Foundational concepts for simulation studies

Populations, samples, pseudo-random number generators, for-loops, and parameter recovery

Ian Hussey

06 Mai, 2024

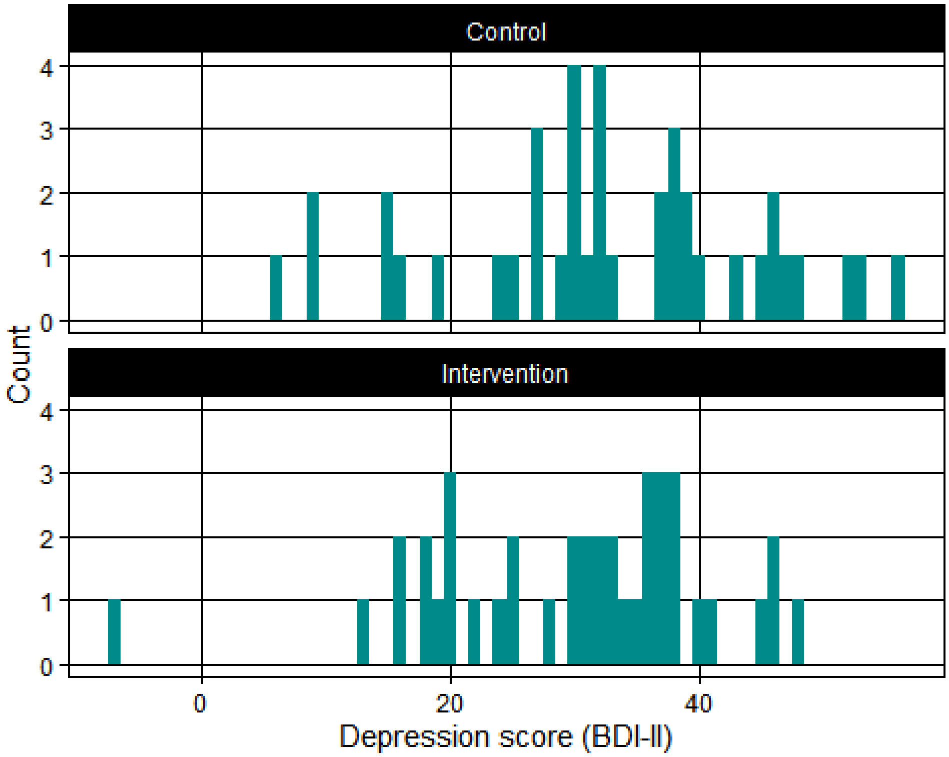
Table of Contents

```r
# dependencies
library(tidyverse)
library(scales)
library(knitr)
library(kableExtra)
library(janitor)
# install.packages("devtools")
# library(devtools)
# devtools::install_github("ianhussey/simulateR")
library(simulateR) # available from github - uncomment the above lines to
install

# set seed
set.seed(42)
```

Why do inferential statistics?

Do these two groups differ from one another? Does the intervention group have a lower average depression score than the control group?

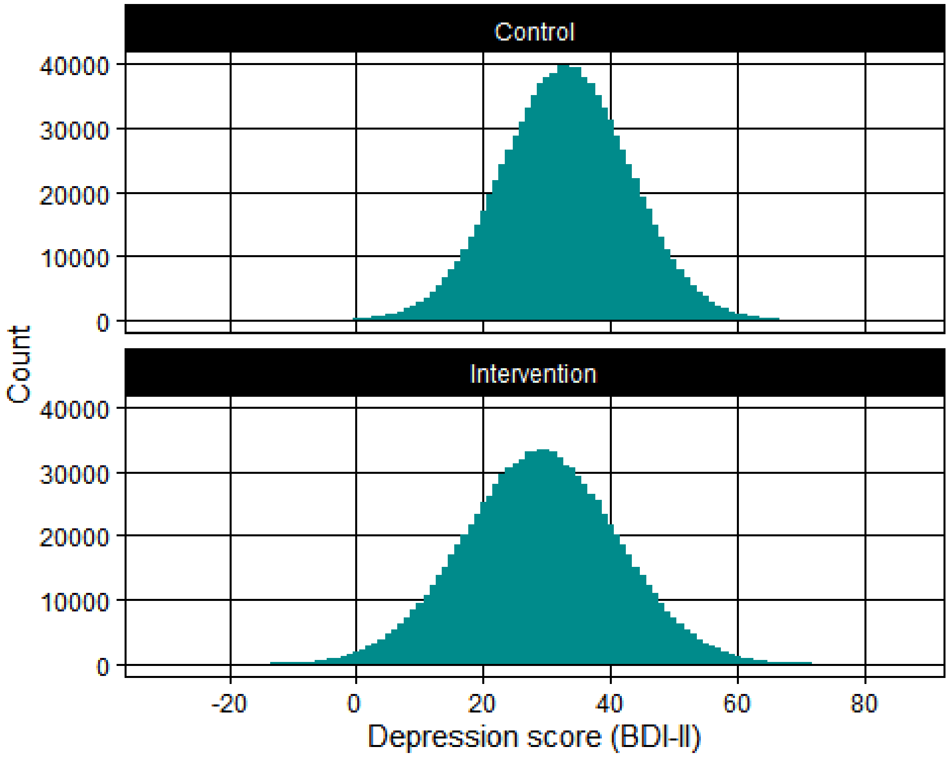
These plots contain data that is simulated but realistic for a hypothetical RCT comparing a psychotherapeutic intervention versus waiting list control. The (hypothetical) outcome variable is a popular measure of depression, the Beck Depression Inventory II.



Statistical inference is about making inferences about populations from limited samples drawn from those populations.

Generally speaking, psychology studies are less interested in the specific participants that we study, and more interested in making generalisations about the populations they are drawn from. E.g., would people in general, other than these participants we studied here, benefit from this therapy?

The previous plots sampled 40 participants per condition from a true population. Imagine we were able to see data from the whole population - which we never can in real life.



It’s easier to see from this plot that the therapy does indeed work. The values are also realistic: the means, SDs, and effect sizes are reasonable for BDI-II scores for an effective therapy. Unfortunately, we almost never have access to the full population we are interested in knowing about. Inferential statistics allow us to use data from smaller samples to make inferences about the larger populations they are drawn from.

Why do stimulation studies?

When you apply an inferential test to real-world data, you are never sure whether the result of your test is truly correct or not, you can only make probability statements about being right (e.g., the long-run false-positive rate, controlled using a test’s alpha value). This makes it hard to know if statistical tests are working correctly as planned, because you never have access to ground-truth.

Simulation studies are very useful because they allow you to construct this ground truth and then apply tests to it. If you’re interested in knowing whether your statistical test is good at detecting population effects of a given size in small sample sizes, you can create an arbitrary number of samples drawn from this precise population.

To take a concrete example: we are often taught that “violating statistical assumptions is bad” in some way. But how severe is the violation, and how negative a consequence does this have? In what contexts, and why? Simulation lets you answer questions like these. In doing so, it can change our understanding of statistical analyses from a set of rules we have learned to a deeper and principled understanding of why we are choosing to analyse data in a given way.

Pseudo-random number generators

Randomness is impossible to achieve. All "random" number generators are actually pseudo-random number generators (PRNGs). Computer scientists and mathematicians spend lots of time trying to increase the randomness of our random number generators, because any degree of predictability adds bias to any models built on them. Pseudo-random numbers are at the core of simulation studies, as they allow us to (pseudo) randomly sample simulated data from known population distributions.

Sampling from uniform distributions

A uniform distributions is when every value is as likely as every other value, and are selected from a given range. E.g., "pick a number between 1 and 10" where the picker is just as likely to say "1" as any other number in that range.

`runif()` is a random generation function (the "r" part) for a uniform distribution (the "unif" part). Not 'run if', which confuses some people.

