# Faster Computing - Part 1

## Contents



## Overview

Sometimes it's nice to have code that takes a long time to run; it feels like you're working even when you're not! But most of the time it's really useful when you can speed up code so that it takes seconds instead of minutes, or minutes instead of hours, or hours instead of days. Most of the time faster computing can be achieved by creativity. It's therefore wise to take the time to think of how you might achieve the same goal in a different manner, and to search online (e.g., https://stackexchange.com) for similar problems to see if others have solutions or inspirational ideas. If what you're trying to achieve requires a large number of steps, search CRAN (https://cran.r-project.org) to see if someone has created a package with useful functions. These have likely been optimized for efficiency (e.g., by performing computations in `C++` that are "wrapped" in `R` code). That said, there are many other ways to achieve faster code. In this and the next class, we'll step through a few options, moving from the simplest forms of code efficiency to the use of high performance clusters (HPCs). Note that there are a number of additional ways for computing faster that we will not discuss (e.g., using `Rcpp` to compile your own `C++` code).

## Benchmarking

The first useful tool for optimizing code is a way to quantitatively measure and compare code performance. That's what benchmarking refers to. There's a few ways to do it, but here are examples of two simple methods using functions that are built in to base-`R`, and a third example using the `microbenchmark` package.

For demonstration purposes, let's consider the task of performing a stochastic simulation of a population that is, on average, growing geometrically. That is, given a mean growth rate $\lambda = 1.01$ whose year-to-year

variation is described by a normal distribution with standard deviation $\sigma^2 = 0.2$, we wish to simulate

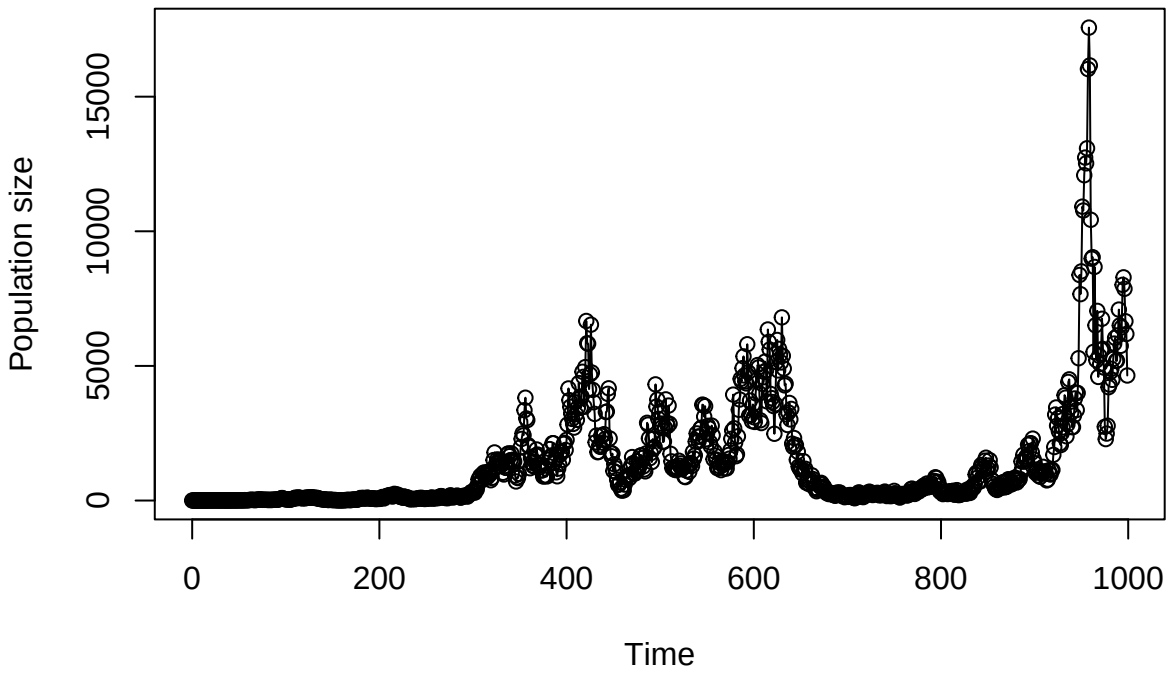
$$N_{t+1} = N_t(\lambda e^{\epsilon}) \tag{1}$$

