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, and to a lesser extent from a session on linear models. 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 and DOCX documents.

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, or in some cases to actually run statistical analyses (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`, `runif`, `rbinom`, `rpois`, or `rgamma`; all of these functions sample univariate data (i.e., one variable) from a specified distribution. The function `sample` can be used to sample elements from an R object with or without replacement. Using the MASS library, the `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 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 with a range from a (`min`) to b (`max`) such that $X \sim \mathcal{U}(a, b)$, where $-\infty < a < b < \infty$. The default is to draw from a standard uniform distribution (i.e., $a = 0$ and $b = 1$).

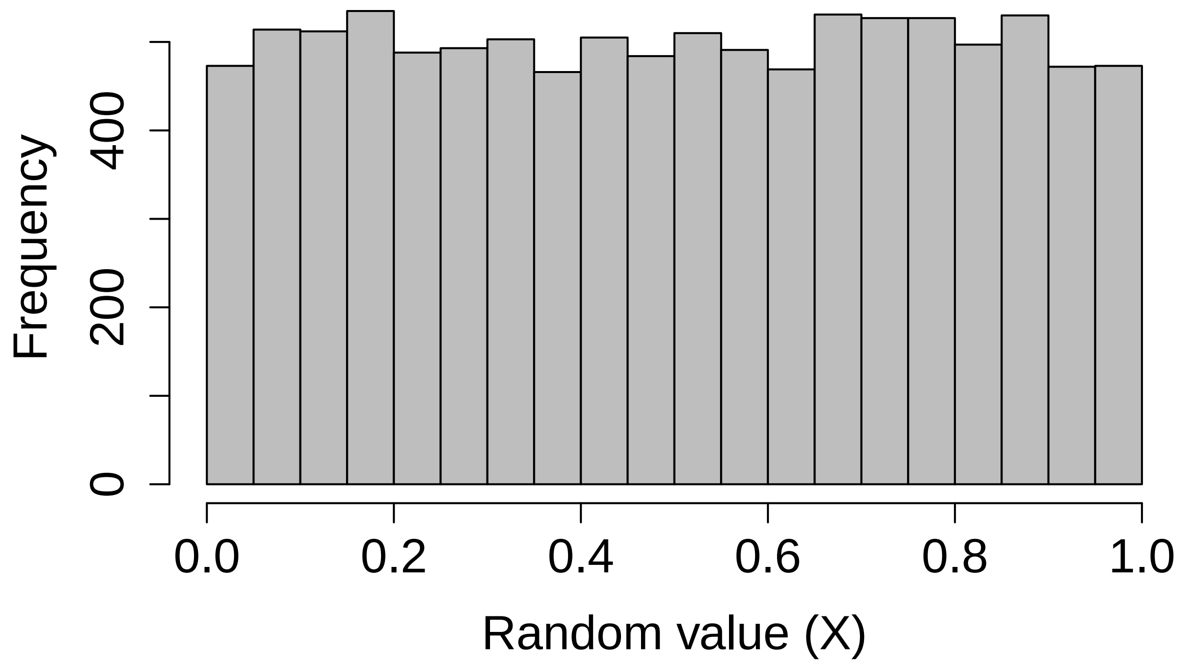
```
rand_unifs_10 <- runif(n = 10, min = 0, max = 1);
```

The above code stores a vector of ten numbers `rand_unifs_10`, shown below.

```
##  [1] 0.39467963 0.47804766 0.11898876 0.35539301 0.37628481 0.95330203
##  [7] 0.10846517 0.06609417 0.72372574 0.41649937
```

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, inverse transform sampling, or the Box Muller 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 (μ; `mean`) and standard deviation (σ; `sd`), such that $X \sim \mathcal{N}(\mu, \sigma^2)$. The default is to draw from a standard normal (a.k.a., “Gaussian”) distribution (i.e., $\mu = 0$ and $\sigma = 1$).

