R Help Desk

How Can I Avoid This Loop or Make It Faster?

by Uwe Ligges and John Fox

Introduction

There are some circumstances in which optimized and efficient code is desirable: in functions that are frequently used, in functions that are made available to the public (e.g., in a package), in simulations taking a considerable amount of time, etc. There are other circumstances in which code should *not* be optimized with respect to speed — if the performance is already satisfactory. For example, in order to save a few seconds or minutes of CPU time, you do not want to spend a few hours of programming, and you do not want to break code or introduce bugs applying optimization patches to properly working code.

A principal rule is: Do not optimize unless you really need optimized code! Some more thoughts about this rule are given by Hyde (2006), for example. A second rule in R is: When you write a function from scratch, do it the vectorized way initially. If you do, then most of the time there will be no need to optimize later on.

If you really need to optimize, measure the speed of your code rather than guessing it. How to *profile* R code in order to detect the bottlenecks is described in Venables (2001), R Development Core Team (2008a), and the help page `?Rprof`. The CRAN packages **proftools** (Tierney, 2007) and **profr** (Wickham, 2008) provide sets of more extensive profiling tools.

The convenient function `system.time()` (used later in this article) simply measures the time of the command given as its argument.[1] The returned value consists of user time (CPU time R needs for calculations), system time (time the system is using for processing requests, e.g., for handling files), total time (how long it really took to process the command) and — depending on the operating system in use — two further elements.

Readability and clarity of the code is another topic in the area of optimized code that has to be considered, because readable code is more maintainable, and users (as well as the author) can easily see what is going on in a particular piece of code.

In the next section, we focus on vectorization to optimize code both for speed and for readability. We describe the use of the family of `*apply` functions, which enable us to write condensed but clear code. Some of those functions can even make the code perform faster. How to avoid mistakes when writing loops and how to measure the speed of code is described in a subsequent section.

Vectorization!

R is an interpreted language, i.e., code is parsed and evaluated at runtime. Therefore there is a speed issue which can be addressed by writing vectorized code (which is executed vector-wise) rather than using loops, if the problem can be vectorized. Loops are not necessarily bad, however, if they are used in the right way — and if some basic rules are heeded: see the section below on loops.

Many vector-wise operations are obvious. Nobody would want to replace the common component-wise operators for vectors or matrices (such as `+`, `-`, `*`, ...), matrix multiplication (`%*%`), and extremely handy vectorized functions such as `crossprod()` and `outer()` by loops. Note that there are also very efficient functions available for calculating sums and means for certain dimensions in arrays or matrices: `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()`.

If vectorization is not as obvious as in the cases mentioned above, the functions in the 'apply' family, named `[s,l,m,t]apply`, are provided to apply another function to the elements/dimensions of objects. These 'apply' functions provide a compact syntax for sometimes rather complex tasks that is more readable and faster than poorly written loops.

Matrices and arrays: `apply()`

The function `apply()` is used to work vector-wise on matrices or arrays. Appropriate functions can be applied to the columns or rows of a matrix or array without explicitly writing code for a loop. Before reading further in this article, type `?apply` and read the whole help page, particularly the sections 'Usage', 'Arguments', and 'Examples'.

As an example, let us construct a 5×4 matrix `X` from some random numbers (following a normal distribution with $\mu = 0, \sigma = 1$) and apply the function `max()` column-wise to `X`. The result will be a vector of the maxima of the columns:

```
R> (X <- matrix(rnorm(20), nrow = 5, ncol = 4))
R> apply(X, 2, max)
```

Dataframes, lists and vectors: `lapply()` and `sapply()`

Using the function `lapply()` (*l* because the value returned is a *l*ist), another appropriate function can be quickly applied element-wise to other objects, for example, dataframes, lists, or simply vectors. The resulting list has as many elements as the original object to which the function is applied.