So, this code, which generates a random number between 1 and 10, will generate 3s just as often as it generates 7s. You can re run this code yourself many times to see it generate different numbers between 1 and 10.

```
runif(n = 1,
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 7
```

Random number generators are not truly random

You don't need to understand how PRNGs work, but you do need to know that the these "random" numbers can be predictably reproduced. The 'seed' value from which a PRNG starts can be set to control which random numbers are generated.

```
set.seed(43) # set the starting seed value for generating the random numbers

runif(n = 1,
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 5
```

```
set.seed(43) # set it again to the same value starting seed value for
generating the random numbers

runif(n = 1,
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 5
```

Note that if you run the function a second time without resetting to a known seed, the second value will be different to the first one.

```
set.seed(43) # set the starting seed value for generating the random numbers

runif(n = 1,
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 5

runif(n = 1,
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 9
```

This is because the Nth value of any sequence from a given seed is knowable, whether its run once or in multiple runs.

```
set.seed(43) # set the starting seed value for generating the random numbers

# generate both of the above numbers in one function call
runif(n = 2, # generate two numbers rather than one
      min = 1,
      max = 10) |>
  round_half_up(0)

## [1] 5 9
```

Sampling from normal distributions

`rnorm()` is a random generation function (the "r" part) for normal distributions (the "norm" part).

Note that normal distributions are sometimes referred to as Gaussian distributions. This can be useful as it can rid us of the impression that Gaussian distributions are typical/standard/default, or that other distributions are abnormal in some way. However, "normal" is more common so we'll use it.

rnorm() is like magic, because it allows you to create data that follows the assumptions of our most common statistical analyses. The difference between simulated data, e.g., from rnorm(), and real data from participants, is that we can know what the real population value - the data generating signals - are in simulated data. Whereas with real participant data we don't ever know this for sure. We use real data to make inferences - best guesses - about (unobserved, unknowable) true populations.

The below code generates data from a normal distribution, where the population mean (usually notated as

$$\mu$$

) is 7.52 and the population standard deviation (usually noted as $\sigma$) is 3.18. Let's sample 100 simulated "participants".

```
#$$\mu$$ -> mu symbol $$\sigma$$ -> sigma symbol

rnorm(n = 100,
      mean = 7.52,
      sd = 3.18)

##   [1]  2.5127580  5.9746233  8.9992922  4.6449681  6.6377637  8.7488614
##   [7]  7.3279149  5.3379484  1.4584850 13.2559560  4.4453442  6.3970839
##  [13] 11.0400612  9.3209115 14.0845018 12.1923034  2.2682295  8.1643840
##  [19]  5.2272841  7.0142457  9.8549099  6.3646957  7.4879010  9.3732909
##  [25]  6.5873431  3.0858640  7.2494402  5.3318568  5.0506602 13.0666590
##  [31]  8.9718465  7.1441995 12.8478098  3.8345012  7.3908623 10.9827326
##  [37] 12.3285839 10.3364826  8.5203292 13.5059072  6.2019522  4.6132806
##  [43]  9.2641947  4.9493314  9.2013831  8.7541560  6.7459866  5.9333882
##  [49]  7.6283154  8.4542556  8.7938039  6.9650353  8.2835166  3.5734530
##  [55]  0.8822097  5.9012722  7.5620074  4.3220909 10.8450172  7.1096607
##  [61]  7.6058013  7.3282377  9.2515998 10.0033800  3.7063326  3.3847643
##  [67]  8.1233128  4.1991932  9.4244811 10.4642459  9.4238010  8.3643321
##  [73]  6.6275344  8.1354815 12.9231734  7.9451509  3.5126656  7.9916667
##  [79]  9.8873840  7.2659647  5.4109653  3.9660905  1.1826540  9.9667953
##  [85] 11.0521923 12.6627685  9.3208589  2.3869794  5.2934703 11.4747428
##  [91] 13.0419833  7.9750700  9.2221276  5.5052833  6.3293344  6.5556286
##  [97] 14.0296200  7.2754612 15.8419109  4.8086489
```

Parameter recovery

But ... how do we know that rnorm() actually does what it claims to? How do we know it generates data with the parameters we tell it to, and from a normal distribution?

-> By checking!

Generate a *very large* sample of participants from a known population (i.e., specified by the arguments given to rnorm()), then quantify if those parameters are what is found in the simulated data.

This ability to compare a known ground truth with what is observed in the simulated data is called **parameter recovery**.

For example, the following code defines a population mean (μ = 7.52) and population SD (σ = 3.18), simulates 1 million participants from this population, and the calculates the mean and SD in the sample. In such a large sample, the sample summary statistics should be almost equal to the population that it was drawn from *if the `rnorm()` function does as it claims to*. This is an extremely simple form of simulation study where, rather than take R for its word that `rnorm()` samples correctly, we test it for ourselves.

```
simulated_scores <- rnorm(n = 1000000, # note that we need lots and lots of
data to get a precise estimate
                          mean = 7.52,
                          sd = 3.18)

mean(simulated_scores) |> round_half_up(2)

## [1] 7.52

sd(simulated_scores) |> round_half_up(2)

## [1] 3.18
```

Having access to the ground truth (the true population effect), i.e., being able to control the data generation process and then check that your tests recover these properties, is at the heart of simulation studies. It is the special magic that allows you to be confident that you understand what a given analysis can and can't do, that you're using it correctly.

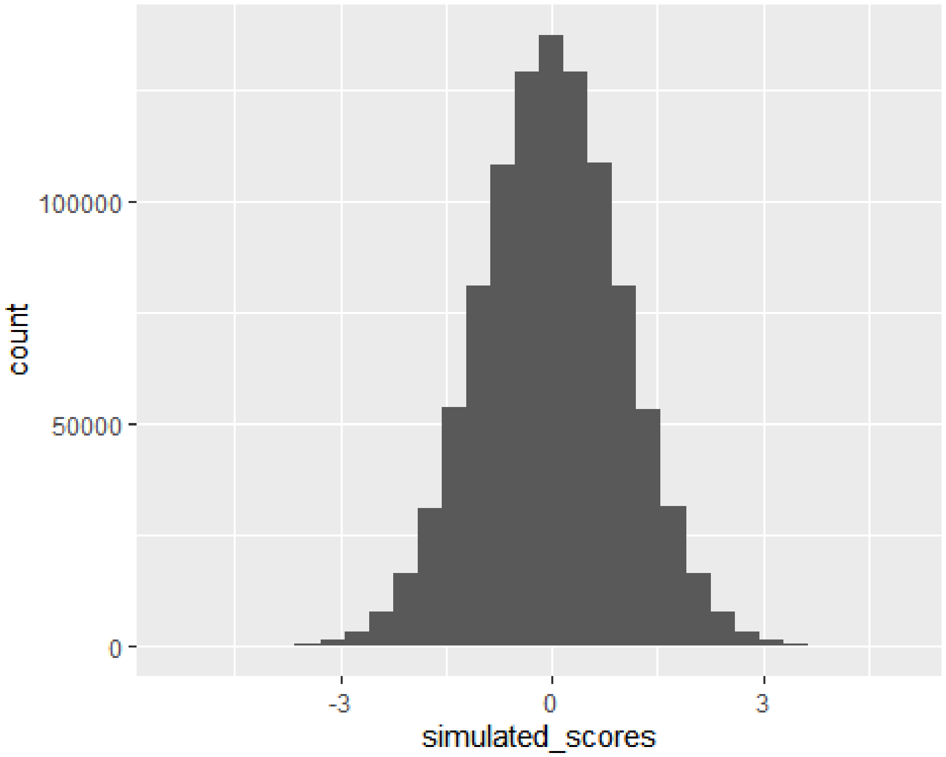
Plotting the normal distribution

Summary statistics like `mean()` and `sd()` allow us to check that we have recovered those population parameters. But `rnorm()` also claims to sample data from a specific distribution: the normal distribution. Rather than take `rnorm()`'s word for this, we still need to examine the distribution of the data it generates to check that it is indeed normally distributed. It is also just generally useful to be able to plot distribution of simulated data. So, I have created some helper functions that allow you to make these plots in just a line or two of code.

Basic ggplot

Ok, but a little ugly. Making it prettier would mean lots more lines of code.