$$\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{2}$$

for a total of $T = 999$ time-steps starting from an initial size population size $N_0 = 2$. We'll use a `for`-loop to do it and, for convenience, will wrap it into a function with the desired parameter values as defaults.

```
set.seed(1) # for reproducibility
geom_growth_base <- function(N0 = 2,
                             T = 999,
                             lambda = 1.01,
                             sigma = 0.2){
  Nvals <- vector('numeric') # initiate a place to put the values
  Nvals[1] <- N0
  for (t in 1:T){
    Nvals[t+1] <- Nvals[t]*(lambda*exp(rnorm(1,0,sigma)))
  }
  return(Nvals)
}

# Run the simulation
out <- geom_growth_base()
# Plot the results
plot(0:999,
     out,
     xlab='Time',
     ylab='Population size',
     type='o')
```




Eerily similar to the early dynamics of COVID, huh?

Now let's quantify how long it takes our function to run.

### Benchmarking using `Sys.time()`

The simplest way is to ask `R` what time it is before and after running our function. We can use `Sys.time()` for that.

```
# Default number of time-points
start_time <- Sys.time()
  out <- geom_growth_base()
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.001021147 secs
```

```
# Repeat with greater number of time-points
start_time <- Sys.time()
  out <- geom_growth_base(T = 9E5)
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.494211 secs
```

```
# Note that the time won't be exactly the same each time (unless the seed is the same)
start_time <- Sys.time()
  out <- geom_growth_base(T = 9E5)
end_time <- Sys.time()

end_time - start_time
```

```
## Time difference of 0.4814329 secs
```

Using `Sys.time()` makes it easy to measure the run-time of any section of code, no matter how long it is, because you don't have to wrap the code in anything.

### Benchmarking using `system.time()`

The function `system.time()` lets you evaluate the run-time of any expression (function), and provides two additional ways of counting time.

```
system.time(geom_growth_base(T=9E5))
```

```
##    user  system elapsed
##   0.485   0.009   0.494
```

`user` gives the CPU time spent by the R session (i.e. the current process). `system` gives the CPU time spent by the kernel (the operating system) on behalf of the R session. The kernel CPU time will include time spent opening files, doing input or output, starting other processes, etc. (i.e. operations involving resources shared with other system processes). `elapsed` gives the time as we measured using `Sys.time()`.

### Using the `microbenchmark` package

There are a few benchmarking packages out there (including `tictoc` and `rbenchmark`). They're similar in that they simplify the task of comparing the speed of multiple functions. `microbenchmark` is nice because it will evaluate each function repeatedly (default `neval`=100 times) and return summary statistics. It also plays well with `ggplot` to enable quick visual comparisons.

```
library(microbenchmark)
```

```
## Warning: package 'microbenchmark' was built under R version 4.4.1
```

