Constructing User Defined Functions in R

Ashley I Naimi

Spring 2024

Contents

1 Constructing User Defined Functions in R

Functions are the most useful operational units in the R programming language. Every command run in R is a function. They are also essential when running simulation studies. In this section, we will cover some basic concepts behind writing functions in R, and then demonstrate how they can be tailored to Monte Carlo simulations.

The basic of a function in R is as follows:

```
a_function <- function(<arguments>){

  <internal code with arguments>

  <return statement>
}
```

There are several functions that we can construct for our purposes in doing a simulation study. For example, when using logistic regression, we'll need two functions that we've already seen: the logistic and inverse logistic functions. Let's explore these:

```
expit <- function(x) {
    exp(x)/(1 + exp(x))
}


logit <- function(x) {
    log(x/(1 - x))
}
```

The first function takes a log-odds and returns a probability. The second takes a probability and returns a log-odds. For example, if we have a simple logistic regression model as defined as:

$$\text{logit}\, P(Y = 1 \mid X) = \beta_0 + \beta_1 X$$

where $\beta_0 = -1.5$ and $\beta_1 = -0.5$ for $X \in [0, 1]$, then, for individuals with $X = 1$ and $X = 0$, we have:

```
expit(-2)
```

```
## [1] 0.1192029
```

```
expit(-1.5)
```

```
## [1] 0.1824255
```

```
logit(0.2)
```

```
## [1] -1.386294
```

```
logit(0.18)
```

```
## [1] -1.516347
```

In the above functions, the arguments are `x`, and the `return` statement is implicit: because the only thing created within these functions are the expit and logit objects, they are automatically returned in the function.

2 A Basic Simulation Function

Here is a basic function that simulates a continuous covariate `c` a binary outcome `y`. The function takes three arguments: `nsim`, which is an indexing argument whose use will become clear soon; `sample_size`, and `parameter`, which is the odds ratio for the association between the covariate `c` and the outcome `y`.

The function outputs four items: the first is the indexing argument; the second is the sample size; the third is the mean of the outcome `y`, which will depend on the intercept value and the odds ratio parameter; the fourth is the estimated log-odds ratio for the association between `c` and `y`. Here is the function:

```r
set.seed(123)

simulation_function <- function(nsim, sample_size, parameter) {

    # data generation
    c <- rnorm(sample_size, mean = 0, sd = 1)

    p_y <- expit(-2 + log(parameter) * c)

    y <- rbinom(sample_size, size = 1, p = p_y)

    a_data <- data.frame(c, y)

    # analysis

    mY <- mean(a_data$y)

    glm_fit <- glm(y ~ c, data = a_data, family = binomial("logit"))

    glm_res <- summary(glm_fit)$coefficients[2, 1]

    sim_res <- c(nsim, sample_size, parameter, mY, glm_res)

    return(sim_res)

}
```

Running the above code initiates the function. Let's run a little simulation:

```r
simulation_function(nsim = 1, sample_size = 500, parameter = 2)
```

```
## [1]   1.0000000 500.0000000   2.0000000   0.1180000   0.9608086
```

Of course, running a single iteration of this simulation is not going to help us. Instead, we want to be able to generate a large number of log-odds ratio estimates to we can evaluate some properties of this distribution. We can do

this with one of the many iterative functions available to us in R. These include `for` loops as well as the `apply` family of functions.

3 For Loops in R

We'll start with using for loops to run the simulation function several times.

```
simulation_results <- NULL

for(i in 1:10){
  simulation_results <- rbind(
    simulation_results,
    simulation_function(nsim = i, sample_size = 500, parameter = 2)
  )
}

simulation_results
```

```
##       [,1] [,2] [,3]  [,4]      [,5]
##  [1,]    1  500    2 0.136 0.7794415
##  [2,]    2  500    2 0.120 0.9628786
##  [3,]    3  500    2 0.140 0.6097010
##  [4,]    4  500    2 0.146 0.6789281
##  [5,]    5  500    2 0.140 0.6448806
##  [6,]    6  500    2 0.138 0.7061543
##  [7,]    7  500    2 0.144 0.9168108
##  [8,]    8  500    2 0.134 0.7159953
##  [9,]    9  500    2 0.134 0.7376969
## [10,]   10  500    2 0.144 0.7773598
```

The above code creates an empty R object we called `simulation_results` and then ran a for loop which assigned the results of the first iteration of the loop to the empty object. The second iteration results are then appended to the first using the `rbind` function, and the process continues until the last for loop iteration.

