Applied Class #2 - Simulation Basics in R

Monte Carlo Simulation: What & Why?

In Monte Carlo simulation, we use a computer to repeatedly carry out a random experiment, keep track of the outcomes and compute relative frequencies. We then make plots and tables to summarize the results. In a simulation, we **know** the truth. This allows us check how estimators, confidence intervals, and tests perform in finite samples. More generally, we can also check probability calculations by simulation or approximate probabilities that can't be computed analytically.

So here's a dirty secret: computers **cannot generate "truly" random numbers**. Computers are *deterministic*: if we supply the same input, we'll always get the same output. So-called "random" draws made on a computer are really **pseudorandom**, a fancy way of saying "as if random." While they are generated by a deterministic algorithm, pseudorandom numbers are statistically random in that they are indistinguishable from *genuine* random numbers using statistical tests. Fortunately, pseudorandom draws are good enough for all of our purposes here. I'll use the abbreviation RNG for "pseudo-random number generator" below.

`sample()` - Draw from a Vector in R

The simplest command for Monte Carlo simulation in R is `sample()` This function generates a random sample of length `size` from the vector `x`. If `x` is a positive integer rather than a vector, `sample()` draws from the vector `1:x`. By default `sample()` draws *without replacement*; to draw *with replacement* set `replace = TRUE`. By default, elements of `x` are sampled with *equal probability*. If you want to sample elements of `x` with different probabilities, supply these probabilities using the argument `prob`.

```
sample(x, size, replace = FALSE, prob = NULL)
```

Now let's some simple examples. The following command draws two marbles without replacement from a "bowl" with three marbles (really a vector of character data):

```
marbles <- c('red', 'blue', 'green') # a bowl with three marbles
sample(marbles, size = 2) # draw two marbles without replacement
```

```
[1] "blue"  "green"
```

This next example makes five draws without replacement from the vector `1:10`

```
sample(10, size = 5) # five draws without replacement from 1:10
```

```
[1] 8 5 4 1 2
```

while this one samples with replacement:

```
# Sample with replacement
sample(c('Oxford', 'Summer', 'School'), 10, replace = TRUE)
```

```
 [1] "School" "Summer" "School" "School" "Summer" "School" "Oxford" "School"
 [9] "School" "Summer"
```

Finally, we have an example of sampling with unequal probabilities:

```
# Unequal probabilities: P(FALSE) = 0.7
sample(c(TRUE, FALSE), 20, replace = TRUE, prob = c(0.3, 0.7))
```

```
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
[13] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
```

Setting the Seed in R

You probably got different results from me when you ran the code from above. If I were to run the code again on my own computer, I'd get different results too: it's (pseudo)random! But as we've discussed, RNG is *deterministic*. For the purposes of replicating our own work in the future, or replicating someone else's work, it can be helpful to make sure that we obtain the **same random draws** when we re-run a chunk of simulation code. To do this we **set the seed** of the RNG. Remember that computers draws pseudo-random numbers using a deterministic algorithm: the same inputs give the same outputs. The **seed** of an RNG is its **initial condition**. If we set the same seed again, we'll get the same results. To do this in R, we supply a single numeric value (interpreted as an integer) as the argument of the function `set.seed()`. In other words:

```
set.seed(YOUR_SEED_GOES_HERE)
```

Here are a few examples. Notice how the first two calls of `sample(100, 2)` give different results, whereas the third gives the same result as the first. This is because we set the same seed before the third `sample(100, 2)`

```
set.seed(1983)
sample(100, 2)
```

```
[1] 67 18
```

```
sample(100, 2)
```

```
[1] 77 94
```

```
set.seed(1983)
sample(100, 2)
```

```
[1] 67 18
```

In gerneral you should set the seed *once* at the beginning of your simulation study to ensure replicability. If you like, you can even choose a *genuine* random number to use as your seed by visiting random.org!

Mind your Ps and Qs (and Ds and Rs).

Base R has functions for many common random variables and their distributions. Each random variable has a standardized abbreviation: e.g. `norm` for normal. Associated with each of the random variables supported in base R is a set of four related functions. These functions begin with the letters `d`, `p`, `q`, and `r`, indicating the purpose of the function in question:

- `d` = density if continuous, mass function if discrete
- `p` = CDF
- `q` = quantile function (inverse CDF)
- `r` makes random draws

For example, `dnorm()` is the normal density function, `pnorm()` is the normal CDF, `qnorm()` is the normal quantile function, and `rnorm()` make normally distributed pseudorandom draws. The following table lists common random variables and their R commands. If you need to work with a random variable that isn't in this table, I suggest consulting the Cran Task View: Probability Distributions.

