Vectorization, Timing, and Parallelization – Solutions

Stats 216

27 January 2016

In this class we’re going to take a step back from statistical analysis and instead take a closer look at what’s going on under the hood when you run a program in R. This is incredibly important, because when you work with large real-world datasets, even small inefficiencies can lead to large costs.

We will cover:

- vectorization
- how to time functions
- a closer look at function calls in R
- parallelization
- some additional information about data structures and the apply class of functions

Constructing squares

Suppose we need to construct a vector of perfect squares. More specifically, let’s say we want to create a function `my_squares` that takes an integer `n` and returns a vector containing all of the squares of the numbers between 1 and `n`. Eventually we may want to call this function on a large value of `n`, say `n = 100000`.

Problem:

Think of at least three different ways to implement the function. In other words, complete the following code in at least three ways:

```r
my_squares <- function(n) {
  # insert your code here
}
```

*(Hint: At first, you may want to test your function with a value of `n` smaller than 100000. Try to write your function using (a) a for loop, (b) sapply or a related function, (c) neither of the above.)*

Solution:

Here are six ways to implement the square function. We store them in a list, which will make it a bit easier to do a simulation later.

```r
# Initialize a list of functions
my_squares <- list()

# Bad for loop. Concatenate each number on to the end of the vector.
my_squares[[1]] <- function(n) {
  result = NULL
  for (ii in 1:n) result <- c(result, ii^2)
  result
}
```
Also bad for loop. Assign each number by index to the end of the vector.

```
my_squares[[2]] <- function(n) {
  result = NULL
  for (ii in 1:n) result[ii] = ii^2
  result
}
```

Better for loop. Initialize the entire vector first.

```
my_squares[[3]] <- function(n) {
  result = rep(0, n)
  for (ii in 1:n) result[ii] = ii^2
  result
}
```

Use sapply.

```
my_squares[[4]] <- function(n) {
  sapply(1:n, function(ii){ii^2})
}
```

Use vapply. Like sapply but specifies the type of function output.

```
my_squares[[5]] <- function(n) {
  vapply(1:n, function(ii){ii^2}, FUN.VALUE=1)
}
```

Directly square the entire vector.

```
my_squares[[6]] <- function(n) {
  (1:n)^2
}
```

Label the functions.

```
names(my_squares) <- c("bad for 1", "bad for 2", "better for", "sapply", "vapply", "(1:n)^2")
```

Let’s check that all the functions actually produce the same output.

```
n <- 1e4
all(identical(my_squares[[1]](n), my_squares[[2]](n)),
    identical(my_squares[[2]](n), my_squares[[3]](n)),
    identical(my_squares[[3]](n), my_squares[[4]](n)),
    identical(my_squares[[4]](n), my_squares[[5]](n)),
    identical(my_squares[[5]](n), my_squares[[6]](n)))
```

## [1] TRUE

Which solution is best?

Which of these six (or more!) functions is best to use? In general there are several ways we may want to optimize our code, but for now let’s just focus on speed.

R provides some simple tools that we can use to test which of these methods is the fastest.
Problem:

Try to figure out how to use either `proc.time` or `system.time` to evaluate how long each method takes to run, and write some code to perform those tests. (You may need to look at some documentation.) Then try to think about why each method takes as long as it does (relative to the others).

Solution:

Let's write a couple functions to help us test the methods with varying values of \( n \).

```r
# Returns the time taken to run function f with argument n.
calculate_time <- function(n, f) {
  start_time <- proc.time()
  f(n)
  (proc.time() - start_time)["elapsed"]
}

# Returns a vector with length(n_values) elements containing the
# results of applying calculate_time for the function f to the
# vector lengths specified in n_values.
simulate <- function(f, n_values) {
  result <- sapply(n_values, FUN=calculate_time, f=f)
  names(result) <- n_values
  result
}
```

Now we actually want to try all of the methods, so we'll call `sapply` again, on the `simulate` function we just wrote.

```r
# This takes about 2 minutes to run.
n_values <- c(1e3, 2e3, 5e3, 1e4, 2e4, 5e4, 1e5)
times <- sapply(my_squares, FUN=simulate, n_values=n_values)
```

Let's take a look at the results.

```r
times
```

```
## bad for 1 bad for 2 better for sapply vapply (1:n)^2
## 1000 0.003 0.002 0.001 0.000 0.001 0.000
## 2000 0.010 0.006 0.001 0.002 0.001 0.000
## 5000 0.052 0.025 0.004 0.003 0.004 0.000
## 10000 0.196 0.106 0.009 0.008 0.010 0.000
## 20000 0.903 0.536 0.016 0.014 0.013 0.000
## 50000 5.639 3.678 0.040 0.042 0.043 0.000
## 1e+05 25.814 18.708 0.081 0.116 0.073 0.001
```