We can use another approach with `for` loops in R. This second approach

starts with constructing an empty list to store the results in, as opposed to a `NULL` object:

```
simulation_results <- list()

for(i in 1:10){
  simulation_results[[i]] <- simulation_function(nsim = i, sample_size = 500, parameter = 2)
}

simulation_results
```

```
## [[1]]
## [1]   1.0000000 500.0000000   2.0000000   0.1460000   0.7485483
##
## [[2]]
## [1]   2.000000 500.000000   2.000000   0.120000   0.534538
##
## [[3]]
## [1]   3.0000000 500.0000000   2.0000000   0.1400000   0.8073003
##
## [[4]]
## [1]   4.0000000 500.0000000   2.0000000   0.1560000   0.9765668
##
## [[5]]
## [1]   5.0000000 500.0000000   2.0000000   0.1620000   0.9902378
##
## [[6]]
## [1]   6.000000 500.000000   2.000000   0.122000   0.600954
##
## [[7]]
## [1]   7.0000000 500.0000000   2.0000000   0.1400000   0.6138803
##
## [[8]]
## [1]   8.0000000 500.0000000   2.0000000   0.1380000   0.4679613
##
## [[9]]
```

```
## [1]   9.0000000 500.0000000   2.0000000   0.1500000   0.6392804
##
## [[10]]
## [1]  10.0000000 500.0000000   2.0000000   0.1040000   0.6449406
```

This approach is sometimes a little faster to run, but requires some post-processing to fix the list. We can do this using the `do.call` function, which takes another function (in this case, `rbind`) as an argument, and applies it to the second argument[1]:

[1] We can also use the `rbindlist` function, which is available in the `data.table` package

```
library(data.table)

sim_res <- do.call(rbind, simulation_results)

sim_res
```

```
##       [,1] [,2] [,3]  [,4]      [,5]
##  [1,]    1  500    2 0.146 0.7485483
##  [2,]    2  500    2 0.120 0.5345380
##  [3,]    3  500    2 0.140 0.8073003
##  [4,]    4  500    2 0.156 0.9765668
##  [5,]    5  500    2 0.162 0.9902378
##  [6,]    6  500    2 0.122 0.6009540
##  [7,]    7  500    2 0.140 0.6138803
##  [8,]    8  500    2 0.138 0.4679613
##  [9,]    9  500    2 0.150 0.6392804
## [10,]   10  500    2 0.104 0.6449406
```

4 The Apply Family of Functions

Instead of a `for` loop, we can use candidate algorithms in the `apply` family of functions. These functions include `apply`, `lapply`, `sapply`, `mapply`, and `tapply`. The `apply` and `tapply` functions are often used to apply a function to one or more columns of data. They can be useful in simulation studies, but we're going to focus here on `lapply`, `sapply`, and `mapply`.

4.1 lapply

The one I use most frequently in the context of simulation studies is the `lapply` function. It operates very much like a `for` loop that populates an empty list.

```
simulation_results <- lapply(1:10, function(x)
  simulation_function(nsim = x, sample_size = 500, parameter = 2)
)

simulation_results
```

```
## [[1]]
## [1]   1.0000000 500.0000000   2.0000000   0.1260000   0.8316456
##
## [[2]]
## [1]   2.0000000 500.0000000   2.0000000   0.1280000   0.3415096
##
## [[3]]
## [1]   3.0000000 500.0000000   2.0000000   0.1280000   0.6634622
##
## [[4]]
## [1]   4.0000000 500.0000000   2.0000000   0.1320000   0.8254062
##
## [[5]]
## [1]   5.0000000 500.0000000   2.0000000   0.1320000   0.6364461
##
## [[6]]
## [1]   6.0000000 500.0000000   2.0000000   0.1120000   0.6431364
##
## [[7]]
## [1]   7.0000000 500.0000000   2.0000000   0.1220000   0.7788064
##
## [[8]]
## [1]   8.0000000 500.0000000   2.0000000   0.1240000   0.7048815
##
```

```
## [[9]]
## [1]   9.0000000 500.0000000   2.0000000   0.1300000   0.8443099
##
## [[10]]
## [1]  10.0000000 500.0000000   2.0000000   0.1220000   0.6223795
```

```
do.call(rbind, simulation_results)
```

```
##       [,1] [,2] [,3]  [,4]      [,5]
##  [1,]    1  500    2 0.126 0.8316456
##  [2,]    2  500    2 0.128 0.3415096
##  [3,]    3  500    2 0.128 0.6634622
##  [4,]    4  500    2 0.132 0.8254062
##  [5,]    5  500    2 0.132 0.6364461
##  [6,]    6  500    2 0.112 0.6431364
##  [7,]    7  500    2 0.122 0.7788064
##  [8,]    8  500    2 0.124 0.7048815
##  [9,]    9  500    2 0.130 0.8443099
## [10,]   10  500    2 0.122 0.6223795
```

