Apply and Sapply function over arrays and rows
Evaluate one Defined or Anonymous Function
Across Rows of a Matrix or Elements of an Array

Fan Wang

2022-07-23

Contents

1 Apply and Sapply

Go to the **RMD**, **R**, **PDF**, or **HTML** version of this file. Go back to fan's REconTools research support package, R4Econ examples page, PkgTestR packaging guide, or Stat4Econ course page.

- r apply matrix to function row by row
- r evaluate function on grid
- Apply a function to every row of a matrix or a data frame
- r apply
- r sapply
- sapply over matrix row by row
- function as parameters using formulas
- do

We want evaluate linear function f(x_i, y_i, ar_x, ar_y, c, d), where c and d are constants, and ar_x and ar_y are arrays, both fixed. x_i and y_i vary over each row of matrix. More specifically, we have a functions, this function takes inputs that are individual specific. We would like to evaluate this function concurrently across N individuals.

The function is such that across the N individuals, some of the function parameter inputs are the same, but others are different. If we are looking at demand for a particular product, the prices of all products enter the demand equation for each product, but the product's own price enters also in a different way.

The objective is either to just evaluate this function across N individuals, or this is a part of a nonlinear solution system.

What is the relationship between apply, lapply and vectorization? see Is the "*apply" family really not vectorized?.

1.1 Set up Input Arrays

There is a function that takes $M = Q + P$ inputs, we want to evaluate this function N times. Each time, there are M inputs, where all but Q of the M inputs, meaning P of the M inputs, are the same. In particular, $P = Q * N$.

$$M = Q + P = Q + Q * N$$

```
# it_child_count = N, the number of children
it_N_child_cnt = 5
# it_heter_param = Q, number of parameters that are
# heterogeneous across children
it_Q_hetpa_cnt = 2

# P fixed parameters, nN is N dimensional, nP is P dimensional
ar_nN_A = seq(-2, 2, length.out = it_N_child_cnt)
ar_nN_alpha = seq(0.1, 0.9, length.out = it_N_child_cnt)
ar_nP_A_alpha = c(ar_nN_A, ar_nN_alpha)

# N by Q varying parameters
mt_nN_by_nQ_A_alpha = cbind(ar_nN_A, ar_nN_alpha)

# display
kable(mt_nN_by_nQ_A_alpha) %>%
  kable_styling_fc()
```

ar_nN_A	ar_nN_alpha
-2	0.1
-1	0.3
0	0.5
1	0.7
2	0.9

1.2 Using apply

1.2.1 Named Function

First we use the apply function, we have to hard-code the arrays that are fixed for each of the N individuals. Then apply allows us to loop over the matrix that is N by Q, each row one at a time, from 1 to N.

```
# Define Implicit Function
ffi_linear_hardcode <- function(ar_A_alpha){
  # ar_A_alpha[1] is A
  # ar_A_alpha[2] is alpha

  fl_out = sum(ar_A_alpha[1]*ar_nN_A +
                 1/(ar_A_alpha[2] + 1/ar_nN_alpha))

  return(fl_out)
}
```

```
# Evaluate function row by row
ar_func_apply = apply(mt_nN_by_nQ_A_alpha, 1, ffi_linear_hardcode)
```

1.2.2 Anonymous Function

- apply over matrix

Apply with anonymous function generating a list of arrays of different lengths. In the example below, we want to drawn N sets of random uniform numbers, but for each set the number of draws we want to have is Q_i. Furthermore, we want to rescale the random uniform draws so that they all become proportions that sum u pto one for each i, but then we multply each row's values by the row specific aggregates.