[1] Timings in this article have been measured on the following platform: AMD Athlon 64 X2 Dual Core 3800+ (2 GHz), 2 Gb RAM, Windows XP Professional SP2 (32-bit), using an optimized 'Rblas.dll' linked against ATLAS as available from CRAN.

Analogously, the function `sapply()` (*s* for *s*implify) works like `lapply()` with the exception that it tries to simplify the value it returns. This means, for example, that if the resulting object is a list containing just vectors of length one, the result simplifies to a vector (or a matrix, if the list contains vectors of equal lengths). If `sapply()` cannot simplify the result, it returns the same list as `lapply()`.

A frequently used R idiom: Suppose that you want to extract the *i*-th columns of several matrices that are contained in a list `L`. To set up an example, we construct a list `L` containing two matrices `A` and `B`:

```
R> A <- matrix(1:4, 2, 2)
R> B <- matrix(5:10, 2, 3)
R> (L <- list(A, B))
    [[1]]
        [,1] [,2]
    [1,]    1    3
    [2,]    2    4

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

The next call can be read as follows: 'Apply the function `[()` to all elements of `L` as the first argument, omit the second argument, and specify 2 as the third argument. Finally return the result in the form of a list.' The command returns the second columns of both matrices in the form of a list:

```
R> lapply(L, "[", , 2)
    [[1]]
    [1] 3 4

    [[2]]
    [1] 7 8
```

The same result can be achieved by specifying an anonymous function, as in:

```
R> sapply(L, function(x) x[ , 2])
        [,1] [,2]
    [1,]    3    7
    [2,]    4    8
```

where the elements of `L` are passed separately as `x` in the argument of the anonymous function given as the second argument in the `lapply()` call. Because all matrices in `L` contain equal numbers of rows, the call returns a matrix consisting of the second columns of all the matrices in `L`.

Vectorization via `mapply()` and `Vectorize()`

The `mapply()` function (*m* for *m*ultivariate) can simultaneously vectorize several arguments to a function that does not normally take vector arguments. Consider the `integrate()` function, which approximates definite integrals by adaptive quadrature, and which is designed to compute a single integral. The following command, for example, integrates the standard-normal density function from -1.96 to 1.96:

```
R> integrate(dnorm, lower=-1.96, upper=1.96)
   0.9500042 with absolute error < 1.0e-11
```

`integrate()` returns an object, the first element of which, named `"value"`, contains the value of the integral. This is an artificial example because normal integrals can be calculated more directly with the vectorized `pnorm()` function:

```
> pnorm(1.96) - pnorm(-1.96)
[1] 0.9500042
```

`mapply()` permits us to compute several normal integrals simultaneously:

```
R> (lo <- c(-Inf, -3:3))
   [1] -Inf   -3   -2   -1    0    1    2    3
R> (hi <- c(-3:3, Inf))
   [1]  -3  -2  -1   0   1   2   3 Inf

R> (P <- mapply(function(lo, hi)
 +   integrate(dnorm, lo, hi)$value, lo, hi))
   [1] 0.001349899 0.021400234 0.135905122
   [4] 0.341344746 0.341344746 0.135905122
   [7] 0.021400234 0.001349899

R> sum(P)
   [1] 1
```

`vectorize()` takes a function as its initial argument and returns a vectorized version of the function. For example, to vectorize `integrate()`:

```
R> Integrate <- Vectorize(
 +   function(fn, lower, upper)
 +     integrate(fn, lower, upper)$value,
 +   vectorize.args=c("lower", "upper")
 + )
```

Then