4.2 sapply

We can use `sapply` too, but need to format the output of this function slightly differently.

```
simulation_results <- sapply(1:10, function(x)
  simulation_function(nsim = x, sample_size = 500, parameter = 2),
  simplify = T
)

simulation_results
```

```
##             [,1]        [,2]        [,3]        [,4]        [,5]        [,6]
## [1,]   1.0000000   2.0000000   3.0000000   4.0000000   5.0000000   6.0000000
## [2,] 500.0000000 500.0000000 500.0000000 500.0000000 500.0000000 500.0000000
## [3,]   2.0000000   2.0000000   2.0000000   2.0000000   2.0000000   2.0000000
```

```
## [4,]   0.1340000   0.1420000   0.1400000   0.1220000   0.1340000   0.1320000
## [5,]   0.5730563   0.6960024   0.6341226   0.6185285   0.5704796   0.8771361
##             [,7]        [,8]        [,9]       [,10]
## [1,]   7.0000000   8.0000000   9.0000000  10.0000000
## [2,] 500.0000000 500.0000000 500.0000000 500.0000000
## [3,]   2.0000000   2.0000000   2.0000000   2.0000000
## [4,]   0.1540000   0.1180000   0.1540000   0.1240000
## [5,]   0.8272415   0.7592709   0.6899982   0.4811386
```

```
# note the transpose!
t(simulation_results)
```

```
##       [,1] [,2] [,3]  [,4]      [,5]
##  [1,]    1  500    2 0.134 0.5730563
##  [2,]    2  500    2 0.142 0.6960024
##  [3,]    3  500    2 0.140 0.6341226
##  [4,]    4  500    2 0.122 0.6185285
##  [5,]    5  500    2 0.134 0.5704796
##  [6,]    6  500    2 0.132 0.8771361
##  [7,]    7  500    2 0.154 0.8272415
##  [8,]    8  500    2 0.118 0.7592709
##  [9,]    9  500    2 0.154 0.6899982
## [10,]   10  500    2 0.124 0.4811386
```

In my experience, `sapply` is sometimes a slower function, possibly because it builds a set of results along columns as opposed to a list. However, we will later do a test to evaluate the performance speed of `sapply` and `lapply` (we will do this in the section on computing and profiling functions).

