# 13. Simulation

Solutions to Swirl's R Programming Exercises

07-09-2022

Acknowledgements: R Language Concepts and code questions (with minor modifications) are used here from the swirl package. https://www.r-project.org/nosvn/pandoc/swirl.html

Important note: We don't require to use library(swirl) and swirl() here because we are not going to run the R script in RStudio Console.

One of the great advantages of using a statistical programming language like R is its vast collection of tools for simulating random numbers.

This lesson assumes familiarity with a few common probability distributions, but these topics will only be discussed with respect to random number generation. Even if you have no prior experience with these concepts, you should be able to complete the lesson and understand the main ideas.

The first function we'll use to generate random numbers is sample(). Use ?sample to pull up the documentation.

```
?sample
```

```
## starting httpd help server ... done
```

Let's simulate rolling four six-sided dice: sample(1:6, 4, replace = TRUE).

```
sample(1:6, 4, replace = TRUE)
```

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

Now repeat the command to see how your result differs. (The probability of rolling the exact same result is (1/6)^4 = 0.00077, which is pretty small!)

```
sample(1:6, 4, replace = TRUE)
```

```
## [1] 1 4 6 6
```

sample(1:6, 4, replace = TRUE) instructs R to randomly select four numbers between 1 and 6, WITH replacement. Sampling with replacement simply means that each number is "replaced" after it is selected, so that the same number can show up more than once. This is what we want here, since what you roll on one die shouldn't affect what you roll on any of the others.

Now sample 10 numbers between 1 and 20, WITHOUT replacement. To sample without replacement, simply leave off the 'replace' argument.

```
sample(1:20, 10)
```

```
##  [1]  9  2  6 13  1 12  4  8  3 15
```

Since the last command sampled without replacement, no number appears more than once in the output.

LETTERS is a predefined variable in R containing a vector of all 26 letters of the English alphabet. Take a look at it now.

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

The sample() function can also be used to permute, or rearrange, the elements of a vector. For example, try sample(LETTERS) to permute all 26 letters of the English alphabet.

```
sample(LETTERS)
```

```
##  [1] "D" "Y" "B" "U" "S" "A" "X" "P" "W" "O" "R" "J" "V" "M" "G" "F" "E" "T" "L"
## [20] "K" "Q" "C" "H" "Z" "N" "I"
```

This is identical to taking a sample of size 26 from LETTERS, without replacement. When the 'size' argument to sample() is not specified, R takes a sample equal in size to the vector from which you are sampling.

Now, suppose we want to simulate 100 flips of an unfair two-sided coin. This particular coin has a 0.3 probability of landing 'tails' and a 0.7 probability of landing 'heads'.

Let the value 0 represent tails and the value 1 represent heads. Use sample() to draw a sample of size 100 from the vector c(0,1), with replacement. Since the coin is unfair, we must attach specific probabilities to the values 0 (tails) and 1 (heads) with a fourth argument, prob = c(0.3, 0.7). Assign the result to a new variable called flips.

```
flips <- sample(c(0,1), size = 100, replace = TRUE, prob = c(0.3, 0.7))
```

View the contents of the flips variable.

```
flips
```

```
##   [1] 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1
##  [38] 1 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1
##  [75] 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0
```

Since we set the probability of landing heads on any given flip to be 0.7, we'd expect approximately 70 of our coin flips to have the value 1. Count the actual number of 1s contained in flips using the sum() function.

```
sum(flips)
```

```
## [1] 75
```

A coin flip is a binary outcome (0 or 1) and we are performing 100 independent trials (coin flips), so we can use rbinom() to simulate a binomial random variable. Pull up the documentation for rbinom() using ?rbinom.

```
?rbinom
```

Each probability distribution in R has an r*** function (for "random"), a d*** function (for "density"), a p*** (for "probability"), and q*** (for "quantile"). We are most interested in the r*** functions in this lesson, but I encourage you to explore the others on your own.

A binomial random variable represents the number of 'successes' (heads) in a given number of independent 'trials' (coin flips). Therefore, we can generate a single random variable that represents the number of heads in 100 flips of our unfair coin using rbinom(1, size = 100, prob = 0.7). Note that you only specify the probability of 'success' (heads) and NOT the probability of 'failure' (tails). Try it now.

```
rbinom(1,100,0.7)
```

```
## [1] 73
```

Equivalently, if we want to see all of the 0s and 1s, we can request 100 observations, each of size 1, with success probability of 0.7. Give it a try, assigning the result to a new variable called flips2.

```
flips2 <- rbinom(n = 100, size = 1, prob = 0.7)
```

View the contents of flips2.

```
flips2
```

```
##   [1] 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1
##  [38] 1 1 0 1 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0
##  [75] 0 1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1
```

Now use sum() to count the number of 1s (heads) in flips2. It should be close to 70!

```
sum(flips2)
```

```
## [1] 69
```

Similar to rbinom(), we can use R to simulate random numbers from many other probability distributions. Pull up the documentation for rnorm() now.

```
?rnorm
```