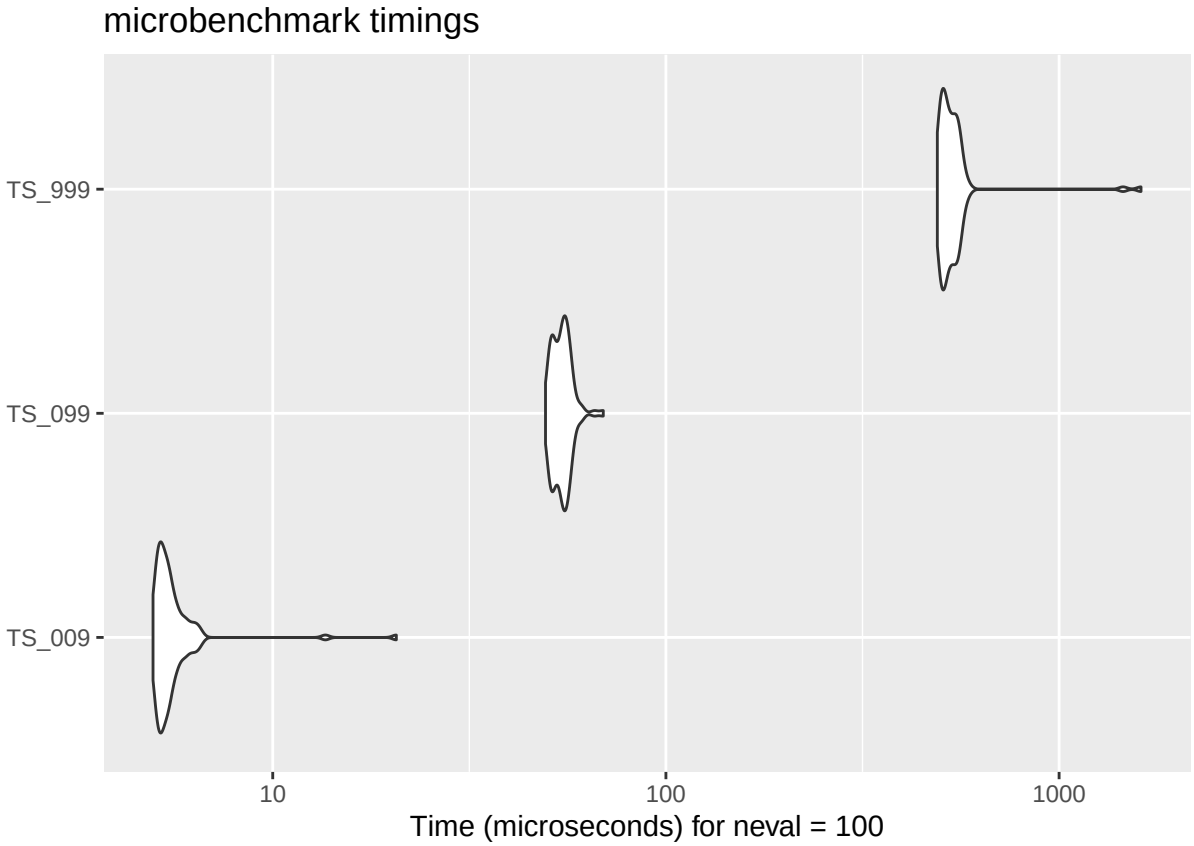
```
comp <- microbenchmark(TS_009 = {geom_growth_base(T = 9)},
                       TS_099 = {geom_growth_base(T = 99)},
                       TS_999 = {geom_growth_base(T = 999)})
```

```
## Warning in microbenchmark(TS_009 = {: less accurate nanosecond times to avoid
## potential integer overflows
```

```
comp
```

```
## Unit: microseconds
##    expr     min       lq      mean median       uq      max neval cld
##  TS_009   4.961   5.1660   5.66866   5.33   5.6375   20.623   100 a
##  TS_099  49.405  51.6395  54.36518  54.53  56.2930   69.290   100  b
##  TS_999 490.114 504.4845 545.90475 521.11 551.2655 1615.523   100   c
```

```
library(ggplot2)
autoplot(comp)
```




# Faster `for`-loops

## Pre-allocated space

Now let's start writing faster code! You may be surprised to learn that even `for`-loops can be made faster with just a tiny adjustment: pre-specifying the length of the container that will hold the results. Let's do that and compare to our old `for`-loop simulation.

```
set.seed(1) # for reproducibility
geom_growth_preallocated <- function(N0 = 2,
```

```
                                    T = 999,
                                    lambda = 1.01,
                                    sigma = 0.2){
  Nvals <- vector('numeric', length = T+1) # here's the only change
  Nvals[1] <- N0
  for (i in 1:T){
    Nvals[i+1] <- Nvals[i]*(lambda*exp(rnorm(1,0,sigma)))
  }
  return(Nvals)
}

# Compare the old and new simulation functions
comp <- microbenchmark(Old = {geom_growth_base(T = 9999)},
                       New = {geom_growth_preallocated(T = 9999)})
comp
```

```
## Unit: milliseconds
##  expr      min       lq     mean   median       uq      max neval cld
##   Old 5.041524 5.109113 5.248608 5.141421 5.184122 7.064546   100  a
##   New 4.262073 4.334274 4.457683 4.360822 4.399915 6.201783   100   b
```

Pre-allocating the size of containers – whether they're vectors, arrays, or lists – can make a big difference when you're using a `for`-loop – or any function – repeatedly.

### Progress bar

Next we'll learn how to avoid `for`-loops altogether using vectorization, which is typically much faster. But oftentimes, slow `for`-loops are impossible to avoid given the nature of the problem (like in our population simulation example). In such cases it's often nice to know how far along your code is in its computation. You could, of course, simply include a `print()` statement just before the end of the `for`-loop (e.g., `print(paste(t, "of", T, "completed"))`), but that will quickly eat up console space unless you also use `flush.console()`. Alternatively, use a progress bars (for which there are a few alternative options). The simplest that does all you really need comes with base-`R`:

```
total <- 10
pb <- txtProgressBar(min = 0, max = total, style = 3)
```

```
##   |                                                                      |
```

```
for(i in 1:total){
   Sys.sleep(0.1)
   setTxtProgressBar(pb, i) # update progress bar
}
```

```
##   |                                                                      |=======
```

```
close(pb)
```

(Note that it's only because of `knitR` that each iteration of the progress bar is printed. It'll remain on a single line in the console.)

