Exploring Simulations

Simulating Games of Chance in R

Gaston Sanchez

Contents

About

	roll1	roll2	roll3	roll4
game1	6	2	1	2
game2	4	4	1	6
game3	5	3	5	3
game4	5	5	1	4
game5	3	2	1	4

This manuscript describes a couple of examples for implementing simulations of games of chance.

Chapter 1

Introduction

We are going to use a classic example in the history of probability: the problem posed by French gambler Antoine Gombaud (1607-1684), better known by his *nome de plume* "Chevalier De Méré". By the way, he was not a nobleman, but just an amateur mathematician. Most important, for the history of mathematics and probability, he was an avid gambler.

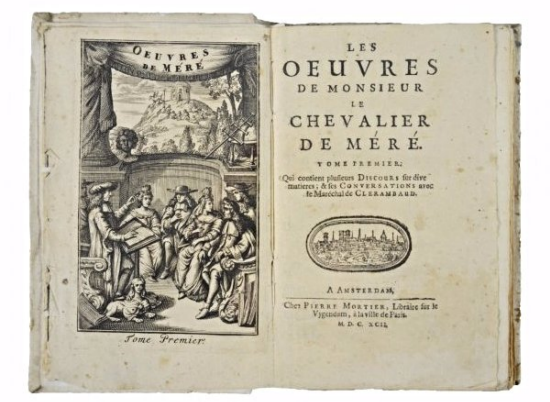


Figure 1.1: Old manuscript with Antoine Gombaud's texts

Specifically, let's focus on one type of gambling scenario which we will refer to as "Game-A"

1.1 Description of Game-A

This is a fairly simple game of dice in which you roll a die four times, and you win if you get at least one six.

Figure 1.2: Rolling 4 dice in Game version A

Think about what could happen when rolling one die four times. For instance, some possible outcomes may be:

- `1, 3, 5, 2` (you lose)
- `2, 4, 6, 5` (you win)
- `5, 6, 3, 6` (you win)
- `4, 2, 1, 3` (you lose)
- `1, 3, 5, 2` (you win)

This is how De Méré was reasoning about Game-A:

- the chance of getting a six in one roll of a die is 1/6 (this is correct)
- in four rolls of a die, the chance of getting one six would be $4 \times 1/6 = 4/6 = 2/3$ (this is incorrect)

Based on this (incorrect) reasoning, he assumed that the odds were definitely in his favor. Interestingly, despite De Méré's faulty reasoning about the probability of getting one six in four rolls, he was able to make money by playing this game.

Why was he able to make money? Let's find out the probability of winning in this type of game.

1.2 Probability of winning Game-A

Finding the probability of winning Game-A translates into finding the probability of getting at least one six in four rolls of a (fair) die:

$$Prob(\text{winning Game-A}) = Prob(\text{getting at least one 6 in four rolls})$$

This probability can easily be calculated with the *complementary probability rule* which states that:

> the probability of one event is equal to 1 minus the probability of its complement event.

Or in symbols:

$$Prob(\text{E}) = 1 - Prob(\text{Not E})$$

and equivalently:

$$Prob(\text{Not E}) = 1 - Prob(\text{E})$$

where:

- E and Not E are complement events (i.e. they cannot occur simultaneously)

A classic example of this type of complement events is "E = getting heads when flipping a coin" and "Not E = getting tails when flipping a coin".

$$Prob(\text{getting heads}) = 1 - Prob(\text{getting tails}) \text{and} Prob(\text{getting tails}) = 1 - Prob(\text{getting heads})$$

So, if we consider event "E = getting at least one six in four rolls", then its complement event is "Not E = getting no six in four rolls"

1.2.1 No six in four rolls

The probability of getting no six in four rolls involves not getting a six in the first roll, *AND* not getting a six in the second roll, *AND* not getting a six in the third roll, *AND* not getting a six in the fourth roll. Because each roll is independent from the others, we can express this probability as the product of the probability in each roll:

$$Prob(\text{no 6 in 4 rolls}) = Prob(\text{no 6 in 1st roll}) \times \cdots \times Prob(\text{no 6 in 4th roll})$$