The standard normal distribution has mean 0 and standard deviation 1. As you can see under the 'Usage' section in the documentation, the default values for the 'mean' and 'sd' arguments to rnorm() are 0 and 1, respectively. Thus, rnorm(10) will generate 10 random numbers from a standard normal distribution. Give it a try.

```
rnorm(10, 0, 1)
```

```
##  [1] -0.3634224 -0.4778257 -0.3632362  0.3680531  0.4546152  0.3691083
##  [7]  1.3668744 -0.6722612  0.8028871  1.1280235
```

Now do the same, except with a mean of 100 and a standard deviation of 25.

```
rnorm(10, 100, 25)
```

```
##  [1]  87.15256  78.47470 112.83078 146.34744 112.35412 117.66363 129.75925
##  [8] 103.72941 143.13982 121.57865
```

Finally, what if we want to simulate 100 *groups* of random numbers, each containing 5 values generated from a Poisson distribution with mean 10? Let's start with one group of 5 numbers, then I'll show you how to repeat the operation 100 times in a convenient and compact way.

Generate 5 random values from a Poisson distribution with mean 10. Check out the documentation for rpois() if you need help.

Use rpois(5, 10) to generate 5 random numbers from a Poisson distribution with mean 10.

```
rpois(5, 10)
```

```
## [1] 13 11  6 12  6
```

Now use replicate(100, rpois(5, 10)) to perform this operation 100 times. Store the result in a new variable called my_pois.

```
my_pois <- replicate(100, rpois(5, 10))
```

Take a look at the contents of my_pois.

```
my_pois
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
## [1,]    9   11    5   13   13   11    6   18   13     5     9     9     7    14
## [2,]    7    4   10    9   14    9   10   10   11    13     9     7     9     8
## [3,]    8   17    4   12   10    5   14   13    9     9    14    14    17    11
## [4,]    9    8   14    5   15   12   10   14   10    11    10    13    13    13
## [5,]    9   11   10    8   13    7    9    6   10    13     9    16    11     9
##      [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26]
## [1,]    10    15     6     6    12    10    12     8     8     8     9    11
## [2,]    13    11    18     6     6     7    12     9    17    14    12    12
## [3,]    14     9    11    10     8    10    13     9     7     8     4    10
## [4,]     7    11    12    13     7     9    13     9    11     4     9     5
## [5,]    11    10    12     4    13    10    16     4    10    12    10    15
##      [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35] [,36] [,37] [,38]
## [1,]    10    11     8    13    13    12    12     9    10     8     8     8
## [2,]    10     8    11     8    11    13     6     5    13    11    13    12
## [3,]    10    13     8     4     6    12    13    13    12    18     9    11
## [4,]    11    16    15    11     6     9    15    11    10    13    11     6
## [5,]    13     6    15    10     6     4     9    10     9    14    12    13
##      [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46] [,47] [,48] [,49] [,50]
## [1,]    12     9    11     7    11     8     7    10     7    13     7    17
## [2,]    11    11     9     5    13    11    15     9    13    10    10     6
## [3,]    10     8    12    12    12    10    15     7    11    13    14    11
## [4,]    10    11    10    15    17    12    11    12     9    14    14    10
## [5,]    11    11     8     5    14    13     6    10    11     7    12    15
##      [,51] [,52] [,53] [,54] [,55] [,56] [,57] [,58] [,59] [,60] [,61] [,62]
```

```
## [1,]    12     5     8     2     8     6     9     7     9    11    13    10
## [2,]    13    21    14    17     5    12    14    13    16    11     5    10
## [3,]     8     9     7    12     7     8    13    10     8    11    10     6
## [4,]    11     9    13     9     9    10     4     5    12     8     7    10
## [5,]     7    14     8    13    10     8     7    10    16     7     7     6
##      [,63] [,64] [,65] [,66] [,67] [,68] [,69] [,70] [,71] [,72] [,73] [,74]
## [1,]    11     9     4    16     4    11     9    15     9     9     9     7
## [2,]    14    10     7    12    14    11     8    12     8    14     9     8
## [3,]    10     4     8     8     7    10    13    11     7    13    13    14
## [4,]     9    14    14     8    13     9     9    16     9     8     5    13
## [5,]    10    13    17     3    10    15     8     9    13     9     7    14
##      [,75] [,76] [,77] [,78] [,79] [,80] [,81] [,82] [,83] [,84] [,85] [,86]
## [1,]     6     9     9     5    14     7     8    11    13     8    10    14
## [2,]    13     8     9    11    12    10     9     9     9    10     6     4
## [3,]    11     4     9     7    14     9    11     5     5    11    10    14
## [4,]     5    10     7     7    11    10     8     8    18    14    15    12
## [5,]    14    11    11     8    11     8     5     1    11    15    11     9
##      [,87] [,88] [,89] [,90] [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98]
## [1,]    14    11    11     6    11    10    10    10     9    13     6    16
## [2,]     7     7    11    13     9     7    10    13     8     8    11    15
## [3,]     7     8     9     9     5    14     9    11     6    10    10    17
## [4,]     9    14     9    13     8    15    10    12     7     7     7     9
## [5,]    16    10    19     9     8    10    16    11    12     9     9     7
##      [,99] [,100]
## [1,]     9     10
## [2,]     7     14
## [3,]     9      9
## [4,]    18     10
## [5,]    14      6
```