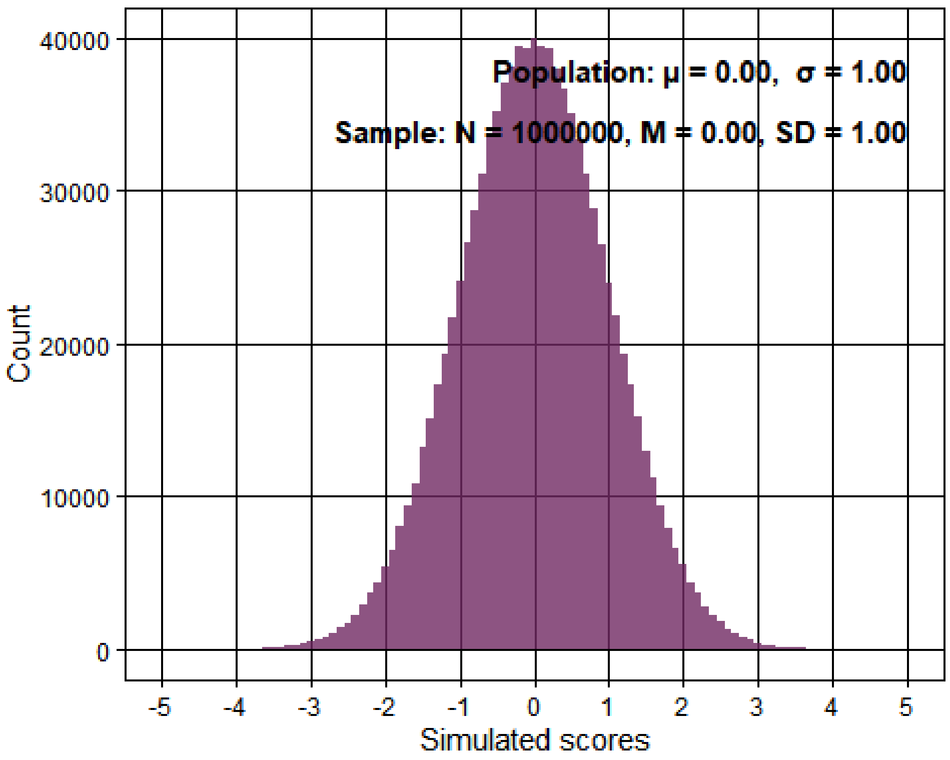
```
simulated_scores <-
  rnorm(n = 1000000, # sample n
        mean = 0, # population mean (μ or mu)
        sd = 1) # population sd (σ or sigma)

dat <- data.frame(simulated_scores = simulated_scores)

ggplot(dat, aes(x = simulated_scores)) +
  geom_histogram()
```



simulateR::rnorm_histogram()

The simulateR R package is in very early development. Its `rnorm_histogram()` function does both data generation and plotting for you. It plots not only the sample summary statistics, but also the parameters used to generate the data.

```
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 1)
```



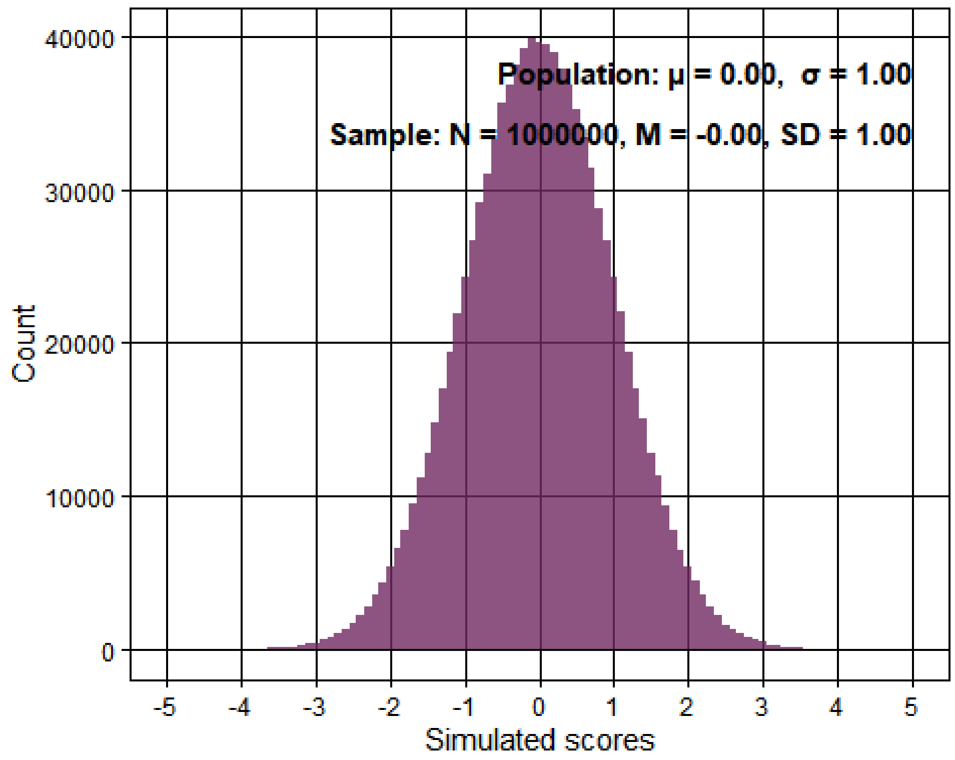
Understanding μ, σ, and n

In order to really understand `rnorm()`'s parameters (μ, σ, and n) it useful to vary them.

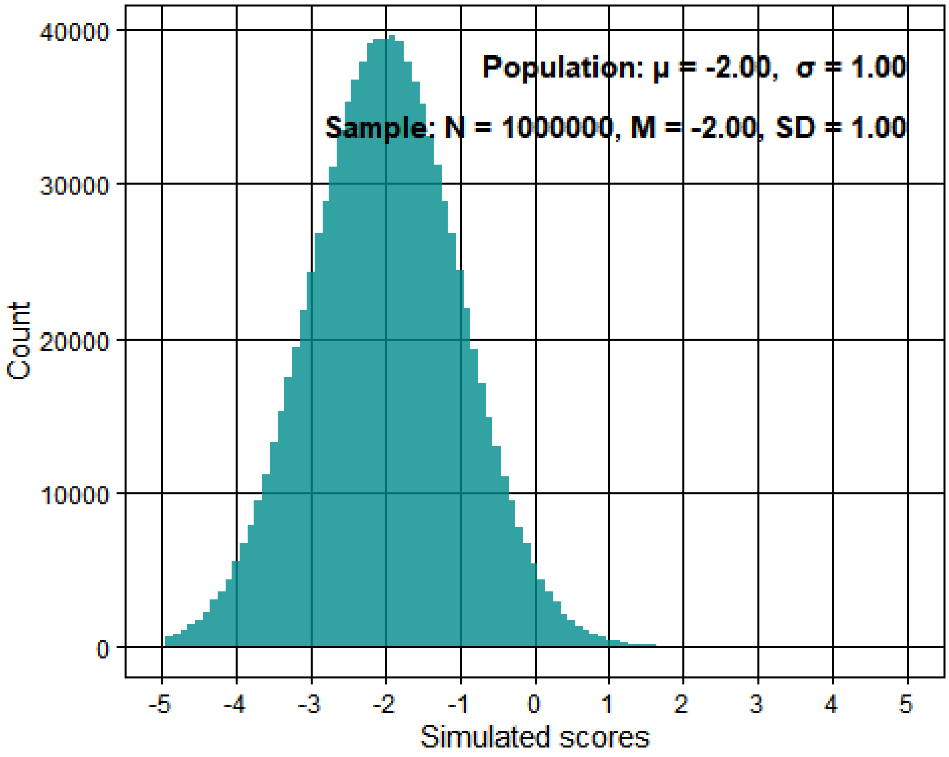
Varying the population mean (μ)

As if wasn't already complicated enough, note that population mean μ) is also sometimes referred to as "location".

```
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 1)
```