4.3 `mapply`

Another function that we can use is the `mapply` function, which is someone different from the previous two. To extend our example, let's say that instead of looping over the simulation index `nsim`, suppose we also wanted to look at the impact of different sample sizes (say 250 and 500). This creates a situation where we have to loop the function over two arguments instead of one. This is where `mapply` can come in useful:

```
simulation_results <- mapply(simulation_function,
                             nsim = 1:10, # number of simulations
                             sample_size = rep(c(250,500), each=10), # sample size
                             parameter = 2, # other function arguments
                             SIMPLIFY = T) # need to simplify for proper formatting

t(simulation_results)
```

```
##       [,1] [,2] [,3]  [,4]      [,5]
##  [1,]    1  250    2 0.104 0.8059524
##  [2,]    2  250    2 0.164 0.7439719
##  [3,]    3  250    2 0.160 0.4177259
##  [4,]    4  250    2 0.128 0.8803293
##  [5,]    5  250    2 0.144 0.7543569
##  [6,]    6  250    2 0.148 0.5727400
##  [7,]    7  250    2 0.120 1.0666401
##  [8,]    8  250    2 0.144 0.6444885
##  [9,]    9  250    2 0.144 0.8696708
## [10,]   10  250    2 0.164 0.6472104
## [11,]    1  500    2 0.146 0.7295056
## [12,]    2  500    2 0.124 0.4724984
## [13,]    3  500    2 0.146 0.7664200
## [14,]    4  500    2 0.146 0.7065716
## [15,]    5  500    2 0.140 0.7841973
## [16,]    6  500    2 0.120 0.7528922
## [17,]    7  500    2 0.160 0.7487138
## [18,]    8  500    2 0.160 0.6217614
## [19,]    9  500    2 0.134 0.9040787
## [20,]   10  500    2 0.144 0.6845667
```

```
sim_res <- t(simulation_results)

colnames(sim_res) <- c("index", "sample_size", "parameter", "meanY", "log_OR")
```

```
head(sim_res, 3)
```

```
##      index sample_size parameter meanY    log_OR
## [1,]     1         250         2 0.104 0.8059524
## [2,]     2         250         2 0.164 0.7439719
## [3,]     3         250         2 0.160 0.4177259
```

```
tail(sim_res, 3)
```

```
##       index sample_size parameter meanY    log_OR
## [18,]     8         500         2 0.160 0.6217614
## [19,]     9         500         2 0.134 0.9040787
## [20,]    10         500         2 0.144 0.6845667
```

This `mapply` function provides us with a general way to explore a range of different simulation specifications. However, it can sometimes be a little tricky to keep track of all the arguments in a function, their range of values, and whether the `mapply` function is returning results for each combination of interest. In the next section, we cover a technique that can sometimes come in handy when trying to loop over multiple arguments.

4.4 A Generalization of `sapply` and `lapply` to Multiple Argument Loops

Consider a scenario where we might be interested in modifying the sample size and the parameters for the logistic regression model used to simulate our data. Let's say, in particular, we want to do 10 simulations, under two sample sizes (250 and 500), and under two parameter values (0.5 and 2). We can do this with `mapply` as follows:

```
simulation_results <- mapply(simulation_function,
                             nsim = 1:10, # number of simulations
                             sample_size = rep(c(250, 500), each = 10), # sample size
                             parameter = rep(c(.5, 2), each = 20), # other function arguments
                             SIMPLIFY = T) # need to simplify for proper formatting

sim_res <- t(simulation_results)
```

However, notice that for the `rep` function, the `each` argument is different between the `sample_size` and `parameter` arguments passed to the `mapply` function. At some point, this can become difficult to track, especially when there are multiple arguments to the simulation function, each with multiple different dimensions.

Instead of `mapply`, we can use a dataset based approach using another function called `expand.grid`. The first step is to set up a dataset that includes values for all the parameters we are interested in:

```
parm_data <- expand.grid(
  index = 1:10,
  n = c(250, 500, 1000),
  parms = c(.5, 2)
)
```

With this, we can modify our function to look like the following:

```
simulation_results <- lapply(1:nrow(parm_data), function(x)
  simulation_function(nsim = parm_data[x,]$index,
                      sample_size = parm_data[x,]$n,
                      parameter = parm_data[x,]$parms)
)

sim_res <- do.call(rbind, simulation_results)

colnames(sim_res) <- c("index", "sample_size", "parameter", "meanY", "log_OR")

head(sim_res, 3)
```

```
##      index sample_size parameter meanY     log_OR
## [1,]     1         250       0.5 0.144 -0.9534398
## [2,]     2         250       0.5 0.124 -0.9143330
## [3,]     3         250       0.5 0.116 -0.6507084
```

```
tail(sim_res, 3)
```

```
##       index sample_size parameter meanY    log_OR
## [58,]     8        1000         2 0.139 0.6218280
## [59,]     9        1000         2 0.148 0.7823719
## [60,]    10        1000         2 0.140 0.5546487
```

Importantly, we are using the `lapply` function, but instead of indexing over the number of Monte Carlo samples we want, we are indexing over all of the combinations of unique parameter values in the `parm_data` dataset. By putting all the parameters in a data.frame using the `expand.grid` function, we can deploy a simulation function with multiple arguments and only a single looping index. This strategy can become useful with particularly complex simulation functions with multiple arguments of varying lengths.

5 Understanding Seeds

One essential component of successfully deploying a simulation study is the appropriate use of seeds to fix the random number generator that R relies on to generate pseudo-randomness so that we might be able to reproduce the results. In this section, we'll cover how seeds work in R, how to use them, and common mistakes that should be avoided.