R commands	Distribution	R commands	Distribution
d/p/q/rbeta	Beta	d/p/q/rlogis	Logistic
d/p/q/rbinom	Binomial	d/p/q/rlnorm	Log Normal
d/p/q/rcauchy	Cauchy	d/p/q/rnbinom	Negative Binomial
d/p/q/rchisq	Chi-Squared	d/p/q/rnorm	Normal
d/p/q/rexp	Exponential	d/p/q/rpois	Poisson
d/p/q/rf	F	d/p/q/rt	Student's t
d/p/q/rgamma	Gamma	d/p/q/runif	Uniform
d/p/q/rgeom	Geometric	d/p/q/rweibull	Weibull
d/q/p/rhyper	Hypergeometric		Other RVs...

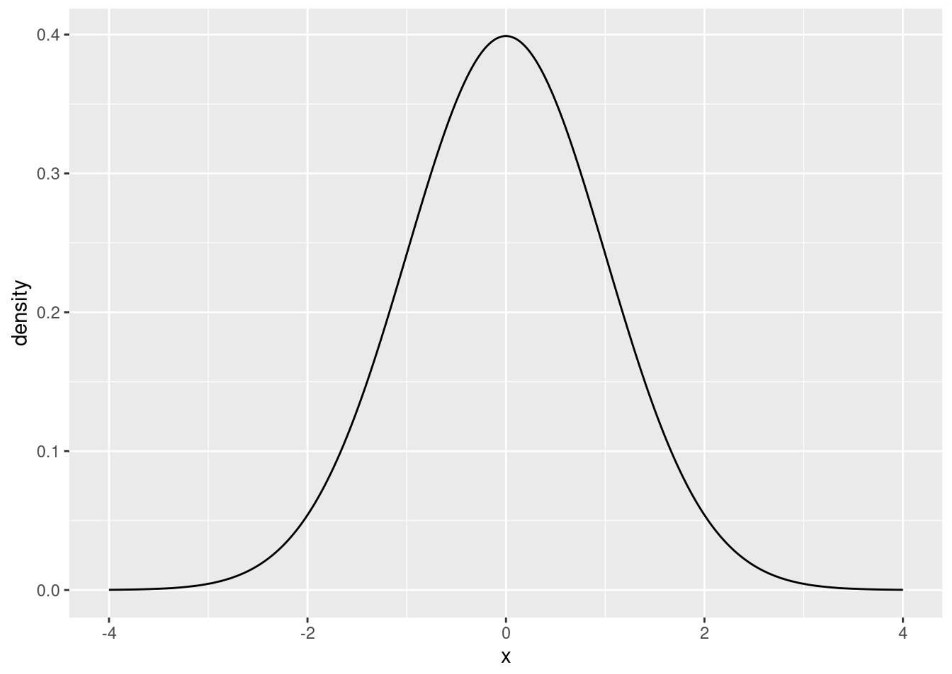
Example: Standard Normal Distribution

To understand how these `d/p/q/r` functions work, we'll look at two common examples. First is the normal distribution. The key thing you need to know about the normal distribution is that R parameterizes it in terms of its mean and **standard deviation** rather than its variance. For example, we can plot the standard normal density $\varphi(\cdot)$ with `dnorm()` as follows:

```
library(tidyverse)
mu <- 0
sigma <- 1

normal_density <- tibble(x = seq(from = -4, to = 4, by = 0.01)) |>
  mutate(density = dnorm(x, mu, sigma))
```

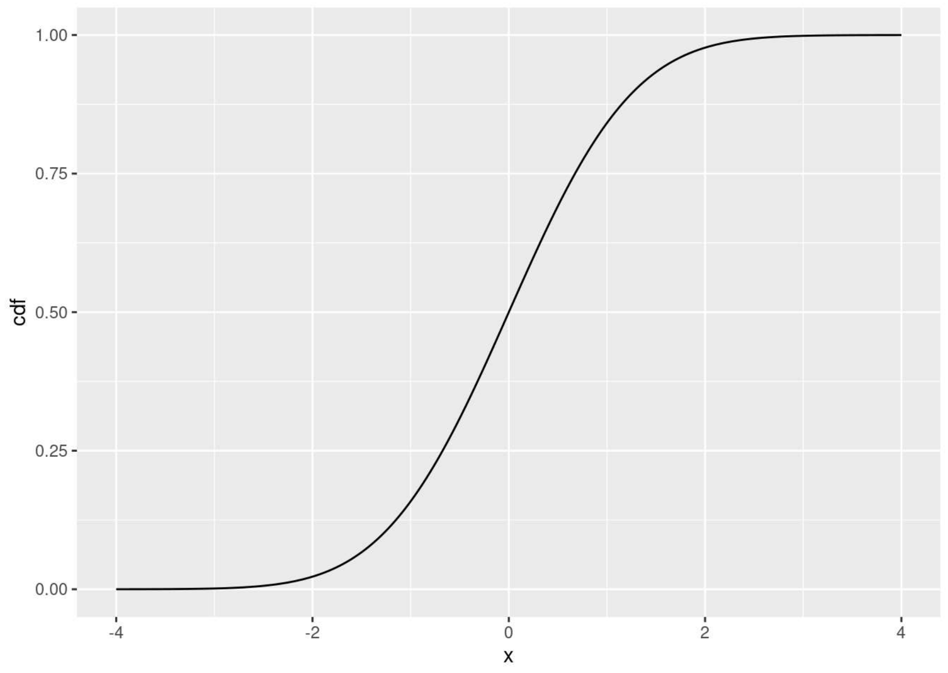
```
normal_density |>
  ggplot(aes(x, density)) +
  geom_line()
```