In any single roll, the probability of getting no six is:

$$Prob(\text{no six in one roll}) = \frac{5}{6}$$

Therefore:

$$\begin{aligned} Prob(\text{no six in 4 rolls}) &= Prob(\text{no 6 in 1st roll}) \times \cdots \times Prob(\text{no 6 in 4th roll}) \\ &= \frac{5}{6} \times \frac{5}{6} \times \frac{5}{6} \times \frac{5}{6} \\ &= \left(\frac{5}{6}\right)^4 \\ &= \frac{625}{1296} = 0.482253 \end{aligned}$$

Consequently, the probability of winning Game-A is:

$$\begin{aligned} Prob(\text{at least one six in 4 rolls}) &= 1 - Prob(\text{no six in 4 rolls}) \\ &= 1 - \left(\frac{5}{6}\right)^4 \\ &= 0.517747 \end{aligned}$$

Note that this probability of 0.517747 is not that different from 0.5 (just slightly greater). But the difference is enough so that if you gamble in this game, you should expect to have a positive gain … **in the long run**.

Chapter 2

Coding Game-A

In the preceding chapter we described Game-A, and we were able to derive the probability of winning this game:

$$\begin{aligned} Prob(\text{at least one six in 4 rolls}) &= 1 - Prob(\text{no six in 4 rolls}) \\ &= 1 - \left(\frac{5}{6}\right)^4 \\ &= 0.517747 \end{aligned}$$

In this example, we are able to find the exact probability associated to the event of "winning Game-A". However, not all probability problems have an analytical solution. When this is the case, we often rely on computers to implement simulations that allow us to find an approximate solution, and have a better understanding of the long-term systematic patterns that arise in random processes.

To go ahead with our exploration of simulations, let us pretend that the probability of winning Game-A had no analytical solution. Instead, let's see how to use R for running various simulations to get an idea of the probabilities involved behind Game-A.

2.1 Rolling one die

Because Game-A involves rolling a die four times, the first step is to create—in a computational sense—an object die, and learn how to roll it.

Perhaps the most straightforward way to create a `die` object is with a numeric vector. One option for this is with a numeric sequence as follows:

```
die = 1:6

die
```

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

Once we have an object `die`, how do we roll it? Rolling a die, computationally speaking, implies getting a sample of size 1 from the elements in vector `die`. To do this in R, we can use the function `sample()`. This function takes an input vector and draws a random sample—of a given size—from the vector's elements.

```
# random seed (for reproducibility purposes)
set.seed(5)
sample(die, size = 1)
```

```
## [1] 2
```

Because `sample()` is a function that generates random numbers, every time you invoke it you will get a different output.

```
# another "roll"
sample(die, size = 1)
```

```
## [1] 3
```

```
# one more "roll"
sample(die, size = 1)
```

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

To be able to reproduce results with any random generator function, you can use `set.seed()` to set the so-called *random seed* that the algorithms behind `sample()` and friends employ to generate those values. By setting this seed, you can obtain the same reproducible output:

```
# random seed (for reproducibility purposes)
set.seed(5)
sample(die, size = 1)
```

```
## [1] 2
```

2.2 Rolling dice with `sample()`

What about rolling a pair of dice? We could repeat the `sample()` command twice:

```
set.seed(5)
roll1 = sample(die, size = 1)
roll2 = sample(die, size = 1)
```

To avoid repeating the `sample()` command multiple times we can also change the value of the `size` argument in `sample()`:

```
set.seed(5)
sample(die, size = 2)
```

```
## [1] 2 3
```

The issue, not evident here, is that `sample()`–by default—draws samples **without replacement**. This means that if we want to get a sample of size six, we would get a simple rearrangement of the elements in `die`, which may not be what we really want:

```
set.seed(5)
sample(die, size = 6)
```

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

Even worse, for bigger samples of size 7 or greater, we would run into some limitations:

```
set.seed(5)
sample(die, size = 7)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the popu
```

So, how do we draw samples **with replacement**? To draw samples with replacement we need to use the `replace` argument as follows:

```
set.seed(5)
sample(die, size = 2, replace = TRUE)
```

```
## [1] 2 3
```

2.3 Rolling four dice

So far, so good. We have a mechanism to simulate rolling a die four times:

```
set.seed(20)
four_rolls = sample(die, size = 4, replace = TRUE)
four_rolls
```

```
## [1] 6 3 2 1
```