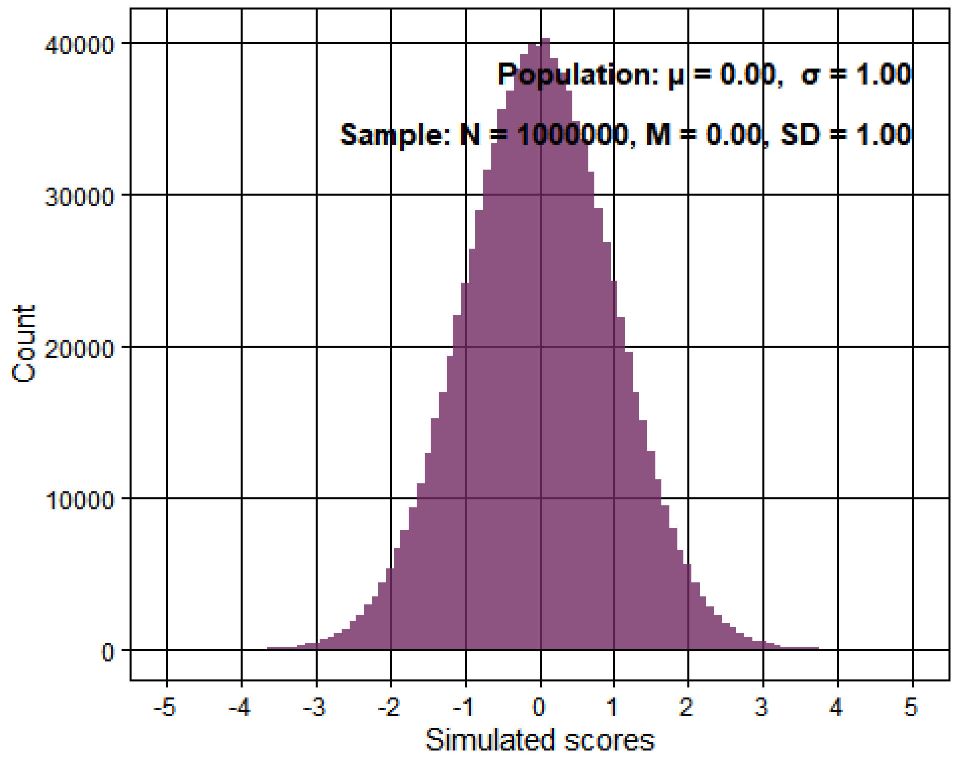
```
rnorm_histogram(n = 1000000,
                mean = -2,
                sd = 1,
                fill = "darkcyan")
```



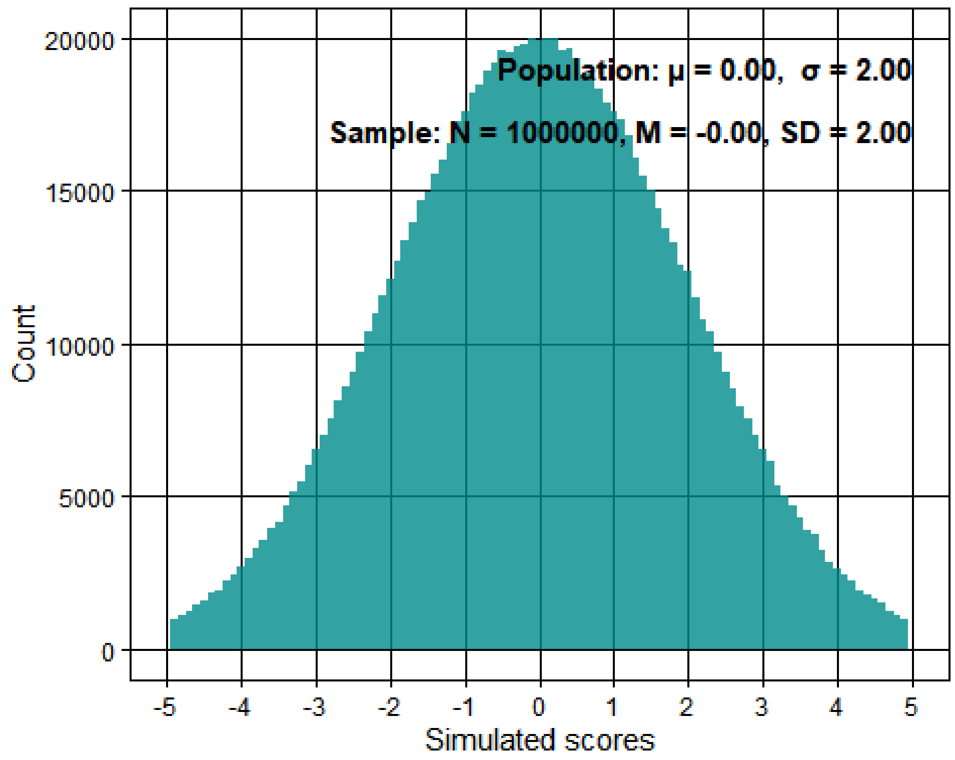
Varying the population SD (σ)

As if wasn't already complicated enough, note that population SD σ) is closely related to the concept of variance, and both are ways of talking about dispersion (i.e., spread) of scores around means.

```r
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 1)
```

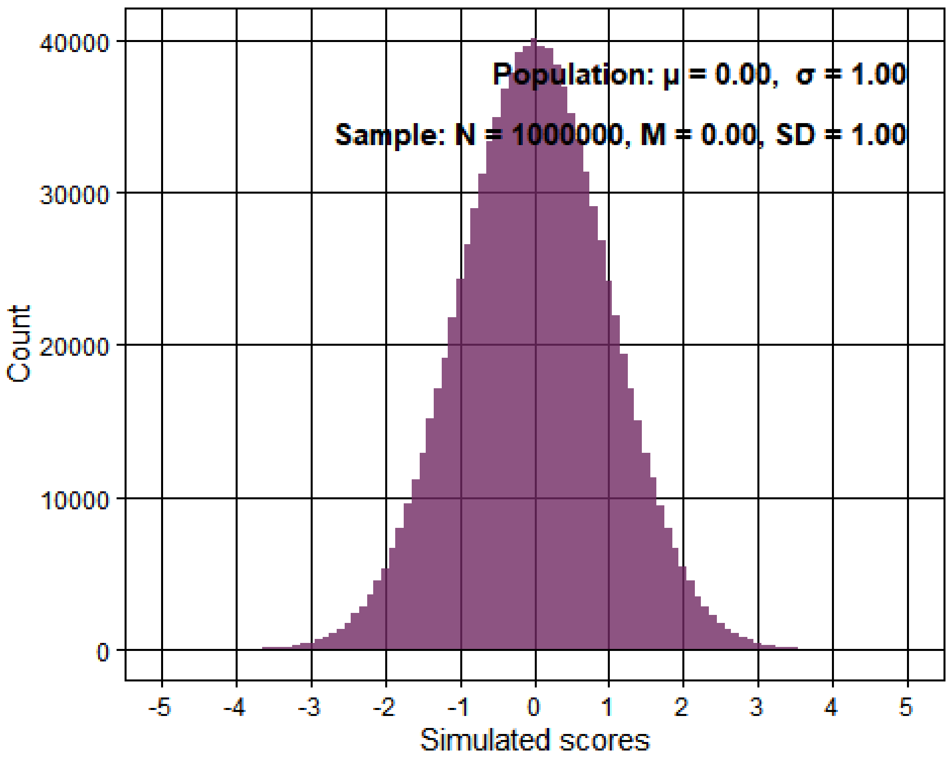


```
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 2,
                fill = "darkcyan")
```

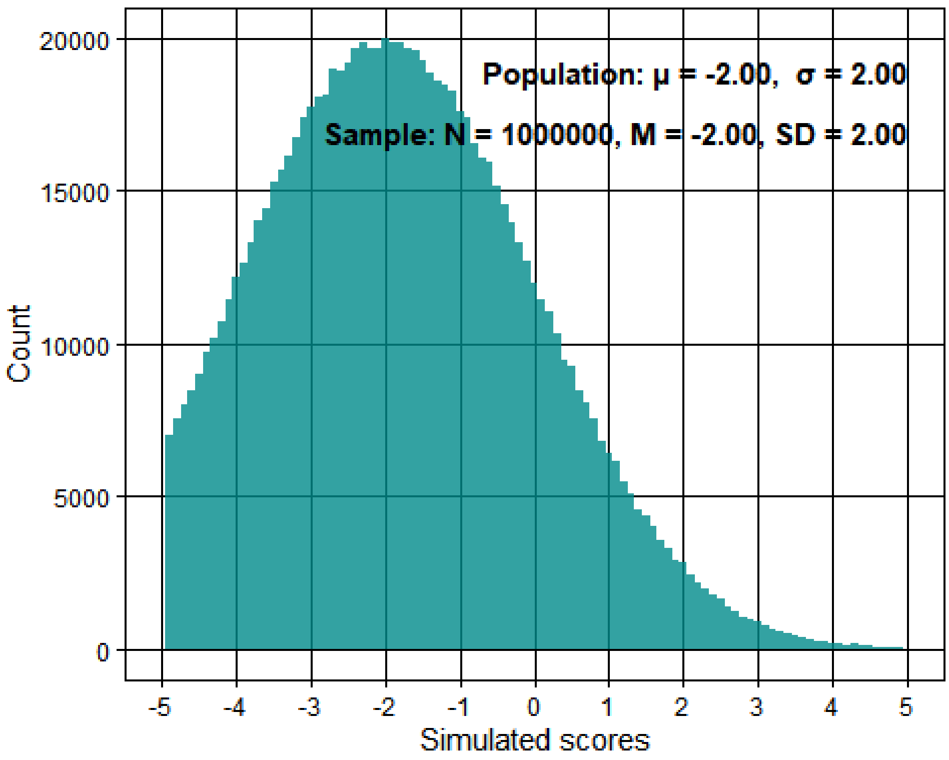


Varying both the population mean (μ) and SD (σ)

```
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 1)
```



```
rnorm_histogram(n = 1000000,
                mean = -2,
                sd = 2,
                fill = "darkcyan")
```

