19 Profiling R Code

> Let's solve the problem but let's not make it worse by guessing. —*Gene Kranz, Apollo 13 Lead Flight Director*

Watch a video of this section

R comes with a profiler to help you optimize your code and improve its performance. In generall, it's usually a bad idea to focus on optimizing your code at the very beginning of development. Rather, in the beginning it's better to focus on translating your ideas into code and writing code that's coherent and readable. The problem is that heavily optimized code tends to be obscure and difficult to read, making it harder to debug and revise. Better to get all the bugs out first, then focus on optimizing.

Of course, when it comes to optimizing code, the question is what should you optimize? Well, clearly should optimize the parts of your code that are running slowly, but how do we know what parts those are?

This is what the profiler is for. Profiling is a systematic way to examine how much time is spent in different parts of a program.

Sometimes profiling becomes necessary as a project grows and layers of code are placed on top of each other. Often you might write some code that runs fine once. But then later, you might put that same code in a big loop that runs 1,000 times. Now the original code that took 1 second to run is taking 1,000 seconds to run! Getting that little piece of original code to run faster will help the entire loop.

It's tempting to think you just *know* where the bottlenecks in your code are. I mean, after all, you write it! But trust me, I can't tell you how many times I've been surprised at where exactly my code is spending all its time. The reality is that *profiling is better than guessing*. Better to collect some data than to go on hunches alone. Ultimately, getting the biggest impact on speeding up code depends on knowing where the code spends most of its time. This cannot be done without some sort of rigorous performance analysis or profiling.

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil —*Donald Knuth*

The basic principles of optimizing your code are:

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don't guess.
- If you're going to be scientist, you need to apply the same principles here!

19.1 Using `system.time()`

They `system.time()` function takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression. The `system.time()` function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred. The function returns an object of class `proc_time` which contains two useful bits of information: But Sys.time() outputs the current time.

- *user time*: time charged to the CPU(s) for this expression
- *elapsed time*: "wall clock" time, the amount of time that passes for *you* as you're sitting there

Usually, the user time and elapsed time are relatively close, for straight computing tasks. But there are a few situations where the two can diverge, sometimes dramatically. The elapsed time may be *greater than* the user time if the CPU spends a lot of time waiting around. This commonly happens if your R expression involes some input or output, which depends on the activity of the file system and the disk (or the Internet, if using a network connection).

The elapsed time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them). For example, multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL) can greatly speed up linear algebra calculations and are commonly installed on even desktop systems these days. Also, parallel processing done via something like the `parallell` package can make the elapsed time smaller than the user time. When you have multiple processors/cores/machines working in parallel, the amount of time that the collection of CPUs spends working on a problem is the same as with a single CPU, but because they are operating in parallel, there is a savings in elapsed time.

Here's an example of where the elapsed time is greater than the user time.

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed
  0.004   0.002   0.431
```

Most of the time in this expression is spent waiting for the connection to the web server and waiting for the data to travel back to my computer. This doesn't involve the CPU and so the CPU simply waits around for things to get done. Hence, the user time is small.

In this example, the elapsed time is smaller than the user time.

```
## Elapsed time < user time
> hilbert <- function(n) {
+         i <- 1:n
+         1 / outer(i - 1, i, "+")
+ }
> x <- hilbert(1000)
> system.time(svd(x))
   user  system elapsed
  1.035   0.255   0.462
```

In this case I ran a singular value decomposition on the matrix in `x`, which is a common linear algebra procedure. Because my computer is able to split the work across multiple processors, the elapsed time is about half the user time.

19.2 Timing Longer Expressions

You can time longer expressions by wrapping them in curly braces within the call to `system.time()`.

```
> system.time({
+         n <- 1000
+         r <- numeric(n)
+         for(i in 1:n) {
+                 x <- rnorm(n)
+                 r[i] <- mean(x)
+         }
+ })
   user  system elapsed
  0.081   0.004   0.085
```

If your expression is getting pretty long (more than 2 or 3 lines), it might be better to either break it into smaller pieces or to use the profiler. The problem is that if the expression is too long, you won't be able to identify which part of the code is causing the bottleneck.

19.3 The R Profiler

Watch a video of this section

Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time. However, this approach assumes that you already know where the problem is and can call `system.time()` on it that piece of code. What if you don't know where to start?

This is where the profiler comes in handy. The `Rprof()` function starts the profiler in R. Note that R must be compiled with profiler support (but this is usually the case). In conjunction with `Rprof()`, we will use the `summaryRprof()` function which summarizes the output from `Rprof()` (otherwise it's not really readable). Note that you should NOT use `system.time()` and `Rprof()` together, or you will be sad.

`Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function. By default, the profiler samples the function call stack every 0.02 seconds. This means that if your code runs very quickly (say, under 0.02 seconds), the profiler is not useful. But of your code runs that fast, you probably don't need the profiler.

The profiler is started by calling the `Rprof()` function.

