Week 4:Simulation & Profiling

Katherine Ramírez Cubillos

23/8/2020

The srt Function

srt: Compactly display the internal structure of an R object

- A diagnostic funciton and an alternative to 'summary'.
- It is especially well studied to compactly display the (abbreviated) contents of (possibily nested) lists.
- Roughly one line per basic object.
- The basic goal of str is to answer the question: What is in the object?

```
str(str)
```

```
## function (object, ...)
```

```
str(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr", model = TRUE,
##     x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL,
##     offset, ...)
```

It shows the funciton arguments for the **lm** function. You can see a very brief summary, it takes the first argument's a formula, the second argument's data, etc.

The next example generates some normal random variables, 100 of the, mean two variant, and standard derivation four.

```
x <- rnorm(100, 2, 4)
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -6.749  -1.452   2.372   1.808   4.911   9.588
```

```
## give the first five numbers in the vector x
str(x)
```

```
##  num [1:100] -3.476 7.892 -0.118 -6.749 5.358 ...
```

```
# The factor has 40 levels and each one is
# repeated ten times so if I call str on it.
f <- gl(40,10)
str(f)
```

```
##  Factor w/ 40 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
# The output's a llittle bit different.
# It gives the number of elements in each of the 40 different levels.
summary(f)
```

```
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
## 27 28 29 30 31 32 33 34 35 36 37 38 39 40
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

It is possible to call str to get a little some different output. In a data-frame it tells that there's a 153 observations, so there are 153 rows in the data frame, with six variables and then for each variable it gives a little output.

```
library(datasets)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
#Quick examination of data that you might have in R.
# And what the structure of different R objects is.
```

Matrix with random normals in there. It will be a 10 by 10 matrix. The function srt will give a little bit more information. It'll say that it's a two-dimensional array.

```
m <- matrix(rnorm(100), 10,10)
str(m)
```

```
##  num [1:10, 1:10] -0.0781 -2.1811 -3.2038 0.9876 -0.4566 ...
```

The next example is a list by using the split funciton and see how srt can look at the list and give a compact summary of it. It takes the airquality data frame and split it by the month.

```
s <- split(airquality, airquality$Month)
str(s)
```

```
## List of 5
##  $ 5:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
##   ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
##   ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##   ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
##   ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 6:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
##   ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
##   ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
##   ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
##   ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 7:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
##   ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
##   ..$ Wind   : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
##   ..$ Temp   : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
##   ..$ Month  : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 8:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
##   ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
##   ..$ Wind   : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
##   ..$ Temp   : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
##   ..$ Month  : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 9:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
##   ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
##   ..$ Wind   : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
##   ..$ Temp   : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
##   ..$ Month  : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

This list contains five different data frames where each data frame corresponds to the data for a given month.

Simulation

Funcitons for probability distribution in R:

- **rnorm**: generate random Normal variates with a given mean and standar deviation.

- **dnorm**: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- **pnorm**: evaluate the cumulative distribution function for a normal distribution.
- **rpois** generate random Poisson variates with a given rate.

Generating Random Numbers

Probability distribution functions usually have four funcitons associated with them. The function are prefixed with a:

- **d** for density.
- **r** for random number generation.
- **p** for cumulative distribution.
- **q** for quantile funciton.

Working with the Normal distributions requires using four functions.