With an output vector of four rolls, the next step involves determining if any of the numbers is a six. To do this, we use a comparison to see which elements in `four_rolls` are equal to 6:

```
four_rolls == 6
```

```
## [1]  TRUE FALSE FALSE FALSE
```

This comparison returns a logical vector, which we can then pass to `sum()` in order to count the number of `TRUE` values:

```
count_sixes = sum(four_rolls == 6)
count_sixes
```

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

Because all we care about is knowing if there's at least one six, we can also use the `any()` function; here are a couple of examples of what this function does when testing equality to 6:

```
any(c(6, 3, 2, 1) == 6)
```

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

```
any(c(5, 3, 2, 1) == 6)
```

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

```
any(c(4, 6, 2, 6) == 6)
```

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

If `any()` returns `TRUE`, we win that game. If it returns `FALSE`, we lose.

2.4 Playing Game-A once

To summarize, here's some code that allows us to play Game-A once, determining the winning status:

```
# playing Game-A once
set.seed(20)
four_rolls = sample(die, size = 4, replace = TRUE)
win = any(four_rolls)
win
```

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

Chapter 3

Playing Game-A a Few Times

By now you know that Game-A involves rolling a (fair) die four times, and you win if you get at least one six.

Figure 3.1: Rolling 4 dice in Game version A

You've also learned that we can write some code to play this game once, determining the winning status:

```
# playing Game-A once
set.seed(20)
die = 1:6
four_rolls = sample(die, size = 4, replace = TRUE)
win = any(four_rolls)
win
```

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

3.1 Playing Game-A Five Times

We are ready to write code and use it to play Game-A five times

```
set.seed(133)

die = 1:6

game_1 = sample(die, size = 4, replace = TRUE)
game_2 = sample(die, size = 4, replace = TRUE)
game_3 = sample(die, size = 4, replace = TRUE)
game_4 = sample(die, size = 4, replace = TRUE)
game_5 = sample(die, size = 4, replace = TRUE)

# win (or lose)?
wins = c(
  "game_1" = any(game_1 == 6),
  "game_2" = any(game_2 == 6),
  "game_3" = any(game_3 == 6),
  "game_4" = any(game_4 == 6),
  "game_5" = any(game_5 == 6))

# number of wins
total_wins = sum(wins)
total_wins
```

```
## [1] 3
```

```
# proportion of wins
prop_wins = sum(wins) / 5
prop_wins
```

```
## [1] 0.6
```

As you can tell, the above code works, but there is a substantial amount of unnecessary repetition. Imagine copying-pasting the `sample()` command if we wanted to play Game-A 50 times, or 100 times, or 1000 times.

3.2 Using a `for()` loop

The above code generates the five following games (each game with 4 rolls)

	roll1	roll2	roll3	roll4
game1	1	6	4	1
game2	5	1	1	6
game3	2	5	5	4
game4	2	6	6	1
game5	4	1	1	1

To avoid repeating the same commands multiple times, we can take advantage of loops, for example with a `for()` loop.

Instead of using individual vectors `game_1`, `game_2`, etc, we are going to use a matrix to store the rolls of all games. One option for this matrix is to initialize it in a way that rows correspond to games, while columns correspond to rolls:

	roll1	roll2	roll3	roll4
game1				
game2				
game3				
game4				
game5				

Figure 3.2: Structure of output matrix with 5 rows and 4 columns.

The idea is to have an iterative process in which each iteration is associated to a game. In other words, at each iteration we play a single game, and we store the output of the rolls in the associated row of the matrix. Here's how.

```
set.seed(133)

die = 1:6
number_games = 5

# initialize output matrix (full of zeros)
games = matrix(0, nrow = number_games, ncol = 4)

# each iteration corresponds to a single game
for (game in 1:number_games) {
  games[game, ] = sample(die, size = 4, replace = TRUE)
}

rownames(games) = paste0("game", 1:number_games)
colnames(games) = paste0("roll", 1:4)
games
```

```
##       roll1 roll2 roll3 roll4
## game1     1     6     4     1
```

```
## game2     5     1     1     6
## game3     2     5     5     4
## game4     2     6     6     1
## game5     4     1     1     1
```

Observe the calls of `rownames()` and `colnames()` to assign names to both the rows and the columns of the `games` matrix. In turn, the row and column names are created with `paste0()` to generate the necessary character vectors.

