Manuel J. A. Eugster

Programming R

Chapter 6: Vectorizing Computations

Last modification on June 11, 2012

Draft of the R programming book I always wanted to read
http://mjaeugster.github.com/progr

6 Vectorizing Computations

Programming without explicit loops.

The main literature for this section is:

- *R Language Definition* by R Core Team (2012)
- *Software for Data Analysis: Programming with R* by Chambers (2008)
- *The Art of R Programming* by Matloff (2011)

6.1 Vectorization

The idea of **vectorizing** comes from the contrast between a single expression applied to one or more R vectors, compared to a loop that computes corresponding single values.

For example, the addition of elements of two vectors of equal lengths,

```
> set.seed(1234)
> x <- rnorm(5)
> y <- runif(5)
```

can be done in R by making use of the implemented vectorized addition operator:

```
> x + y

[1] -0.5135  0.8224  1.3672 -1.4223  0.7214
```

One can think of an **implicit loop** iterating over the vectors' elements. We can receive the same result using an **explicit loop**:

```
> z <- numeric(length(x))
> for ( i in seq(along = x) ) {
+   z[i] <- x[i] + y[i]
+ }
> z

[1] -0.5135  0.8224  1.3672 -1.4223  0.7214
```

However, the vectorized statement is more compact, more readable and faster (for “big” vectors).

```
> set.seed(1234)
> x <- rnorm(10^6)
> y <- runif(10^6)
>
> ## Implicit loop:
> system.time(x + y)

   user  system elapsed
   0.02    0.00    0.01


>
+ ## Explicit loop:
> z <- numeric(length(x))
> system.time(for ( i in seq(along = x) ) z[i] <- x[i] + y[i])

   user  system elapsed
   3.31    0.02    3.33
```

The implicit loop is much faster then the explicit loop. Even through R internally loops over the two vectors, this is done in native machine code—which results in this speedup. **Therefore, whenever it is possible to use R’s vectorization, use it!**

One note of caution: be aware of the recycling rules. If one tries to operate on two vectors with different number of elements, then the shortest is recycled to length of longest. Only if the length of the longer vector is not a multiple of the shorter one, a warning is given.

```
> c(1, 2, 3) + 1:6

[1] 2 4 6 5 7 9

> c(1, 2, 3) + 1:5

Warning message: longer object length is not a multiple of shorter object length
[1] 2 4 6 5 7
```

6.2 Implicit loops

Here, we discuss some examples for implicit loops.

Operators. All operators `+`, `-`, etc. (see `?Arithmetic`, `?Comparison`) are vectorized, i.e., implicit loops iterate over the two vectors.

For example,

```
> set.seed(1234)
> x <- sample(100, 5)
> ((x %% 2) == 0)

[1]  TRUE  TRUE  TRUE FALSE FALSE
```

returns `TRUE` if an element of the vector is even, otherwise `FALSE`. On can think of two implicit loops, the first computes the `x %% 2` and the second `.  == 0`.

Statistical measures. Most of the common statistical measures—`mean()`, `median()`, `sd()`, etc.—imply an implicit loop. For example, the arithmetic mean of a vector x_i $(i = 1, \ldots, n)$ is defined as

$$\frac{1}{n}\sum_{i=1}^{n} x_i.$$

The sum sign means that we have to loop over the elements to add them. The R function does this implicitly,

```
> mean(x)

[1] 55.6
```

Another way with an implicit loop is,

```
> sum(x) / length(x)

[1] 55.6
```

where `sum()` loops over the elements to add them. The version with the explicit loop is:

```
> z <- 0
> for ( i in seq(along = x) ) {
+   z <- z + x[i]
+ }
> z <- z / length(x)
> z

[1] 55.6
```

In case we have not only a vector but a `data.frame` we may want to compute column means. This would imply two loops, one over the columns, and for each column one loop over the rows. R provides a function with these two implicit loops, `colMeans()`.

```
> data("cars", package = "datasets")
> str(cars)

'data.frame': 50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...

> colMeans(cars)

speed  dist
15.40 42.98
```

`rowMeans()` is the equivalent for computing row means; `rowSums()` and `colSums` for computing the row and column sums.

Subsets. Extractions (or replacements) of subsets, i.e., expressions of the form `x[i]`, contain implicit loops as well.