## Vectorize it!

As already mentioned, `for`-loops are typically slow. Learning to avoid them is your best ticket to writing faster (and more compact) code. It takes practice though, so we'll start with a simple example.

## Thinking in vectors

Let's say we had population dynamics data, like that which we simulated above, from which we want to calculate the population's growth rate between each pair of successive years.
Using a `for`-loop, we would do:

```
data <- geom_growth_preallocated(T = 99999)

start_time <- Sys.time()
  growth_rates <- vector('numeric', (length(data)-1))
  for(i in 1:(length(data)-1)){
    growth_rates[i] <- data[i+1] / data[i]
  }
end_time <- Sys.time()
end_time-start_time
```

```
## Time difference of 0.006042004 secs
```

But what are we actually doing here? Instead of looping through all the data points, we could do the same by considering them as comprising nothing more than a vector. Because of the way `R` treats vectors, we could take all data points excepting the first, and divide them by all data points excepting the last:

```
start_time <- Sys.time()
  growth_rates <- data[-1] / data[-length(data)]
end_time <- Sys.time()
end_time-start_time
```

```
## Time difference of 0.0009372234 secs
```

We've improved the speed of our code by an order of magnitude! Granted, writing `for`-loops is a good way to figure out what you want to do, but quite often it's good to consider them more like pseudo-code that helps you think through how to write faster functions using vector-based operations.

## `apply()`-family functions

Base-`R` has a number of functions for vector-based operations, and there are many more in various packages (e.g., in the TidyVerse). We'll touch on the `apply()`-family here (which includes `lapply`, `sapply`, `mapply` and more). The key to successful vector-based operations often lies in figuring out how to write the function in order to "apply" it to your data.

We'll demonstrate by continuing on with our population dynamics example. First, let's create a dataset that contains multiple population time-series.

```
n <- 5 # number of time-series to create
# use replicate() to create n time-series, each in a different matrix column
dat_array <- replicate(n, geom_growth_preallocated(T = 9999))
colnames(dat_array) <- paste0('Site_', 1:n)
head(dat_array)
```

```
##         Site_1   Site_2   Site_3   Site_4   Site_5
## [1,] 2.0000000 2.000000 2.000000 2.000000 2.000000
## [2,] 0.9690090 1.886143 2.926192 1.696935 1.776995
## [3,] 1.1994399 1.823862 1.963421 2.050277 1.465000
## [4,] 1.1560687 1.650698 1.955786 1.521136 1.617340
## [5,] 0.8845911 1.440549 1.769213 1.603350 1.709409
## [6,] 0.8374126 1.811698 2.237555 1.638299 2.709000
```

Now let's define a vector-based function to calculate growth rates between all time-steps in a time-series, and apply it to each site (i.e. each column):

```
calc_growth_rates <- function(x){
  gr <- x[-1] / x[-length(x)]
  return(gr)
}
system.time({
  apply(dat_array, 2, calc_growth_rates)
})
```

```
##    user  system elapsed
##   0.001   0.000   0.002
```

```
# margin = 2 means apply to each column
# margin = 1 means apply to each row
```

In order to compare what we just did to the use of `lapply`, we'll first convert the data that's currently in an `array` format (i.e. a matrix) into a `list` format. That is, we'll take each column (i.e. each population time series) and place it in its own list element.

```
dat_list <- as.list(as.data.frame(dat_array))
```

```
system.time({
  growth_rates <- lapply(dat_list, calc_growth_rates)
})
```

```
##    user  system elapsed
##       0       0       0
```

```
lapply(growth_rates, head) # look only at head of each list element
```

```
## $Site_1
## [1] 0.4845045 1.2378006 0.9638405 0.7651718 0.9466662 0.9764813
##
## $Site_2
## [1] 0.9430715 0.9669795 0.9050567 0.8726905 1.2576445 0.8296649
##
## $Site_3
## [1] 1.4630962 0.6709817 0.9961111 0.9046046 1.2647180 0.9350358
##
## $Site_4
## [1] 0.8484676 1.2082235 0.7419173 1.0540477 1.0217974 0.7870187
##
## $Site_5
## [1] 0.8884976 0.8244253 1.1039861 1.0569263 1.5847587 1.5072356
```

