Simulating data is an invaluable tool. We use simulations to conduct power analyses, probe how robust methods are to violating assumptions, and examine how different methods handle different types of data. If I’m learning something new or writing a model from scratch, We’ll simulate data so that I know the correct answer—and make sure my model gives me that answer.

But simulations can be complicated. Many other programming languages require for loops to do a process multiple times; nesting many conditional statements and other for loops within for loops can quickly be difficult to read and debug. In this post, I’ll show how I do modular simulations by writing R functions and using the apply family of R functions to repeat processes. I use examples from Paul Nahin’s book, *Digital Dice: Computational Solutions to Practical Probability Problems*, and I show how his MATLAB code differs from what is possible in R.

My background is in the social sciences; We learned statistics as a tool to answer questions about psychology and behavior. Despite being a quantitative social scientist professionally now, We was not on the advanced math track in high school, and I never took a proper calculus class. We don’t know the theoretical math or how to derive things, but We are good at R programming and can simulate instead! All of these problems have derivations and theoretically-correct answers, but Nahin writes the book to show how simulation studies can achieve the same answer.

**Example 1: Clumsy Dishwasher**

Imagine 5 dishwashers work in a kitchen. Let’s name them dishwashers *A*, *B*, *C*, *D*, and *E*. One week, they collectively broke 5 plates. And dishwasher *A* was responsible for 4 of these breaks. His colleagues start referring to him as clumsy, but he says that this was a fluke and could happen to any of them. This is the first example in Nahin’s book, and he tasks us with finding the probability that dishwasher *A* was responsible for 4 or more of the 5 breaks. We are to do our simulation assuming that each dishwasher is of equal skill; that is, the probability of any dishwasher breaking a dish is the same.

What I’ll do first is define some parameters we are interested in. Let *N* be the number of dishwashers and *K* be the number of broken dishes. We will run 5 million simulations:

iter <- 5000000 # number of simulations

n <- 5 # number of dishwashers

k <- 5 # number of dish breaks

First, I adapted Nahin’s solution from MATLAB code to R code. It looks like this:

set.seed(1839)

clumsy <- 0

for (zzz in seq\_len(iter)) {

broken\_dishes <- 0

for (yyy in seq\_len(k)) {

r <- runif(1)

if (r < (1 / n))

broken\_dishes <- broken\_dishes + 1

}

if (broken\_dishes > 3)

clumsy <- clumsy + 1

}

clumsy / iter

## [1] 0.0067372

First, he sets clumsy to zero. This will be a variable that counts how many times dishwasher *A* broke more than 3 of the plates. We see a nested for loop here. The first one loops through all 5 million iterations; the second loops through all broken dishes. We draw a random number between 0 and 1. If this is less than 1 / *N* (the probability of any one dishwasher breaking a dish), we assign the broken dish to dishwasher A. If there are more than 3 of these, we call them “clumsy” and increment the clumsy vector by 1. At the end, we divide how many times dishwasher *A* was clumsy and divide that by the number of iterations to get the probability that this dishwasher broke 4 or 5 plates, given that all of the dishwashers have the same skill. We arrive at about .0067.

These nested for loops and if statements can be difficult to handle when simulations get more complicated. What would a modular simulation look like? I break this into two functions. First, we simulate which dishwashers broke the plates in a given week. sim\_breaks will give us a sequence of *N* letters from the first *K* letters of the alphabet. Each letter is drawn with equal probability, simulating the situation where all dishwashers are at the same skill level. Then, a\_breaks counts up how many times dishwasher *A* was at fault. Note that this function has no arguments of its own; it only has ..., which passes all arguments to sim\_breaks. The sapply function tells R to apply a function to all numbers 1 through iter. Since we don’t actually want to use those values—we just want them as dummy numbers to do something iter many times—I put a dummy argument of zzz in the function that we will be applying to each number 1 through iter. This function is a\_breaks(n, k) > 3). result will be a logical vector, where TRUE denotes dishwasher *A* broke more than 3 dishes and FALSE denotes otherwise. Since R treats TRUE as numeric 1 and FALSE as numeric 0, we can get the mean of result to tell us the probability of *A* breaking more than 3 dishes, given that all dishwashers are at the same skill level:

# simulate k dishwashers making n breaks in a week:

sim\_breaks <- function(n, k) {

sample(letters[seq\_len(k)], n, replace = TRUE)

}

# get the number of breaks done by the target person:

a\_breaks <- function(...) {

sum(sim\_breaks(...) == "a")

}

# how often will dishwasher a be responsible for 4 or 5 breaks?

set.seed(1839)

result <- sapply(seq\_len(iter), function(zzz) a\_breaks(n, k) > 3)

mean(result)

## [1] 0.0067372

We again arrive at about .0067.