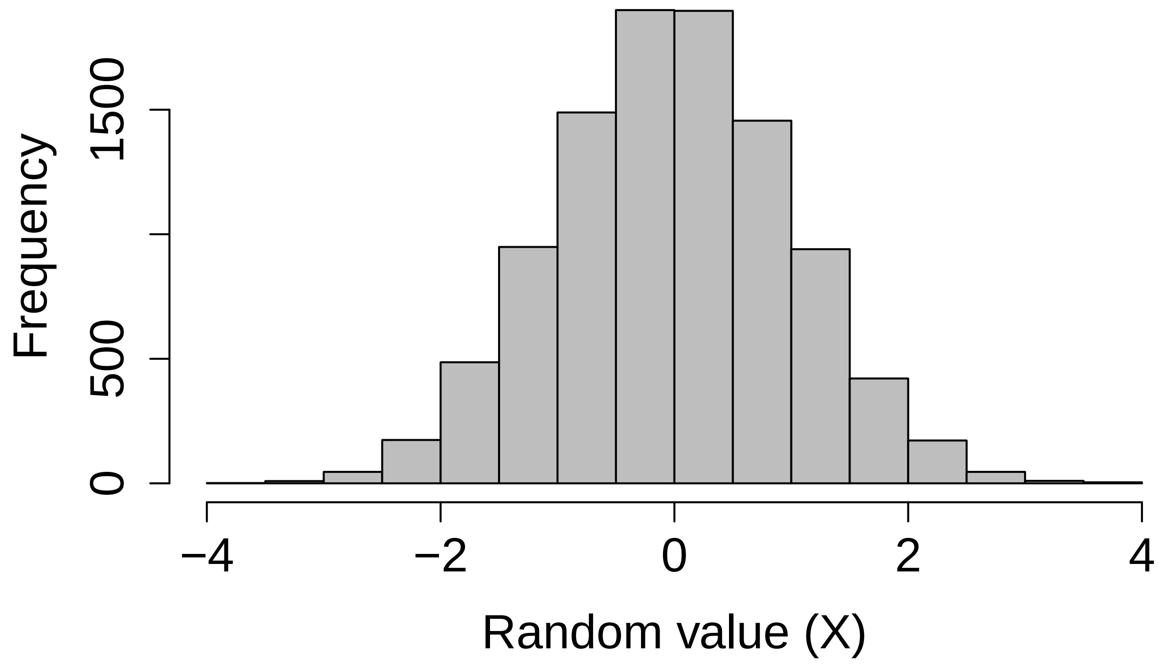
```
rand_norms_10 <- rnorm(n = 10, mean = 0, sd = 1);
```

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

```
##  [1]  0.60437831  0.00788268  0.92972575 -0.50330342 -0.65440283  0.46421298
##  [7] -1.51329634  0.04582298 -0.53638155 -0.55462332
```

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

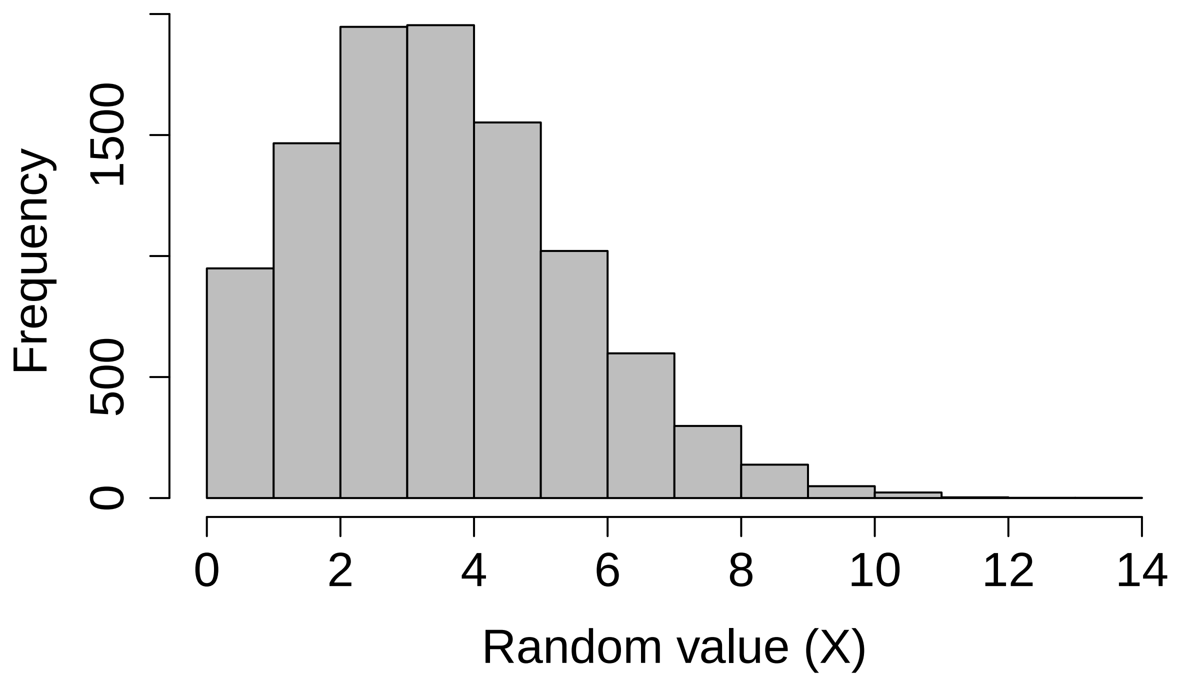
Many processes in biology can be described by a Poisson distribution. A Poisson process describes events happening with some given probability over an area of time or space such that $X \sim Poisson(\lambda)$, where the rate parameter λ is both the mean and variance of the Poisson distribution (note that by definition, $\lambda > 0$, 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] 0 1 1 1 1 2 0 1 0 0
```

There are no default values for `rpois`. We can plot a histogram of a large number of values to see the distribution when $\lambda = 4$ 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 describes the number of 'successes' for some number of independent trials ($\Pr(success) = p$). 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] 492
```

