University of Duisburg-Essen
Faculty of Business Administration and Economics
Chair of Econometrics

A Functional Approach to (Parallelised) Monte Carlo Simulation

Advanced R for Econometricians

Final Project

Submitted to the Faculty of
Business Administration and Economics
at the
University of Duisburg-Essen

from:

Alexander Langnau, Öcal Kaptan, Sunyoung Ji

Matriculation Number:	232907, 230914, 229979
Study Path:	M.Sc. Econometircs
Reviewer:	Prof. Dr. Christoph Hanck
Secondary Reviewer:	M.Sc. Martin C. Arnold, M.Sc. Jens Klenke
Semester:	1st Semester
Graduation (est.):	Summer Term 2022
Deadline:	09. 09. 2022

Contents

List of Tables

List of Figures

1 Introduction

Monte Carlo, named after a casino in Monaco, simulates complex probabilistic events using simple random events, such as the tossing of a pair of dice to simulate the casino's overall business model. In Monte Carlo computing, a pseudo-random number generator is repeatedly called which returns a real number in [0, 1], and the results are used to generate a distribution of samples that is a fair representation of the target probability distribution under study. (**Barbu**) Monte Carlo Method is combined with programming in modern research and contributes to various studies.

Monte Carlo simulations are and will stay an important method in the tool box of any econometrician, statistican or data scientist. Since these simulations may be needed on a regular basis or are run over a complex set of functions and parameters, its time well spend to implement some tools, that allow the user to easily create a variety of different Monte Carlo studies.

This paper was the final project of the course "Advanced R for econometricians" at the chair of econometrics at university Duisburg Essen. The goal is to use a functional programming aproach to create a collection of different wrapper functions in R, that - providing a convenient interface for Monte Carlo Simulations - create a paramter grid - iterate homogenous function calls over the parameter grid - provides an informative summary of the simulation results - can be visualized by ggplot-methods - offers the possibility to use parallelised processing (using `furrr` package)

A functional programming approach is well suited to implement the different steps. The structure of this paper underlying code in general follows this approach:

In chapter xyz we introduce different functions, that each specifically solve the task of the bullet points mentioned above. In the beginning we´ll underline the motivation and problem behind each function and showcase the code.

At the end of each section we provide a minimal working example, that illustrates the function and its output. We tried to implement in a way, that the function works for as much cases, as possible. If there are some restrictions regarding the usage of those functions, we´ll briefly discuss them as well.

2 Preprocess / Helper functions

2.1 Function for creating grid

In order to efficiently run a Monte Carlo simulation, we first need to specify for which set of parameters we want to run the simulation process. In case more than one variable is defined, it reasonable to create a parameter grid for each different combination of parameters.

```
create_grid <- function(parameters, nrep){
  input <- parameters
```

```r
  storage <- list()
  name_vec <- c()

  for(i in 1:length(input)){ #1:3
    a <- as.numeric(input[[i]][[2]])
    b <- as.numeric(input[[i]][[3]])
    c <- as.numeric(input[[i]][[4]])
    output <- seq(from=a, to=b, by=c)
    storage[[i]] <-  output
    name_vec[i] <- input[[i]][[1]]
  }

  grid <- expand_grid(unlist(storage[1])
                    , unlist(storage[2])
                    , unlist(storage[3])
                    , unlist(storage[4])
                    , unlist(storage[5])
                    , c(1:nrep))

  names(grid) <- c(name_vec, "rep")

  return(grid)
}
```

`create_grid` is the function hat creates a parameter grid with all permutations of the given parameters. The user has to input the parameters as a list, thats specified in the following way:

`parameter_list <- list(c("variable name 1", from, to, by) ,c("variable name 2", from, to, by)  ,c("variable name 3", from, to, by)  ,c("variable name 4", from, to, by))`

The function works with a minimum of 1 and a maximum of 4 variables. The structure of the remaining arguments is kept similar to the way the way R´s build in function`seq()` is specified: The first argument after the variable name defines the start of the sequence, the second one the end and the last one the steps, by which each variable specified for the parameter grid.