Lastly, R gives us functions to draw randomly from distributions. Simulating how many dishes were broken by dishwasher *A* can also be seen as coming from a binomial distribution with *K* trials and a probability of *1 / N*. We can make iter draws from that distribution and see how often the number is 4 or 5:

set.seed(1839)

mean(rbinom(iter, k, 1 / n) > 3)

## [1] 0.0067196

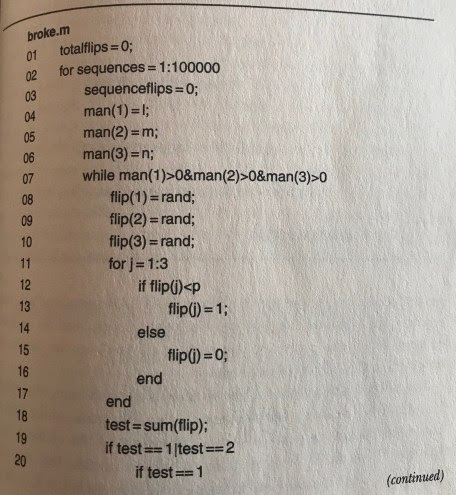
All three simulations give us about the same answer, which basically agree with the mathematically-derived answer of .00672. How do we interpret this? If you have a background in classical frequentist statistics (that is, focusing on *p*-values), you’ll notice that our interpretation is about the same as a *p*-value. If all dishwashers had the same probability of breaking a dish, the probability that dishwasher *A* broke 4 or 5 of them is .0067. Note that we are simulating from what could be called a “null hypothesis” that all dishwashers are equally clumsy. What we observed (dishwasher *A* breaking 4)—or more extreme data than we observed (i.e., 4 or more dishes) had a probability of .0067 of occurring. In most situations, we would “reject” this null hypothesis, because what we observed would have been so rare under the null. Thus, most people would say that dishwasher *A*’s 4 breaks in a week was not due to chance, but probably due to *A* having a higher latent clumsiness.

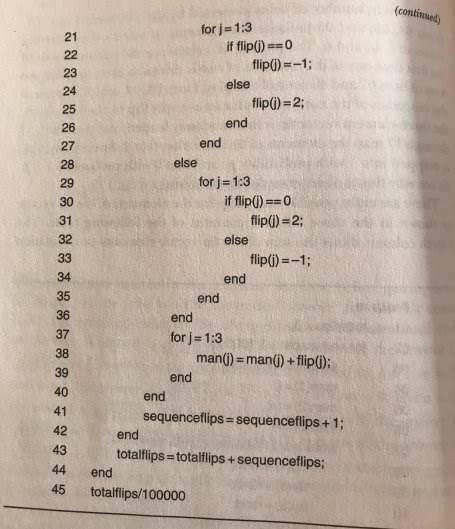
**Example 2: Curious Coin Flip Game**

Nahin tells us that this was originally a challenge question from the August-September 1941 issue of *American Mathematical Monthly*, and it was not solved until 1966. Imagine there are three people playing a game. Each of these three people have a specific number of quarters. One person has *L* quarters, another has *M* quarters, and another has *N* quarters. Each round involves all three people flipping one of their quarters. If all three coins come up the same (i.e., three heads or three tails), then nothing happens during that round. Otherwise, two of the players will have their coins come up the same, and one person will be different. The one that is different takes the other two players coins from that round.

So, for example, let’s say George has 3 quarters, Elaine has 3 quarters, and Jerry has 3 quarters. They all flip. George and Elaine get heads, while Jerry gets tails. George and Elaine would give those quarters to Jerry. So after that round, George and Elaine would have 2 quarters, while Jerry would have 5.

