Apply Functions

Alexander Vining

November 7, 2017

The goal of this vignette is to teach you how to use the apply function and its variants. Apply essentially acts like a for loop, applying some process or function iteratively over a set of values. It is often cited as being faster than for loops in R, though this isn't always the case. The apply function can also be more concise and easier to interpret than a for loop. Finally, regardless of the benefits, R users often use the apply function, so it is important to know how it works. I'd like you start by reading this page on functionals (apply is an example of a functional) down to (but not including) the section on paralellism. You probably won't understand everything, but it will give you a primer to what the apply function actually is; then you can come back here and go through some examples to help you understand better.

http://adv-r.had.co.nz/Functionals.html (http://adv-r.had.co.nz/Functionals.html)

Now that you've read about what the apply function is and why we use it, let's practice. The basic apply function iterates over a matrix or data frame. These iterations can be done by row, column, or both.

```
example1 <- matrix(1,nrow = 4, ncol = 4)
example1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    1    1    1
```

```
example1 <- apply(X = example1, MARGIN = c(1,2), FUN = runif) #There are three main ar
guments to apply. X is the data to be used, MARGIN indicates whether to iterate over r
ows (1), columns (2), or both (c(1,2)). FUN is the function that should be applied to
each value. By default, the current iteration of X is passed to the first argument of
FUN. Here that means for each iteration, a 1 is passed to the n argument of runif (a s
ingle value is drawn).
example1
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.9822405 0.5195005 0.5696176 0.3128406
## [2,] 0.3365324 0.9600567 0.7478967 0.9856150
## [3,] 0.2503005 0.2178113 0.4032665 0.3562903
## [4,] 0.6936951 0.8871289 0.2329949 0.3214794
```

Now lets use apply to take the mean of each column or each row:

```
apply(X = example1, MARGIN = 2, FUN = mean) #column means
```

```
## [1] 0.5656921 0.6461243 0.4884439 0.4940563
```

```
apply(example1, 1, mean) #row means
```

```
## [1] 0.5960498 0.7575252 0.3069172 0.5338246
```

We can also use apply to pass values to arguments other than the first one in a function

```
example2 <- matrix(rep(1:4, each = 100), nrow = 100, ncol = 4)
head(example2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    1    2    3    4
## [3,]    1    2    3    4
## [4,]    1    2    3    4
## [5,]    1    2    3    4
## [6,]    1    2    3    4
```

```
example2 <- apply(X = example2, MARGIN = c(1,2), FUN = function(x) rnorm(1, x)) #Here, we define a function separately that takes our iterated value as the variable x and passes it to the second argument of rnorm (the mean of the distribution)
head(example2)
```

```
##           [,1]      [,2]      [,3]     [,4]
## [1,] 1.4359749 1.7126019 2.0323671 3.559052
## [2,] 0.7340776 2.3783196 3.4037642 4.042140
## [3,] 1.7599045 2.8114743 4.0550464 5.407136
## [4,] 1.3161002 0.3884142 2.6404904 3.044459
## [5,] 2.6433129 2.1070830 0.8720783 4.453421
## [6,] 1.3933669 0.6982530 4.1263045 4.664747
```

Because the first column in our matrix was 1s, the second 2s, etc. and these are the numbers passed to the ‘mean’ argument of rnorm, when we take the mean of each column it should be close to the number the column was filled with!

```
apply(X = example2, MARGIN = 2, FUN = mean)
```

```
## [1] 1.045460 2.090961 2.843705 3.914801
```

