Creating simulated data sets in R

Brad Duthie

8 July 2020

# Contents

**The ability to simulate data is a useful tool for better understanding statistical analyses and planning experimental designs. These notes illustrate how to simulate data using a variety of different functions in the R programming language, then discuss how data simulation can be used in research. These notes borrow heavily from a Stirling Coding Club session on** [**randomisation**](https://stirlingcodingclub.github.io/randomisation/randomisation_notes.html)**, and to a lesser extent from a session on** [**linear models**](https://stirlingcodingclub.github.io/linear_modelling/)**. After working through these notes, the reader should be able to simulate their own data sets and use them to explore data visualisations and statistical analysis. These notes are also available as** [**PDF**](https://stirlingcodingclub.github.io/linear_modelling/index.pdf) **and** [**DOCX**](https://stirlingcodingclub.github.io/linear_modelling/index.docx) **documents.**

* [Introduction: Simulating data](#intro)
* [Univariate random numbers](#univariate)
  + [Random uniform: runif](#runif)
  + [Random normal: rnorm](#rnorm)
  + [Random poisson: rpois](#rpois)
  + [Random binomial: rbinom](#rbinom)
* [Random sampling using sample](#sample)
  + [Sampling random numbers from a list](#sample_numbers)
  + [Sampling random characters from a list](#sample_characters)
* [Simulating data with known correlations](#mvtnorm)
* [Simulating a full data set](#fullsim)
* [Conclusions](#conclusions)
* [Literature Cited](#refs)

# Introduction: Simulating data

The ability generate simulated data is very useful in a lot of research contexts. Simulated data can be used to [better understand statistical methods](https://stirlingcodingclub.github.io/linear_modelling/#makeup), or in some cases to actually [run statistical analyses](https://stirlingcodingclub.github.io/randomisation/randomisation_notes.html#rand_m) (e.g., simulating a null distribution against which to compare a sample). Here I want to demonstrate how to simulate data in R. This can be accomplished with base R functions including [rnorm](#rnorm), [runif](#runif), [rbinom](#rbinom), [rpois](#rpois), or rgamma; all of these functions sample univariate data (i.e., one variable) from a specified distribution. The function [sample](#sample) can be used to sample elements from an R object with or without replacement. Using the [MASS](https://cran.r-project.org/web/packages/MASS/MASS.pdf) library, the [mvtnorm](#mvtnorm) function will sample multiple variables with a known correlation structure (i.e., we can tell R how variables should be correlated with one another) and normally distributed errors.

Below, I will first demonstrate how to use some common functions in R for simulating data. Then, I will illustrate how these simulated data might be used to better understand common statistical analyses and data visualisation.

# Univariate random numbers

Below, I introduce some base R functions that simulate (pseudo)random numbers from a given distribution. Note that most of what follows in this section is a recreation of a similar section in the notes for [randomisation](https://stirlingcodingclub.github.io/randomisation/randomisation_notes.html) analysis in R.

**Sampling from a uniform distribution**

Like the rnorm function, the runif function returns some number (n) of random numbers, but from a [uniform distribution](https://en.wikipedia.org/wiki/Uniform_distribution_(continuous)) with a range from (min) to (max) such that , where . The default is to draw from a standard uniform distribution (i.e., and ).

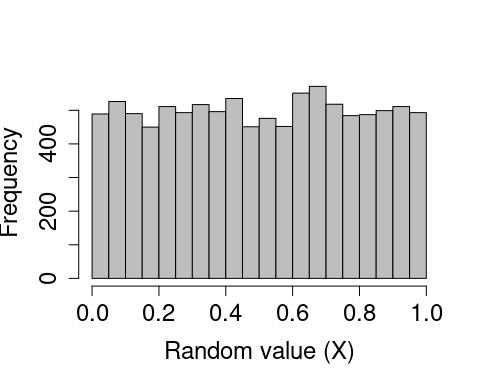
rand\_unifs\_10 <- runif(n = 10, min = 0, max = 1);

The above code stores a vector of ten numbers rand\_unifs\_10, shown below.

## [1] 0.3483086 0.5185688 0.3694390 0.9461924 0.6466858 0.9857185 0.3296636  
## [8] 0.3127385 0.6657488 0.9957011

We can visualise the standard uniform distribution that is generated by plotting a histogram of a very large number of values created using runif.

rand\_unifs\_10000 <- runif(n = 10000, min = 0, max = 1);  
hist(rand\_unifs\_10000, xlab = "Random value (X)", col = "grey",  
 main = "", cex.lab = 1.5, cex.axis = 1.5);



The random uniform distribution is special in some ways. The algorithm for generating random uniform numbers is the starting point for generating random numbers from other distributions using methods such as [rejection sampling](https://en.wikipedia.org/wiki/Rejection_sampling), [inverse transform sampling](https://en.wikipedia.org/wiki/Inverse_transform_sampling), or the [Box Muller](https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform) method (Box and Muller 1958).

