

For many, R code that works properly is good enough. However, if you are planning to share, package, or use your code repeatedly, you might consider the efficiency of your code. Running your code and timing it is a good starting point.

This page will demonstrate two R commands for timing code: **proc.time** and **system.time**. It will also illustrate the first rule of making R code efficient: avoid loops! We will start with a vector of 100,000 values sampled randomly from a standard normal. We want to add 1 to each of these values. We will do this with and without looping, timing both.

The **proc.time** command essentially works as a stop-watch: you initialize it to a starting time, run all the code desired, and then stop it by subtracting the starting time from the ending time.

We can first use the slow, looping method to add 1 to each value in our vector:

```
g <- rnorm(100000)
h <- rep(NA, 100000)

# Start the clock!
ptm <- proc.time()

# Loop through the vector, adding one
for (i in 1:100000){
  h[i] <- g[i] + 1
}

# Stop the clock
proc.time() - ptm
```

user	system	elapsed
0.34	0.06	0.41

Alternatively, we can use a non-looping approach:

```
ptm <- proc.time()
```

```
h <- g + 1
proc.time() - ptm
```

user	system	elapsed
0.00	0.02	0.01

The values presented (**user**, **system**, and **elapsed**) will be defined by your operating system, but generally, the **user** time relates to the execution of the code, the **system** time relates to system processes such as opening and closing files, and the **elapsed** time is the difference in times since you started the stopwatch (and will be equal to the sum of **user** and **system** times if the chunk of code was run altogether and single-threaded). Note that the elapsed time may be shorter than the sum of the user time and system time if multiple threads were used to execute the expression. While the difference of .42 seconds may not seem like much, this gain in efficiency is huge!

The **system.time** command takes a single R expression as its argument. Thus, to repeat the steps above using **system.time**, we wrote function wrappers around our fast and slow methods. Again we see that the looping method is much slower. Note that **system.time** is simply calling **proc.time**! So the better one to use just depends on the nature of the code you wish to time.

```
quickadd <- function(g){
  return(g+1)
}
```

```
slowadd <- function(g){  
  h <- rep(NA, length(g))  
  for (i in 1:length(g)){  
    h[i] <- g[i] + 1  
  }  
  return(h)  
}
```

```
system.time(a <- slowadd(g))  
  user  system elapsed  
0.34    0.00    0.34
```

```
system.time(a <- quickadd(g))  
  user  system elapsed  
0      0      0
```

References

Lim, A., & Tjhi, W. (2015). *R High Performance Programming*. Packt Publishing Ltd.

[Click here to report an error on this page or leave a comment](#)

[How to cite this page \(https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-how-do-i-cite-web-pages-and-programs-from-the-ucla-statistical-consulting-group/\)](https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-how-do-i-cite-web-pages-and-programs-from-the-ucla-statistical-consulting-group/)