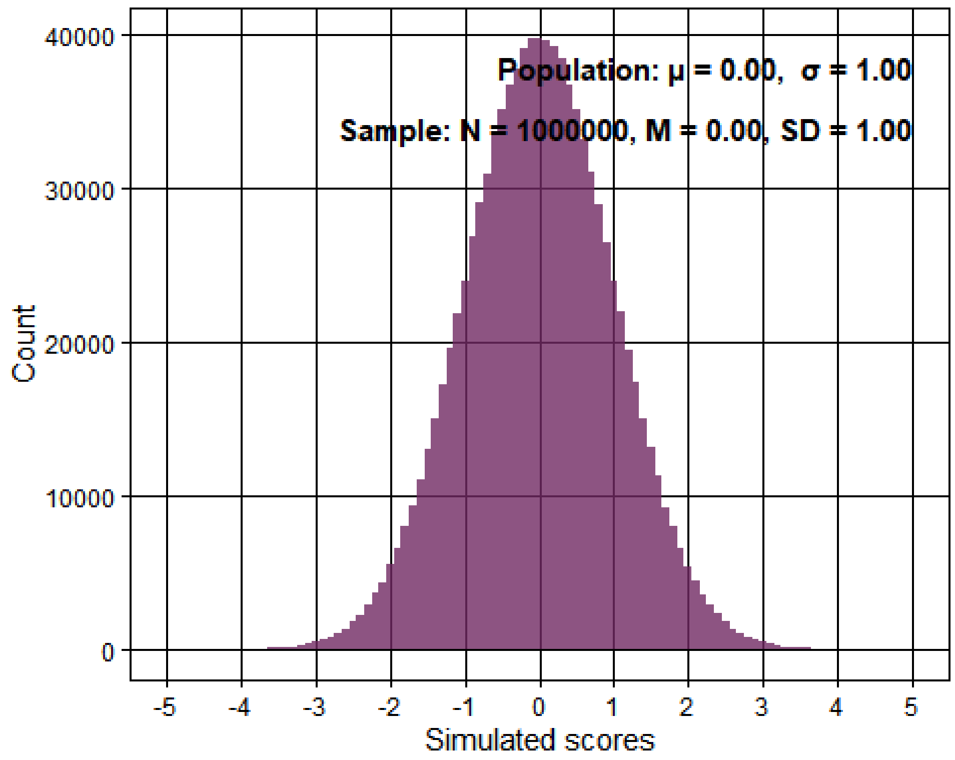


Varying the sample N

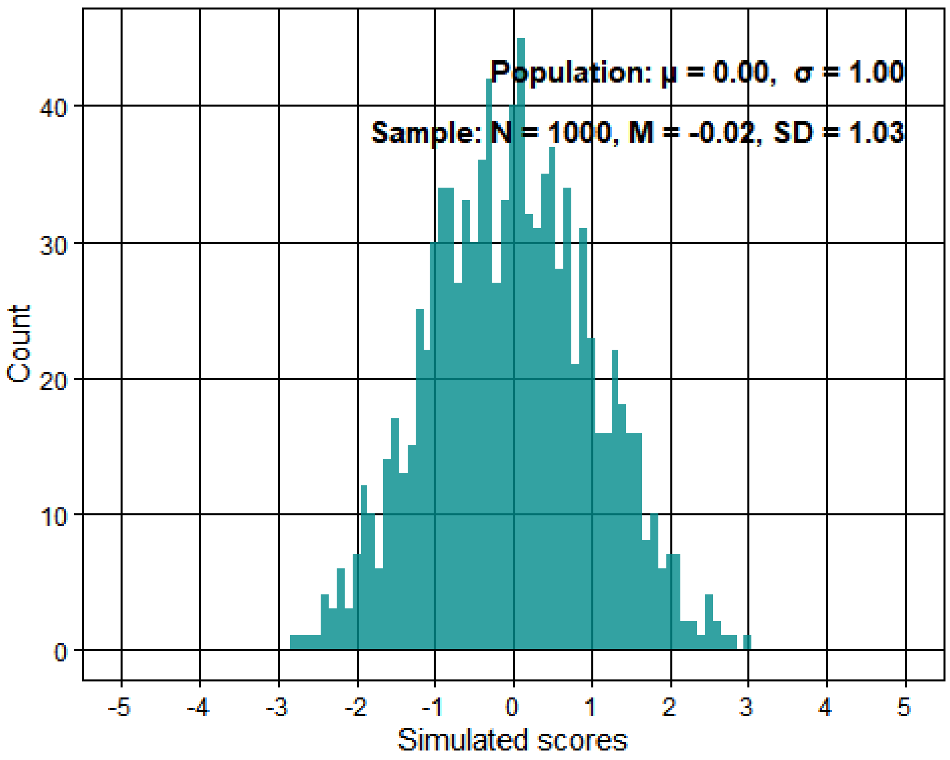
Aside from location and dispersion, we can also change the number of simulated participants we sample from the population.

If we radically lower the sample sizes, from one million to one thousand to one hundred, this will change how rough/granular/noisy the sampled data looks, and how much the sample summary statistics (M and SD) differ from the population parameters (μ and σ).

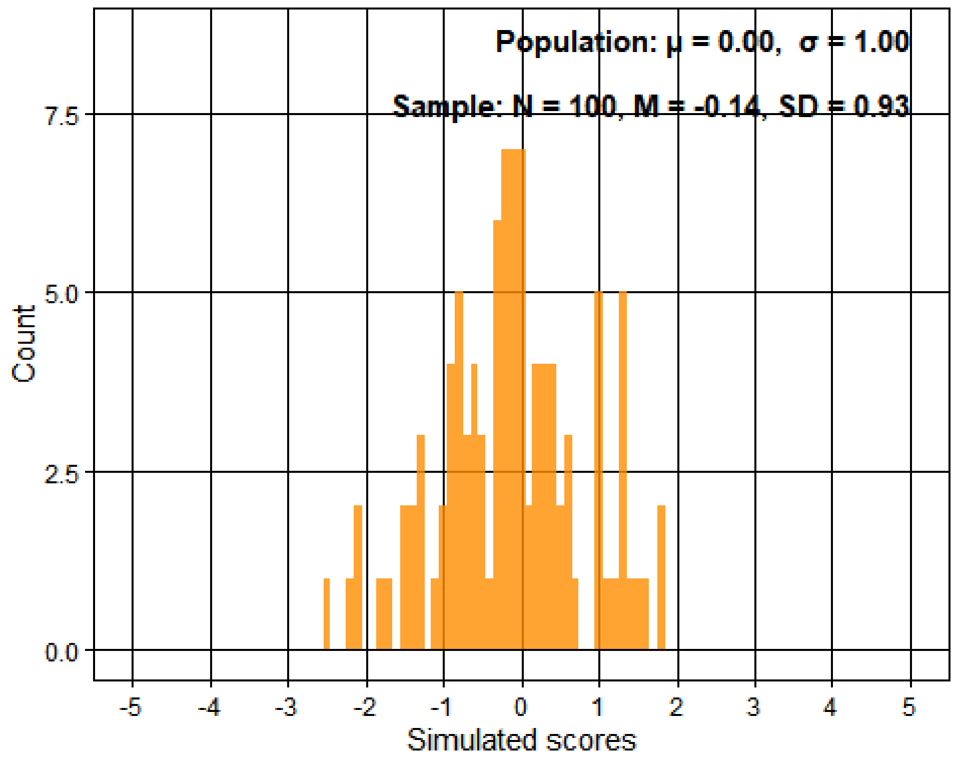
```
rnorm_histogram(n = 1000000,
                mean = 0,
                sd = 1)
```



```
rnorm_histogram(n = 1000,
                mean = 0,
                sd = 1,
                fill = "darkcyan")
```