We can use this same method to apply functions that are entirely our own to a matrix

```
arbitraryOperations <- function(a, b) { #don't worry about understanding what this fun
ction does. The point is to illustrate the we can do whatever we want!
  if (a == 0) return(0)
  out <- (b*a) + (b/a)
  while (out < 1) {
    a <- a - 0.1
    out <- out * a
  }
  out
}
example3 <- data.frame("A" = runif(10, -10, -1), "B" = runif(10, -1, 1), "C" = runif(1
0, 1, 10))
example3
```

```
##            A          B        C
## 1  -6.835180 -0.5800308 9.083719
## 2  -1.382631 -0.7091176 4.306426
## 3  -5.504763  0.9843521 1.791297
## 4  -6.968477 -0.2563225 3.028171
## 5  -1.984420 -0.4788901 6.229966
## 6  -7.033033  0.4490192 6.939546
## 7  -8.513252  0.9935723 1.150289
## 8  -9.673484  0.2225849 6.109090
## 9  -6.241974 -0.6579895 5.098529
## 10 -6.499586 -0.9538025 5.847612
```

```
apply(X = example3, MARGIN = c(1,2), FUN = arbitraryOperations, b = 0.2) #our arbitrar
yOperations function requires two arguments. The first 'a', is passed to our function
by apply and comes from X (example3). The second we tell apply what it is directly by
including the argument name and its value our call to apply.
```

```
##               A        B         C
##  [1,]  9.683566 1.864649  1.838761
##  [2,]  1.662904 1.890093  3.818289
##  [3,]  6.374212 3.727502  1.264697
##  [4,] 10.054175 1.523793  1.966796
##  [5,]  1.037351 2.114839  1.278096
##  [6,] 10.236216 1.020374  1.416729
##  [7,] 14.867707 1.393503 12.891583
##  [8,] 19.110796 2.621990  1.254556
##  [9,]  8.120492 1.290108  1.058933
## [10,]  8.781993 1.201507  1.203724
```

Of course, sometimes we want to apply loops to a data structure other than a matrix. For this we can use lapply (the l stands for list).

```
coords1 <- vector(mode = "list", length = 10) #create empty list
coords1 <- lapply(X = coords1, FUN = function(x) rnorm(n = 2, mean = 0, sd = 5)) #noti
ce our function doesn't even use its input! We could enter any list and the values in
it wouldn't matter. Do you see why?
move <- function(coords, dist, angle) {  #a simple move function
  coords[1] <- coords[1] + dist * cos(angle)
  coords[2] <- coords[2] + dist * sin(angle)
  coords
}
coords2 <- lapply(X = coords1, FUN = move, dist = rnorm(n=1, mean = 1, sd = 0.2), angl
e = runif(1,0,2*pi))
```

We can use sapply to coerce our output into the simplest possible structure (usually a vector). sapply works just like lapply, but it will turn the output into a vector if possible. Lets use this to make vectors of x and y coordinates we can use to plot our points.

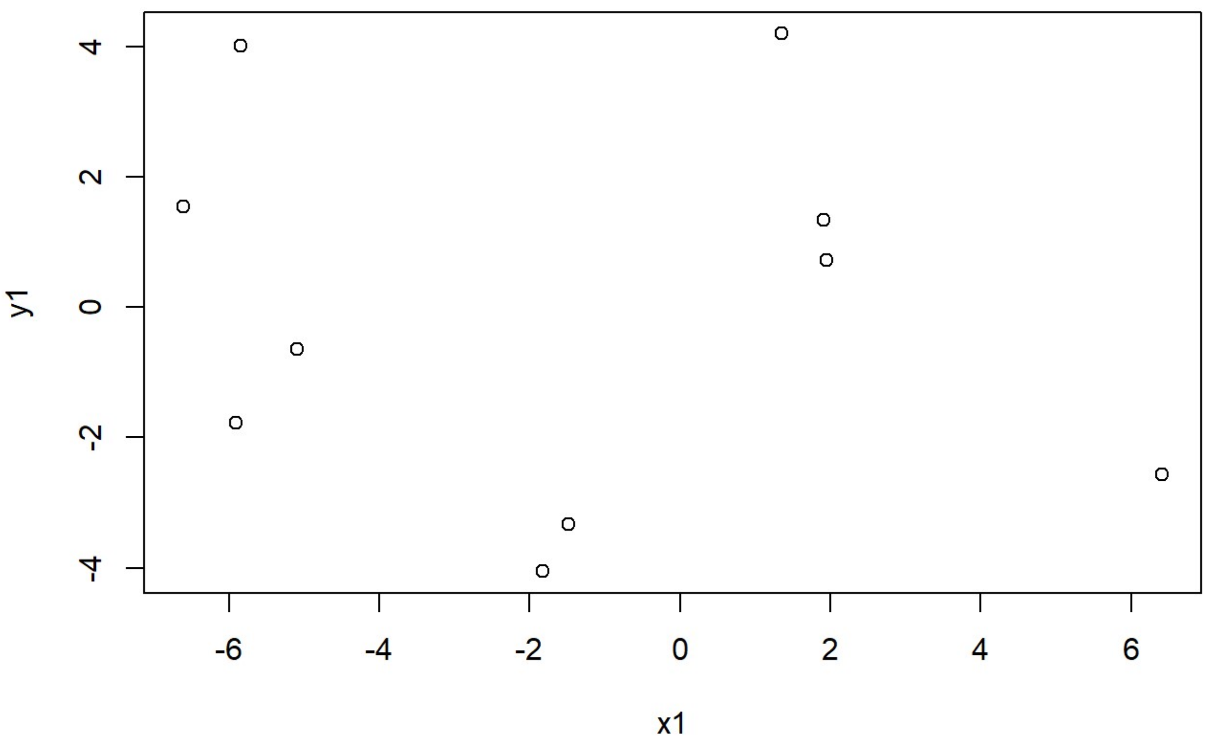
```
x1 <- sapply(coords1, FUN = function(x) x[1]) #puts all the x coordinates in one vecto
r
y1 <- sapply(coords1, FUN = function(x) x[2]) #puts all the y coordinates in one vecto
r
x1
```

```
##  [1] -5.912870 -6.607880 -5.097252  1.905447 -5.843454 -1.483329  1.356680
##  [8]  1.952872  6.411209 -1.820486
```