The nice thing about lists (as opposed to matrices and arrays) is that list elements need not be of the same length (e.g., you could have time-series for different populations that differ in their number of time points). Since `lapply` returns a list, performing additional computations by site (e.g., calculating an arithmetic mean growth rate) is super fast and easy. (No looping required!)

```
growth_rate_means <- lapply( lapply(dat_list, calc_growth_rates), mean)
growth_rate_means
```

```
## $Site_1
## [1] 1.029464
##
## $Site_2
## [1] 1.030702
##
```

```
## $Site_3
## [1] 1.031581
##
## $Site_4
## [1] 1.029763
##
## $Site_5
## [1] 1.03346
```

```
unlist(growth_rate_means)
```

```
##   Site_1   Site_2   Site_3   Site_4   Site_5
## 1.029464 1.030702 1.031581 1.029763 1.033460
```

The function `sapply` is similar to `lapply` but returns a vector, matrix, or array instead of a list.

```
sapply( lapply(dat_list, calc_growth_rates), mean)
```

```
##   Site_1   Site_2   Site_3   Site_4   Site_5
## 1.029464 1.030702 1.031581 1.029763 1.033460
```

The function `mapply` is useful when you want to parameterize a function from multiple vectors. For example, suppose you have a function that has two parameters and you want to run it through a series of parameter combinations:

```
dat_list <- mapply(geom_growth_preallocated,
           N0 =c(Site1 = 1.8, Site2 = 2.1), # initiate simul. with diff. N0's
           T = c(Site1= 4, Site2 = 9)) # initiate simul. with diff. T's
dat_list
```

```
## $Site1
## [1] 1.800000 2.010616 1.524689 2.390441 2.513850
##
## $Site2
##  [1] 2.100000 2.716472 2.887920 2.581156 2.281833 3.093818 2.943172 2.650243
##  [9] 2.142214 2.304969
```

```
sapply( lapply(dat_list, calc_growth_rates), mean)
```

```
##    Site1    Site2
## 1.123694 1.025155
```

# Parallelize it!

## Running `lapply` in parallel

You can gain speed for vector-based operations by using the analogs of the `apply()`-family from the `parallel` package. (There are other packages as well, like `multidplyr` for the TidyVerse.) We'll demonstrate with `mclapply` ("multicore lapply"), but note that there are others as well (e.g., `parSapply`). (*Windows users:* `mclapply` may not work for you, so skip it and move on to the next example!)

```
# Generate a large amount of demonstration data
n <- 1E8
data_list <- list("A" = rnorm(n),
                  "B" = rnorm(n),
                  "C" = rnorm(n),
                  "D" = rnorm(n))
```

```
# Single core
system.time(
  means <- lapply(data_list, mean)
)
```

```
##    user  system elapsed
##   0.569   0.173   0.749
```

Compare that runtime to using `mclapply` on twice the number of cores:

```
library(parallel)
detectCores()
```

```
## [1] 8
```

```
cores <- 2 # use as many as you have, if you'd like,
           # but note that you'll leave less resources
           # for the rest of your computer!

# mclapply may not work on a Windows machine!
system.time(
  means <- mclapply(data_list, mean, mc.cores = cores)
)
```

```
##    user  system elapsed
##   0.000   0.012   0.428
```

```
unlist(means)
```

```
##             A             B             C             D
##  6.080599e-05  9.890973e-05  9.702543e-05 -1.103169e-05
```

(Note that running things in parallel will take longer than will non-parallelized functions for trivial problems; it takes a little bit of resources and time to set things up on all cores.)

## Running `for`-loops in parallel

When you can't avoid using `for`-loops, but each round of your loop is independent of the others (or you've got nested loops that are independent), you can use the `foreach` package which supports a parallelizable operator `dopar` from the `doParallel` package.

```
library(parallel)
library(foreach)
library(doParallel)
```

```
## Loading required package: iterators
```

```
detectCores()
```

```
## [1] 8
```

```
cores <- 2
cl <- makeCluster(cores) # Create cluster
registerDoParallel(cl) # Activate clusters
system.time({
  means <- foreach(i = 1:length(data_list),
                           .combine = c) %dopar% {
                             # replace c with rbind to create a dataframe
                             mean(data_list[[i]])
```

```
                            }
})
```

```
##    user  system elapsed
##   3.353   1.279   7.714
```

```
stopCluster(cl) # Stop cluster to free up resources
means
```

```
## [1]  6.080599e-05  9.890973e-05  9.702543e-05 -1.103169e-05
```

In this particular example it's slower even than base `lapply`, but that's usually an exception. Nevertheless, it's important to not assume that running things in parallel will always be faster! For example, moving lots of data between cores slows things down significantly.