**Sampling from a normal distribution**

The rnorm function returns some number (n) of randomly generated values given a set mean (; mean) and standard deviation (; sd), such that . The default is to draw from a standard [normal (a.k.a., “Gaussian”) distribution](https://en.wikipedia.org/wiki/Normal_distribution) (i.e., and ).

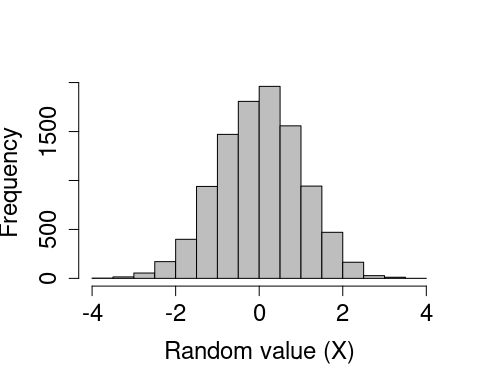
rand\_norms\_10 <- rnorm(n = 10, mean = 0, sd = 1);

The above code stores a vector of 10 numbers, shown below.

## [1] 1.17671190 0.90023495 1.11026872 -1.76899133 -0.63446729 -1.88246810  
## [7] 0.05246716 0.82688295 1.10053697 0.93490105

We can verify that a standard normal distribution is generated by plotting a histogram of a very large number of values created using rnorm.

rand\_norms\_10000 <- rnorm(n = 10000, mean = 0, sd = 1);  
hist(rand\_norms\_10000, xlab = "Random value (X)", col = "grey",  
 main = "", cex.lab = 1.5, cex.axis = 1.5);



Generating a histogram using data from a simulated distribution like this is often a useful way to visualise distributions, or to see how samples from the same distribution might vary. For example, if we wanted to compare the above distribution with a normal distribution that had a standard deviation of 2 instead of 1, then we could simply sample 10000 new values in rnorm with sd = 2 instead of sd = 1 and create a new histogram with hist. If we wanted to see what the distribution of sampled data might look like given a low sample size (e.g., 10), then we could repeat the process of sampling from rnorm(n = 10, mean = 0, sd = 1) multiple times and looking at the shape of the resulting histogram.

**Sampling from a poisson distribution**

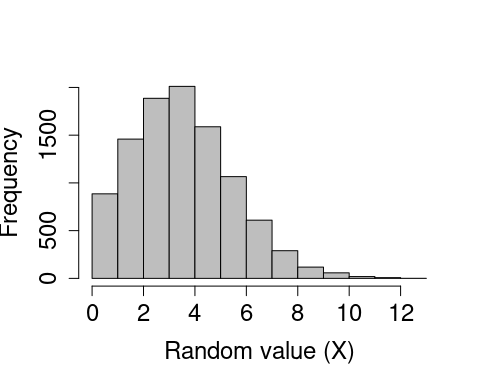
Many processes in biology can be described by a [Poisson distribution](https://en.wikipedia.org/wiki/Poisson_distribution). A Poisson process describes events happening with some given probability over an area of time or space such that , where the rate parameter is both the mean and variance of the Poisson distribution (note that by definition, , and values are always integers, as with count data). Sampling from a Poisson distribution can be done in R with rpois, which takes only two arguments specifying the number of values to be returned (n) and the rate parameter (lambda).

rand\_poissons <- rpois(n = 10, lambda = 1);  
print(rand\_poissons);

## [1] 1 3 0 1 0 0 0 0 0 1

There are no default values for rpois. We can plot a histogram of a large number of values to see the distribution when below.

rand\_poissons\_10000 <- rpois(n = 10000, lambda = 4);  
hist(rand\_poissons\_10000, xlab = "Random value (X)", col = "grey",  
 main = "", cex.lab = 1.5, cex.axis = 1.5);



**Sampling from a binomial distribution**

Sampling from a binomial distribution in R with rbinom is a bit more complex than using runif, rnorm, or rpois. Like those previous functions, the rbinom function returns some number (n) of random numbers, but the arguments and output can be slightly confusing at first. Recall that a [binomial distribution](https://en.wikipedia.org/wiki/Binomial_distribution) describes the number of ‘successes’ for some number of independent trials (). The rbinom function returns the number of successes after size trials, in which the probability of success in each trial is prob. For a concrete example, suppose we want to simulate the flipping of a fair coin 1000 times, and we want to know how many times that coin comes up heads (‘success’). We can do this with the following code.