The anonymous function has hard coded parameters. Using an anonymous function here allows for parameters to be provided inside the function that are shared across each looped evaluation. This is perhaps more convenient than sapply with additional parameters.

```
set.seed(1039)

# Define the number of draws each row and total amount
it_N <- 4
fl_unif_min <- 1
fl_unif_max <- 2
mt_draw_define <- cbind(sample(it_N, it_N, replace=TRUE),
                        runif(it_N, min=1, max=10))
tb_draw_define <- as_tibble(mt_draw_define) %>%
  rowid_to_column(var = "draw_group")
print(tb_draw_define)

# apply row by row, anonymous function has hard
# coded min and max
ls_ar_draws_shares_lvls =
  apply(tb_draw_define,
        1,
        function(row) {
          it_draw <- row[2]
          fl_sum <- row[3]
          ar_unif <- runif(it_draw,
                           min=fl_unif_min,
                           max=fl_unif_max)
          ar_share <- ar_unif/sum(ar_unif)
          ar_levels <- ar_share*fl_sum
          return(list(ar_share=ar_share,
                      ar_levels=ar_levels))
        })

# Show Results as list
print(ls_ar_draws_shares_lvls)
```

```
## [[1]]
## [[1]]$ar_share
## [1] 0.2783638 0.2224140 0.2797840 0.2194381
##
## [[1]]$ar_levels
## [1] 1.492414 1.192446 1.500028 1.176491
##
```

```
##
## [[2]]
## [[2]]$ar_share
## [1] 0.5052919 0.4947081
##
## [[2]]$ar_levels
## [1] 3.866528 3.785541
##
##
## [[3]]
## [[3]]$ar_share
## [1] 1
##
## [[3]]$ar_levels
##       V2
## 9.572211
##
##
## [[4]]
## [[4]]$ar_share
## [1] 0.4211426 0.2909812 0.2878762
##
## [[4]]$ar_levels
## [1] 4.051971 2.799640 2.769765
```

Above, our results is a list of lists. We can conver this to a table. If all results are scalar, would be regular table where each cell has a single scalar value.

```
# Show results as table
kable(as_tibble(do.call(rbind, ls_ar_draws_shares_lvls))) %>%
  kable_styling_fc()
```

ar_share	ar_levels
0.2783638, 0.2224140, 0.2797840, 0.2194381	1.492414, 1.192446, 1.500028, 1.176491
0.5052919, 0.4947081	3.866528, 3.785541
1	9.572211
0.4211426, 0.2909812, 0.2878762	4.051971, 2.799640, 2.769765

We will try to do the same thing as above, but now the output will be a stacked dataframe. Note that within each element of the apply row by row loop, we are generating two variables *ar_share* and *ar_levels*. We will not generate a dataframe with multiple columns, storing *ar_share*, *ar_levels* as well as information on *min*, *max*, number of draws and rescale total sum.

```
set.seed(1039)
# apply row by row, anonymous function has hard coded min and max
ls_mt_draws_shares_lvls =
  apply(tb_draw_define, 1, function(row) {

    it_draw_group <- row[1]
    it_draw <- row[2]
    fl_sum <- row[3]

    ar_unif <- runif(it_draw,
                     min=fl_unif_min,
                     max=fl_unif_max)
```

```
    ar_share <- ar_unif/sum(ar_unif)
    ar_levels <- ar_share*fl_sum

    mt_all_res <- cbind(it_draw_group, it_draw, fl_sum,
                        ar_unif, ar_share, ar_levels)
    colnames(mt_all_res) <-
      c('draw_group', 'draw_count', 'sum',
        'unif_draw', 'share', 'rescale')
    rownames(mt_all_res) <- NULL

    return(mt_all_res)
  })
mt_draws_shares_lvls_all <- do.call(rbind, ls_mt_draws_shares_lvls)
# Show Results
kable(mt_draws_shares_lvls_all) %>% kable_styling_fc()
```

draw_group	draw_count	sum	unif_draw	share	rescale
1	4	5.361378	1.125668	0.1988606	1.066167
1	4	5.361378	1.668536	0.2947638	1.580340
1	4	5.361378	1.419382	0.2507483	1.344356
1	4	5.361378	1.447001	0.2556274	1.370515
2	2	7.652069	1.484598	0.4605236	3.523959
2	2	7.652069	1.739119	0.5394764	4.128110
3	1	9.572211	1.952468	1.0000000	9.572211
4	3	9.621375	1.957931	0.3609352	3.472693
4	3	9.621375	1.926995	0.3552324	3.417824
4	3	9.621375	1.539678	0.2838324	2.730858

1.3 Using sapply

1.3.1 Named Function

- r convert matrix to list
- Convert a matrix to a list of vectors in R

Sapply allows us to not have tohard code in the A and alpha arrays. But Sapply works over List or Vector, not Matrix. So we have to convert the N by Q matrix to a N element list Now update the function with sapply.

```
ls_ar_nN_by_nQ_A_alpha = as.list(data.frame(t(mt_nN_by_nQ_A_alpha)))

# Define Implicit Function
ffi_linear_sapply <- function(ar_A_alpha, ar_A, ar_alpha){
  # ar_A_alpha[1] is A
  # ar_A_alpha[2] is alpha

  fl_out = sum(ar_A_alpha[1]*ar_nN_A +
                 1/(ar_A_alpha[2] + 1/ar_nN_alpha))

  return(fl_out)
}

# Evaluate function row by row
ar_func_sapply = sapply(ls_ar_nN_by_nQ_A_alpha, ffi_linear_sapply,
```

```
                        ar_A=ar_nN_A, ar_alpha=ar_nN_alpha)
```