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

```
##   [1] 9.485630e-04 1.189157e-14 3.961662e-01 5.142691e-11 2.331114e-07
##   [6] 1.086652e-01 4.211311e-02 1.060261e-01 5.898221e-04 2.326794e-01
##  [11] 3.792249e-06 1.023261e-04 3.825260e-04 9.792818e-02 1.309359e-05
##  [16] 2.850441e-04 1.046793e-01 2.353813e-01 3.206485e-01 1.520400e-01
##  [21] 8.017127e-07 1.539465e-02 3.863914e-01 3.953303e-05 1.509046e-07
##  [26] 3.102611e-04 3.295054e-06 1.783946e-02 3.829233e-07 6.842064e-17
##  [31] 6.931255e-04 2.882110e-06 7.492132e-03 1.357030e-03 5.871678e-09
##  [36] 4.255968e-02 2.935129e-14 3.978087e-01 7.416008e-05 3.859279e-01
##  [41] 3.227538e-01 2.178270e-05 2.676172e-02 2.330957e-01 7.288035e-16
##  [46] 6.398635e-11 1.037177e-10 1.322469e-07 1.966190e-02 2.558407e-01
##  [51] 1.133752e-07 7.586252e-09 1.197454e-05 2.265808e-03 2.134850e-02
##  [56] 2.952071e-05 8.747481e-03 6.593892e-07 1.666136e-01 1.240216e-02
##  [61] 1.039657e-02 1.188078e-06 2.862458e-07 5.608792e-02 3.787940e-01
##  [66] 1.562697e-03 4.348326e-21 4.215463e-16 7.993159e-10 7.799814e-08
##  [71] 2.489376e-03 1.126988e-01 2.385597e-01 3.668325e-01 3.491613e-01
##  [76] 5.049573e-04 1.347735e-01 2.810758e-02 3.954515e-01 1.892259e-10
##  [81] 1.437005e-01 1.907454e-14 1.514490e-03 6.946332e-08 9.426809e-05
##  [86] 1.530027e-01 2.223238e-04 3.736788e-17 1.144594e-10 2.935785e-06
##  [91] 1.524356e-06 1.403540e-01 2.173667e-01 2.742312e-03 3.952197e-01
##  [96] 1.642518e-02 6.774872e-03 1.356068e-07 1.274312e-04 9.133956e-04
```

```
# pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
# qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
# rnorm(n, mean = 0, sd = 1)
```

If ϕ is the cumulative distribution for a standar Normal distribution, then $pnorm(q) = \Phi(q)$ and $qnorm(p) = \Phi^{-1}(p)$

```
x <- rnorm(10)
x
```

```
##  [1]  0.3859187  1.6786268  1.6505217 -0.8672358 -0.3984341  0.2592715
##  [7] -0.8295117 -0.2623993  0.1235248 -0.4601073
```

```r
x <- rnorm(10,20,2)
x
```

```
##  [1] 20.78853 18.26480 17.71275 22.29522 24.48039 19.48712 16.06058 20.64381
##  [9] 19.42273 21.92412
```

Setting the random number with **set.seed** ensures reproducibility.

```r
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

```r
rnorm(5)
```

```
## [1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

```r
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Poisson data