coin\_flips <- rbinom(n = 1, size = 1000, prob = 0.5);  
print(coin\_flips);

## [1] 501

The above result shows that the coin came up heads 501 times. Note, however, the (required) argument n above. This allows the user to set the number of sequences to run. In other words, if we set n = 2, then this could simulate the flipping of a fair coin 1000 times once to see how many times heads comes up, then repeating the whole process a second time to see how many times heads comes up again (or, if it is more intuitive, the flipping of two separate fair coins 1000 times).

coin\_flips\_2 <- rbinom(n = 2, size = 1000, prob = 0.5);  
print(coin\_flips\_2);

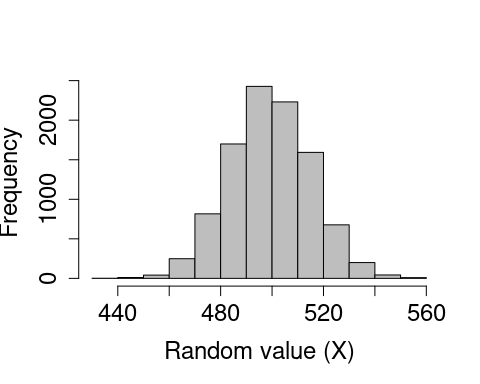
## [1] 497 492

In the above, a fair coin was flipped 1000 times and returned 497 heads, and then another fair coin was flipped 1000 times and returned 492 heads. As with the rnorm and runif functions, we can check to see what the distribution of the binomial function looks like if we repeat this process. Suppose, in other words, that we want to see the distribution of times heads comes up after 1000 flips. We can, for example, simulate the process of flipping 1000 times in a row with 10000 different coins using the code below.

coin\_flips\_10000 <- rbinom(n = 10000, size = 1000, prob = 0.5);

I have not printed the above coin\_flips\_10000 for obvious reasons, but we can use a histogram to look at the results.

hist(coin\_flips\_10000, xlab = "Random value (X)", col = "grey",  
 main = "", cex.lab = 1.5, cex.axis = 1.5);



As would be expected, most of the time ‘heads’ occurs around 500 times out of 1000, but usually the actual number will be a bit lower or higher due to chance. Note that if we want to simulate the results of individual flips in a single trial, we can do so as follows.

flips\_10 <- rbinom(n = 10, size = 1, prob = 0.5);

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

In the above, there are n = 10 trials, but each trial consists of only a single coin flip (size = 1). But we can equally well interpret the results as a series of n coin flips that come up either heads (1) or tails (0). This latter interpretation can be especially useful to write code that randomly decides whether some event will happen (1) or not (0) with some probability prob.

# Random sampling using sample

Sometimes it is useful to sample a set of values from a vector or list. The R function sample is very flexible for sampling a subset of numbers or elements from some structure (x) in R according to some set probabilities (prob). Elements can be sampled from x some number of times (size) with or without replacement (replace), though an error will be returned if the size of the sample is larger than x but replace = FALSE (default).

**Sampling random numbers from a list**

To start out simple, suppose we want to ask R to pick a random number from one to ten with equal probability.

rand\_number\_1 <- sample(x = 1:10, size = 1);

The above code will set rand\_number\_1 to a randomly selected value, in this case 3. Because we have not specified a probability vector prob, the function assumes that every element in 1:10 is sampled with equal probability. We can increase the size of the sample to 10 below.

rand\_number\_10 <- sample(x = 1:10, size = 10);  
print(rand\_number\_10);

## [1] 6 9 1 2 7 3 8 5 10 4

Note that all numbers from 1 to 10 have been sampled, but in a random order. This is becaues the default is to sample with replacement, meaning that once a number has been sampled for the first element in rand\_number\_10, it is no longer available to be sampled again. To change this and allow for sampling with replacement, we can change the default.

rand\_number\_10\_r <- sample(x = 1:10, size = 10, replace = TRUE);  
print(rand\_number\_10\_r);

## [1] 2 3 3 2 6 6 2 10 9 6

Note that the numbers {2, 3, 6} are now repeated in the set of randomly sampled values above. We can also specify the probability of sampling each element, with the condition that these probabilities need to sum to 1. Below shows an example in which the numbers 1-5 are sampled with a probability of 0.05, while the numbers 6-10 are sampled with a probability of 0.15, thereby biasing sampling toward larger numbers.

prob\_vec <- c( rep(x = 0.05, times = 5), rep(x = 0.15, times = 5) );  
rand\_num\_bias <- sample(x = 1:10, size = 10, replace = TRUE, prob = prob\_vec);  
print(rand\_num\_bias);

## [1] 4 10 8 9 7 10 10 4 7 10

Note that rand\_num\_bias above contains more numbers from 6-10 than from 1-5.