To understand seeds, it is important to understand the idea of a pseudo random number generator. The basic idea behind a pseudo-random number generator is to find a sequence of digits with no apparent pattern or regularities to them. For example, if we can establish that the sequence of digits in the constant π doesn't have any regular patterns to it, we can use it as our base to generate randomness:

$$3.14159265358979\cdots$$

The so-called "pseudo-randomness" occurs in the transition from one number in the sequence to the next. For example, if we start the random number sequence at the third digit (4), the next numbers in the sequence will be 1, then 5, then 9, and so on.

Since there is no obvious predictability or regularity in this sequence, we

can use these digits as the base to initialize a range of complex functions that mimic random behavior.

However, because π is a fixed constant, the numbers in the sequence don't change. So, if we want to initialize functions that mimic randomness, but want to be able to reproduce this randomness, we can use a seed value to start the random sequence at the same point every time.

In the above simplified π example, we could say that the "seed" value will be 3, because we relied on the **third** digit in the sequence.

The above description of seeds and random number generators is obviously way simplified. However, this example represents the core idea of a seed, and sets the stage for understanding some of the negative consequences that we can encounter using seed values.

6 Using Seeds in R

Near the beginning of this pdf document, we set a seed value to demonstrate the use of for loops and the apply family for a simple simulation study. We used `set.seed(123)` to do this. It is possible that this seed value will affect all analyses in subsequent code chunks in the document,[2] so let's take the opportunity to reset it here:

[2] I don't actually think this is true, but it's a good opportunity to demonstrate how to reset seeds.

```
set.seed(NULL)
```

Now that we've initialized a clean slate with respect to the random number generator, we can explore the use of seeds in R. Let's start with generating five realizations of a draw from a uniform distribution and a Poisson distribution with a mean of two:

```
a <- runif(5)

b <- rpois(5, lambda = 2)
```

Let's look at the values of `a` and `b` generated from this run:

```
a
```

```
## [1] 0.2235426 0.2335404 0.7526581 0.4292375 0.9095494
```

```
b
```

```
## [1] 1 0 4 1 1
```

I'd like to be able to tell you what these numbers are, but I can't, since every time I render this RMarkdown document, they change. We can use seeds to fix this:

```
set.seed(123)

a <- runif(5)

b <- rpois(5, lambda = 2)

a
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

```
b
```

```
## [1] 0 2 4 2 2
```

Now, every time we run this code chunk, we get exactly the same values for `a` and `b`, which are 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673 for `a` and 0 1 1 1 1 for `b`.

6.1 Setting Seeds Once and Only Once

In most settings, it's important to avoid setting the seed multiple times during a single session, since this can lead to unwanted dependencies between pseudo-random variables generated in the session. A good example of this problem can be demonstrated by continuing the above example. This time, we'll set the seed once before each initialization of the random variable generator.

We'll also increase the sample size to 50,000. These two variables should be independent because the mean of the poisson random variable does not depend on the value of the simulated uniform random variable (and vice versa):

```
# re-initialize the seed
set.seed(NULL)

set.seed(123)
a <- runif(50000)

set.seed(123)
b <- rpois(50000, lambda = 2)
```

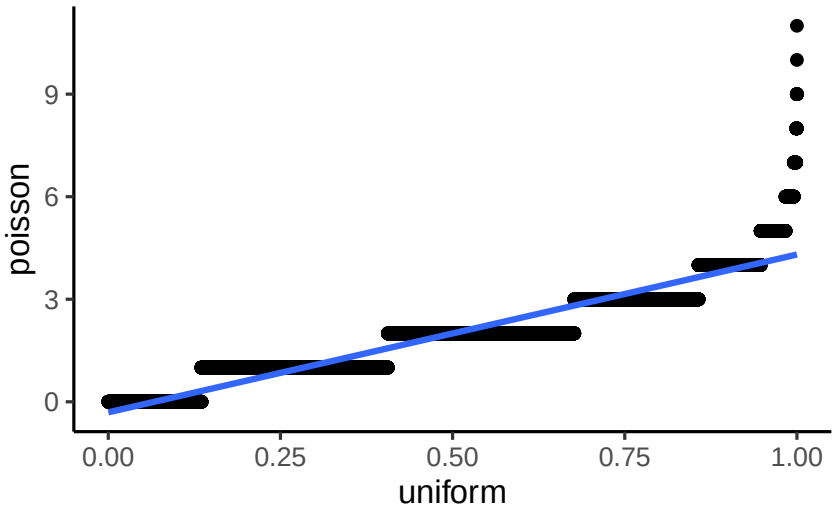
We can fit a regression model to evaluate the statistical relationship between `b` and `a`:

```
summary(lm(b ~ a))$coefficients
```

```
##               Estimate  Std. Error   t value Pr(>|t|)
## (Intercept) -0.3077963 0.004098892 -75.09256        0
## a            4.6156121 0.007129870 647.36270        0
```

This demonstrates a strong association between these two variables that should be independent, which is also established in the following figure:

```
plot_dat <- tibble(uniform = a, poisson = b)

ggplot(plot_dat, aes(x = uniform, y = poisson)) +
  geom_point() +
  geom_smooth(method='lm', se = F)
```



The plot above demonstrates a clear problem: higher/lower values of the uniform distribution are associated with higher/lower values of the Poisson distribution, even though these two variables should be independent[3]

[3] Importantly, note that no information in the uniform distribution is used to define the Poisson distribution, and vice versa.

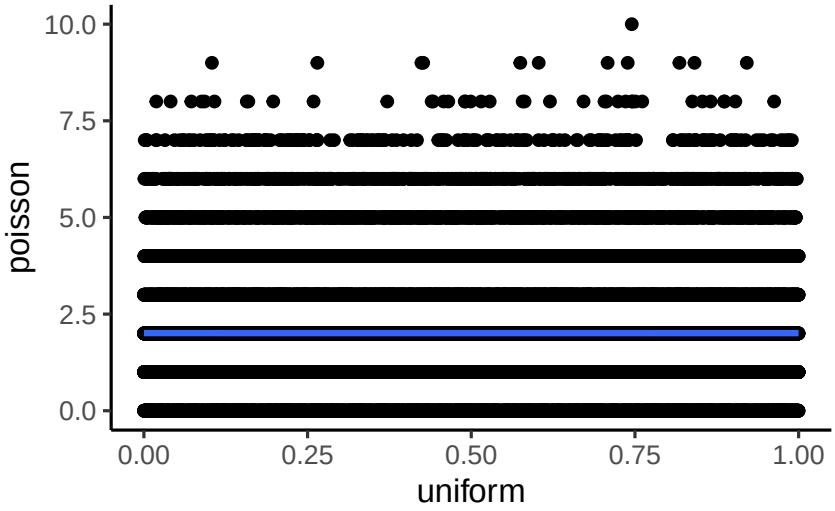
Why is this occurring? It turns out that in R (as in many other statistical software platforms) the way the Poisson random variable is generated is based on a procedure known as the inverse transformation method [Roberts and Casella (2010); p44]. This approach starts with a uniform distribution to generate the Poisson random variable. So the problem that arises is that the exact same uniform random variable that is generated in `a` is used to generate the Poisson random variable defined in `b`.

What should this plot look like if we want to construct two **truly independent** random variables from a uniform and Poisson distribution? We can see this with the use of only a single seed initiated in the code chunk:

```
# re-initialize the seed
set.seed(NULL)

set.seed(123)
a <- runif(50000)

#set.seed(123)
b <- rpois(50000, lambda = 2)

summary(lm(b ~ a))$coefficients
```

```
##                Estimate Std. Error      t value  Pr(>|t|)
## (Intercept)  2.00809358 0.01264939 158.75029241 0.0000000
## a           -0.00119338 0.02200314  -0.05423681 0.9567467
```

```
plot_dat <- tibble(uniform = a, poisson = b)

ggplot(plot_dat, aes(x = uniform, y = poisson)) +
  geom_point() +
  geom_smooth(method='lm', se = F)
```



This example serves to illustrate the danger of setting the seed multiple times in a given session. Notably, if you construct a simulation to evaluate the property of an estimator that assumes some form of independence between random variables, and you generate these variables with a strong dependence between them, the underlying assumptions of the simulation study will be violated, and the conclusions drawn from the study may be misleading.

References

C Roberts and G Casella. *Introducing Monte Carlo Methods with R*. Springer US, New York, NY, 2010.