Now we can plot the data we’ve collected. Below, I’m using the `reshape2` and `ggplot2` packages to produce a nicer-looking plot. For the most part you can ignore the below code chunk, and it’s possible to produce a similar plot using the base R `plot` function. But if you’re interested, you can take a look at the documentation for those packages.
library(ggplot2)
# "melt" the data into a long format so that we can plot separate lines for each function
times <- reshape2::melt(times)
# label the columns
names(times) <- c("n", "method", "time")
# place n on the x-axis, time on the y-axis, and separate lines by method
plot <- ggplot(times, aes(x=n, y=time, colour=method))
# draw lines; if you want points, use geom_point() instead
plot <- plot + geom_line()
# use log-log scale so it's easier to see
plot <- plot + scale_x_log10() + scale_y_log10()
# display plot
plot

(Note: The above plot is on a log-log scale.) So clearly the \((1:n)^2\) function is the best to use! But why?

Some details on R and vectorization

To understand why the functions perform so differently, we have to delve a bit deeper into the way that R actually runs code. R is a “high-level programming language.” This means that it is quite human-readable, but does not necessarily correspond to exactly what a computer does. In contrast, a lower-level language like C or Fortran is written in a way that is more difficult for a human to read, but more directly translates to exactly what the computer will do. (C is essentially one level of abstraction above assembly, which is machine code.)

So, what actually happens when we run a function in R? Well, typically R wraps up the requisite information
needed to execute the function and then ships it off to a faster language to actually run. Compare the
printout for the functions `nrow` and `ndim`.

```
dim

# function (x) .Primitive("dim")
```

```
nrow

# function (x)
# dim(x)[1L]
# <bytecode: 0x7f81141fb6a8>
# <environment: namespace:base>
```

The `.Primitive("dim")` means that `dim` is a function that isn’t written in R. On the other hand, `nrow` is
written in R, and actually makes a call to `dim` and then takes the first element. When you evaluate a `dim`, R
calls some compiled C code. This is good because C is a much faster language, but bad because there is some
overhead in the communication. Ideally, you want to write code that passes back and forth between R and C
as few times as possible.

We can see that the power operator is also a C primitive:

```
# function (e1, e2) .Primitive("^")
```

```
(1:n)^2 only needs to make one function call regardless of the size of n and is thus much faster at large
values. In contrast, the other methods, including the for loops and `sapply`, make at least n function calls.
The principle of writing code that can make a constant number of function calls independent of the length of
the input is typically referred to as **vectorization**, and it is crucial to optimizing R code.

**Parallelization**

Almost all modern personal computers have multiple CPU cores. This means that properly written code can
sometimes utilize more than one of these cores at a time, speeding up the time it takes to run. Note that this
is not possible for all tasks; some tasks are more easily parallelized than others, and here we will only discuss
**embarassingly parallel** problems. A task is embarassingly parallel if it can be easily divided into many
small task that can each be completed without speaking to any of the other tasks.

Some embarassingly parallel problems:

- Doing a group project by having each person independently complete a section.
- Calculating all the perfect squares between 1 and 100000.

Some problems that are not embarassingly parallel:

- Doing a group project where each person gets feedback on their section
- Multiple regression

Two other embarassingly parallel problems that are very common in statistics are cross-validation and the
bootstrap! Each of these methods can be thought of as small problems that are done totally independently.
One easy way to do multicore processing in R is via the `mclapply` function in the `parallel` library.
Problem:
Write a parallel solution to the perfect squares problem. How does the runtime compare to other solutions?

Solution:

```r
library(parallel) # for multicore processing
n <- 50000
system.time((my_squares <- lapply(1:n, function(ii) {ii^2})))
##    user  system elapsed
##       0.037 0.003 0.040

system.time((my_squares_mcl <- mclapply(1:n, function(ii) {ii^2})))
##    user  system elapsed
##       0.039 0.035 0.071

identical(my_squares, my_squares_mcl)
## [1] TRUE
```

We see that the multicore version actually takes longer to run. For any problem there’s a tradeoff between the time gained from running the tasks on multiple cores, and the overhead that R needs to set up the multiple cores. For this toy example, there’s too much overhead in the problem to make the multicore version worth the effort.

Parallel bootstrap

Here is a normal implementation of the bootstrap, taken from the Chapter 5 R session video:

```r
library(boot) # for the boot function
library(ISLR) # contains the data

alpha <- function(x, y) {
  vx <- var(x)
  vy <- var(y)
  cxy <- cov(x,y)
  (vy - cxy) / (vx + vy - 2*cxy)
}

alpha.fn <- function(data, index) {
  with(data[index, ], alpha(X,Y))
}

# calculate alpha on the dataset
alpha.fn(Portfolio, 1:100)
```
# generate one bootstrap replica of alpha
set.seed(1)
alpha.fn(Portfolio, sample(1:100, 100, replace=TRUE))

# generate 1000 replicas
boot.out <- boot(Portfolio, alpha.fn, R=1000)

Problem:
Write a new parallel way of computing boot.out (without using the boot function). Compare your results with the boot function’s multicore option.

Solution:
Let’s define the alpha functions as above.

library(boot)  # for the boot function
library(ISLR)  # contains the data

alpha <- function(x, y) {
  vx <- var(x)
  vy <- var(y)
  cxy <- cov(x,y)
  (vy - cxy) / (vx + vy - 2*cxy)
}

alpha.fn <- function(data, index) {
  with(data[index,], alpha(X,Y))
}

First, try the multicore option of the boot function:

set.seed(1)
system.time((boot.out=boot(Portfolio, alpha.fn, R=1000,
parallel="multicore", ncpus=2)))

## user  system elapsed
## 0.035  0.032  0.154

Next, try an implementation with mclapply:

require(parallel)
system.time((mcl.boot.out=simplify2array(mclapply(1:1000,
function(ii) {
  set.seed(ii)
  alpha.fn(Portfolio, sample(1:100, 100, replace=TRUE))
},
mc.cores=2)))))
Some additional practice

The apply functions

Problem:

1. What function would you use to calculate the row means of a matrix? What about the column means? Verify your answer using the following matrix.

   ```r
   X <- matrix(1:15, nrow=3)
   ```

   2. What is the difference between `lapply`, `sapply`, and `vapply`? Run the following lines to find out.

   ```r
   l <- lapply(1:10, function(x) {x^2})
   s <- sapply(1:10, function(x) {x^2})
   v <- vapply(1:10, function(x) {x^2}, 1)
   ```

Solution:

1. The `apply` function is used to apply a function over the rows or columns of a matrix. Specify either 1 or 2 in the second argument for rows or columns, respectively.

   ```r
   apply(X, 1, mean)
   ```
   
   ```r
   #> [1] 7 8 9
   ```

   ```r
   apply(X, 2, mean)
   ```
   
   ```r
   #> [1] 2 5 8 11 14
   ```

2. `lapply` returns a list. `sapply` performs the same operation as `lapply`, but tries to return the same type as the input, which can save you some time. So since `1:10` in the above example is a vector, you get a vector back. `vapply` is also similar, except that it enforces a consistency in the function output type (notice in the documentation that the `FUN.VALUE` argument is not optional). This can help with optimization and error checking.

Data structures

Problem:

Let’s explore the relationship between lists and data frames. First create a data frame and a list:

```r
my_df <- data.frame(a = 1:5, b = 6:10)
my_list <- list(a = 1:5, b = 6:10)
```
1. Try and access the first column of my_df and my_list by using the dollar sign notation (my_df$a) and the double bracket notation (my_df[['a']]) or my_df[[1]]). Now try it using the row-column index notation (my_df[, 'a'] or my_df[,1]). What happens?

2. Try and access the first row of my_df and my_list by using the row-column index notation (my_df[1,]). What happens?

3. Try and create another data frame and another list, again with two columns but where each column has a differing number of elements. What happens?

What do the above exercises tell you about the relationship between data frames and lists? Which is a larger class of objects?

Now explore how atomic vectors (those formed with the combine function c()) differ from lists. Try the following and see what happens:

1. Combine values of two different types into a vector. Types include booleans (TRUE, FALSE), integers (1), numerics (1.5), and strings (stanford).
2. Combine several vectors into one, or combine a vector with a single value (which is really a vector of length 1).

Solution:

Lists are one of the most flexible data structures in R, as it can contain objects of different types (similar to a list in Python). You can even store functions or other lists inside a list.

A data frame is really a list in which the columns are the list elements, and all of the columns are constrained to have the same number of elements. This is why you can access data frame columns using the list indexing $ and [[]], because a data frame is really a list. However, since each column in a data frame has the same number of rows (i.e. it’s not ragged), you can also access elements by row-column index. This won’t work in a general list.

Atomic vectors are homogeneous, meaning all of the elements must be the same type. (The elements also have to be atomic type, which means you can’t store functions in a vector.) If you try and combine elements of two differing types into a vector, one of them will be coerced so that the types agree. Atomic vectors are also flat, meaning you can’t store vectors inside other vectors.

The page adv-r.had.co.nz/Data-structures.html has some more information on data structures in R.