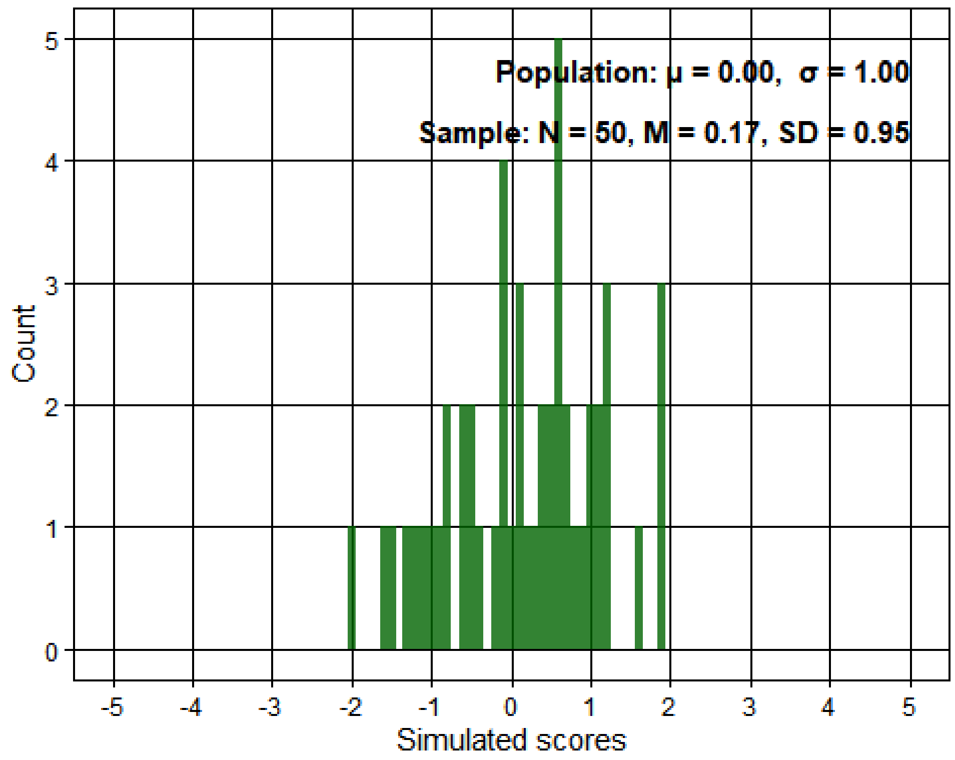
```
rnorm_histogram(n = 100,
                mean = 0,
                sd = 1,
                fill = "darkorange")
```



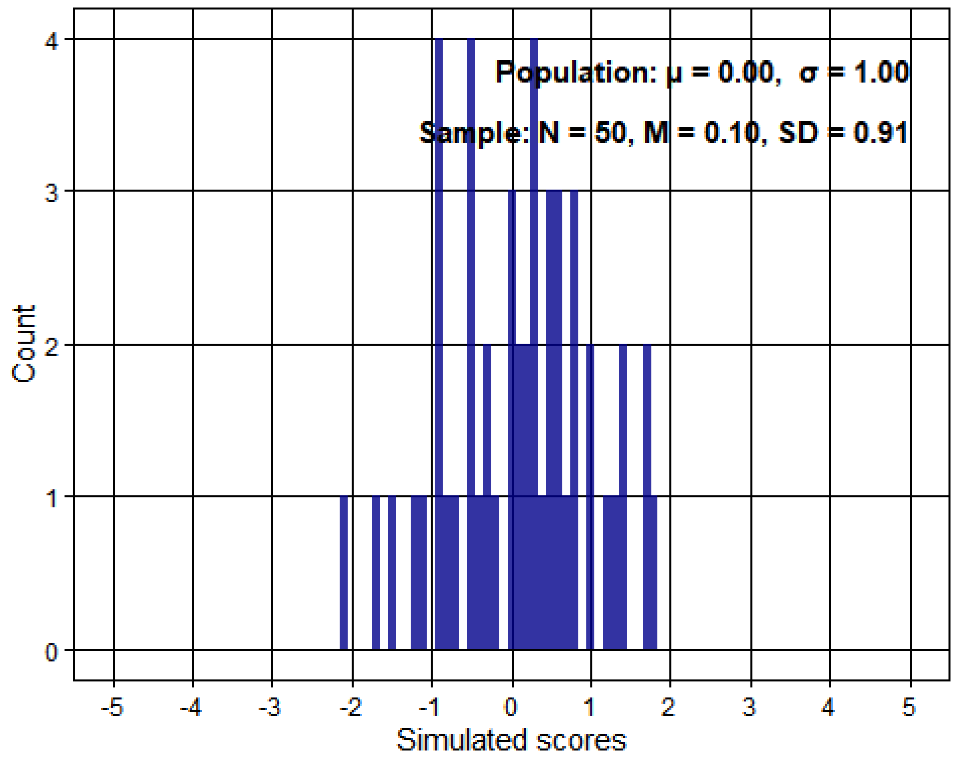
Test yourself: Is this data normally distributed?

Why or why not?

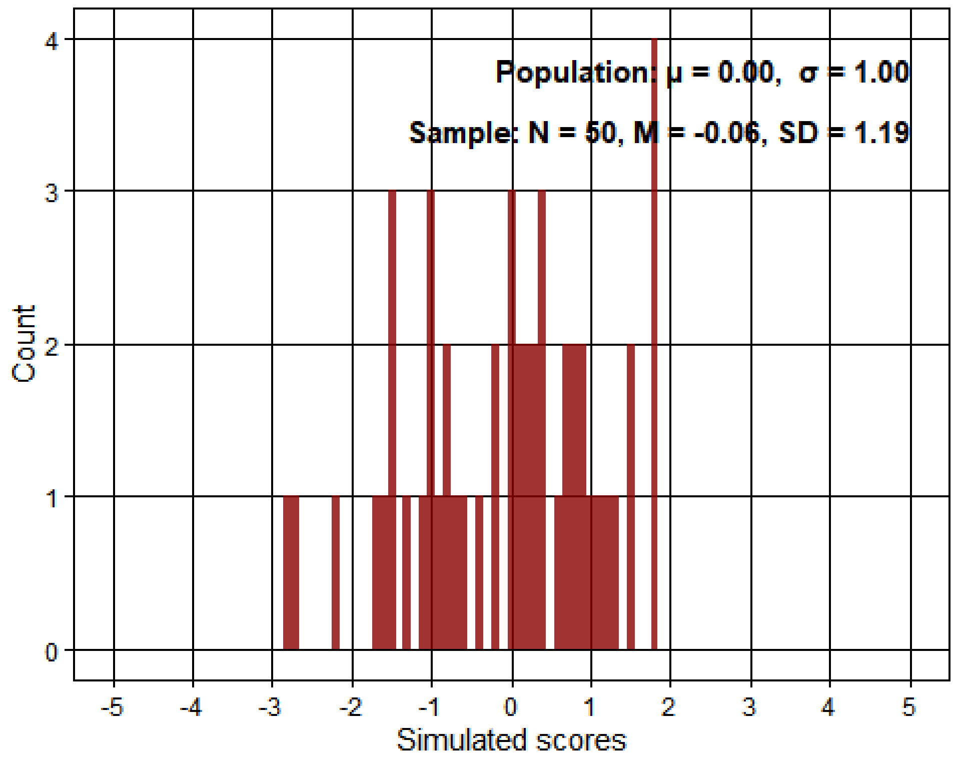
```
set.seed(238)

rnorm_histogram(n = 50,
                mean = 0,
                sd = 1,
                fill = "darkgreen")
```



```
rnorm_histogram(n = 50,
                mean = 0,
                sd = 1,
                fill = "darkblue")
```



```
rnorm_histogram(n = 50,
                mean = 0,
                sd = 1,
                fill = "darkred")
```