**Sampling random characters from a list**

Sampling characters from a list of elements is no different than sampling numbers, but I am illustrating it separately because I find that I often sample characters for conceptually different reasons. For example, if I want to create a simulated data set that includes three different species, I might create a vector of species identities from which to sample.

species <- c("species\_A", "species\_B", "species\_C");

This gives three possible categories, which I can now use sample to draw from. Assume that I want to simulate the sampling of these three species, perhaps with species\_A being twice as common as species\_B and species\_C. I might use the following code to sample 12 times.

sp\_sample <- sample(x = species, size = 12, replace = TRUE,   
 prob = c(0.5, 0.25, 0.25)   
 );

Below are the values that get returned.

## [1] "species\_C" "species\_A" "species\_A" "species\_A" "species\_A" "species\_B"  
## [7] "species\_A" "species\_A" "species\_A" "species\_A" "species\_C" "species\_A"

# Simulating data with known correlations

We can generate variables and that have known correlations with with one another. The code below does this for two standard normal random variables with a sample size of 10000, such that the correlation between them is 0.3.

N <- 10000;  
rho <- 0.3;  
x1 <- rnorm(n = N, mean = 0, sd = 1);  
x2 <- (rho \* x1) + sqrt(1 - rho\*rho) \* rnorm(n = N, mean = 0, sd = 1);

Mathematically, these variables are generated by first simulating the sample (x1 above) from a standard normal distribution. Then, (x2 above) is calculated as below,

Where is a sample from a normal distribution with the same variance as . A simple call to the R function cor will confirm that the correlation does indeed equal rho.

cor(x1, x2);

## [1] 0.3016727

This is useful if we are only interested in two variables, but there is a much more efficient way to generate any number of variables with different variances and correlations to one another. To do this, we need to use the [MASS](https://cran.r-project.org/web/packages/MASS/MASS.pdf) library, which can be installed and loaded as below.

install.packages("MASS");  
library("MASS");

In the [MASS](https://cran.r-project.org/web/packages/MASS/MASS.pdf) library, the function mvrnorm can be used to generate any number of variables for a pre-specified covariance structure.

Suppose we want to simulate a data set of three measurements from a species of organisms. Measurement 1 () has a mean of 159.54 and variance of 12.68, measurement 2 () has a mean of 245.26 and variance of 30.39, and measurement 3 () has a mean of 25.52 and variance of 2.18. Below is a table summarising.

|  |  |  |
| --- | --- | --- |
| measurement | mean | variance |
| M1 | 159.54 | 12.68 |
| M2 | 245.26 | 30.39 |
| M3 | 25.52 | 2.18 |

Further, we want the covariance between and to equal 13.95, the covariance between and to equal 3.07, and the covariance between and to equal 4.7. We can put all of this information into a [covariance matrix](https://en.wikipedia.org/wiki/Covariance_matrix) with three rows and three columns. The diagonal of the matrix holds the variances of each variable, with the off-diagonals holding the covariances (note also that the variance of a variable is just the variable’s covariance with itself; e.g., ).

In R, we can create this matrix as follows.

matrix\_data <- c(12.68, 13.95, 3.07, 13.95, 30.39, 4.70, 3.07, 4.70, 2.18);  
cv\_mat <- matrix(data = matrix\_data, nrow = 3, ncol = 3, byrow = TRUE);  
rownames(cv\_mat) <- c("M1", "M2", "M3");  
colnames(cv\_mat) <- c("M1", "M2", "M3");

Here is what cv\_mat looks like (note that it is symmetrical along the diagonal).

## M1 M2 M3  
## M1 12.68 13.95 3.07  
## M2 13.95 30.39 4.70  
## M3 3.07 4.70 2.18

Now we can add the means to a vector in R.

mns <- c(159.54, 245.26, 25.52);

We are now ready to use the mvrnorm function in R to simulate some number n of sampled organisms with these three measurements. We use the mvrnorm arguments mu and Sigma to specify the vector of means and covariance matrix, respectively.

sim\_data <- mvrnorm(n = 40, mu = mns, Sigma = cv\_mat);

Here are the example data below.