1.3.2 Anonymous Function, list of arrays as output

- sapply anonymous function
- r anoymous function multiple lines

Sapply with anonymous function generating a list of arrays of different lengths. In the example below, we want to drawn N sets of random uniform numbers, but for each set the number of draws we want to have is Q_i. Furthermore, we want to rescale the random uniform draws so that they all become proportions that sum u pto one for each i.

```
it_N <- 4
fl_unif_min <- 1
fl_unif_max <- 2

# Generate using runif without anonymous function
set.seed(1039)
ls_ar_draws = sapply(seq(it_N),
                     runif,
                     min=fl_unif_min, max=fl_unif_max)
print(ls_ar_draws)
```

```
## [[1]]
## [1] 1.125668
##
## [[2]]
## [1] 1.668536 1.419382
##
## [[3]]
## [1] 1.447001 1.484598 1.739119
##
## [[4]]
## [1] 1.952468 1.957931 1.926995 1.539678
```

```
# Generate Using Anonymous Function
set.seed(1039)
ls_ar_draws_shares = sapply(seq(it_N),
                            function(n, min, max) {
                              ar_unif <- runif(n,min,max)
                              ar_share <- ar_unif/sum(ar_unif)
                              return(ar_share)
                            },
                            min=fl_unif_min, max=fl_unif_max)
# Print Share
print(ls_ar_draws_shares)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 0.5403432 0.4596568
##
## [[3]]
## [1] 0.3098027 0.3178522 0.3723451
##
```

```
## [[4]]
## [1] 0.2646671 0.2654076 0.2612141 0.2087113
```

```
# Sapply with anonymous function to check sums
sapply(seq(it_N), function(x) {sum(ls_ar_draws[[x]])})
```

```
## [1] 1.125668 3.087918 4.670717 7.377071
```

```
sapply(seq(it_N), function(x) {sum(ls_ar_draws_shares[[x]])})
```

```
## [1] 1 1 1 1
```

1.3.3 Anonymous Function, matrix as output

Below, we provide another example with sapply, we generate probabilities for discrete random variables that follow the binomial distribution. We do this for twice, with "chance of success" set at different values.

The output in this case is a matrix, where each column stores the output from a different dbinom call.

```
# First, generate the results without sapply
ar_binomprob <- matrix(c(0.1, 0.9), nrow=2, ncol=1)
# Second, generate the results with sapply
# dbinom call: dbinom(x, size, prob, log = FALSE)
# The function requires x, size, and prob.
# we provide x and size, and each element of ar_binomprob
# will be a different prob.
mt_dbinom <- sapply(ar_binomprob, dbinom, x=seq(0,4), size=4)
# Third compare results
print(paste0('binomial p=', ar_binomprob[1]))
```

```
## [1] "binomial p=0.1"
```

```
print(dbinom(seq(0,4), 4, ar_binomprob[1]))
```

```
## [1] 0.6561 0.2916 0.0486 0.0036 0.0001
```

```
print(mt_dbinom[,1])
```

```
## [1] 0.6561 0.2916 0.0486 0.0036 0.0001
```

```
print(paste0('binomial p=', ar_binomprob[2]))
```

```
## [1] "binomial p=0.9"
```

```
print(dbinom(seq(0,4), 4, ar_binomprob[2]))
```

```
## [1] 0.0001 0.0036 0.0486 0.2916 0.6561
```

```
print(mt_dbinom[,2])
```

```
## [1] 0.0001 0.0036 0.0486 0.2916 0.6561
```

1.4 Compare Results

```
# Show overall Results
mt_results <- cbind(ar_func_apply, ar_func_sapply)
colnames(mt_results) <- c('eval_lin_apply', 'eval_lin_sapply')
kable(mt_results) %>% kable_styling_fc()
```

	eval_lin_apply	eval_lin_sapply
X1	2.346356	2.346356
X2	2.094273	2.094273
X3	1.895316	1.895316
X4	1.733708	1.733708
X5	1.599477	1.599477