3.3 Playing Game-A 10 times

Now that we have our `for()` loop in place, we can easily increase the number of games. For instance, consider 10 games. Taking into account the preceding code chunk, all we have to do is change the value of `number_games`

```
set.seed(133)

die = 1:6
number_games = 10

# initialize output matrix
games = matrix(0, nrow = number_games, ncol = 4)

for (game in 1:number_games) {
  games[game, ] = sample(die, size = 4, replace = TRUE)
}

rownames(games) = paste0("game", 1:number_games)
colnames(games) = paste0("roll", 1:4)
games
```

```
##        roll1 roll2 roll3 roll4
## game1      1     6     4     1
## game2      5     1     1     6
## game3      2     5     5     4
## game4      2     6     6     1
## game5      4     1     1     1
## game6      4     3     4     2
## game7      6     3     1     3
## game8      4     2     1     1
## game9      1     6     2     1
## game10     5     1     6     5
```

As you can tell, we now get a matrix `games` with 10 rows and 4 columns.

We still need to determine the status of each game: do we win? do we lose?

Conceptually, we could write another `for()` loop to determine the status of each

game: win or lose; something like this

```
# initialize (logical) vector
wins = logical(length = number_games)

for (game in 1:number_games) {
  wins[game] = any(games[game, ] == 6)
}
wins
```

```
##  [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE
```

What are we doing in the preceding loop? Simply put, we are applying the `any()` function to every row of matrix `games` to see if any of the row elements are equal to six. This is illustrated in the following diagram:

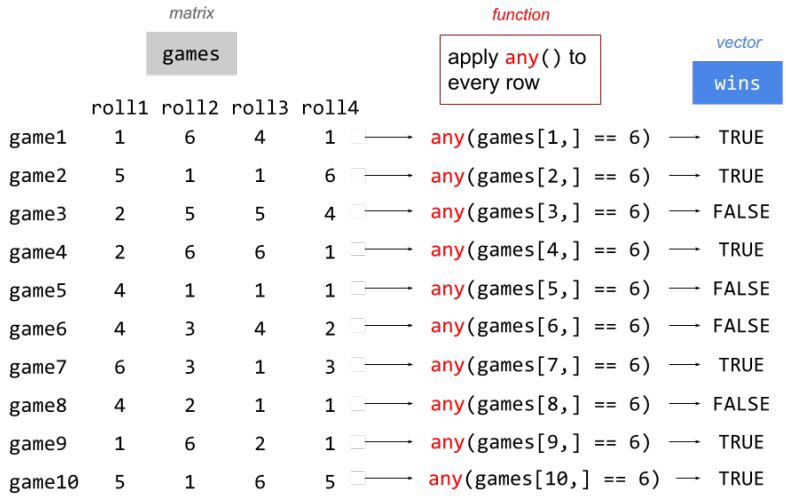


Figure 3.3: Diagram depicting the application of any() to all the rows of matrix games.

While there is nothing conceptually wrong with the above `for()` loop, we can advantage of other functions in R to help us write code in a more compact way. Let's learn about this topic in the next chapter.

Chapter 4

Vectorized loops with `apply()`

We finish the last chapter writing code that simulates playing Game-A 10 times. For recap purposes, the implemented code is displayed below:

```
set.seed(133)

die = 1:6
number_games = 10

# initialize output matrix
games = matrix(0, nrow = number_games, ncol = 4)

for (game in 1:number_games) {
  games[game, ] = sample(die, size = 4, replace = TRUE)
}

rownames(games) = paste0("game", 1:number_games)
colnames(games) = paste0("roll", 1:4)
games
```

The punchline of this piece of code has to do with the `for()` loop, storing the outputs of each game in the rows of the `games` matrix.