|  |  |  |
| --- | --- | --- |
| M1 | M2 | M3 |
| 156.7999 | 237.6301 | 23.76884 |
| 158.4109 | 239.1323 | 24.52847 |
| 154.5124 | 244.2650 | 25.78550 |
| 165.5468 | 254.9793 | 26.41293 |
| 156.1780 | 239.3595 | 23.92292 |
| 157.1597 | 239.6884 | 24.88802 |
| 159.0564 | 245.0888 | 24.97877 |
| 156.4170 | 239.2255 | 25.29391 |
| 159.1641 | 249.4468 | 28.32484 |
| 161.8576 | 246.7828 | 27.16455 |
| 158.7202 | 235.7650 | 24.20640 |
| 165.2677 | 248.1581 | 27.90782 |
| 159.9080 | 245.9444 | 25.69184 |
| 157.8584 | 251.4671 | 25.40560 |
| 160.1644 | 249.6938 | 26.25221 |
| 154.1145 | 238.6140 | 23.37118 |
| 156.3118 | 233.9362 | 23.43543 |
| 157.6097 | 247.7627 | 24.26124 |
| 156.8026 | 245.7292 | 26.42285 |
| 167.5281 | 263.4859 | 28.59048 |
| 155.9269 | 237.4287 | 24.75136 |
| 158.0006 | 249.9820 | 26.11238 |
| 161.5973 | 249.5291 | 26.29375 |
| 162.6388 | 250.1094 | 25.09689 |
| 163.1960 | 248.7763 | 24.23409 |
| 161.5163 | 250.7406 | 24.24654 |
| 161.0379 | 239.7254 | 24.57046 |
| 163.0718 | 252.9002 | 25.67310 |
| 158.1829 | 244.8200 | 26.51034 |
| 156.9078 | 240.7932 | 24.91835 |
| 156.5814 | 241.1537 | 25.07142 |
| 161.4002 | 252.5069 | 27.09345 |
| 155.0329 | 242.6161 | 23.94510 |
| 162.0298 | 244.9221 | 26.67841 |
| 159.6927 | 247.7254 | 26.94708 |
| 158.6999 | 240.7109 | 23.92360 |
| 162.6690 | 249.3751 | 27.83585 |
| 163.6575 | 250.1501 | 26.53301 |
| 159.4483 | 247.3285 | 26.52222 |
| 155.7414 | 236.1683 | 25.94123 |

We can check to confirm that the mean values of each column are correct using apply.

apply(X = sim\_data, MARGIN = 2, FUN = mean);

## M1 M2 M3   
## 159.41044 245.34043 25.58781

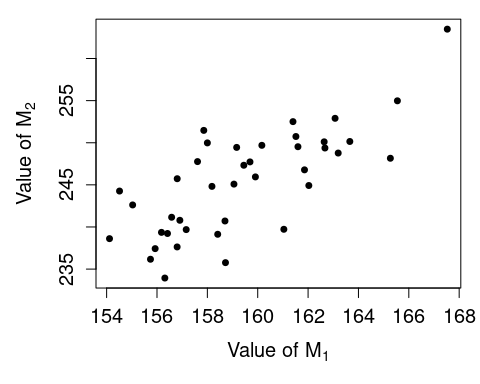
And we can check to confirm that the covariance structure of the data is correct using cov.

cov(sim\_data);

## M1 M2 M3  
## M1 10.367057 14.935778 2.539578  
## M2 14.935778 38.188322 5.569731  
## M3 2.539578 5.569731 1.879835

Note that the values are not exact, but should become closer to the specified values as increase the sample size n. We can visualse the data too; for example, we might look at the close correlation between and using a scatterplot, just as we would for data sampled from the field.

par(mar = c(5, 5, 1, 1));  
plot(x = sim\_data[,1], y = sim\_data[,2], pch = 20, cex = 1.25, cex.lab = 1.25,  
 cex.axis = 1.25, xlab = expression(paste("Value of ", M[1])),  
 ylab = expression(paste("Value of ", M[2])));



We could even run an ordination on these simulated data. For example, we could extract the principle components with prcomp, then plot the first two PCs to visualise these data. We might, for example, want to compare different methods of ordination using a data set with different, pre-specified properties (e.g., Minchin 1987). We might also want to use simulated data sets to investigate how different statistical tools perform. I show this in the next section, where I put a full data set together and run [linear models](https://stirlingcodingclub.github.io/linear_modelling/) on it.

# Simulating a full data set

Putting everything together, here I will create a data set of three different species from which three different measurements are taken. We can just call these measurements ‘length’, ‘width’, and ‘mass’. For simplicity, let us assume that these measurements always covary in the same way that we saw with (i.e., cv\_mat) above. But let’s also assume that we have three species with slightly different mean values. Below is the code that will build a new data set of samples with four columns: species, length, width, and mass.