# Profiling

As you can hopefully sense already, there are many ways to write faster code. The more you practice vector-based thinking, the faster, more compact, and – generally speaking – easier to read your code will be come (as long as you don't overdo it). That said, you also don't want to fall into the trap of wasting time trying to speed up code that doesn't take that long to run in the first place! Rather, you want to spend your time optimizing the part of your code that actually does slow things down. This is where profiling comes in.

There are a few packages available (e.g., `lineprof` and `profvis`), but here we'll only use `Rprof()` from base-`R` to identify sections of code (components of a function) that are slowing it down. To demonstrate, let's create a function that has a few functions nested within it.

```
SimCalc_growth_rates <- function(n){
  N0s <- rlnorm(n)
  Ts <- round( rnorm(n, 50, 10), 0)
  data <- mapply(geom_growth_preallocated,
                 N0 = N0s,
                 T = Ts)
  growth_rates <- lapply(data, calc_growth_rates)
  mean_growth_rates <- sapply(growth_rates, mean, na.rm=TRUE)
  return(mean_growth_rates)
}
```

You use `Rprof()` by initiating it above, and ending it below, your code:

```
Rprof(line.profiling=TRUE)
  out <- SimCalc_growth_rates(1000)
Rprof(NULL)

summaryRprof()
```

```
## $by.self
##                self.time self.pct total.time total.pct
## "<Anonymous>"       0.02       50       0.02        50
## "length"            0.02       50       0.02        50
##
## $by.total
##                           total.time total.pct self.time self.pct
## "block_exec"                    0.04       100      0.00        0
## "call_block"                    0.04       100      0.00        0
## "eng_r"                         0.04       100      0.00        0
## "eval_with_user_handlers"       0.04       100      0.00        0
```

```
## "eval"                          0.04       100      0.00         0
## "evaluate_call"                 0.04       100      0.00         0
## "evaluate::evaluate"            0.04       100      0.00         0
## "evaluate"                      0.04       100      0.00         0
## "handle"                        0.04       100      0.00         0
## "in_dir"                        0.04       100      0.00         0
## "in_input_dir"                  0.04       100      0.00         0
## "knitr::knit"                   0.04       100      0.00         0
## "process_file"                  0.04       100      0.00         0
## "process_group"                 0.04       100      0.00         0
## "rmarkdown::render"             0.04       100      0.00         0
## "SimCalc_growth_rates"          0.04       100      0.00         0
## "timing_fn"                     0.04       100      0.00         0
## "withCallingHandlers"           0.04       100      0.00         0
## "withVisible"                   0.04       100      0.00         0
## "xfun:::handle_error"           0.04       100      0.00         0
## "<Anonymous>"                   0.02        50      0.02        50
## "length"                        0.02        50      0.02        50
## "FUN"                           0.02        50      0.00         0
## "isTRUE"                        0.02        50      0.00         0
## "lapply"                        0.02        50      0.00         0
## "mapply"                        0.02        50      0.00         0
## "mean.default"                  0.02        50      0.00         0
## "sapply"                        0.02        50      0.00         0
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 0.04
```

```
summaryRprof(lines='show')
```

```
## $by.self
##          self.time self.pct total.time total.pct
## <text>#8      0.02       50       0.02        50
## <text>#9      0.02       50       0.02        50
##
## $by.total
##          total.time total.pct self.time self.pct
## <text>#8       0.02        50      0.02       50
## <text>#9       0.02        50      0.02       50
## <text>#4       0.02        50      0.00        0
##
## $by.line
##          self.time self.pct total.time total.pct
## <text>#4      0.00        0       0.02        50
## <text>#8      0.02       50       0.02        50
## <text>#9      0.02       50       0.02        50
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 0.04
```

Note that `Rprof()` is not always able to identify all the components (e.g., when functions aren't named). In these instances it'll indicate said function(s) by calling them "Anonymous".