Additionally, we also wrote a second `for()` loop to determine whether each game—each row in `games`—had at least one six; this was done with the `any()` function. This is illustrated in the following diagram:

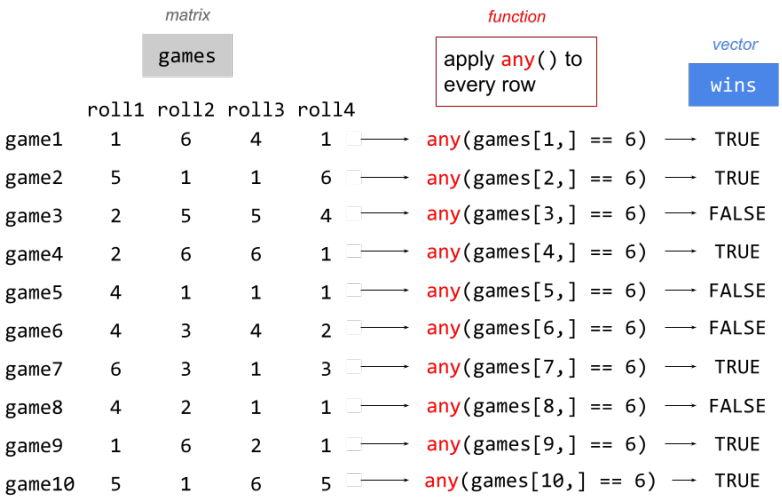


Figure 4.1: Diagram depicting the application of any() to all the rows of matrix games.

4.1 Function `apply()`

Instead of writing a loop to see which games are wins, and which games are losses, we can take advantage of a very interesting function called `apply()`, which R users refer to as a *vectorized loop* function.

As the name indicates, `apply()` lets you **apply** a function to the elements of a matrix. The elements of a matrix can be:

- its rows: `MARGIN = 1`
- its columns: `MARGIN = 2`
- both (rows & cols): `MARGIN = c(1, 2)`

For example, say you want to get the `sum()` of all the elements in each row of `games`:

```
# row sum
apply(X = games, MARGIN = 1, FUN = sum)
```

```
##  game1  game2  game3  game4  game5  game6  game7  game8  game9 game10
##     12     13     16     15      7     13     13      8     10     17
```

We pass three inputs to `apply()`. The first ingredient is the input matrix, the second ingredient specifies the `MARGIN` value, and the third ingredient is the function to be applied.

Here's another example. Say you want to obtain the product of all the elements in each column of `games`:

```
# column product
apply(X = games, MARGIN = 2, FUN = prod)
```

```
## roll1 roll2 roll3 roll4
## 38400 19440  5760   720
```

Or what if you want to get the minimum in each row of `games`:

```
# row minimum
apply(X = games, MARGIN = 1, FUN = min)
```

```
##  game1  game2  game3  game4  game5  game6  game7  game8  game9 game10
##      1      1      2      1      1      2      1      1      1      1
```

4.2 Anonymous functions and `apply()`

Some times, there is no built-in function to do what we want. For instance, say you want to obtain the **range** of each row, that is, the maximum minus the minimum. R has a `range()` function but it does not return a single value, it just gives you the min and the max of an input vector:

```
game_1 = c(1, 6, 4, 1)
range(game_1)
```

```
## [1] 1 6
```

If you want the range, you need to compute the `max()` minus the `min()`

```
game_1 = c(1, 6, 4, 1)
range_1 = max(game_1) - min(game_1)
range_1
```

```
## [1] 5
```

Because R does not have a built-in function that returns the range, we need to provide this function for the `FUN` argument of `apply()`. When the function to be provided is fairly simple, we can create an **anonymous function** inside `apply()`:

```
# row ranges (with anonymous function)
apply(
  X = games,
  MARGIN = 1,
  FUN = function(x) max(x) - min(x))
```

```
##  game1  game2  game3  game4  game5  game6  game7  game8  game9 game10
##      5      5      3      5      3      2      5      3      5      5
```

The reason why the provided function to FUN is called an anonymous function is because the created function has no name.

An alternative option is to first create a function outside apply(), and then pass this function like any other function. This alternative is often preferred when the body of the function to be passed to apply() involves several lines of code.

```
# auxiliary function to compute range
vector_range = function(x) {
  max(x) - min(x)
}

# row ranges
apply(
  X = games,
  MARGIN = 1,
  vector_range)
```

```
##  game1  game2  game3  game4  game5  game6  game7  game8  game9 game10
##      5      5      3      5      3      2      5      3      5      5
```

4.2.1 Number of wins with apply()

Because there's no default function that computes if any() element of a vector is equal to six, we need to create an *anonymous* function for the FUN argument:

```
wins = apply(
  X = games,
  MARGIN = 1,
  FUN = function(x) any(x == 6))

wins
```

```
##  game1  game2  game3  game4  game5  game6  game7  game8  game9 game10
##   TRUE   TRUE  FALSE   TRUE  FALSE  FALSE   TRUE  FALSE   TRUE   TRUE
```