And we can plot the standard normal CDF $\Phi(\cdot)$ with `pnorm()` as follows, again noting that R parameterizes this distribution in terms of the standard deviation rather than the variance:

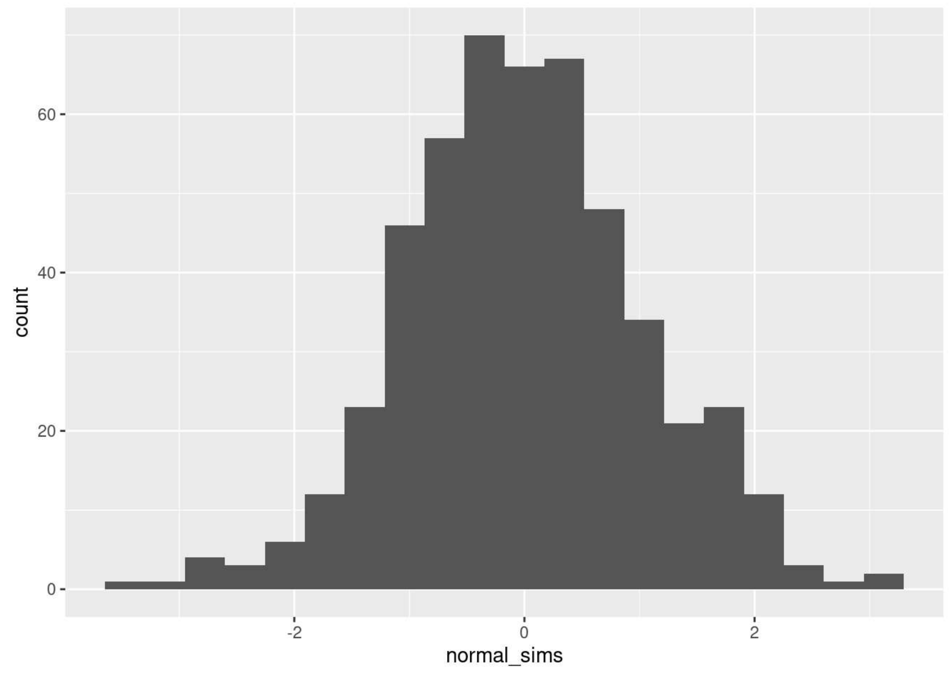
```
normal_cdf <- tibble(x = seq(from = -4, to = 4, by = 0.01)) |>
  mutate(cdf = pnorm(x, mu, sigma))

normal_cdf |>
  ggplot(aes(x, cdf)) +
  geom_line()
```



Finally, we can draw from a standard normal distribution using `rnorm()` as follows:

```
set.seed(1234)
normal_sims <- rnorm(500, mu, sigma)
tibble(normal_sims) |>
  ggplot(aes(x = normal_sims)) +
  geom_histogram(bins = 20)
```



A Very Simple Monte Carlo Simulation

Now that we understand the basics, we're ready to carry out a simple Monte Carlo Simulation: we'll use simulation to approximate the probabilities of every possible sum that you can obtain when rolling a pair of fair, six-sided dice. Here we know what the answer should be, so it will be easy for us to check that our code is working correctly. I'll begin by giving you a **general recipe** that you can use for practically any simulation experiment you will ever need to carry out:

1. Write a function to carry out the experiment *once*.
2. Use iteration to call that function many times.
3. Store and summarize the results.

As described above, you should set the seed **once** before running your simulation study to ensure replicability. Note that, although this recipe can always be applied, it's not the only approach and it may not always be the fastest approach. Still it's a good place to start. Now we'll look at each step in detail for the sum of two dice example.

Step 1: Write a function to carry out the experiment once.