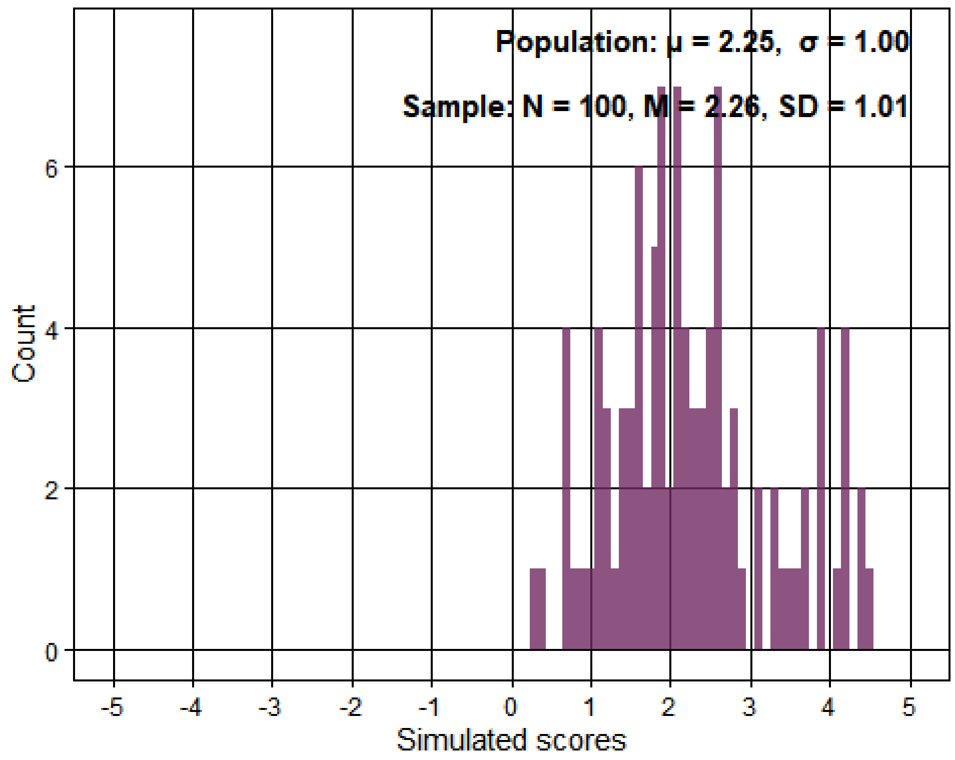


Simulations using “for loops”

Above, we saw how to simulate data for a single sample, how to plot it in a histogram, and even how to do both in a single function (`simulateR::rnorm_histogram()`).

Let’s do this again using new population parameters: μ = 2.25 and σ = 1. We’ll draw 100 samples from this population distribution. This time, we’ll set these population parameters as variables so they can be reused without having to type them each time.

```
# define the parameters
n_samples <- 100 # number of samples in each simulation
mu <- 2.25       # population mean
sigma <- 1       # population standard deviation

# make an annotated histogram
rnorm_histogram(n = n_samples,
                mean = mu,
                sd = sigma)
```



We can also skip the histogram and just simulate the data itself and then calculate the sample mean and SD. We'll do this using the parameters set in the previous chunk.

```
simulated_scores <- 
    rnorm(n = n_samples, 
          mean = mu, 
          sd = sigma)

mean(simulated_scores) |> round_half_up(2)

## [1] 2.3

#sd(simulated_scores) |> round_half_up(2)
```

If you're reading this in the .Rmd file rather than as a .html, click the run button on the previous chunk a few times to re-run the code. Notice that the sample mean and SD are different each time. Each one is somewhat close to the population value, but not exact.

Each time you click run, you are creating a new "iteration" of this very small simulation: the same code, specifying the same population parameters are generating data, and that data is being analyzed in some way (in this case: by calculating a mean and SD).

A full-blown simulation study would typically include thousands of iterations, and conclusions would be made by summarizing across those thousands of iterations.

Multiple iterations of a given simulation

What if I wanted to generate these means of simulated data lots of times?

The following code implements the following logic: “for the ith element in the sequence 1 to 10 (i.e., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), run the following code that appears between the curly brackets {}.

You have seen the code between the curly brackets before above: it generates normal data from the population parameters, calculates the sample mean, and prints it. Putting it in a for loop allows us to run it 10 times. This is very useful when you want to run things an arbitrary and large amount of times, e.g., 10,000.

```
for(i in 1:10){
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and print it
  mean(simulated_scores) |>
    print()
}
```

```
## [1] 2.345271
## [1] 1.999888
## [1] 2.196017
## [1] 2.329672
## [1] 2.389554
## [1] 2.202022
## [1] 2.250702
## [1] 2.173483
## [1] 2.229205
## [1] 2.217192
```

Note that the use of i as the iterating variable in a for loop is just a convention, but it can be any variable. For example, the following code runs identically:

```
for(whatever_varible_name_you_want in 1:10){ # only this line differs from
the previous chunk
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and print it
  mean(simulated_scores) |>
```

```
    print()
}

## [1] 2.252013
## [1] 2.270707
## [1] 2.240264
## [1] 2.333123
## [1] 2.317021
## [1] 2.286971
## [1] 2.285349
## [1] 2.292039
## [1] 2.21084
## [1] 2.033861
```

Note that the loop sequence (previously “1:10”) can be a variable instead.

```
n_iterations <- 10

for(i in 1:n_iterations){
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and print it
  mean(simulated_scores) |>
    print()
}

## [1] 2.308732
## [1] 2.327487
## [1] 2.108914
## [1] 2.330393
## [1] 2.202904
## [1] 2.257978
## [1] 2.318596
## [1] 2.165976
## [1] 2.203594
## [1] 2.059645
```

What if I wanted to save the means of simulated data rather than printing them? Doing anything useful with simulated data will require that we are able to save results to a usefully formatted data structure.

This takes a bit more effort. Let’s take a step back and talk about assignment, vectors, and for loops.

Assignment, vectors, and loops

You already know how variable assignment works. Here we create a new variable `n_iterations` and assign a single integer to it.

```
n_iterations <- 10

# print
n_iterations

## [1] 10
```

We can also create a vector - a one-dimensional, ordered collection of values. This numeric vector - ie a vector containing numeric values only - has `n_iterations` number of elements. The elements each take the default value of 0.

```
results <- numeric(n_iterations)

# print
results

##  [1] 0 0 0 0 0 0 0 0 0 0

# number of elements in the vector == n_iterations
length(results)

## [1] 10
```

We can alter individual elements of a vector. E.g., we can assign the first element of this vector to be 5.

```
results[1] <- 5

# print
results

##  [1] 5 0 0 0 0 0 0 0 0 0
```

We can also assign the fifth element of this vector to be 4.

```
results[5] <- 4

# print
results

##  [1] 5 0 0 0 4 0 0 0 0 0
```

What if we want to assign every element to 7, and we don't want to repeat ourselves? We can use a for loop.

For the `i`th element in the sequence 1:n_iterations (i.e., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), assign the `i`th element of the results vector to be 7.

```
for(i in 1:n_iterations){
  results[i] <- 7
}

# print
results

##  [1] 7 7 7 7 7 7 7 7 7 7
```

What if we want to assign each element not to the same value, but different values following a pattern? In this case, the value should be double the value of i.

```
for(i in 1:n_iterations){
  results[i] <- i * 2
}

# print
results

##  [1]  2  4  6  8 10 12 14 16 18 20
```

Now let's do something more complex. In the `i`th iteration of the loop, we simulate data from a normal distribution, calculate its mean, and then assign the resulting mean to the `i`th element of the results vector.

```
# the only difference compared to the first example at the top of the
"Multiple iterations of a given simulation" section
# is the resulting means are saved to the results vector.
# But it requires you to think about the loop in a deeper way, and the
variable value of i and what its implications are.
for(i in 1:n_iterations){
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and store it
  # in the `i`th element of the results vector
  results[i] <- mean(simulated_scores)
}

# print
results

##  [1] 2.400711 2.193914 2.298349 2.226627 2.080299 2.137809 2.172315
2.287930
##  [9] 2.283247 2.192627
```

Now that I have the results of each iteration stored in a vector, I can also summarize across iterations

```
# calculate the mean of means
mean(results) |> round_half_up(2)

## [1] 2.23
```

Testing yourself: Why doesn’t this accomplish the goal of saving all iterations to the vector?

To check your own understanding, see if you can guess what output this code creates and why.

Try to predict what value it returns, and why.