We can now compute the proportion of wins:

```
prop_wins = sum(wins) / number_games
prop_wins
```

```
## [1] 0.6
```

Chapter 5

Playing Game-A 100 times

With all the coding elements that we have so far, it's time to play Game-A 100 times, and see what the proportion of wins turns out to be:

```
# random seed (for reproducibility purposes)
set.seed(753)

# main inputs
die = 1:6
number_games = 100

# initialize output matrix
games = matrix(0, nrow = number_games, ncol = 4)

# playing Game-A several times
for (game in 1:number_games) {
  games[game, ] = sample(die, size = 4, replace = TRUE)
}

rownames(games) = paste0("game", 1:number_games)
colnames(games) = paste0("roll", 1:4)

# determine each game's win-or-lose output
wins = apply(
  X = games,
  MARGIN = 1,
  FUN = function(x) any(x == 6))

# total proportion of wins
prop_wins = sum(wins) / number_games
prop_wins
```

```
## [1] 0.5
```

5.1 Plotting Cumulative Gains

It would be nice to visualize the sequence of wins and losses. To make things more interesting, let's assume that you are paid $1 if you win a game, but also that you pay $1 if you lose a game. That is:

- gain `1` if you win a game
- gain `-1` if you lose a game

```
# vector of gains
gains = rep(-1, number_games)  # initialize with all -1 elements
gains[wins] = 1                # switch to +1 for every win

# cumulative gains
cumulative_gains = cumsum(gains)
```

As you can tell, in this particular simulation of 100 games, the total proportion of wins is 0.5, and the total gain is 0. You didn't gain any money, but you didn't lose either.

Here's the graph using base `"graphics"` functions:

```
plot(1:number_games, cumulative_gains, type = 'l')
abline(h = 0, col = "tomato", lty = 2)
```

Chapter 6

Running Various Simulations

A single series of 100 games may not be enough to fully understand the probability behavior of Game-A. So let's see how to run various simulations, all of them involving playing 100 games.

Let's start small. Meaning, let's begin with a handful of games, and a few simulations. This will allow us to get a better idea of the objects and commands we need to use in order to later generalize things with more games and simulations.

Our starting point will involve 3 simulations, each one consisting of 5 games as illustrated in the picture below.

Simulation 1

game1	6	2	1	2
game2	4	4	1	6
game3	5	3	5	3
game4	5	5	1	4
game5	3	2	1	4

Simulation 2

roll1	roll2	roll3	roll4
1	5	3	2
4	4	6	3
3	2	2	1
3	2	5	3
5	1	5	2

Simulation 3

5	3	3	1
1	5	6	2
1	1	4	6
5	3	2	6
6	3	1	4

Here's some naive code to run these simulations. We are momentarily considering this piece of code to get a high-level intuition of the things we might need later to write more efficient commands.

```
# ------------------------------------
# three simulations, each of 5 games
# ------------------------------------
```

```r
set.seed(753)

# main inputs
die = 1:6
number_games = 5

# simulation 1
games_sim1 = matrix(0, nrow = number_games, ncol = 4)
for (game in 1:number_games) {
  games_sim1[game, ] = sample(die, size = 4, replace = TRUE)
}
wins_sim1 = apply(games_sim1, 1, function(x) any(x == 6))
prop_wins_sim1 = sum(wins_sim1) / number_games


# simulation 2
games_sim2 = matrix(0, nrow = number_games, ncol = 4)
for (game in 1:number_games) {
  games_sim2[game, ] = sample(die, size = 4, replace = TRUE)
}
wins_sim2 = apply(games_sim2, 1, function(x) any(x == 6))
prop_wins_sim2 = sum(wins_sim2) / number_games


# simulation 3
games_sim3 = matrix(0, nrow = number_games, ncol = 4)
for (game in 1:number_games) {
  games_sim3[game, ] = sample(die, size = 4, replace = TRUE)
}
wins_sim3 = apply(games_sim3, 1, function(x) any(x == 6))
prop_wins_sim3 = sum(wins_sim3) / number_games


# summary: proportion of wins
c('sim1' = prop_wins_sim1,
  'sim2' = prop_wins_sim2,
  'sim3' = prop_wins_sim3)
```