The above result shows that the coin came up heads 492 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);
```

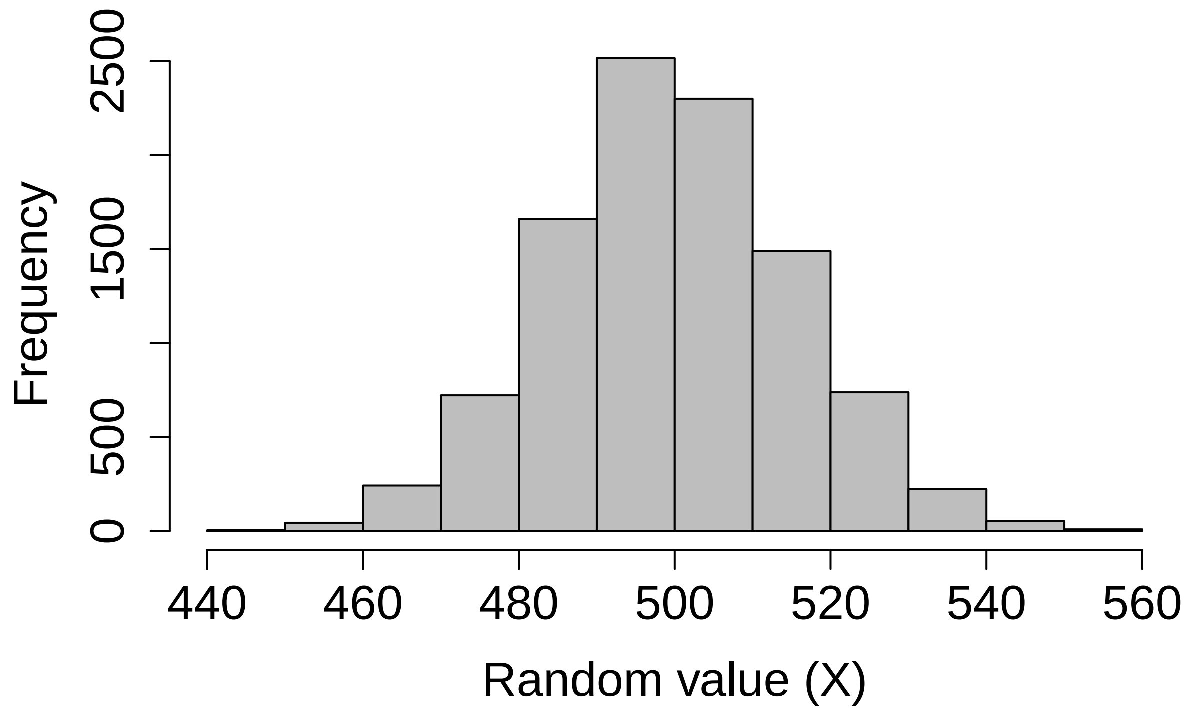
```
## [1] 495 513
```

In the above, a fair coin was flipped 1000 times and returned 495 heads, and then another fair coin was flipped 1000 times and returned 513 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 1 1 0 0 1 0 1 0
```