When someone runs out of coins, they lose the game. The challenge is to find the average number of rounds it takes until someone loses the game (i.e., runs out of coins). We are tasked with doing this at various values of initial starting quarter coins of *L*, *M*, and *N*. This is Nahin’s MATLAB solution:





For my taste, there’s too many for, while, and if else statements nested within one another. This can make it really easy to get confused while you’re writing the code, harder to debug, even harder to read, and a pain if you want to change something later on. Let’s make this modular with R functions.

#' Simulate a Round of Coin Flips

#'

#' This function simulates three coin flips, one for each player in the game.

#' A 1 corresponds to heads, while 0 corresponds to tails.

#'

#' @param p Numeric value between 0 and 1, representing the probability of

#' flipping a heads.

#' @return A numeric vector of length 3, containing 0s and 1s

sim\_flips <- function(p) {

rbinom(3, 1, p)

}

#' Simulate the Winner of a Round

#'

#' This function simulates the winner of a round of the curious coin flip game.

#'

#' @param ... Arguments passed to sim\_flips.

#' @return Either a number (1, 2, or 3) denoting which player won the round or

#' NULL, denoting that the round was a tie and had no winner.

sim\_winner <- function(...) {

x <- sim\_flips(...)

x <- which(x == as.numeric(names(table(x))[table(x) == 1]))

if (length(x) == 0)

else

}

#' Simulate an Entire Game of the Curious Coin Flip Game

#'

#' This function simulates an entire game of the curious coin flip game, and it

#' returns to the user how many rounds happened until someone lost.

#'

#' @param l Number of starting coins for Player 1.

#' @param m Number of starting coins for Player 2.

#' @param n Number of starting coins for Player 3.

#' @param ... Arguments passed to sim\_winner.

#' @return A numeric value, representing how many rounds passed until a player

#' lost.

sim\_game <- function(l, m, n, ...) {

lmn <- c(l, m, n)

counter <- 0

while (all(lmn > 0)) {

winner <- sim\_winner(...)

if (!is.null(winner)) {

lmn[winner] <- lmn[winner] + 2

lmn[-winner] <- lmn[-winner] - 1

}

counter <- counter + 1

}

return(counter)

}

Nahin asks for the answer with a number of different combinations of starting quarter counts *L*, *M*, and *N*. Below, I run the sim\_game function iter number of times for the starting values: 1, 2, and 3; 2, 3, and 4; 3, 3, and 3; and 4, 7, and 9. Giving a vector of calls to sapply will return a matrix where each row represents a different combination of starting quarter values and each column represents a result from that simulation. We can get the row means to give us the average values until someone loses the game for each combination:

set.seed(1839)

iter <- 100000 # setting lower iter, since this takes longer to run

results <- sapply(seq\_len(iter), function(zzz) {

c(

sim\_game(1, 2, 3, .5),

sim\_game(2, 3, 4, .5),

sim\_game(3, 3, 3, .5),

sim\_game(4, 7, 9, .5)

)

})

rowMeans(results)

## [1] 2.00391 4.56995 5.13076 18.64636

These values are practically the same as the theoretical, mathematically-derived solutions of 2, 4.5714, 5.1428, and 18.6667. I find creating the functions and then running them repeatedly through the sapply function to be cleaner, more readable, and easier to adjust or debug than using a series of nested for loops, while loops, and if else statements.

**Example 3: Gamow-Stern Elevator Problem**

As a last example, consider physicists Gamow and Stern. They both had an office in a building seven stories tall. The building had just one elevator. Gamow was on the second floor, Stern on the sixth. Gamow often wanted to visit his colleague Stern and vice versa. But Gamow felt like the elevator was always going down when it first got to his floor, and he wanted to go up. Stern, ostensibly paradoxically, always felt like the elevator was going up when he wanted to go down. Assuming that the elevator is going up-and-down all day, this makes sense: 5/6 of the other floors relative to Gamow (on the second floor) were above him, so 5/6 of the time the elevator would be on its way down. And the same is true for Stern, albeit in the opposite direction.