For example,

```
> set.seed(1234)
> x <- letters[1:10]
> ind <- sample(100, 10)
>
> x[ind %% 2 == 0]

[1] "a" "b" "c" "h"
```

creates the subsets of letters where the corresponding element of the `ind` vector is even. Behind the scenes something like the following explicit loop is happening:

```
> z <- character()
> for ( i in seq(along = x) ) {
+   if ( ind[i] %% 2 == 0 ) {
+     z <- c(z, x[i])
+   }
+ }
> z

[1] "a" "b" "c" "h"
```

A similar result can be obtained using `ifelse()` for conditional element selection.

```
> ifelse(test = ind %% 2 == 0, yes = x, no = NA)

 [1] "a" "b" "c" NA  NA  NA  NA  "h" NA  NA
```

The result is a vector with the same length as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Matrix computations. This concept expands to two- (`matrix`) and higher- (`array`) dimensional structures as well. Obvious examples are matrix multiplication, transposition and subsetting.

For example, the transposition of a $n \times m$ matrix,

```
> set.seed(1234)
> A <- matrix(sample(10), ncol = 2)
> A

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

> t(A)
```

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

would consist of two explicit loops over each row of each column:

```
> n <- nrow(A)
> m <- ncol(A)
> At <- matrix(NA, nrow = m, ncol = n)
> for ( i in seq(length = n) ) {
+   for ( j in seq(length = m) ) {
+     At[j, i] <- A[i, j]
+   }
+ }
> At

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

6.3 The `apply`-family

One common task is to repeatedly call a function for all of the elements of a vector, or for all of the rows or columns of a matrix or `data.frame` (in fact, for all of the dimensions of an array) and to collect the results. The `apply`-family provides functions for this task.

The main reasons to prefer using the `apply`-family to an explicit loop are:

1. The computation becomes more compact and clearer. (often true)
2. The computation is easily parallelizable. (always true; see the `plyr` package)
3. The computation should run faster. (problematic statement; see the discussion by Chambers, 2008, p. 213)

Apply a function over a list or vector. Let us start with a simple (and artificial) example. Given is the following function

```
> ab <- function(x) {
+   if ( x < 10 ) {
+     "a"
+   } else {
+     "b"
+   }
+ }
```

which returns the character "a" if the argument x is smaller than 10, and "b" otherwise.

```
> ab(4)

[1] "a"

> ab(20)

[1] "b"
```

Because of the nature of the if-else control structure, this function cannot be correctly applied to a vector:

```
> ab(5:15)

Warning message: the condition has length > 1 and only the first element will be used
[1] "a"
```

Only the first element of the vector will be used. In order to apply the function to each element of the vector, we have to use one of the functions sapply() or lapply():

```
> sapply(5:15, ab)

 [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "b"
```

The result is then a vector with the same length as the input vector. Each element is the result of applying the function to the corresponding element in the input vector. Note that lapply() will always return a list, whereas sapply() tries to simplify the result to a vector, matrix or higher dimensional array.

A (slightly) more realistic example is that a list with results from different runs of an experiment are given:

```
> set.seed(1234)
> l <- list(exp1 = rnorm(100),
+           exp2 = rnorm(100),
+           exp3 = rnorm(100))
```

If we now want to compute the mean for each element, we can do this, for example, by hand:

```
> c(mean(l$exp1),
+   mean(l$exp2),
+   mean(l$exp3))

[1] -0.15676  0.04124  0.15460
```

This works as long as the list has not too many elements. If this is the case, we can write an explicit loop:

```
> ms <- numeric(length = length(l))
> for ( i in seq(along = l) ) {
+   ms[i] <- mean(l[[i]])
+ }
> ms

[1] -0.15676  0.04124  0.15460
```

This works. It is, however, a lot of (error-prone) code for such a simple task. Using the `apply`-family simplifies this task a lot—it reduces to:

```
> sapply(l, mean)

    exp1     exp2     exp3 
-0.15676  0.04124  0.15460 
```

We will see the difference between `sapply()` and `lapply()` when we compute a function with a vector as result; for example, the `quantile()` function. `lapply()`,

```
> lapply(l, quantile)

$exp1
     0%      25%      50%      75%     100% 
```

```
-2.3457 -0.8953 -0.3846  0.4712  2.5490

$exp2
     0%      25%      50%      75%     100%
-2.8558 -0.5593  0.0328  0.6276  3.0438

$exp3
     0%      25%      50%      75%     100%
-3.2332 -0.3779  0.2779  0.6823  2.9191
```

returns a list of quantiles, whereas `sapply()`,