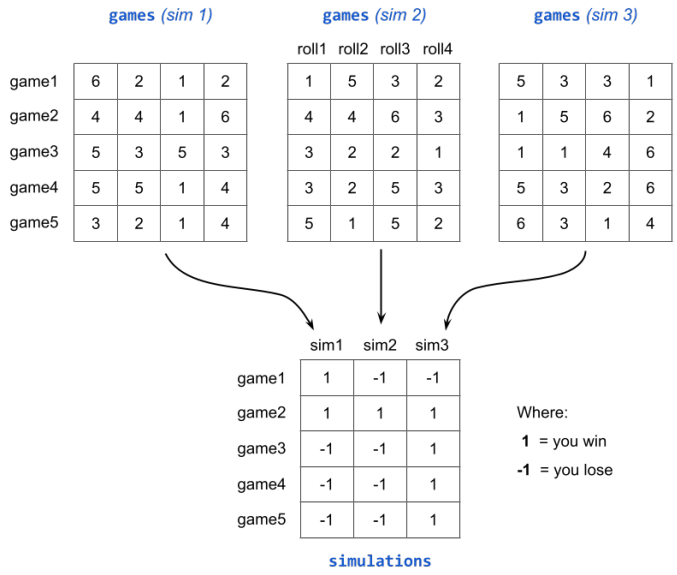
```
## sim1 sim2 sim3
##  0.4  0.2  0.8
```

This piece of code has a lot of unnecessary redundancy. But let's ignore this fact for a second. In each simulation, we create a matrix `games_sim` to store the rolls of each game. Then we use `apply()` to determine which games are wins (`wins_sim`), and finally we calculate the proportion of wins (`prop_wins_sim`).

If you inspect the vectors `wins_sim1`, `wins_sim2`, and `wins_sim3`, you'll see that these are of `"logical"` data-type. `TRUE` means that a given game is a win, whereas `FALSE` means lose.

Alternatively, instead of dealing with the logical vectors `wins_sim`, we can convert their values into numbers: `1` instead of `TRUE`, and `-1` instead of `FALSE`. Why do we need this change from logical to numbers? Quick answer, you don't really need it. But as we'll see, working with `1` and `-1` is more convenient later down the road when we calculate expected gains.

The diagram below depicts this idea of storing the end result of each game into `1` (win) or `-1` (lose), for every simulation:



6.1 Embedded `for()` loops

Let's take the preceding naive code and try to make it more compact. We are going to use embedded `for()` loops: one of them to take care of the simulations, and the other one to take care of the games.

The idea is to iterate through each simulation: 1, 2, and 3. And then, in a given simulation, we iterate through each game: 1, 2, 3, 4, and 5.

```
set.seed(753)

# main inputs
die = 1:6
number_games = 5
```

```
number_sims = 3

# matrix to store simulation outputs
simulations = matrix(0, nrow = number_games, ncol = number_sims)

# 1st loop) iterate through each simulation
for (sim in 1:number_sims) {
  games = matrix(0, nrow = number_games, ncol = 4)

  # 2nd loop) iterate through each game
  for (game in 1:number_games) {
    games[game, ] = sample(die, size = 4, replace = TRUE)
  }
  any_sixes = apply(games, 1, function(x) any(x == 6))
  simulations[ ,sim] = ifelse(any_sixes, 1, -1)
}

rownames(simulations) = paste0("game", 1:number_games)
colnames(simulations) = paste0("sim", 1:number_sims)
simulations
```

```
##       sim1 sim2 sim3
## game1    1   -1   -1
## game2    1    1    1
## game3   -1   -1    1
## game4   -1   -1    1
## game5   -1   -1    1
```

What's going on with this code? Before starting the iterations, we create a matrix `simulations` which will store the numeric values of all games, and all simulations. The rows of this matrix are associated to the games, while the columns are associated to the simulations. At the end of the iterative process, this matrix will be populated with `1` (win game) and `-1` (lose game).

The first `for()` loop iterates through the simulations. At the beginning of each simulation, an auxiliary `games` matrix is initialized, which then gets populated in the embedded `for()` loop. This second `for()` loop iterates through the games, obtaining the rolls of the five games. After the games are completed, we use `apply()` to determine which games are wins (i.e. have at least one 6), and then we convert the `TRUE`'s and `FALSE`'s into `1` and `-1`, respectively, by using `ifelse()`. This numeric vector is stored in the corresponding column of the `simulations` matrix.