It would be fairly easy to adapt this helper function for more parameters, but it is assumed, that a parameter grid with up to 4 parameters offers enough complexity for the simulation. The function basically takes the infromation of the input parameter list and creates a parameter grid with `tidyr::expand_grid()`. The function also makes sure that the columns are named after the correct variable and also creates a different row for each number repetition, that the user specified in the second argument of `create_grid()`, namely `nrep`.

Following is a demonstration of how the input and output of this function looks like:

`create_grid()` Example:

```
#four parameters
param_list0 <- list(c("n", 10, 20, 10)
                    ,c("mu", 0, 0.5, 0.25)
                    ,c("sd", 0, 0.3, 0.1)
                    ,c("gender", 0, 1, 1))

head(create_grid(param_list0, nrep=3), n=20)
```

```
## # A tibble: 20 x 5
##         n    mu    sd gender   rep
##     <dbl> <dbl> <dbl>  <dbl> <int>
##  1     10     0   0        0     1
##  2     10     0   0        0     2
##  3     10     0   0        0     3
##  4     10     0   0        1     1
##  5     10     0   0        1     2
##  6     10     0   0        1     3
##  7     10     0   0.1      0     1
##  8     10     0   0.1      0     2
##  9     10     0   0.1      0     3
## 10     10     0   0.1      1     1
## 11     10     0   0.1      1     2
## 12     10     0   0.1      1     3
## 13     10     0   0.2      0     1
## 14     10     0   0.2      0     2
## 15     10     0   0.2      0     3
## 16     10     0   0.2      1     1
## 17     10     0   0.2      1     2
## 18     10     0   0.2      1     3
## 19     10     0   0.3      0     1
## 20     10     0   0.3      0     2
```

2.2 Data generation function

`data_generation` allows users to flexibly change data while keeping the summary statistics and to choose the number of inputs by using different `purrr` mapping functions: map, map2, and pmap for a input, two inputs, and p inputs respectively.

In the function below, `simulation` means a distribution of data, and `grid` is a list of parameters.

```
data_generation <- function(simulation, grid){
  #this is for use inside the function
  if(ncol(grid)==2){
    var1 <- c(unlist(grid[,1]))
    data <- map(var1, simulation)
    #different purrr-functions depending on how many input variables we use
  }

  if(ncol(grid)==3){
    var1 <- c(unlist(grid[,1]))
    var2 <- c(unlist(grid[,2]))
    data <- map2(var1, var2, simulation)
  }

  if(ncol(grid)==4){
    var1 <- c(unlist(grid[,1]))
    var2 <- c(unlist(grid[,2]))
    var3 <- c(unlist(grid[,3]))
    list1 <- list(var1,var2,var3)
    data <- pmap(list1, .f=simulation)
  }

  return(data)
}
```