Nahin tells us, then, that the probability that the elevator is going down when it gets to Gamow on the second floor is 5/6 (.83333). Interestingly, Gamow and Stern wrote that this probability holds when there is more than one elevator—but they were mistaken. Nahin challenges us to find the probability in the case of two and three elevators. Again, I write R functions with roxygen2 documentation:

#' Simulate the Floor on Elevator Was On, and What Direction It Is Going

#'

#' Given the floor someone is on and the total number of floors in the building,

#' this function returns to a user (a) what floor the elevator was on when

#' a potential passenger hits the button and (b) if the elevator is on its way

#' up or down when it reaches the potential passenger.

#'

#' @param f The floor a someone wanting to ride the elevator is on

#' @param h The total number of floors in the building; its height

#' @return A named numeric vector, indicating where the elevator started from

#' when the person waiting hit the button as well as if the elevator is going

#' down when it reaches that person (1 if yes, 0 if not)

sim\_lift <- function(f, h) {

floors <- 1:h

start <- sample(floors[floors != f], 1)

going\_down <- start > f

return(c(start = start, going\_down = going\_down))

}

#' Simulate Direction of First-Arriving Elevator

#'

#' This function uses sim\_lift to simulate N number of elevators. It takes the

#' one closest to the floor of the person who hit the button and returns

#' whether (1) or not () that elevator was going down.

#'

#' @param n Number of elevators

#' @param f The floor a someone wanting to ride the elevator is on

#' @param h The total number of floors in the building; its height

#' @return 1 if the elevator is on its way down or 0 if its on its way up

sim\_gs <- function(n, f, h) {

tmp <- sapply(seq\_len(n), function(zzz) sim\_lift(f, h))

return(tmp[2, which.min(abs(tmp["start", ] - f))])

}

First, let’s make sure sim\_lift gives us about 5/6 (.83333):

set.seed(1839)

iter <- 2000000

mean(sapply(seq\_len(iter), function(zzz) sim\_lift(2, 7)[[2]]))

## [1] 0.8334135

Great. Now, we can run the simulation for when there are two and three elevators:

set.seed(1839)

results <- sapply(seq\_len(iter), function(zzz) {

c(sim\_gs(2, 2, 7), sim\_gs(3, 2, 7))

})

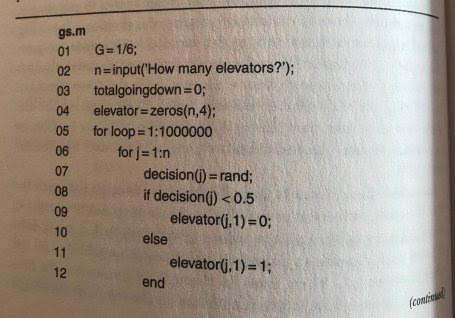
rowMeans(results)

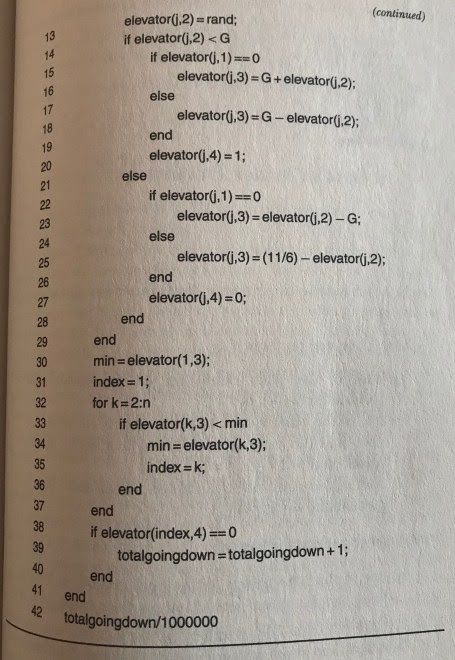
## going\_down going\_down

## 0.7225405 0.6480000

Nahin tells us that the theoretical probability of the elevator going down with two elevators is .72222 and three is .648148; our simulation adheres close to this.

I prefer my modular R functions to Nahin’s MATLAB solution:





**Conclusion**

Simulate data! Use it for power simulations, testing assumptions, verifying models, and solving fun puzzles. But make it modular by writing a few R functions, with documentation, and combine them all in a call to an apply-family function to do them in a way that is readable, clean, and easier to debug and modify.