N <- 20;  
matrix\_data <- c(12.68, 13.95, 3.07, 13.95, 30.39, 4.70, 3.07, 4.70, 2.18);  
cv\_mat <- matrix(data = matrix\_data, nrow = 3, ncol = 3, byrow = TRUE);  
mns\_1 <- c(159.54, 245.26, 25.52);  
sim\_data\_1 <- mvrnorm(n = N, mu = mns, Sigma = cv\_mat);  
colnames(sim\_data\_1) <- c("Length", "Width", "Mass");  
# Below, I bind a column for indicating 'species\_1' identity  
species <- rep(x = "species\_1", times = 20); # Repeats 20 times  
sp\_1 <- data.frame(species, sim\_data\_1);

Let us add one more data column. Suppose that we can also sample the number of offspring each organism has, and that the mean number of offspring that an organism has equals one tenth of the organism’s mass. To do this, we can use rpois, and take advantage of the fact that the argument lambda can be a vector rather than a single value. So to get the number of offspring for each organism based on its body mass, we can just insert the mass vector sp\_1$Mass times 0.1 for lambda.

offspring <- rpois(n = N, lambda = sp\_1$Mass \* 0.1);  
sp\_1 <- cbind(sp\_1, offspring);

I have also bound the offspring number to the data set sp\_1. Here is what it looks like below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| species | Length | Width | Mass | offspring |
| species\_1 | 158.4565 | 243.1370 | 23.38561 | 2 |
| species\_1 | 157.2741 | 244.2342 | 24.94022 | 4 |
| species\_1 | 158.2928 | 248.7526 | 26.38998 | 2 |
| species\_1 | 156.2168 | 245.9917 | 25.96801 | 1 |
| species\_1 | 163.0832 | 245.8923 | 27.57875 | 2 |
| species\_1 | 161.3169 | 242.3061 | 26.32443 | 3 |
| species\_1 | 165.5562 | 251.5265 | 25.69130 | 1 |
| species\_1 | 161.7600 | 248.6971 | 26.86490 | 1 |
| species\_1 | 155.4882 | 241.2396 | 24.70706 | 2 |
| species\_1 | 156.7792 | 244.2795 | 25.65491 | 0 |
| species\_1 | 161.9879 | 243.5086 | 23.84894 | 3 |
| species\_1 | 164.5975 | 255.0058 | 28.28061 | 3 |
| species\_1 | 160.0662 | 244.7753 | 23.87984 | 2 |
| species\_1 | 161.3359 | 239.8836 | 24.38817 | 3 |
| species\_1 | 153.8743 | 237.2926 | 24.18724 | 5 |
| species\_1 | 159.7522 | 244.6814 | 25.89826 | 4 |
| species\_1 | 157.4403 | 246.2192 | 23.86332 | 4 |
| species\_1 | 160.2325 | 250.6766 | 26.15072 | 4 |
| species\_1 | 156.0560 | 241.6316 | 22.26129 | 4 |
| species\_1 | 155.8918 | 243.6194 | 24.50011 | 8 |

To add two more species, let us repeat the process two more times, but change the expected mass just slightly each time. The code below does this, and puts everything together in a single data set.

# First making species 2  
mns\_2 <- c(159.54, 245.26, 25.52 + 3); # Add a bit  
sim\_data\_2 <- mvrnorm(n = N, mu = mns, Sigma = cv\_mat);  
colnames(sim\_data\_2) <- c("Length", "Width", "Mass");  
species <- rep(x = "species\_2", times = 20); # Repeats 20 times  
offspring <- rpois(n = N, lambda = sim\_data\_2[,3] \* 0.1);  
sp\_2 <- data.frame(species, sim\_data\_2, offspring);  
# Now make species 3  
mns\_3 <- c(159.54, 245.26, 25.52 + 4.5); # Add a bit more  
sim\_data\_3 <- mvrnorm(n = N, mu = mns, Sigma = cv\_mat);  
colnames(sim\_data\_3) <- c("Length", "Width", "Mass");  
species <- rep(x = "species\_3", times = 20); # Repeats 20 times  
offspring <- rpois(n = N, lambda = sim\_data\_3[,3] \* 0.1);  
sp\_3 <- data.frame(species, sim\_data\_3, offspring);  
# Bring it all together in one data set  
dat <- rbind(sp\_1, sp\_2, sp\_3);

Our full data set now looks like the below.