`data_generation()` Example:

```
param_list1 <- list(c("n", 10, 20, 10))

param_list2 <- list(c("n", 10, 20, 10)
                  ,c("mu", 0.5, 1, 0.25))
#create_grid(param_list1, nrep=10)
#create_grid(param_list1, nrep=1)

'grid1 <- create_grid(param_list1, nrep=3)
tail(data_generation(simulation=rnorm, grid=grid1),1)'
```

```
## [1] "grid1 <- create_grid(param_list1, nrep=3)\ntail(data_generation(simulation=rnorm, gr
```

```
grid2 <- create_grid(param_list2, nrep=3)
data_generation(simulation=rnorm, grid=grid2)
```

```
## $n1
##  [1] -0.06047565  0.26982251  2.05870831  0.57050839  0.62928774  2.21506499
##  [7]  0.96091621 -0.76506123 -0.18685285  0.05433803
##
## $n2
##  [1]  1.72408180  0.85981383  0.90077145  0.61068272 -0.05584113  2.28691314
##  [7]  0.99785048 -1.46661716  1.20135590  0.02720859
##
## $n3
##  [1] -0.5678237  0.2820251 -0.5260044 -0.2288912 -0.1250393 -1.1866933
##  [7]  1.3377870  0.6533731 -0.6381369  1.7538149
##
## $n4
##  [1] 1.1764642 0.4549285 1.6451257 1.6281335 1.5715811 1.4386403 1.3039177
##  [8] 0.6880883 0.4440373 0.3695290
##
## $n5
##  [1]  0.05529302  0.54208272 -0.51539635  2.91895597  1.95796200 -0.37310858
##  [7]  0.34711516  0.28334465  1.52996512  0.66663093
##
## $n6
##  [1]  1.0033185  0.7214532  0.7071295  2.1186023  0.5242290  2.2664706
##  [7] -0.7987528  1.3346137  0.8738542  0.9659416
##
## $n7
##  [1]  1.37963948  0.49767655  0.66679262 -0.01857538 -0.07179123  1.30352864
##  [7]  1.44820978  1.05300423  1.92226747  3.05008469
##
## $n8
##  [1]  0.5089688 -1.3091689  2.0057385  0.2907992  0.3119914  2.0255714
##  [7]  0.7152270 -0.2207177  1.1813035  0.8611086
##
## $n9
##  [1] 1.0057642 1.3852804 0.6293400 1.6443765 0.7795134 1.3317820 2.0968390
##  [8] 1.4351815 0.6740684 2.1488076
##
## $n10
##  [1]  1.4935039  1.0483970  0.7387317 -0.1279061  1.8606524 -0.1002596
##  [7]  2.6873330  2.0326106  0.2642996 -0.5264209 -0.2104066  0.7568837
## [13]  0.2533081  0.1524574 -0.4516186  0.4549723 -0.2849045 -1.1679419
## [19]  0.1197735  1.4189966
##
## $n11
##  [1] -0.075346963  1.107964322 -1.117882708  0.444438034  1.019407204
##  [6]  0.801153362  0.605676194 -0.140706008 -0.349704346 -0.524128791
## [11]  0.617646597 -0.447474614  0.009442556  0.243907808  2.343862005
## [16] -0.151949902  0.735386572  0.577960850 -0.461856634  0.428691914
##
## $n12
##  [1]  1.94455086  0.95150405  0.54123292  0.07750317 -1.55324722  1.63133721
##  [7] -0.96064007  1.23994751  2.40910357 -0.94389316  1.20178434  0.23780251
## [13] -1.07214416 -1.01466765 -1.10153617 -0.03090652 -0.96175558  1.18791677
## [19]  2.60010894 -0.78703048
##
## $n13
##  [1]  1.5377388  1.5190422  1.0822026 -0.2583766  0.6305474  0.4696047
##  [7]  1.3129895  0.3775612  1.7269734  0.3754191  1.8027115 -0.2991770
## [13] -0.5101552  3.9910399  0.3331424  1.0482276  1.3865697  0.2662194
## [19]  1.2668620  1.1189645
##
## $n14
##  [1]  0.534619492  0.815293034  0.715932746  2.878451899  0.008663904
##  [6] -0.345996267  0.787788399  1.060480749  1.186523479  0.291634667
## [11] -0.313326134  2.013185176  0.400349612 -0.115512863  0.513720431
## [16]  0.552824106  1.859920290  0.834737292  1.504053785  0.250707983
##
## $n15
##  [1]  0.96444531  0.42531409  0.84458353 -0.14536336 -0.56080153  2.74721338
##  [7]  1.35070882 -0.50127136  0.13883408 -0.43548008  2.94881035  2.06241298
## [13]  0.48485494  1.29319406  0.33566005  0.27375311 -0.03860284  0.15538273
## [19]  2.40090747  0.69597187
##
## $n16
##  [1]  1.11924524  1.24368743  2.23247588  0.48393617  0.00749285  2.67569693
##  [7]  0.55883678  0.27693403 -0.23627312 -0.28471572  0.42602652  1.61798582
## [13]  2.10984814  1.70758835  0.63634270  1.05974994  0.29540354  0.28278184
## [19]  1.88465050 -0.01559258
##
## $n17
##  [1]  2.9552940  0.9096804  1.2145388  0.2614723  0.4256113 -0.3170161
##  [7]  0.8170746  1.4189824  1.3243043  0.2184635  0.2113780  0.4978013
## [13]  2.4960607 -0.1373036  0.8209484  2.9023618  0.8990251 -0.3598407
## [19]  0.3352306  1.4854600
##
## $n18
##  [1]  0.6243971  0.4381236  0.6560828  1.0904966  2.5985088  0.9114349
##  [7]  2.0807995  1.6307541  0.8863601 -0.5329020  0.4788827  0.5101295
## [13]  1.0471544  2.3001987  3.2930790  2.5475811  0.8668490 -0.7565274
## [19]  0.6112201  1.0892072
```