```
R> Integrate(dnorm, lower=lo, upper=hi)
```

produces the same result as the call to `mapply()` above.

Optimized BLAS for vectorized code

If vector and matrix operations (such as multiplication, inversion, decomposition) are applied to very large matrices and vectors, optimized BLAS (Basic Linear Algebra Subprograms) libraries can be used in order to increase the speed of execution dramatically, because such libraries make use of the specific architecture of the CPU (optimally using caches, pipelines, internal commands and units of a CPU). A well known optimized BLAS is ATLAS (Automatically Tuned Linear Algebra Software, `http://math-atlas.sourceforge.net/`, Whaley and Petitet, 2005). How to link R against ATLAS, for example, is discussed in R Development Core Team (2008b).

Windows users can simply obtain precompiled binary versions of the file 'Rblas.dll', linked against ATLAS for various CPUs, from the directory

'/bin/windows/contrib/ATLAS/' on their favourite CRAN mirror. All that is necessary is to replace the standard file 'Rblas.dll' in the 'bin' folder of the R installation with the file downloaded from CRAN. In particular, it is not necessary to recompile R to use the optimized 'Rblas.dll'.

Loops!

Many comments about R state that using loops is a particularly bad idea. This is not necessarily true. In certain cases, it is difficult to write vectorized code, or vectorized code may consume a huge amount of memory. Also note that it is in many instances much better to solve a problem with a loop than to use recursive function calls.

Some rules for writing loops should be heeded, however:

Initialize new objects to full length before the loop, rather than increasing their size within the loop.

If an element is to be assigned into an object in each iteration of a loop, and if the final length of that object is known before the loop starts, then the object should be initialized to full length prior to the loop. Otherwise, memory has to be allocated and data has to be copied in each iteration of the loop, which can take a considerable amount of time.

To initialize objects we can use functions such as

- `logical()`, `integer()`, `numeric()`, `complex()`, and `character()` for vectors of different modes, as well as the more general function `vector()`;
- `matrix()` and `array()`.

Consider the following example. We write three functions, `time1()`, `time2()`, and `time3()`, each assigning values element-wise into an object: For $i = 1, \ldots, n$, the value i^2 will be written into the i-th element of vector `a`. In function `time1()`, `a` will not be initialized to full length (very bad practice, but we see it repeatedly: `a <- NULL`):

```
R> time1 <- function(n){
 +     a <- NULL
 +     for(i in 1:n) a <- c(a, i^2)
 +     a
 + }
R> system.time(time1(30000))
   user  system elapsed
   5.11    0.01    5.13
```

In function `time2()`, `a` will be initialized to full length [`a <- numeric(n)`]:

```
R> time2 <- function(n){
 +     a <- numeric(n)
 +     for(i in 1:n) a[i] <- i^2
 +     a
 + }
R> system.time(time2(30000))
   user  system elapsed
   0.22    0.00    0.22
```

In function `time3()`, `a` will be created by a vectorwise operation without a loop.

```
R> time3 <- function(n){
 +     a <- (1:n)^2
 +     a
 + }
R> system.time(time3(30000))
   user  system elapsed
      0       0       0
```

What we see is that

- it makes sense to measure and to think about speed;
- functions of similar length of code and with the same results can vary in speed — drastically;
- the fastest way is to use a vectorized approach [as in `time3()`]; and
- if a vectorized approach does not work, remember to initialize objects to full length as in `time2()`, which was in our example more than 20 times faster than the approach in `time1()`.

It is always advisable to initialize objects to the right length, if possible. The relative advantage of doing so, however, depends on how much computational time is spent in each loop iteration. We invite readers to try the following code (which pertains to an example that we develop below):

```
R> system.time({
 +   matrices <- vector(mode="list", length=10000)
 +   for (i in 1:10000)
 +     matrices[[i]] <-
 +       matrix(rnorm(10000), 100, 100)
 + })

R> system.time({
 +   matrices <- list()
 +   for (i in 1:10000)
 +     matrices[[i]] <-
 +       matrix(rnorm(10000), 100, 100)
 + })
```

Notice, however, that if you deliberately build up the object as you go along, it will slow things down a great deal, as the entire object will be copied at every step. Compare both of the above with the following:

```
R> system.time({
 +   matrices <- list()
 +   for (i in 1:1000)
 +     matrices <- c(matrices,
 +      list(matrix(rnorm(10000), 100, 100)))
 + })
```

Do not do things in a loop that can be done outside the loop.

It does not make sense, for example, to check for the validity of objects within a loop if checking can be applied outside, perhaps even vectorized.

It also does not make sense to apply the same calculations several times, particularly not n times within a loop, if they just have to be performed one time.

Consider the following example where we want to apply a function [here `sin()`] to $i = 1, \ldots, n$ and multiply the results by 2π. Let us imagine that this function cannot work on vectors [although `sin()` does work on vectors, of course!], so that we need to use a loop:

```
R> time4 <- function(n){
 +     a <- numeric(n)
 +     for(i in 1:n)
 +         a[i] <- 2 * pi * sin(i)
 +     a
 + }
R> system.time(time4(100000))
   user  system elapsed
   0.75    0.00    0.75

R> time5 <- function(n){
 +     a <- numeric(n)
 +     for(i in 1:n)
 +         a[i] <- sin(i)
 +     2 * pi * a
 + }
R> system.time(time5(100000))
   user  system elapsed
   0.50    0.00    0.50
```

Again, we can reduce the amount of CPU time by heeding some simple rules. One of the reasons for the performance gain is that `2*pi` can be calculated just once [as in `time5()`]; there is no need to calculate it $n = 100000$ times [as in the example in `time4()`].

Do not avoid loops simply for the sake of avoiding loops.

Some time ago, a question was posted to the *R-help* email list asking how to sum a large number of matrices in a list. To simulate this situation, we create a list of 10000 100×100 matrices containing random-normal numbers:

```
R> matrices <- vector(mode="list", length=10000)
R> for (i in seq_along(matrices))
 +     matrices[[i]] <-
 +         matrix(rnorm(10000), 100, 100)
```

One suggestion was to use a loop to sum the matrices, as follows, producing, we claim, simple, straightforward code:

```
R> system.time({
 +     S <- matrix(0, 100, 100)
 +     for (M in matrices)
 +         S <- S + M
 + })
    user  system elapsed
    1.22    0.08    1.30
```

In response, someone else suggested the following 'cleverer' solution, which avoids the loop:

```
R> system.time(S <- apply(array(unlist(matrices),
 +     dim = c(100, 100, 10000)), 1:2, sum))
Error: cannot allocate vector of size 762.9 Mb
```

Not only does this solution fail for a problem of this magnitude on the system on which we tried it (a 32-bit system, hence limited to 2Gb for the process), but it is slower on smaller problems. We invite the reader to redo this problem with 10000 10×10 matrices, for example.

A final note on this problem:

```
R> S <- rowSums(array(unlist(matrices),
 +     dim = c(10, 10, 10000)), dims = 2)
```

is approximately as fast as the loop for the smaller version of the problem but fails on the larger one.

The lesson: Avoid loops to produce clearer and possibly more efficient code, not simply to avoid loops.

Summary

To answer the frequently asked question, 'How can I avoid this loop or make it faster?': Try to use simple vectorized operations; use the family of apply functions if appropriate; initialize objects to full length when using loops; and do not repeat calculations many times if performing them just once is sufficient.

Measure execution time before making changes to code, and only make changes if the efficiency gain really matters. It is better to have readable code that is free of bugs than to waste hours optimizing code to gain a fraction of a second. Sometimes, in fact, a loop will provide a clear and efficient solution to a problem (considering both time and memory use).

Acknowledgment

We would like to thank Bill Venables for his many helpful suggestions.

Bibliography

R. Hyde. The fallacy of premature optimization. *Ubiquity*, 7(24), 2006. URL `http://www.acm.org/ubiquity`.

R Development Core Team. *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2008a. URL `http://www.R-project.org`.

R Development Core Team. *R Installation and Administration*. R Foundation for Statistical Computing, Vienna, Austria, 2008b. URL `http://www.R-project.org`.

L. Tierney. *proftools: Profile Output Processing Tools for R*, 2007. R package version 0.0-2.