The function `dice_sum()` simulates the experiment of rolling a pari of fair dice and computing their sum. Notice that the function `dice_sum()` has *no arguments*: it doesn't need any.

```
dice_sum <- \() {
# Roll a pair of fair, six-sided dice and return their sum
  die1 <- sample(1:6, 1)
  die2 <- sample(1:6, 1)
  die1 + die2
}

dice_sum()
```

```
[1] 4
```

```
dice_sum()
```

```
[1] 6
```

```
dice_sum()
```

```
[1] 6
```

Step 2: Use iteration to call that function many times.

We *could* use a `for()` loop, but there's a better way: **functional programming**. In our next applied class we'll talk about functional programming in more detail, using the `purrr` package. But for today I want to keep things relatively simple, so rather than trying to explain `purrr` I'll show you how to use the base R function `replicate()`. In effect, `replicate(5, dice_sum())` does the same thing as running `dice_sum()` five times, except that we don't have to do all the typing and R stores all the results for us in a vector:

```
replicate(5, dice_sum())
```

```
[1]  6  9  8  6 10
```

Using `replicate()` we can run our simulation a large number of times and store the results. The vector `sims` contains 10,000 pseudorandom draws from our dice simulation:

```
set.seed(59283)
sims <- replicate(1e4, dice_sum())

head(sims)
```

```
[1] 11  7  7  6  7 11
```

Notice that we still have to set up `f()`. This is because `map_dbl()` requires a function that takes a single argument, but `dice_sum()` doesn't have any arguments. So we trick R into doing what we want by creating a new function with a "dummy" argument. If this is confusing, don't worry about it: just copy the pattern from above.

Step 3: Summarize the results

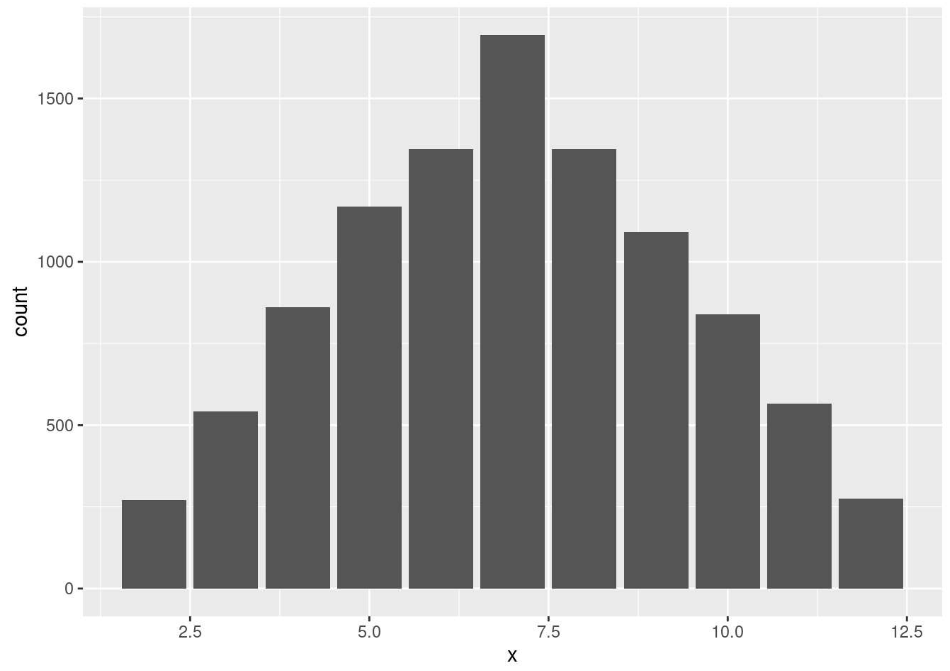
There are lots of different questions we could answer using the vector `sims`. Here's one: what is the probability of getting a seven when you roll two fair dice? Hooray: we got the right answer!

```
## What is the probability of rolling a 7?
mean(sims == 7)
```

```
[1] 0.1695
```

Indeed, our simulation produced the correct probabilities for each possible sum of two dice: the law of large numbers in action!

```
tibble(x = sims) |>
  ggplot(aes(x)) +
  geom_bar()
```



Exercise

This problem was initially posed by the famous 17th century gambler Antoine Gombaud, more commonly known as the Chevalier de Méré. Fermat and Pascal discussed its solution in their legendary correspondence that began the study of probability as we know it. Here's the Chevalier's question:

> Which is more likely: (A) getting at least one six when rolling a single die four times or (B) getting at least one pair of sixes when rolling a pair of dice twenty-four times?

Answer the Chevalier's question using Monte Carlo Simulation. Assume that all dice are fair and six-sided.