Having obtained the matrix `simulations`, we then proceed to count the number of wins `total_wins` (see code below). Notice the use of `apply()` to count the number of wins for each column of `simulations`. The last step involves computing the proportion of wins `prop_wins`:

```
total_wins = apply(simulations, 2, function(x) sum(x==1))
prop_wins = total_wins / number_games
prop_wins
```

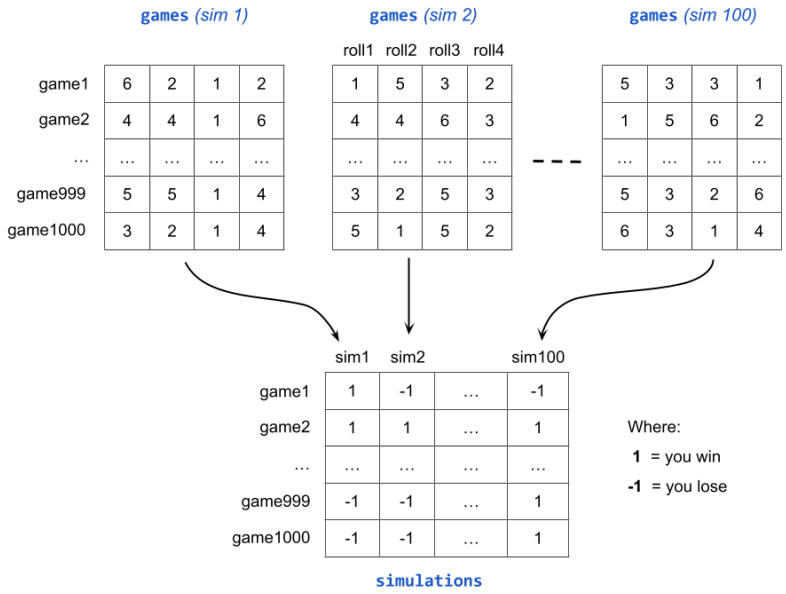
```
## sim1 sim2 sim3
##  0.4  0.2  0.8
```

Chapter 7

Running Even More Simulations

In the previous section we discussed one possible strategy to run three simulations, each one consisting of five games. From the programming point of view, the core of this strategy is the utilization of embedded `for()` loops.

Now, three simulations and five games is obviously not enough. To get a better feeling of the probability of winning Game-A—approximated by the proportion of wins—we need to take things to the next level: with more simulations, and many more games.



The following code uses 1000 games and 100 simulations. This allows us to play Game-A for a total of $1,000 \times 100 = 100,000$ times. With this many games, we can safely calculate the average proportion of wins, and use this value to estimate the probability of winning Game-A:

```
set.seed(753)

# main inputs
die = 1:6
number_games = 1000
number_sims = 100

# matrix to store simulation outputs
simulations = matrix(0, nrow = number_games, ncol = number_sims)

for (sim in 1:number_sims) {
  games = matrix(0, nrow = number_games, ncol = 4)
  for (game in 1:number_games) {
    games[game, ] = sample(die, size = 4, replace = TRUE)
  }
  any_sixes = apply(games, 1, function(x) any(x == 6))
  simulations[ ,sim] = ifelse(any_sixes, 1, -1)
}

rownames(simulations) = paste0("game", 1:number_games)
colnames(simulations) = paste0("sim", 1:number_sims)
```

```
total_wins = apply(simulations, 2, function(x) sum(x==1))
prop_wins = total_wins / number_games

# mean win-proportion
mean(prop_wins)
```

```
## [1] 0.51674
```

As you can tell, the average proportion of wins turns out to be 0.51674 which is pretty close to the theoretical probability of winning Game-A

$$\begin{aligned} Prob(\text{at least one six in 4 rolls}) &= 1 - Prob(\text{no six in 4 rolls}) \\ &= 1 - \left(\frac{5}{6}\right)^4 \\ &= 0.517747 \end{aligned}$$

Of course, if you change the random seed and rerun this simulation, you will very likely obtain a slightly different proportion of wins. But still close enough to the actual theoretical probability of winning Game-A.