Efficient R: practical

Dr Colin Gillespie

Course R package

Before starting, we first install `drat`[1]

```
install.packages("drat")
```

Then install the course R package via

```
drat::addRepo("rcourses")
install.packages("nclRefficient", type="source")
```

This R package contains copies of the practicals, solutions and data sets that we require. To load the package, use

```
library("nclRefficient")
```

Each practical corresponds to a chapter in the notes.

Practical 1

1. Reproduce the timing results in chapter 2 using the `benchmark` function from the `rbenchmark` package. Remember to load the package using

   ```
   library("rbenchmark")
   ```

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:[2]

   ```
   N = 1e5
   m = as.data.frame(matrix(runif(N), ncol=1000))
   write.csv(m, file="example.csv", row.names=FALSE)
   ```

   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. We will now look ways of speeding up this step.

   (a) 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:

[1] If you have already installed `drat` you can skip this step.

[2] If setting `N=1e6` is too large for your machine, reduce it at bit. For example, `N=50,000`.

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

(b) Use the `saveRDS` and `readRDS` functions:

```
saveRDS(m, file="example.RData")
readRDS(file="example.RData")
```

(c) Install the `readr` package via

```
install.packages("readr")
```

Then load the package in the usual way

```
library("readr")
```

This package contains the function `read_csv`; a replacement function for `read.cvs`. Is this new function much better than the default?

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?

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) = paste0("c", 1:ncol(d_df))
```

(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.

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

This code calculates the values:

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

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:

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

```
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
  ##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")
```