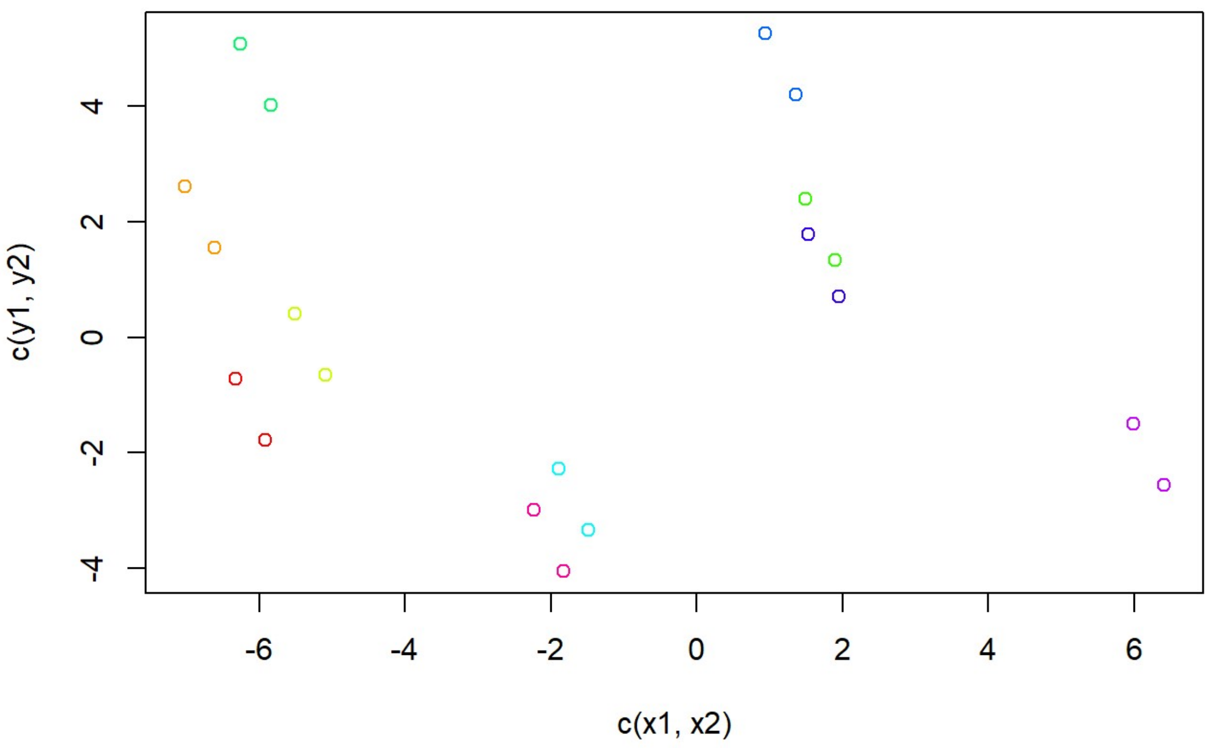
```
y1
```

```
##  [1] -1.7818757  1.5441641 -0.6466571  1.3365745  4.0130525 -3.3299233
##  [7]  4.1921340  0.7128447 -2.5606660 -4.0524981
```

```
plot(x1,y1)
```

