Notes on Vectorization

Robert McDonald

June 19, 2018

Contents

1 Introduction

Many of the pricing functions in the *derivmkts* package are vectorized. This document summarizes some of the issues that arose when implementing vectorization.

2 Vectorization

Where possible, I have tried to make sure that the pricing functions return vectors when inputs are vectors. This is automatic in many cases (for example, with the Black-Scholes formula), but there are situations in which achieving robust vectorization requires care when constructing a function. The purpose of this section is to explain the problems I encountered and the solutions I considered. Perhaps I overlooked the obvious or I am ignorant of some details of R. In either case I hope you will let me know! Otherwise, I hope this discussion is helpful to others.

2.1 What about R's Vectorize function?

R's `Vectorize` function transparently invokes `mapply` to create vectorized output. It is extraordinarily useful, but it is slow. For example if you compute 500 European call prices, automatic vectorization is about 70 times faster than the use of `Vectorize`. You can run this to see for yourself:

```
library(derivmkts)
library(microbenchmark)
S <- seq(0.5, 250, by=0.5)
bsv <- Vectorize(bscall)
microbenchmark(out1 <- bsv(S, 40, .3, .08, .25, 0))
microbenchmark(out2 <- bscall(S, 40, .3, .08, .25, 0))
all.equal(out1, out2)
```

We won't further discuss `Vectorize`, but it is useful, for example, when performing binomial pricing calculations, which are not vectorized by default.

2.2 Automatic Vectorization

```
f = function(a, b, k) a*b + k
f(3, 5, 1)

[1] 16

f(1:5, 5, 1)

[1]  6 11 16 21 26

f(1:6, 1:2, 1)

[1]  2  5  4  9  6 13
```

In this example, R automatically vectorizes the multiplication using the recycling rule. It's worth noting that the third example, in which both arguments are vectorized, but with different length vectors, is an unusual programming construct.[1] This property makes it trivial to perform what-if calculations for an option pricing formula.

2.3 Limitations of Automatic Vectorization

A problem with automatic vectorization occurs when there are conditional statements. With barrier options, for example, it is necessary to check whether the asset price is past the barrier. R's `if` statement is not vectorized, and the `ifelse` function returns output that has the dimension of the conditional.

[1] You can produce the same s output in python using the itertools module.

```
cond1 <- function(a, b, k) {
    if (a > b) {
        a*b + k
    } else {
        k
    }
}
cond1(5, 3, 1)
```

```
[1] 16
```

```
cond1(5, 7, 1)
```

```
[1] 1
```

```
cond1(3:7, 5, 1)
```

```
Warning in if (a > b) {: the condition has length > 1 and only the first element will
be used
```

```
[1] 1
```

The third invocation of `cond1` causes an error because the `if` statement is not vectorized. This can be fixed by rewriting the conditional using `ifelse`, which is vectorized. The following examples all compute correctly because if either a or b are vectors, the conditional statement is vectorized:

```
cond2 <- function(a, b, k) {
    ifelse(a > b,
           a*b + k,
           k
           )
}
cond2(5, 3, 1)
```

```
[1] 16
```

```
cond2(5, 7, 1)
```

```
[1] 1
```

```
cond2(3:7, 5, 1)
```

```
[1]  1  1  1 31 36
```

There will, however, be a problem if only k is a vector. Suppose we set $a = 5$, $b = 7$, $k = 1 : 3$. Because $a < b$, we want to produce the output $1, 2, 3$. The following example does not work as desired because neither of the variables in the conditional (a and b) are a vector. Thus the calculation is not vectorized:

```
cond2(5, 7, 1:3)
```

```
[1] 1
```

The `ifelse` function returns output with the dimension of the conditional expression, which in this case is a vector of length 1.

2.4 Three Solutions

One solution is to write the function so as to vectorize all the inputs to match the vector length of the longest input. There are at least three ways to do this.

2.4.1 Use a Booleans in Place of `ifelse`

We can create a boolean variable that is true if $a > b$. This will then control which expression is returned:

```
cond2b <- function(a, b, k) {
    agtb <- (a > b)
    agtb*(a*b + k) + (1-agtb)*k
}

cond2b(5, 3, 1)

[1] 16

cond2b(5, 7, 1)

[1] 1

cond2b(3:7, 5, 1)

[1]  1  1  1 31 36

cond2b(5, 7, 1:3)

[1] 1 2 3
```

Whether this solution works in other functions depends on the structure of the calculation and the nature of the output. In particular, if the value of a boolean controls the data structure the function returns (a vector vs a list, for example), then this solution does not work.