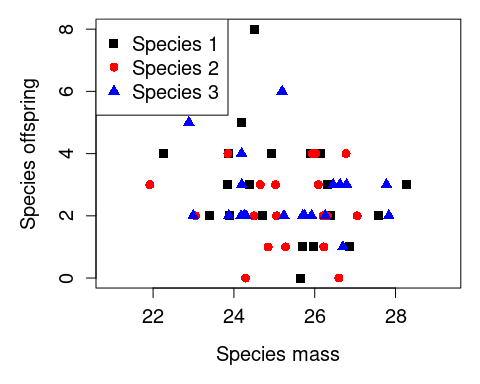
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| species | Length | Width | Mass | offspring |
| species\_1 | 158.4565 | 243.1370 | 23.38561 | 2 |
| species\_1 | 157.2741 | 244.2342 | 24.94022 | 4 |
| species\_1 | 158.2928 | 248.7526 | 26.38998 | 2 |
| species\_1 | 156.2168 | 245.9917 | 25.96801 | 1 |
| species\_1 | 163.0832 | 245.8923 | 27.57875 | 2 |
| species\_1 | 161.3169 | 242.3061 | 26.32443 | 3 |
| species\_1 | 165.5562 | 251.5265 | 25.69130 | 1 |
| species\_1 | 161.7600 | 248.6971 | 26.86490 | 1 |
| species\_1 | 155.4882 | 241.2396 | 24.70706 | 2 |
| species\_1 | 156.7792 | 244.2795 | 25.65491 | 0 |
| species\_1 | 161.9879 | 243.5086 | 23.84894 | 3 |
| species\_1 | 164.5975 | 255.0058 | 28.28061 | 3 |
| species\_1 | 160.0662 | 244.7753 | 23.87984 | 2 |
| species\_1 | 161.3359 | 239.8836 | 24.38817 | 3 |
| species\_1 | 153.8743 | 237.2926 | 24.18724 | 5 |
| species\_1 | 159.7522 | 244.6814 | 25.89826 | 4 |
| species\_1 | 157.4403 | 246.2192 | 23.86332 | 4 |
| species\_1 | 160.2325 | 250.6766 | 26.15072 | 4 |
| species\_1 | 156.0560 | 241.6316 | 22.26129 | 4 |
| species\_1 | 155.8918 | 243.6194 | 24.50011 | 8 |
| species\_2 | 160.0025 | 241.9459 | 26.77441 | 4 |
| species\_2 | 154.1757 | 235.3846 | 26.08901 | 3 |
| species\_2 | 163.9573 | 249.7366 | 26.30147 | 2 |
| species\_2 | 159.1799 | 246.0470 | 24.84603 | 1 |
| species\_2 | 151.9027 | 237.7081 | 21.92144 | 3 |
| species\_2 | 159.2292 | 242.9459 | 25.02942 | 3 |
| species\_2 | 164.1135 | 249.8544 | 26.02879 | 4 |
| species\_2 | 156.9236 | 241.4771 | 24.64972 | 3 |
| species\_2 | 156.5620 | 234.3267 | 24.28750 | 0 |
| species\_2 | 161.8048 | 241.4408 | 25.28024 | 1 |
| species\_2 | 163.8265 | 251.2648 | 25.94367 | 4 |
| species\_2 | 160.3772 | 243.2569 | 26.59856 | 0 |
| species\_2 | 153.9088 | 245.3636 | 23.04643 | 2 |
| species\_2 | 160.3276 | 247.8055 | 26.30887 | 2 |
| species\_2 | 159.9237 | 247.4858 | 23.85211 | 4 |
| species\_2 | 157.1908 | 248.8533 | 27.05069 | 2 |
| species\_2 | 168.5553 | 250.9182 | 26.22410 | 1 |
| species\_2 | 157.2060 | 240.2435 | 24.49341 | 2 |
| species\_2 | 164.7081 | 257.1248 | 26.20848 | 2 |
| species\_2 | 158.0917 | 245.9145 | 25.05511 | 2 |
| species\_3 | 156.9107 | 236.0195 | 24.19059 | 4 |
| species\_3 | 159.0342 | 245.3388 | 26.45779 | 3 |
| species\_3 | 156.9186 | 246.1912 | 25.74907 | 2 |
| species\_3 | 155.7678 | 239.6978 | 22.88479 | 5 |
| species\_3 | 158.0937 | 242.3673 | 24.19582 | 3 |
| species\_3 | 155.0716 | 238.4135 | 22.99116 | 2 |
| species\_3 | 166.7306 | 256.9142 | 27.77511 | 3 |
| species\_3 | 161.4738 | 244.1493 | 26.26333 | 2 |
| species\_3 | 155.7215 | 238.2564 | 24.26026 | 2 |
| species\_3 | 163.3106 | 248.7752 | 23.87138 | 2 |
| species\_3 | 162.1702 | 247.7806 | 26.62700 | 3 |
| species\_3 | 161.5203 | 250.1563 | 25.92316 | 2 |
| species\_3 | 157.5109 | 241.0652 | 24.17754 | 2 |
| species\_3 | 155.5164 | 243.9490 | 24.30092 | 2 |
| species\_3 | 162.1374 | 246.5908 | 25.70360 | 2 |
| species\_3 | 159.7511 | 236.9142 | 25.24058 | 2 |
| species\_3 | 163.8569 | 253.2958 | 26.78874 | 3 |
| species\_3 | 158.9074 | 250.1859 | 25.19471 | 6 |
| species\_3 | 162.7958 | 254.1816 | 27.83501 | 2 |
| species\_3 | 162.8978 | 256.9045 | 26.69767 | 1 |