replicate() created a matrix, each column of which contains 5 random numbers generated from a Poisson distribution with mean 10. Now we can find the mean of each column in my_pois using the colMeans() function. Store the result in a variable called cm.

```
colMeans(my_pois)
```

```
##   [1]  8.4 10.2  8.6  9.4 13.0  8.8  9.8 12.2 10.6 10.2 10.2 11.8 11.4 11.0 11.0
##  [16] 11.2 11.8  7.8  9.2  9.2 13.2  7.8 10.6  9.2  8.8 10.6 10.8 10.8 11.4  9.2
##  [31]  8.4 10.0 11.0  9.6 10.8 12.8 10.6 10.0 10.8 10.0 10.0  8.8 13.4 10.8 10.8
##  [46]  9.6 10.2 11.4 11.4 11.8 10.2 11.6 10.0 10.6  7.8  8.8  9.4  9.0 12.2  9.6
##  [61]  8.4  8.4 10.8 10.0 10.0  9.4  9.6 11.2  9.4 12.6  9.2 10.6  8.6 11.2  9.8
##  [76]  8.4  9.0  7.6 12.4  8.8  8.2  6.8 11.2 11.6 10.4 10.6 10.6 10.0 11.8 10.0
##  [91]  8.2 11.2 11.0 11.4  8.4  9.4  8.6 12.8 11.4  9.8
```

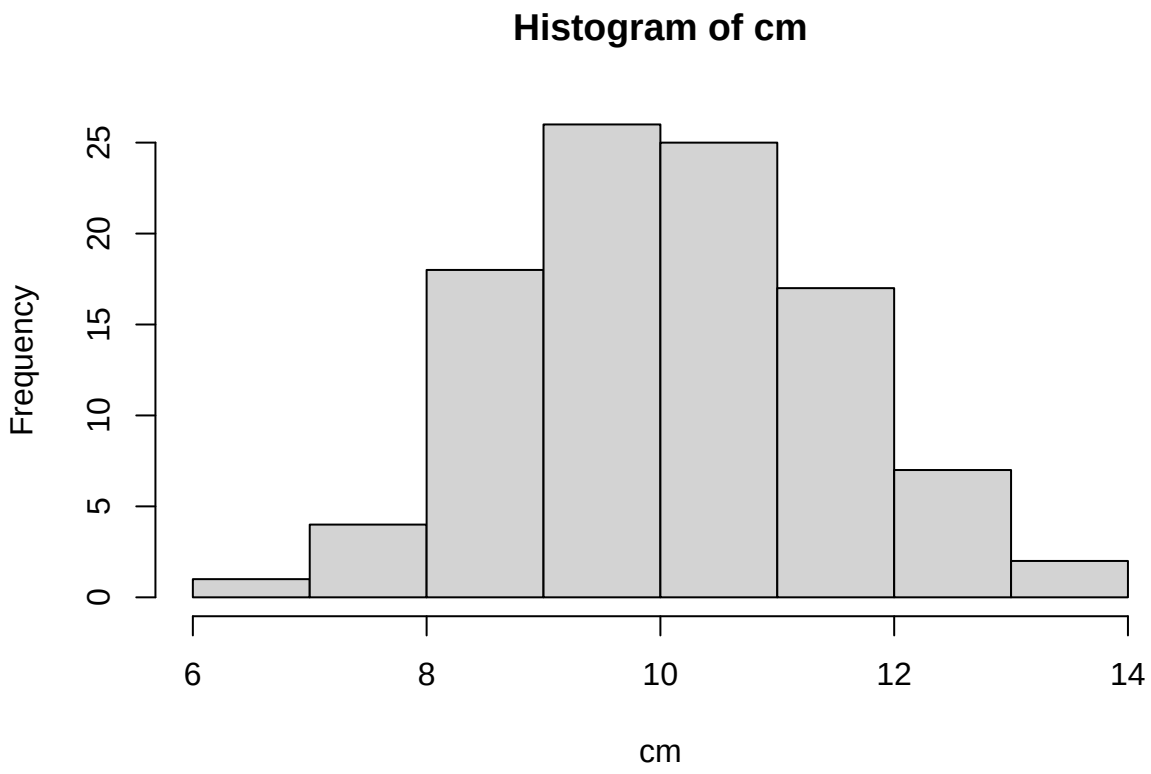
One more time. You can do it! Or, type info() for more options.

Use cm <- colMeans(my_pois) to create a vector of column means, storing the result in cm.

```
cm <- colMeans(my_pois)
```

And let's take a look at the distribution of our column means by plotting a histogram with hist(cm).

```
hist(cm)
```




Looks like our column means are almost normally distributed, right? That's the Central Limit Theorem at work, but that's a lesson for another day! All of the standard probability distributions are built into R, including exponential (rexp()), chi-squared (rchisq()), gamma (rgamma()), .... Well, you see the pattern.

Simulation is practically a field of its own and we've only skimmed the surface of what's possible. I encourage you to explore these and other functions further on your own.