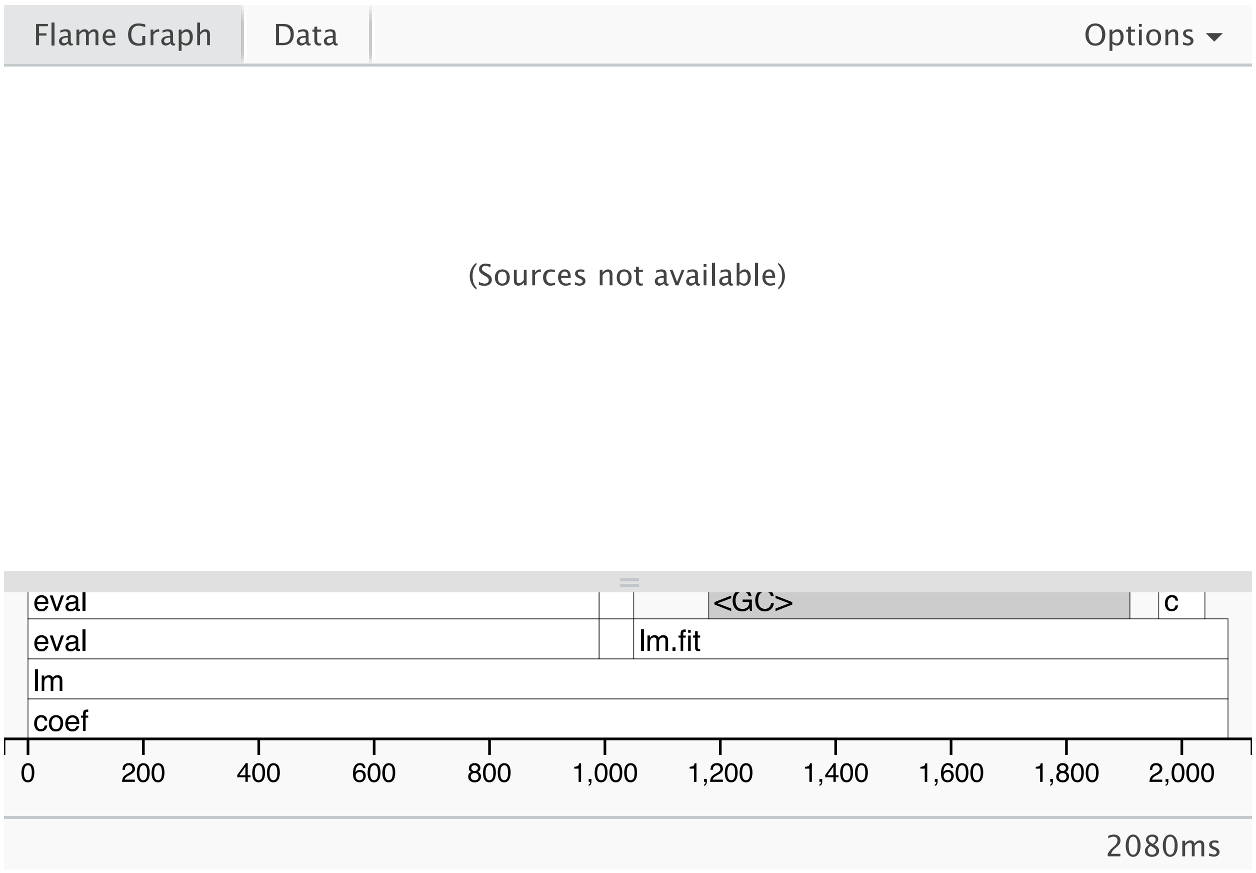
```
##    user  system elapsed
##   0.632   0.191   0.889
```

Visualising profiles

There are two ways to use profvis:

- From the Profile menu in RStudio.
- Use the `profvis` function

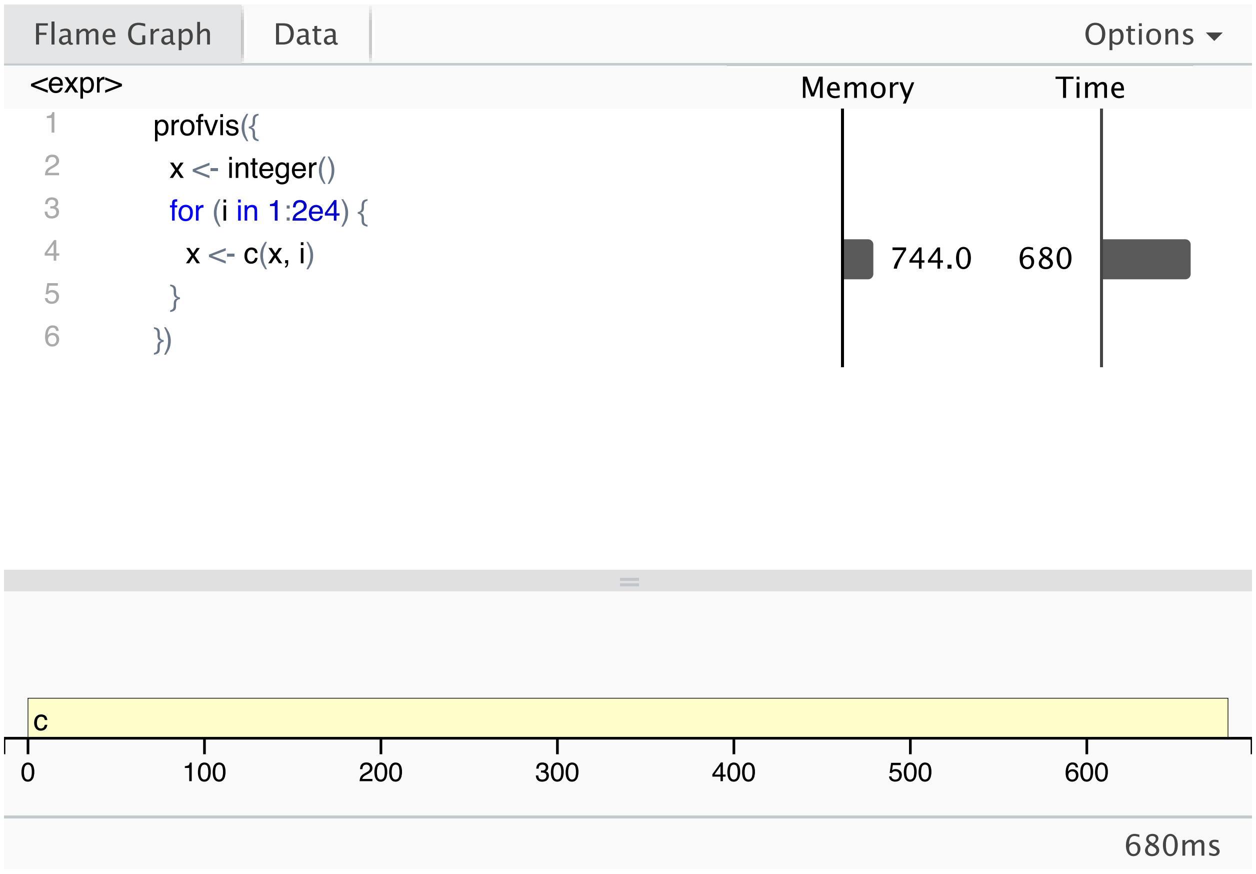
```r
profvis(coef(lm(y ~ x)))
```



Memory profiling

When an object in R is not referenced by any other objects, it will get GC'ed (garbage collected). If is taking a lot of time, it's usually an indication that you're creating many short-lived objects.

```
profvis({
  x <- integer()
  for (i in 1:2e4) {
    x <- c(x, i)
  }
})
```



```
profvis({
  x <- integer()
  for (i in 1:2e4) {
    x <- c(x, i)
  }
})
```

Each time when `x <- c(x, i)` is execulated, the previous `x` is de-referenced and marked as pending to be GC'ed. R will GC those `x`'s at some point down the line.

Microbenchmark

A microbenchmark is a measurement of the performance of a very small piece of code, something that might take milliseconds (ms), microseconds (µs), or nanoseconds (ns) to run.

The following code compares the speed of two approaches to computing a square root.

```
x <- runif(100)
bench::mark(
  sqrt(x),
  x^0.5
)
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sqrt(x)       373ns    444ns   898189.      848B        0
## 2 x^0.5        2.51us    2.7us   213618.      848B        0
```

```r
bench::mark(
  sqrt(x),
  x^0.5,
  relative = TRUE
)
```

```
## # A tibble: 2 x 6
##   expression   min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
## 1 sqrt(x)     1      1          2.23         1      NaN
## 2 x^0.5       6.38   5.86       1            1      NaN
```

`sqrt(x)` is is about 5x faster than `x ^ 0.5`

The simple linear regression example

```r
slr <- function(x, y) {
  # it is meant to be inefficient
  sxy <- sum((x - mean(x)) * (y - mean(y)))
  sxx <- sum((x - mean(x))^2)
  slope <- sxy / sxx
  intercept <- mean(y) - slope * mean(x)
  c(intercept, slope)
}

slr2 <- function(x, y) {
  mux <- mean(x)
  muy <- mean(y)
  sxy <- sum((x - mux) * (y - muy))
  sxx <- sum((x - mux)^2)
  slope <- sxy / sxx
  intercept <- muy - slope * mux
  c(intercept, slope)
}
```

