

Efficient R: practical solutions

Dr Colin Gillespie

Before starting the questions, make sure you can load the rbenchmark package

```
library(rbenchmark)
```

Each practical corresponds to a chapter in the notes.

Practical 1

1. Reproduce the timing results in chapter 1 using the benchmark function from the rbenchmark package.
2. **Case study** In this example, we are going to investigate loading a large data frame. First, we'll generate a large matrix of random numbers and save it as a csv file:¹

```
N = 1e+05  
m = as.data.frame(matrix(runif(N), ncol = 1000))  
write.csv(m, file = "example.csv", row.names = FALSE)
```

¹ If setting N=1e6 is too large for your machine, reduce it a bit. For example, N=50,000.

We can read the file the back in again using read.csv

```
dd = read.csv("example.csv")
```

To get a baseline result, time the read.csv function call above.

```
system.time(read.csv("example.csv"))  
##      user  system elapsed  
##    0.376    0.004    0.383
```

We will now look ways of speeding up this step.

- (a) Use the nrows argument to set the number of rows that will be read from your file.²
- (b) Set comment.char="" to turn off interpretation of comments.
- (c) Explicitly define the classes of each column using colClasses in read.csv, for example, if we have 1000 columns that all have data type numeric, then:

```
read.csv(file="example.csv",  
         colClasses=rep("numeric", 1000))
```

² Hint, use nrow(m) to determine how many rows are in your matrix.

- (d) Use the save and load functions:

```
save(m, file = "example.RData")  
load(file = "example.RData")
```

Which of the above give the biggest speed-ups? Are there any downsides to using these techniques? Do your results depend on the number of columns or the number of rows?

```
## 1. Using RData files is the fastest - although
## you have to read the data in first. Set
## colClasses also produces an good speed-up.

## 2. Setting colClasses R is no longer checking
## your data types. If your data is changing - for
## example it's coming from the web or a database,
## this may be problem.

## 3. The results do depend on the number of
## columns, as this code demonstrates
N = c(1, 10, 100, 1000, 1000, 10000)
l = numeric(5)
for (i in seq_along(N)) {
  m = as.data.frame(matrix(runif(N[6]), ncol = N[i]))
  write.csv(m, file = "example.csv", row.names = FALSE)
  cc = rep("numeric", N[i])
  l[i] = system.time(read.csv("example.csv", colClasses = cc))[3]
}
l

## Notice that when we have a large number of
## columns, we get a slow down in reading in data
## set (even though we have specified the column
## classes). The reason for this slow down is that
## we are creating a data frame and each column has
## to be initialised with a particular class.
```

Practical 2

1. In this question, we'll compare matrices and data frames. Suppose we have a matrix, `d_m`

```
## For fast computers d_m = matrix(1:1000000,
## ncol=1000) Slower computers
d_m = matrix(1:10000, ncol = 100)
dim(d_m)

## [1] 100 100
```

and a data frame `d_df`:

```
d_df = as.data.frame(d_m)
colnames(d_df) = paste("c", 1:ncol(d_df), sep = "")
```

- (a) Using the following code, calculate the relative differences between selecting the first column/row of a data frame and matrix.

```
benchmark(replications=1000,
          d_m[1,], d_df[1,], d_m[,1], d_df[,1],
          columns=c('test', 'elapsed', 'relative'))
```

Can you explain the result? Try varying the number of replications.

```
## Two things are going on here 1. The very large
## difference when selecting columns and rows (in
## data frames) is because the data is stored in
## column major-order. Although the matrix is also
## stored in column major-order, because everything
## is the same type, we can efficiently select
## values.

## 2. Matrices are also more memory efficient:
m = matrix(runif(10000), ncol = 10000)
d = data.frame(m)
object.size(m)

## 80200 bytes

object.size(d)

## 1120568 bytes
```

- (b) When selecting columns in a data frame, there are a few different methods. For example,

```
d_df$c10
d_df[, 10]
d_df[, "c10"]
d_df[, colnames(d_df) == "c10"]
```

Compare these four methods.

2. Consider the following piece of code:

```
a = c()
for(i in 1:n)
  a = c(a, 2 * pi * sin(i))
```

This code calculates the values:

$$2\pi \sin(1), 2\pi \sin(2), 2\pi \sin(3), \dots, 2\pi \sin(n)$$

and stores them in a vector. Two obvious ways of speeding up this code are:

- Pre-allocate the vector `a` for storing your results.
- Remove $2 \times \pi$ from the loop, i.e. at the end of the loop have the statement: `2*pi*a`.

Try the above techniques for speeding up the loop. Vary n and plot your results.

3. R is an interpreted language; this means that the interpreter executes the program source code directly, statement by statement. Therefore, every function call takes time.³ Consider these three examples:

```
n = 1e6
## Example 1
I = 0
for(i in 1:n) {
  10
  I = I + 1
}
## Example 2
I = 0
for(i in 1:n){
  ((((((((((10))))))))))
  I = I + 1
}
## Example 3
I = 0
for(i in 1:n){
  ##This is a comment
  ##But it is still parsed
  ##So takes time
```

³ This example is for illustrative purposes. Please don't start worrying about comments and brackets.

```
##But not a lot  
##So don't worry!  
10  
I = I + 1  
}
```

Using the benchmark function, time these three examples.

Practical 3: parallel programming

1. To begin, load the parallel package and determine how many cores you have

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

2. Run the parallel apply example in the notes.
 - On your machine, what value of N do you need to use to make the parallel code run quicker than the standard serial version?
 - When I ran the benchmarks, I didn't include the makeCluster and stopCluster functions calls. Include these calls in your timings. How does this affect your benchmarks?
3. Run the dice game Monte-Carlo example in the notes. Vary the parameter M.⁴

⁴ Try setting M=50 and varying N.

Solutions

Solutions are contained within this package:

```
library(nclRefficient)
vignette("solutions1", package = "nclRefficient")
```