2.4.2 Create a Data Frame

We can enforce the recycling rule for all variables by creating a data frame consisting of the inputs and assigning the columns back to the original variables:

```
cond2c <- function(a, b, k) {
    tmp <- data.frame(a, b, k)
    for (i in names(tmp)) assign(i, tmp[[i]])
    ifelse(a > b,
           a*b + k,
           k
           )
}

cond2c(5, 3, 1)

[1] 16
```

```
cond2c(5, 7, 1)

[1] 1

cond2c(3:7, 5, 1)

[1]  1  1  1 31 36

cond2c(5, 7, 1:3)

[1] 1 2 3
```

One drawback of this solution is that we have to be careful to update the `data.frame()` definition within the function if we change the function inputs. It may be easy to overlook this when editing the function. The next solution is a more robust version of the same idea.

2.4.3 Create a Vectorization Function

A final alternative is to create a vectorization function that exploits R's functional capabilities and does not require modifications if the function definition changes. This approach can become quite complicated, but is relatively easy to understand in simple cases. We create a `vectorizeinputs()` function that creates a data frame and vectorizes all variables:

```
vectorizeinputs <- function(e) {
    ## e is the result of match.call() in the calling function
    e[[1]] <- NULL
    e <- as.data.frame(as.list(e))
    for (i in names(e)) assign(i, eval(e[[i]]),
                               envir=parent.frame())
}
```

This function assumes that `match.call()` has been invoked in the calling function. The result of that invocation is manipulated to provide information about the parameters passed to the function and used to create the data frame and pass the variables back to the calling function.

```
cond3 <- function(a, b, k) {
    vectorizeinputs(match.call())
    ifelse(a > b, a*b + k, k)
}
cond3(5, 7, 1:3)

[1] 1 2 3

cond3(3:7, 5, 1)

[1]  1  1  1 31 36

cond3(3:7, 5, 1:5)

[1]  1  2  3 34 40
```

```
cond3(k=1:5, 3:7, 5)

[1]  1  2  3 34 40
```

This approach becomes more complicated if there are implicit parameters in the function. If truly implicit, these will not be available via `match.call()`, but they can affect the solution. Here is an example:

```
cond4 <- function(a, b, k, multby2=TRUE) {
    vectorizeinputs(match.call())
    ifelse(multby2,
           2*(a*b + k),
           a*b + k
           )
}
cond4(5, 7, 1:3)

[1] 72

cond4(3:7, 5, 1)

[1] 32

cond4(3:7, 5, 1:5)

[1] 32

cond4(k=1:5, 3:7, 5)

[1] 32
```

The output is not vectorized because the implicit parameter `multby2` is implicit — it is not explicit in the function call — and therefore it is not vectorized. One way to handle this case is to rewrite the `vectorizeinputs` function to retrieve the full set of function inputs for the called function. The name of the function is available through `match.call()[[1]]`, and the function parameters are available using the `formals` function. We can then add the implicit parameters to the vectorized set of inputs. The function `vectorizeinputs2` takes this approach:

```
vectorizeinputs2 <- function(e) {
    funcname <- e[[1]]
    fvals <- formals(eval(funcname))
    fnames <- names(fvals)
    e[[1]] <- NULL
    e <- as.data.frame(as.list(e))
    implicit <- setdiff(fnames, names(e))
    if (length(implicit) > 0) e <- data.frame(e, fvals[implicit])
    for (i in names(e)) assign(i, eval(e[[i]]),
                               envir=parent.frame())
}

cond5 <- function(a, b, k, multby2=TRUE, altmult=1) {
    vectorizeinputs2(match.call())
    ifelse(multby2,
```

```
           2*(a*b + k),
           altmult*(a*b + k)
           )
}
cond5(5, 7, 1:3)

[1] 72 74 76

cond5(3:7, 5, 1)

[1] 32 42 52 62 72

cond5(3:7, 5, 1:5)

[1] 32 44 56 68 80

cond5(k=1:5, 3:7, 5)

[1] 32 44 56 68 80

cond5(k=1:5, 3:7, 5, multby2=FALSE)

[1] 16 22 28 34 40

cond5(k=1:5, 3:7, 5, multby2=FALSE, altmult=5)

[1]  80 110 140 170 200
```

2.5 Why Worry About Vectorization?

R provides looping constructs and `apply` functions. It might seem that it's not necessary to worry about vectorization. There are two reasons that I choose to vectorize where feasible.

First, I personally find vectorization convenient and transparent. I find vectorized code easier to read and it fits the way I like to work. This is an aesthetic argument.

Second, in my experience the apply functions are a real hurdle for new R users. Automatic vectorization makes it possible to perform complicated calculations in a straightforward and intuitive way.