```
> sapply(l, quantile)


        exp1    exp2    exp3
0%   -2.3457 -2.8558 -3.2332
25%  -0.8953 -0.5593 -0.3779
50%  -0.3846  0.0328  0.2779
75%   0.4712  0.6276  0.6823
100%  2.5490  3.0438  2.9191
```

simplifies the list of quantiles to a matrix.

See also: `mapply()` is a multivariate version of `sapply()`; and `replicate()` for repeated evaluation of expressions.

Apply a function over a matrix, an array, or a `data.frame`. The function `apply()` allows to call a function for each dimension of a matrix, an array, or a `data.frame`.

If we, for example, want to find the maximum element for each row and column of the `cars` data set, we can write two explicit loops or use `apply()`:

```
> data("cars", package = "datasets")
>
> apply(cars, 1, which.max)

 [1] 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

> apply(cars, 2, which.max)

speed  dist
   50    49
```

The MARGIN argument gives the subscripts which the function will be applied over; e.g., 1 means the first dimension (i.e., rows), 2 means the second dimension (i.e., columns), c(1, 2) means rows and columns (i.e., cells).

Note that data.frames are in fact lists of equal-length vectors (one list element for each column). Therefore, lapply() will work as well and apply a function on each column of a data.frame.

The plyr package. The plyr package by H. Wickham provides a sound generalization of the apply-family with parallelization.

6.4 Anonymous functions

A important concept in combination with the apply-family, is the concept of anonymous functions. An anonymous function is a function object which is not assigned to an identifier.

For example,

```
> function(x) {
+   x + 1
+ }

function(x) {
  x + 1
}
```

creates a "function object which has no name". In this case, the object somehow gets lost and is not usable. Nevertheless, this concept is useful for defining functions which are only used for one apply-family function call.

For example, the function ab() we defined above. Imagine we need this function only once, therefore we can define this function as an anonymous function directly in the sapply() call:

```
> sapply(5:15,
+        function(x) {
+          if ( x < 10 ) {
+            "a"
+          } else {
+            "b"
+          }
+        })
```

```
 [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "b"
```

6.5 apply-family gems

The do.call(, lapply()) gem. It is very common (at least in my way of programming) that I want to combine the return values of individual `lapply()`-iterations into one “big” object.

For example, the return values of the individual iterations are `data.frame`s,

```
> ds <- lapply(1:10, function(i) data.frame(a = runif(4), b = gl(2, 2)))
> str(ds, 1)

List of 10
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
 $ :'data.frame': 4 obs. of  2 variables:
```

and we want to combine them into one big `data.frame`. In such cases, the `do.call(...)` call is very handy.

```
> d <- do.call(rbind, ds)
> str(d)

'data.frame': 40 obs. of  2 variables:
 $ a: num  0.281 0.174 0.17 0.561 0.429 ...
 $ b: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
```

The first argument of the `do.call()` function is the function to be called, and the second argument is a list of arguments to the function call. Note that `sapply()` does not return the wanted result.

The `lapply(, "[[", )` gem. If you want to access only specific parts of a list's elments, you can utilize the fact that subset operators are functions.

For example, to get the column `b` of every `data.frame` in the above list, we can use:

```
> bs <- lapply(ds, "[[", "b")
> str(bs)

List of 10
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
 $ : Factor w/ 2 levels "1","2": 1 1 2 2
```

6.6 Functional programming

The foundations of "vectorizing" arise from functional programming. This is a programming paradigm that treats computation as the evaluation of functions. In contrast to imperative programming which treats computation in terms of a program state and statements that change the program state (see, e.g., Wikipedia, 2012a).

One important concept is the concept of **higher-order functions**. This is a function that does at least one of the following (cf. Wikipedia, 2012b):

- take one or more functions as an input
- output a function

Therefore, the functions of the `apply`-family are higher-order functions. In contrast, all other functions (like `mean()`) are so-called first order functions.

R supports—as many other languages—multiple programming paradigms, and, among others, the functional programming paradigm. In this sense, R provides further common higher-order functions; see `?Map`.

Bibliography

John Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008. ISBN 9780387759357.

Norman Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, 2011. ISBN 1593273843.

R Core Team. *The R language definition*, 2012. URL http://cran.r-project.org/doc/manuals/R-lang.html.

Wikipedia. Functional programming. Wikipedia entry, accessed on 2012-06-04, 2012a. URL http://en.wikipedia.org/wiki/Functional_programming.

Wikipedia. Higher-order function. Wikipedia entry, accessed on 2012-06-04, 2012b. URL http://en.wikipedia.org/wiki/Higher-order_function.