```
> Rprof()    ## Turn on the profiler
```

You don't need any other arguments. By default it will write its output to a file called `Rprof.out`. You can specify the name of the output file if you don't want to use this default.

Once you call the `Rprof()` function, everything that you do from then on will be measured by the profiler. Therefore, you usually only want to run a single R function or expression once you turn on the profiler and then immediately turn it off. The reason is that if you mix too many function calls together when running the profiler, all of the results will be mixed together and you won't be able to sort out where the bottlenecks are. In reality, I usually only run a single function with the profiler on.

The profiler can be turned off by passing `NULL` to `Rprof()`.

```
> Rprof(NULL)    ## Turn off the profiler
```

The raw output from the profiler looks something like this. Here I'm calling the `lm()` function on some data with the profiler running.

```
## lm(y ~ x)

sample.interval=10000
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
```

At each line of the output, the profiler writes out the function call stack. For example, on the very first line of the output you can see that the code is 8 levels deep in the call stack. This is where you need the `summaryRprof()` function to help you interpret this data.

19.4 Using `summaryRprof()`

The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spent in which function. There are two methods for normalizing the data.

- “by.total” divides the time spend in each function by the total run time
- “by.self” does the same as “by.total” but first subtracts out time spent in functions above the current function in the call stack. I personally find this output to be much more useful.

Here is what `summaryRprof()` reports in the “by.total” output.

```
$by.total
                        total.time total.pct self.time self.pct
"lm"                          7.41    100.00      0.30     4.05
"lm.fit"                      3.50     47.23      2.99    40.35
"model.frame.default"         2.24     30.23      0.12     1.62
"eval"                        2.24     30.23      0.00     0.00
"model.frame"                 2.24     30.23      0.00     0.00
"na.omit"                     1.54     20.78      0.24     3.24
"na.omit.data.frame"          1.30     17.54      0.49     6.61
"lapply"                      1.04     14.04      0.00     0.00
"[.data.frame"                1.03     13.90      0.79    10.66
"["                           1.03     13.90      0.00     0.00
"as.list.data.frame"          0.82     11.07      0.82    11.07
"as.list"                     0.82     11.07      0.00     0.00
```

Because `lm()` is the function that I called from the command line, of course 100% of the time is spent somewhere in that function. However, what this doesn't show is that if `lm()` immediately calls another function (like `lm.fit()`, which does most of the heavy lifting), then in reality, most of the time is spent in *that* function, rather than in the top-level `lm()` function.

The "by.self" output corrects for this discrepancy.

```
$by.self
                        self.time self.pct total.time total.pct
"lm.fit"                     2.99    40.35       3.50     47.23
"as.list.data.frame"         0.82    11.07       0.82     11.07
"[.data.frame"               0.79    10.66       1.03     13.90
"structure"                  0.73     9.85       0.73      9.85
"na.omit.data.frame"         0.49     6.61       1.30     17.54
"list"                       0.46     6.21       0.46      6.21
"lm"                         0.30     4.05       7.41    100.00
"model.matrix.default"       0.27     3.64       0.79     10.66
"na.omit"                    0.24     3.24       1.54     20.78
"as.character"               0.18     2.43       0.18      2.43
"model.frame.default"        0.12     1.62       2.24     30.23
"anyDuplicated.default"      0.02     0.27       0.02      0.27
```

Now you can see that only about 4% of the runtime is spent in the actual `lm()` function, whereas over 40% of the time is spent in `lm.fit()`. In this case, this is no surprise since the `lm.fit()` function is the function that actually fits the linear model.

You can see that a reasonable amount of time is spent in functions not necessarily associated with linear modeling (i.e. `as.list.data.frame`, `[.data.frame`). This is because the `lm()` function does a bit of pre-processing and checking before it actually fits the model. This is common with modeling functions—the preprocessing and checking is useful to see if there are any errors. But those two functions take up over 1.5 seconds of runtime. What if you want to fit this model 10,000 times? You're going to be spending a lot of time in preprocessing and checking.

The final bit of output that `summaryRprof()` provides is the sampling interval and the total runtime.

```
$sample.interval
[1] 0.02

$sampling.time
[1] 7.41
```

19.5 Summary

- `Rprof()` runs the profiler for performance of analysis of R code
- `summaryRprof()` summarizes the output of `Rprof()` and gives percent of time spent in each function (with two types of normalization)
- Good to break your code into functions so that the profiler can give useful information about where time is being spent
- C or Fortran code is not profiled