Users can apply many distributions such as normal, uniform, poisson distributions by putting existing functions in r as `simulation`.

```
# Application to Uniform distribution
param_list_runif <- list(c("n", 10, 30, 10)
                         ,c("min", 0, 0, 0)
                         ,c("max", 1, 1, 0))



grid_unif <- create_grid(param_list_runif, nrep=3)
tail(data_generation(simulation=runif, grid=grid_unif),1)
```

```
## $n9
##  [1] 0.039780637 0.634800510 0.539560914 0.140103550 0.283711777 0.583030595
##  [7] 0.165025892 0.096301111 0.428613091 0.355761566 0.844933540 0.260132474
## [13] 0.023144492 0.862399540 0.334587961 0.631788872 0.546426259 0.376444493
## [19] 0.185873015 0.428940591 0.630773599 0.520842351 0.659621337 0.729360931
## [25] 0.486822873 0.384456616 0.006833509 0.003684228 0.994936440 0.107886880
```

```
# Application to Poisson distribution

param_list_rpois <- list(c("n", 10, 30, 10)
                         , c("lambda", 0, 10, 1))

grid_pois <- create_grid(param_list_rpois, nrep=3)
tail(grid_pois,2) # nrow(grid_pois) = 99
```

```
## # A tibble: 2 x 3
##       n lambda   rep
##   <dbl>  <dbl> <int>
## 1    30     10     2
## 2    30     10     3
```

```
tail(data_generation(simulation=rpois, grid=grid_pois),1)
```

```
## $n99
##  [1] 14 13  9  9  8 14  8 10  6 13 12  6 12  9  9 12 13 21 11  6  9 14  9  8  8
## [26]  9  8 10  8  9
```

2.3 Summary function

`summary_function` offers summary statistics that users can choose.

```r
#summary function for one input
summary_function <- function(sum_fun, data_input){

  count <- length(data_input)
  summary_matrix <- matrix(nrow=count, ncol=1)

  for(i in 1:count){
    input <- list(data_input[[i]])
    output <- sapply(sum_fun, do.call, input)
    summary_matrix[i] <- output
  }
  #output <- as.data.frame(summary_matrix)
  #names(output) <- sum_fun
  colnames(summary_matrix) <- sum_fun
  return(summary_matrix)
}
```

`summary_function` Example:

```r
param_list3 <- list(c("n", 10, 20, 10)
                    ,c("mu", 0, 1, 0.25)
                    ,c("sd", 0, 0.3, 0.1))

grid_test <- create_grid(param_list3, nrep=3)
test_data <- data_generation(simulation=rnorm, grid=grid_test)
tail(summary_function(sum_fun=list("mean"), data_input=test_data),2)
```

```
##             mean
## [119,] 1.0418098
## [120,] 0.9500136
```