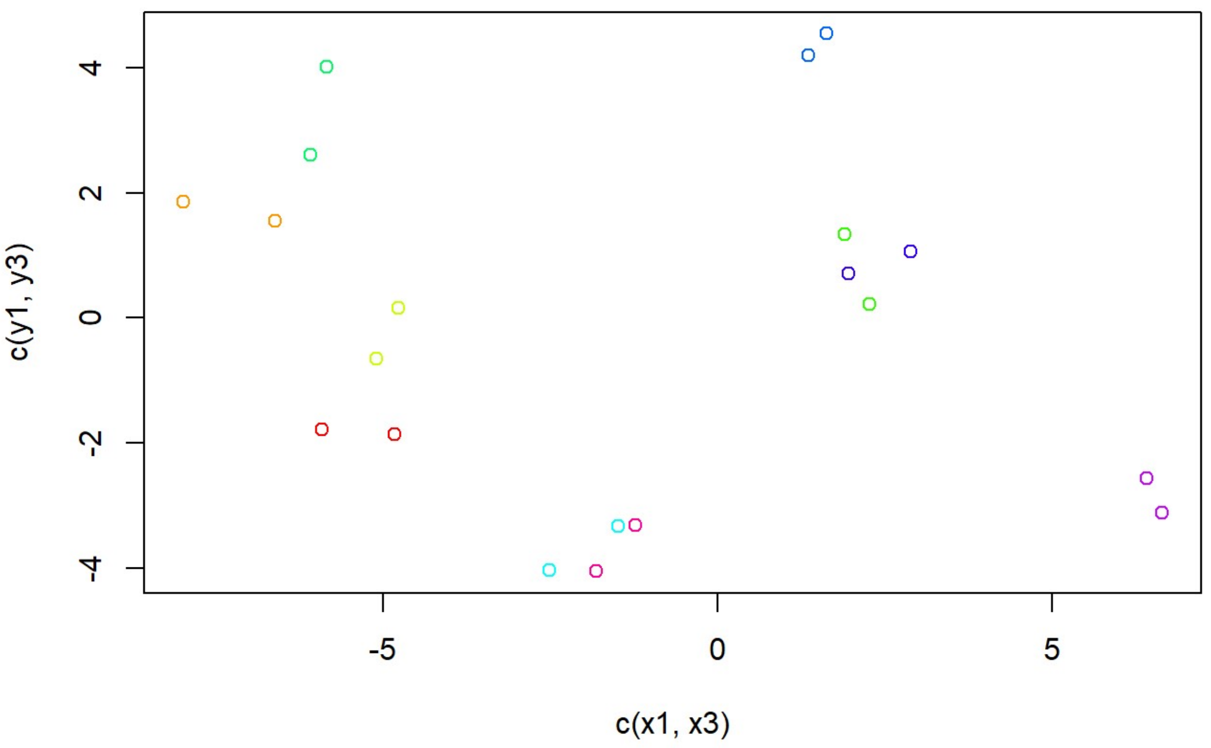


```
x2 <- sapply(coords2, FUN = function(x) x[1])
y2 <- sapply(coords2, FUN = function(x) x[2])
plot(c(x1,x2), c(y1,y2), col = rainbow(length(x1)))
```