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector slr_cpp(NumericVector x, NumericVector y) {
  double mux = mean(x);
  double muy = mean(y);
  double sxy = sum((x - mux)*(y - muy));
  double sxx = sum(pow(x - mux, 2));
  double slope = sxy / sxx;
  double intercept = muy - slope * mux;
  return NumericVector::create(intercept, slope);
}
```

Remark: No worries, we will introduce a quick course in c++

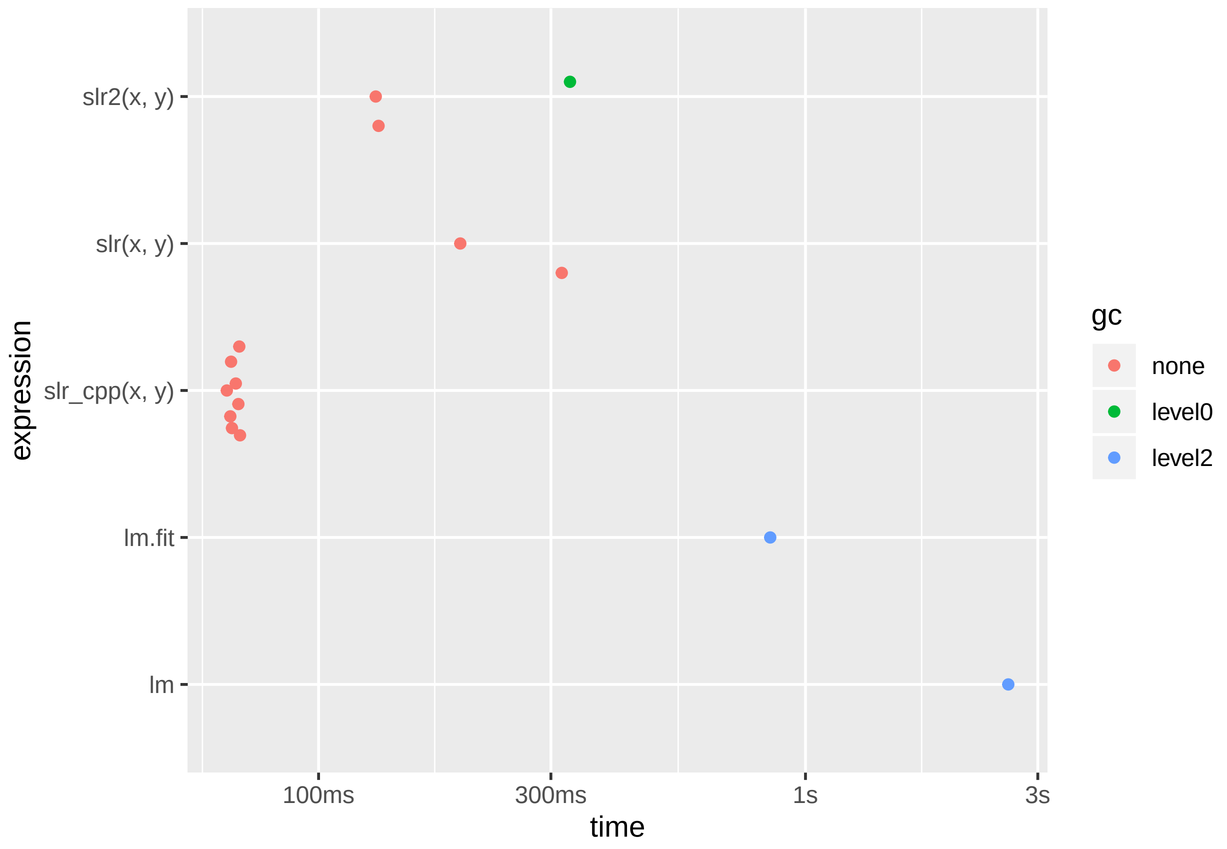
```r
x <- runif(1e7)
y <- runif(1e7)

(result <- bench::mark(
  slr(x, y),
  slr2(x, y),
  slr_cpp(x, y),
  lm = as.double(coef(lm(y ~ x))),
  lm.fit = as.double(lm.fit(cbind(1, x), y)$coefficients)
))
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is disabled.

## # A tibble: 5 x 6
##   expression         min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>    <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 slr(x, y)     195.46ms 255.64ms     3.91    228.88MB     0
## 2 slr2(x, y)    131.05ms  132.8ms     5.07    228.88MB     1.69
## 3 slr_cpp(x, y)   64.8ms  67.01ms    14.9       2.49KB     0
## 4 lm               2.61s    2.61s     0.383     1.61GB     1.15
## 5 lm.fit         846.5ms  846.5ms     1.18    686.65MB     2.36
```

```r
autoplot(result)
```



There are three levels of collections. - level 0 collects only the youngest generation - level 1 collects the two youngest generations - level 2 collects all generations.

After 20 level-0 collections the next collection is at level 1, and after 5 level-1 collections at level 2.

Reference

- Advanced R https://adv-r.hadley.nz/perf-measure.html
- R Programming for Data Science https://bookdown.org/rdpeng/rprogdatascience/profiling-r-code.html