2.4 Summary array funcation

The outcome of `create_array_function` illustrates the combination of user defined grid and the summary statistics. This function product dataframes with all permutations and results that allow, thus users can look any possible parameter regarding specific grid.

```r
create_array_function <- function(comb, parameters, nrep){
  storage <- list()
  name_vec <- c()

  for(i in 1:length(parameters)){
    #this creates the sequences of parameters
    a <- as.numeric(parameters[[i]][[2]])
    b <- as.numeric(parameters[[i]][[3]])
    c <- as.numeric(parameters[[i]][[4]])
    output <- seq(from=a, to=b, by=c)
    storage[[i]] <-  output
    name_vec[i] <- parameters[[i]][[1]]
    #this just stores the names of the variables
  }


  matrix.numeration <-  paste("rep","=", 1:nrep, sep = "")

  if(length(parameters)==1){
    comb_ordered <-  comb %>% arrange(comb[,2])
    seq1 <- c(unlist(storage[1]))

    row.names <- paste(name_vec[1],"=",seq1, sep = "")

    dimension_array <- c(length(seq1), nrep)
    dim_names_list <- list(row.names, matrix.numeration)
  }

  if(length(parameters)==2){
    comb_ordered <-  comb %>% arrange(comb[,2])  %>% arrange(comb[,3])
    seq1 <- c(unlist(storage[1]))
    seq2 <- c(unlist(storage[2]))

    row.names <- paste(name_vec[1],"=",seq1, sep = "")
    column.names <-  paste(name_vec[2],"=",seq2, sep = "")

    dimension_array <- c(length(seq1), length(seq2), nrep)
    dim_names_list <- list(row.names, column.names, matrix.numeration)
  }

  if(length(parameters)==3){
    comb_ordered <-  comb %>% arrange(comb[,2])  %>%
      arrange(comb[,3]) %>% arrange(comb[,4])
    seq1 <- c(unlist(storage[1]))
    seq2 <- c(unlist(storage[2]))
    seq3 <- c(unlist(storage[3]))

    row.names <- paste(name_vec[1],"=",seq1, sep = "")
    column.names <-  paste(name_vec[2],"=",seq2, sep = "")
    matrix.names1 <-  paste(name_vec[3],"=",seq3, sep = "")

    dimension_array <- c(length(seq1), length(seq2), length(seq3), nrep)
    dim_names_list <- list(row.names, column.names,
                           matrix.names1, matrix.numeration)

  }



  array1 <- array(comb_ordered[,ncol(comb)]
                  #change to automatically adjust dim
                  , dim = dimension_array
                  , dim_names_list)
  return(array1)
}
```

create_array_function Example:

```r
# PREP TEST `create_array_function`
main_function_array_test <-  function(parameters #list of parameters
                                      , nrep #number of repetitions
                                      , simulation #data genereation
                                      , sum_fun){ #summary statistics

  grid <- create_grid(parameters, nrep) #Step 1: create grid
  raw_data <- data_generation(simulation, grid) #Step 2: simlate data
  summary <- summary_function(sum_fun, data_input=raw_data) #Step 3: Summary statistics
  comb <- cbind(grid, summary) #Step 4: Combine resuluts with parameters
  array_1 <- create_array_function(comb, parameters, nrep) #Step 5: Create array

  return(comb)
}

param_list3x <- list(c("n", 10, 20, 10)
                     ,c("mu", 0, 5, 1)
                     ,c("sd", 0, 1, 1))
```

```
comb1 <- main_function_array_test(parameters=param_list3x
                                  , nrep = 1
                                  , simulation = rnorm
                                  , sum_fun="mean")

head(comb1,2)
```

```
##    n mu sd rep      mean
## 1 10  0  0   1 0.0000000
## 2 10  0  1   1 0.5813486
```

```
create_array_function(comb=comb1, parameters=param_list3x, nrep=1)
```

```
## , , sd=0, rep=1
##
##      mu=0 mu=1 mu=2 mu=3 mu=4 mu=5
## n=10    0    1    2    3    4    5
## n=20    0    1    2    3    4    5
##
## , , sd=1, rep=1
##
##             mu=0      mu=1     mu=2     mu=3     mu=4     mu=5
## n=10  0.58134864 0.8653182 2.625167 2.869693 4.061384 5.575393
## n=20 -0.01528894 0.5086328 1.861943 2.822552 4.252306 4.919115
```

3 Monte Carlo Simulation Funcion

4 Examples

5 Conclusion

The above section illustrates the power of our implemented model and gives the fairly easy to use tool, that still allows for a variety of different specifications in terms of used parameters, data generation processes and summary functions. Researchers, who use Monte Carlo studys on a regular basis, may save a lot of time using a tool like this in the long run.

By nature, there may be cases, where our implementation doesnt satisfy the needs of the user to the fullest, but for a wide variety of examples we showed, that it worked well and served the goal that we aimed for. Our functional programming approach allows for easy and flexible adjustments in case the use of our functions should be expanded, f.e. if a grid of more than 3 (or 4?) parameters is needed.

Theoretically, this work could be implemented as an R package to share it with the R community. But since the `MonteCarlo()` function of the `vigniette` package already provides a well working alternative to our project besides some minor differences, there is currently no need in doing that.

6 References

7 Contributions

Eidesstattliche Versicherung

Ich versichere an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Essen, den ______________________ ______________________

Alexander Langnau, Öcal Kaptan, Sunyoung Ji