```r
rpois(10,1)
```

```
##  [1] 0 0 1 1 2 1 1 4 1 2
```

```r
rpois(10,2)
```

```
##  [1] 4 1 2 0 1 1 0 1 4 1
```

```r
rpois(10,20)
```

```
##  [1] 19 19 24 23 22 24 23 20 11 22
```

```r
ppois(2,2)   ## Cumulative distribution
```

```
## [1] 0.6766764
```

```r
             ## Pr(x <= 2)
ppois(4,2)
```

```
## [1] 0.947347
```

```r
              ## Pr(x <= 4)
ppois(6,2)
```

```
## [1] 0.9954662
```

```r
              ## Pr(x <= 6)
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   16.06   18.55   20.07   20.11   21.64   24.48
```

Generating Random Numbers From a Linear Model

Suppose we want to simulate from the following linear model:

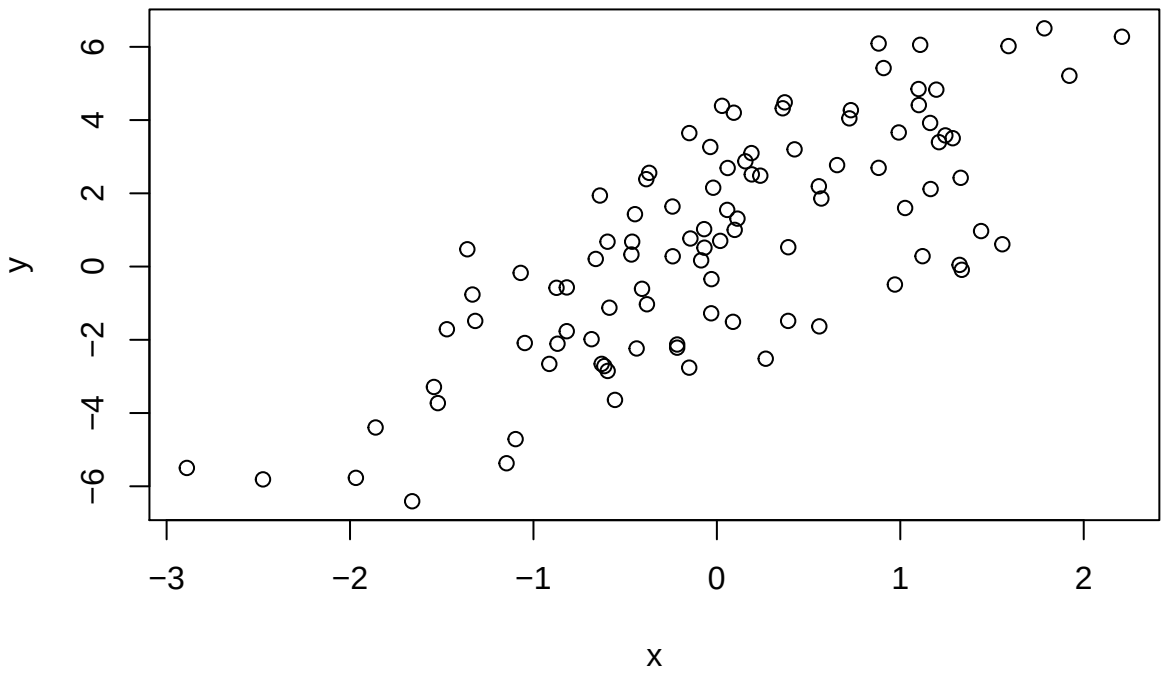
$y = \beta_0 + \beta_1 x + \epsilon$

where ϵ $N(0, 2^2)$. Assume x $N(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$

```r
set.seed(20)
x <- rnorm(100)
e <- rnorm(100, 0, 2) # std = 0 & media =2.
y <- 0.5 + 2*x + e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -6.4084 -1.5402  0.6789  0.6893  2.9303  6.5052
```

```r
plot(x,y)
```

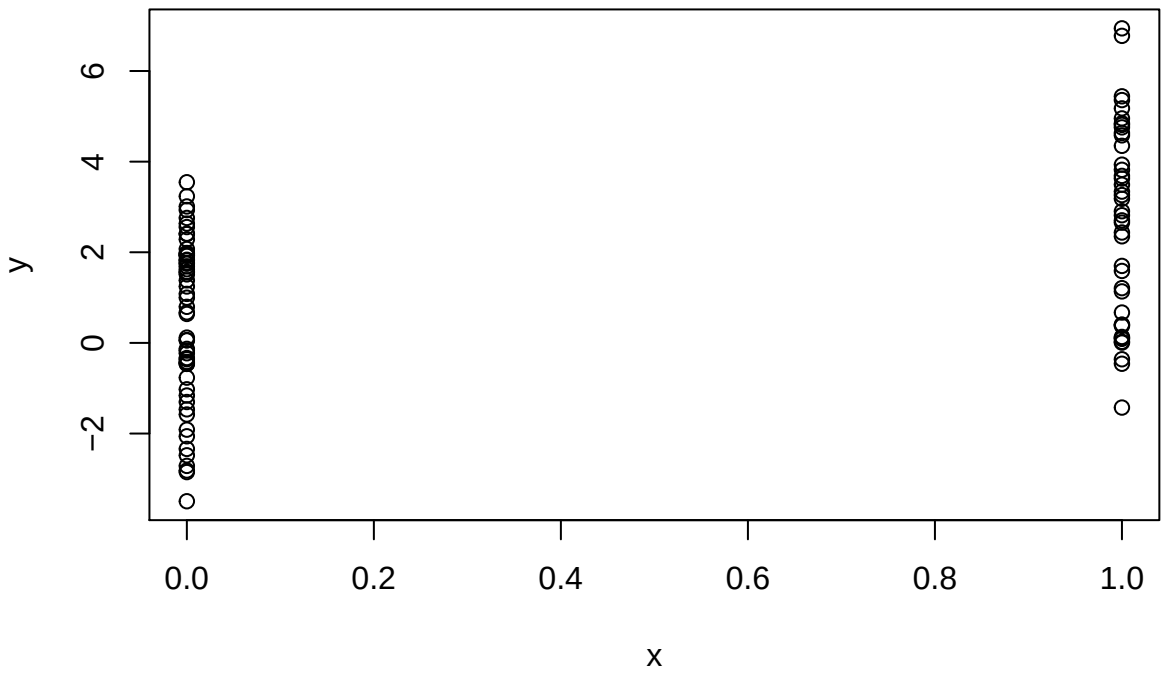


What if **x** is binary?