Why doesn’t it achieve what the above code does, and what you need it to? After all, it looks simpler. Indeed, it would be great if it could accomplish what the previous code does (but it can’t).

Parameter recovery

Each iteration of a simulation is often intended to correspond with a semi-realistic real life study or experiment. Real life studies usually don’t have a million participants, maybe they have more like 100. To know that `rnorm` is generating data from the population parameters we tell it to, even in smaller and more realistic sample sizes where each individual sample isn’t very informative, we can check the long run of studies.

Each individual study will have noise and random variation around it, as we saw in the histograms above with smaller sample sizes. But the long run of studies should recover the population parameters. Each iteration might be relatively small (n_samples = 100), but we can run a large number of iterations to average over (n_iterations = 1000).

```
# we increase the number of iterations to simulate a longer run of
experiments
n_iterations <- 10000

for(i in 1:n_iterations){
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and store it
  # in the `i`th element of the results vector
  results[i] <- mean(simulated_scores)
}

# calculate the mean of means
```

```
mean(results) |>
  round_half_up(2)

## [1] 2.25

# check that the mean of sample means is equal to the population mean (mu)
mean(results) |> round_half_up(2) == mu

## [1] TRUE
```

We can know with greater confidence that our simulations are working by running an experiment for ourselves: when we change the population parameters to other values, does the simulation also recover those? After all, perhaps there is a chance that our simulation (for whatever reason) simply always returns a mean of means of the same value that we used as our population parameter.

The code for this simulation is self contained: it doesn't rely on variables from previous chunks. This is the complete, working simulation. Not of something very interesting, admittedly: it just checks that the μ and σ values that we pass as arguments to the `rnorm()` function do indeed results in datasets with those means and SDs in the long run. If this were not the case, any other simulation relying on `rnorm()` would produce invalid results.

```
set.seed(42)

# new values
n_samples <- 100
n_iterations <- 10000
mu <- -2.84
sigma <- 5.10

# create two new results vectors
results_means <- numeric(n_iterations)
results_sds <- numeric(n_iterations)

for(i in 1:n_iterations){
  # generate data sampled from a normal population using rnorm
  simulated_scores <-
    rnorm(n = n_samples,
          mean = mu,
          sd = sigma)

  # compute the mean for this simulation and store it
  # in the `i`th element of each results vector
  results_means[i] <- mean(simulated_scores)
  results_sds[i] <- sd(simulated_scores)
}

# compute the mean of means
mean(results_means) |> round_half_up(2)
```

```
## [1] -2.84

# check that the mean of sample means is equal to the population mean (mu)
mean(results_means) |> round_half_up(2) == mu

## [1] TRUE

# compute the mean of SDs
mean(results_sds) |> round_half_up(2)

## [1] 5.09

# check that the mean of sample SDs is equal to the population SD (sigma)
mean(results_sds) |> round_half_up(2) == sigma

## [1] FALSE
```

Understanding this simulation in terms of frequentist statistics and vice-versa

Whether you realized it until now or not, this stimulation follows and indeed formalizes the same logic as frequentist statistics:

It imagines that smaller, finite studies are run. Each of them have a realistic number of participants, not the million samples that created the perfect normal curves in the histograms we saw earlier. Each study contains both some indication of the data generating signal - the normal distribution following specific population parameters that gave rise to the data. At the same time, each study contains much uncertainty and noise due to its finite size and random chance.

But, in the long run of studies - either a long run of real, actual replication studies (if unbiased etc), or an arbitrarily long run of simulation iterations - we can see that the population values are being uncovered (or recovered, in the case of the simulations).

We have used this simulation to prove that the (incredibly useful) `rnorm()` function functions correctly, and generates normally distributed data following known population means (μ) and SDs (σ).

Simulating the false-positive rate of a t-test

Now that we know `rnorm()` produces the type of data it claims to, we can use it for more interesting things.

By definition, a frequentist test's alpha value (e.g., .05) should equal the test's false-positive rate. But this is not always the case, under suboptimal conditions. Knowing when and where this is violated tells us about the assumptions of that test, and therefore the conditions under which it will work more or less well. Let's simulate 10,000 datasets, each drawn from a population effect that is null, and check that we do indeed find significant results in only 5% of cases (as the alpha value implies).

Notice that the comments in this code chunk highlight the key components of a simulation study, similar to that described in Hallgren (2013) “Conducting Simulation Studies in the R Programming Environment”.

```
# set seed for reproducibility
set.seed(42)

# simulation parameters
n_control      <- 50
n_intervention <- 50
mu_control      <- 0 # both mu values are zero: population effect is null.
mu_intervention <- 0
sigma_control      <- 1
sigma_intervention <- 1
n_iterations <- 10000

# create results vector
results_ps <- numeric(n_iterations)

# for loop used to repeat this many times
for(i in 1:n_iterations){
  # data generation
  data_control      <- rnorm(n = n_control,      mean = mu_control,      sd =
sigma_control)
  data_intervention <- rnorm(n = n_intervention, mean = mu_intervention, sd =
sigma_intervention)

  # data analysis
  p <- t.test(x = data_control,
              y = data_intervention,
              var.equal = TRUE,
              alternative = "two.sided")$p.value

  results_ps[i] <- p
}

# summarise results across iterations
# compute the false positive rate (proportion of significant p values when
population effect is null)
mean(results_ps < .05) |> round_half_up(2)

## [1] 0.05
```

Results demonstrate that the false positive rate for a Student’s t-test with 100 participants is indeed 5%, as implied by the alpha value. However, there are other situations where this does not hold. If you like, try changing the simulation parameters and find what combination of them produce an inflated false-positive rate. We will return to this question is a later lesson.

Session info

```
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 11 x64 (build 22631)
## 
## Matrix products: default
## 
## 
## locale:
## [1] LC_COLLATE=German_Switzerland.utf8  LC_CTYPE=German_Switzerland.utf8   
## [3] LC_MONETARY=German_Switzerland.utf8 LC_NUMERIC=C                       
## [5] LC_TIME=German_Switzerland.utf8    
## 
## time zone: Europe/Zurich
## tzcode source: internal
## 
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base     
## 
## other attached packages:
##  [1] simulateR_0.1    janitor_2.2.0    kableExtra_1.4.0 knitr_1.46      
##  [5] scales_1.3.0     lubridate_1.9.3  forcats_1.0.0    stringr_1.5.1   
##  [9] dplyr_1.1.4      purrr_1.0.2      readr_2.1.5      tidyr_1.3.1     
## [13] tibble_3.2.1     ggplot2_3.5.0    tidyverse_2.0.0 
## 
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.4        xfun_0.43           lattice_0.22-6     
##  [4] numDeriv_2016.8-1.1 mathjaxr_1.6-0      tzdb_0.4.0         
##  [7] quadprog_1.5-8      vctrs_0.6.5         tools_4.3.2        
## [10] generics_0.1.3      stats4_4.3.2        parallel_4.3.2     
## [13] fansi_1.0.6         highr_0.10          pkgconfig_2.0.3    
## [16] Matrix_1.6-5        lifecycle_1.0.4     farver_2.1.1       
## [19] compiler_4.3.2      munsell_0.5.1       mnormt_2.1.1       
## [22] codetools_0.2-20    snakecase_0.11.1    htmltools_0.5.8.1  
## [25] yaml_2.3.8          pillar_1.9.0        furrr_0.3.1        
## [28] metadat_1.2-0       nlme_3.1-164        parallelly_1.37.1  
## [31] lavaan_0.6-17       tidyselect_1.2.1    digest_0.6.35      
## [34] stringi_1.8.3       future_1.33.2       listenv_0.9.1      
## [37] labeling_0.4.3      fastmap_1.1.1       grid_4.3.2         
## [40] colorspace_2.1-0    cli_3.6.2           metafor_4.6-0      
## [43] magrittr_2.0.3      utf8_1.2.4          broom_1.0.5        
## [46] future.apply_1.11.2 pbivnorm_0.6.0      effsize_0.8.1      
## [49] withr_3.0.0         backports_1.4.1     timechange_0.3.0
## [52] rmarkdown_2.26      globals_0.16.3      hms_1.1.3
## [55] evaluate_0.23       viridisLite_0.4.2   rlang_1.1.3
## [58] glue_1.7.0          xml2_1.3.6          svglite_2.1.3
## [61] rstudioapi_0.16.0   R6_2.5.1            systemfonts_1.0.6
```