To summarise, we now have a simulated data set of measurements from three different species, all of which have known variances and covariances of length, width, and mass. Each species has a slightly different mean mass, and for all species, each unit of mass increases the expected number of offspring by 0.1. Because we know these properties of the data for certain, we can start asking questions that might be useful to know about our data analysis. For example, given this covariance structure and these small differences in mass, is a sample size of 20 really enough to even get a significant difference among species masses using an ANOVA? What if we tried to test for differences among masses using some sort of [randomisation approach](https://stirlingcodingclub.github.io/randomisation/randomisation_notes.html#rand_h) Instead? Would this give us more or less power? Let us run an ANOVA to see if the difference between group means (which we know exists) is recovered.

aov\_result <- aov(Mass ~ species, data = dat);  
summary(aov\_result);

## Df Sum Sq Mean Sq F value Pr(>F)  
## species 2 0.14 0.0699 0.034 0.966  
## Residuals 57 116.44 2.0428

It appears not! What about the relationship between body mass and offspring production that we know exists? Below is a scatterplot of the data for the three different species.



This looks like there might be a postive relationship. We can use a generalised linear model to test it with species as a random effect, as we might do if these were data sampled from the field.

library(lme4);

## Loading required package: Matrix

mod <- glmer(offspring ~ Mass + (1 | species), data = dat, family = "poisson");

## boundary (singular) fit: see ?isSingular

summary(mod);

## Generalized linear mixed model fit by maximum likelihood (Laplace  
## Approximation) [glmerMod]  
## Family: poisson ( log )  
## Formula: offspring ~ Mass + (1 | species)  
## Data: dat  
##   
## AIC BIC logLik deviance df.resid   
## 213.5 219.8 -103.8 207.5 57   
##   
## Scaled residuals:   
## Min 1Q Median 3Q Max   
## -1.6730 -0.4985 -0.2223 0.4981 3.1630   
##   
## Random effects:  
## Groups Name Variance Std.Dev.  
## species (Intercept) 0 0   
## Number of obs: 60, groups: species, 3  
##   
## Fixed effects:  
## Estimate Std. Error z value Pr(>|z|)   
## (Intercept) 2.94915 1.43294 2.058 0.0396 \*  
## Mass -0.07905 0.05690 -1.389 0.1648   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Correlation of Fixed Effects:  
## (Intr)  
## Mass -0.998  
## convergence code: 0  
## boundary (singular) fit: see ?isSingular

There does not appear to be any effect here either! To get one, it appears that we will need to simulate a larger data set (or a bigger effect size – or just get lucky when re-simulating a new data set).

Note that I have run a linear model that might be reasonable given the structure of our data. But the advantage of working with simulated data and knowing for certain what the relationship is between the underlying variables is that we can explore different statistical techniques. For example, we know that our response variable offspring is count data, so we are supposed to specify a Poisson error structure using the family = "poisson" argument above, right? But what would happen if we just used a normal error structure anyway? Would this really be so bad? Now is the opportunity to test because we *know* what the correct answer is supposed to be! Trying statistical methods that are normally ill-advised can actually be useful here, as it can help us see for ourselves when a technique is bad – or perhaps when it really is not (e.g., Ives 2015).

# Conclusions

Simulating data can be a powerful tool for learning and investigating different statistical analyses. The main benefits of using simulated data are flexibility and certainty. Simulation gives us the flexibility to explore any number of hypotheticals, including different sample sizes, effect sizes, relationships between variables, and error distributions. It also works from a point of certainty; we know what the real relationship is between variables, and what the actual effect sizes are because we define them when generating random samples. So if we want to better understand what would happen if we were unable to sample an important variable in our system, or if we were to use a biased estimator, or if we we were to violate key model assumptions, simulated data is a very useful tool.

# Literature cited

Box, G E P, and Mervin E Muller. 1958. “A note on the generation of random normal deviates.” *The Annals of Mathematical Statistics* 29 (2): 610–11. <https://doi.org/10.1214/aoms/1177706645>.

Ives, Anthony R. 2015. “For testing the significance of regression coefficients, go ahead and log-transform count data.” *Methods in Ecology and Evolution* 6: 828–35. <https://doi.org/10.1111/2041-210X.12386>.

Minchin, Peter R. 1987. “An evaluation of the relative robustness of techniques for ecological ordination.” *Vegetatio* 69 (1-3): 89–107. <https://doi.org/10.1007/BF00038690>.