```
set.seed(10)
x <- rbinom(100, 1, 0.5)
e <- rnorm(100, 0, 2)
y <- 0.5 + 2*x + e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -3.4936 -0.1409  1.5767  1.4322  2.8397  6.9410
```

```
plot(x, y)
```



Generating Random Numbers From a Generalized Linear Model

Suppose we want to simulate from a Poisson model where:

$Y = Poisson\ (u)$

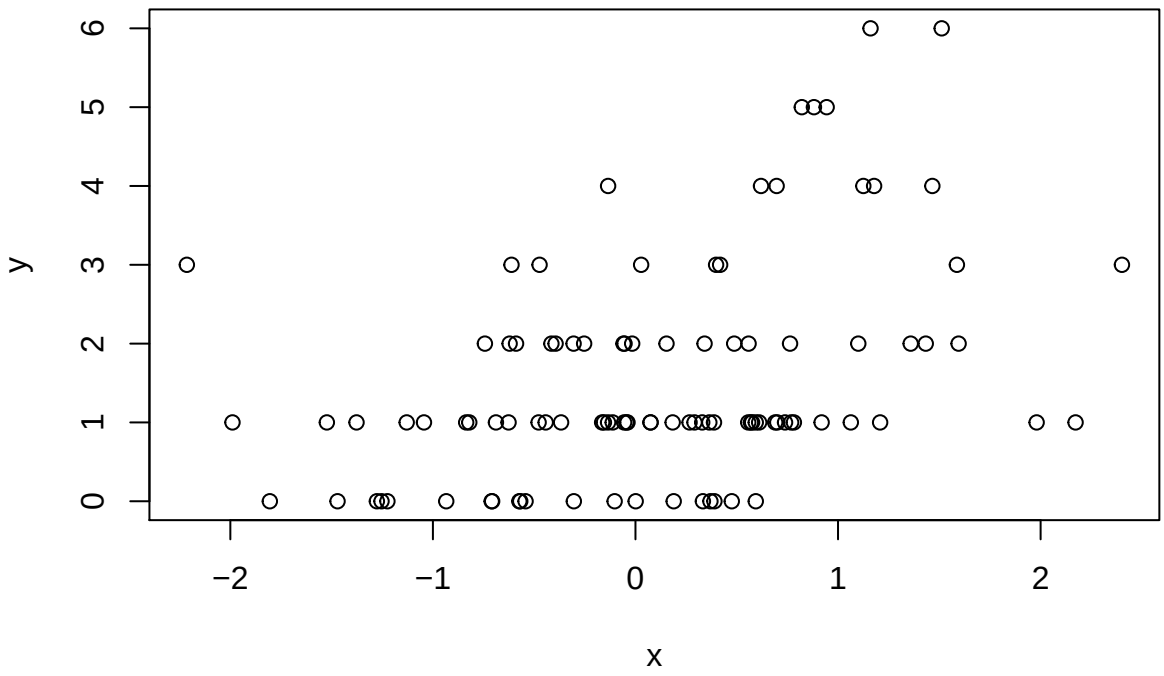
$\log u = \beta_0 + \beta_1 x$

And $\beta_0 = 0.5$ and $\beta_1 = 0.5$. We need to use the **rpois** function for this.