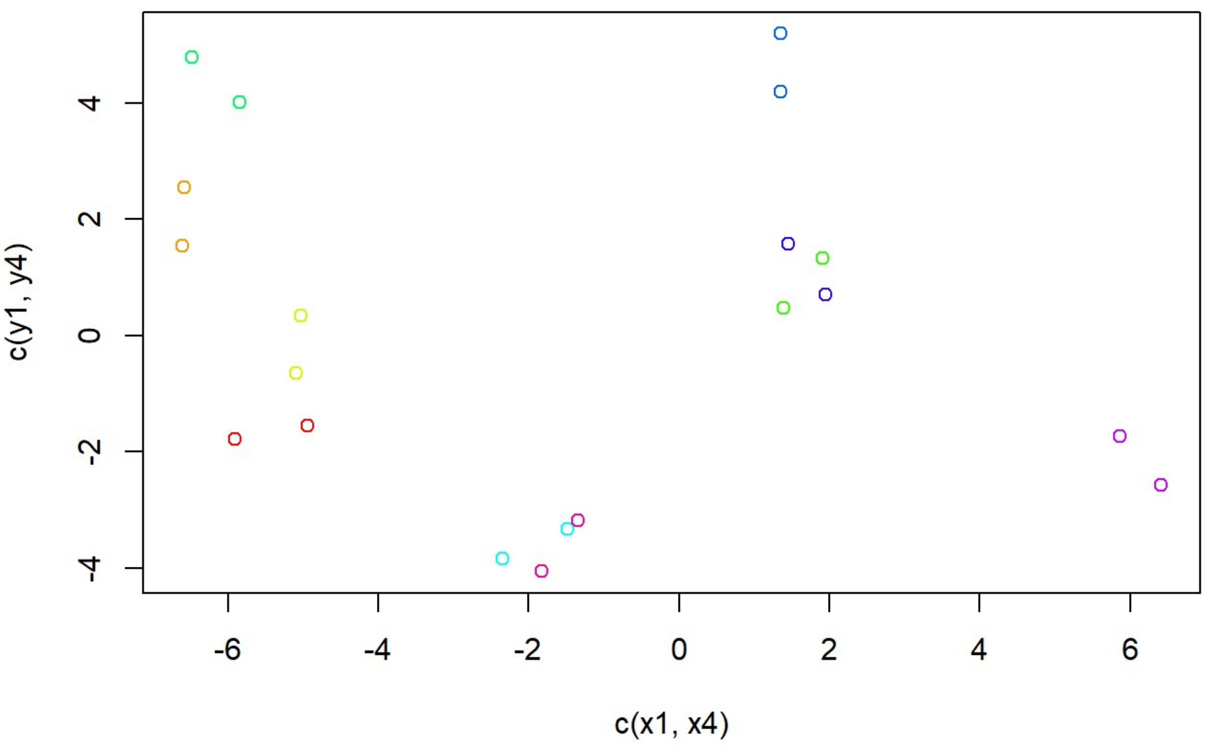
This mostly works great, but all of our agents are moving the same distance and angle! We can use mapply to have a function iterate over different values for multiple arguments (the m stands for multiple)

```
coords3 <- mapply(FUN = move, coords1, dist = rnorm(n = length(coords1), mean = 1, sd
= 0.2), angle = runif(length(coords1), 0, 2*pi), SIMPLIFY = FALSE) #Notice the additio
n of the SIMPLIFY argument; mapply, like sapply, will try to simplify the data ourput
unless we tell it not to. Also note the argument for our data is no longer X. In fact
there is no argument name for our data, any unnamed arguments are just assumed to be t
he data. As a challenge, think back to the reading you did at the start this exercise
and see if you can figure out why.
x3 <- sapply(coords3, function(x) x[1])
y3 <- sapply(coords3, function(x) x[2])
plot(c(x1,x3), c(y1,y3), col = rainbow(length(x1)))
```



Finally, let's say we want to use mapply, but there are some arguments we DON'T want to iterate. For this, we can use the MoreArgs argument. For example, lets say we want to keep the distance travelled the same and only get a random angle for each iteration

```
coords4 <- mapply(FUN = move, coords1, angle = runif(length(coords1),0,2*pi), MoreArg
s = list(dist = 1), SIMPLIFY = FALSE) #The argument MoreArgs takes a list where the na
me of each element is an argument of FUN and the value of each element is the value t
o be passed to that argument.
x4 <- sapply(coords4, function(x) x[1])
y4 <- sapply(coords4, function(x) x[2])
plot(c(x1,x4), c(y1,y4), col = rainbow(length(x1)))
```


c(y1, y4)
c(x1, x4)
-6
-4
-2
0
2
4
6
-4
-2
0
2
4