```
set.seed(1)
x <- rnorm(100)
log.mu <- 0.5 + 0.3*x
y <- rpois(100, exp(log.mu))
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.00    1.00    1.55    2.00    6.00
```

```
plot(x, y)
```



Random Sampling

The **sample** function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
set.seed(1)
# I pass the vector of integers of one to ten.
# I want o sample randomly four of them, without replacements.
sample(1:10, 4)
```

```
## [1] 9 4 7 1
```

```
sample(1:10, 4)
```

```
## [1] 2 7 3 6
```

```
sample(letters, 5)
```

```
## [1] "r" "s" "a" "u" "w"
```

```
sample(1:10)   ## permutation
```

```
##  [1] 10  6  9  2  1  5  8  4  3  7
```

```r
sample(1:10)
```

```
##  [1]  5 10  2  8  6  1  4  3  9  7
```

```r
sample(1:10, replace=TRUE)  ## Sample w/replacement
```

```
##  [1]  3  6 10 10  6  4  4 10  9  7
```

Simulation

- Drawing samples from specific probability distributions can be done with **r** functions.
- Standar distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The **sample** function can be used to draw random samples from arbitrary vectors.
- Setting the random number generator seed via set.seed is critical for reproducibility.

Profiling R Code

Why is my code so slow?

- Proffing is a systematic way to examine how much time is spend in different parts of a program.
- Useful when trying to optimize your code.
- Often code runs fine once, but what if you have to put it in a loop for 1,000 interation? Is it still fast enough?
- Profiling is better than guessing.

On optimizing your code

- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time.
- This cannot be done without 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)

General principles of optimization

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

Using system.time()

- Usually, the user time and elapsed time are relatively close, for straight computing tasks.
- Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around.
- Elapsted time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them)
- Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
- Parallel processing via the **parallel** package.

```r
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
```

```
##    user  system elapsed
##    0.31    0.17    2.68
```

```r
## 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
##    7.56    0.05    7.72
```

Timing longer expressions

```r
system.time({
  n <- 1000
  r <- numeric(n)
  for(i in 1:n){
    x <- rnorm(n)
    r[i] <- mean(x)
  }
})
```

```
##    user  system elapsed
##    0.28    0.03    0.31
```

Beyond system.time()

- Using **system.time()** allows you to test certain functions or code blocks to see if they are taking excessive amounts of time.
- Assumes you already know where the problem is and can call **system.time()** on it.
- What if you don't know where to start?

The R Profiler

- The **Rprof()** funciton starts the profiler in R.
- R must be compiled with profiler support (but this is usually the case)
- The **summaryRprof()** funciotn summarizes the output from **Rprof()** (otherwise it's not readable)
- Do not use **system.time()** and **Rprof()** together or you will be sad.
- **Rprof()** keeps track of the function call stack at regularly sample intervals and tabulates how much time is spend in each funciton.

- Defult sampling interval is 0.02 seconds.
- NOTE: If your code runs very quicly, the profiler is not useful, but then you probably don't need it in that case.

Using summaryRprof()

- The **summaryRprof()** funciton tabulates the R profiler output and calculates how much time is spend in which funciton.
- There are two methods for normalizing the data.
- "by.total" divides the time spend in each funciton by the total run time.
- "by.self" does the same but first subtracts out time spent in functions above in the call stack.

```
## lm(y~x)
sample.interval = 10000
# $by.total
# $by.self
# $sample.interval
# $sample.time
```

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 of normalization)
- Good to break your code into functions so that the profiler can give useful informacion about where time is being spent.
- C or Fortran code is not profiled.