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 2. 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]  3  8  9  7  6  1  4  5 10  2
```

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]  7  1  9  1  9  7  8  1 10  6
```

Note that the numbers {1, 7, 9} 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]  6  1  3  7 10  7  6  5  8  9
```

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_C" "species_B" "species_C" "species_C"
##  [7] "species_B" "species_B" "species_C" "species_A" "species_A" "species_A"
```

Simulating data with known correlations

We can generate variables X_1 and X_2 that have known correlations ρ 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 x_1 (`x1` above) from a standard normal distribution. Then, x_2 (`x2` above) is calculated as below,

$x_2 = \rho x_1 + \sqrt{1-\rho^2} x_{rand}$,

Where x_{rand} is a sample from a normal distribution with the same variance as x_1. A simple call to the R function `cor` will confirm that the correlation does indeed equal `rho`.

```r
cor(x1, x2);
```

```
## [1] 0.3186687
```

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 library, which can be installed and loaded as below.

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

In the MASS 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 (M_1) has a mean of $\mu_{M_1} = 159.54$ and variance of $Var(M_1) = 12.68$, measurement 2 (M_2) has a mean of $\mu_{M_1} = 245.26$ and variance of $Var(M_2) = 30.39$, and measurement 3 (M_2) has a mean of $\mu_{M_1} = 25.52$ and variance of $Var(M_3) = 2.18$. 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 M_1 and M_2 to equal $Cov(M_1, M_2) = 13.95$, the covariance between M_1 and M_3 to equal $Cov(M_1, M_3) = 3.07$, and the covariance between M_2 and M_3 to equal $Cov(M_2, M_3) = 4.7$. We can put all of this information into a covariance matrix **V** 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 M is just the variable's covariance with itself; e.g., $Var(M_1) = Cov(M_1, M_1)$).

$$V = \begin{pmatrix} Var(M_1), & Cov(M_1, M_2), & Cov(M_1, M_3) \\ Cov(M_2, M_1), & Var(M_2), & Cov(M_2, M_3) \\ Cov(M_3, M_1), & Cov(M_3, M_2), & Var(M_3) \end{pmatrix}.$$

In R, we can create this matrix as follows.

```r
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
157.6022	244.7718	25.43982
157.4697	241.9243	23.95531
161.4644	252.4756	25.22423
154.6151	241.3430	25.47135
159.0964	246.7688	25.57913
154.1276	246.5419	28.14036
158.4892	238.5954	24.37487
161.1631	246.2832	26.22794
158.7663	254.4756	24.87081
161.5991	243.7458	25.42041
160.8843	244.5918	26.15107
157.1943	245.6360	26.57818
159.8916	243.3182	23.59380
165.3895	252.3520	27.74332
159.6425	245.8475	23.65327
165.6919	257.4951	29.04742
161.1758	243.8639	28.34780
162.2669	242.9349	24.80435
162.1760	247.8829	27.99384
158.8688	246.7905	24.25969
153.3596	239.0236	23.19760
162.3828	244.0643	24.94607
163.4313	243.0460	24.69998
159.2231	243.9788	26.19499
155.8779	235.5618	22.44783
161.0277	245.8152	26.18067
161.3104	250.3137	23.92232
162.2955	246.0536	24.72083
159.2961	245.6363	25.50928
161.2576	254.0271	27.03230
159.7353	243.0549	26.89184
162.5449	253.9657	28.12329
165.1509	249.9838	27.86599
153.0378	234.7000	23.84580
161.4127	244.3355	24.53738
163.2672	251.0519	25.53865
153.9410	241.3024	22.87275
161.6742	246.6190	26.70570
160.2237	245.9867	24.81407
160.9353	243.3741	24.48150

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
## 159.97399 245.73831  25.53515
```

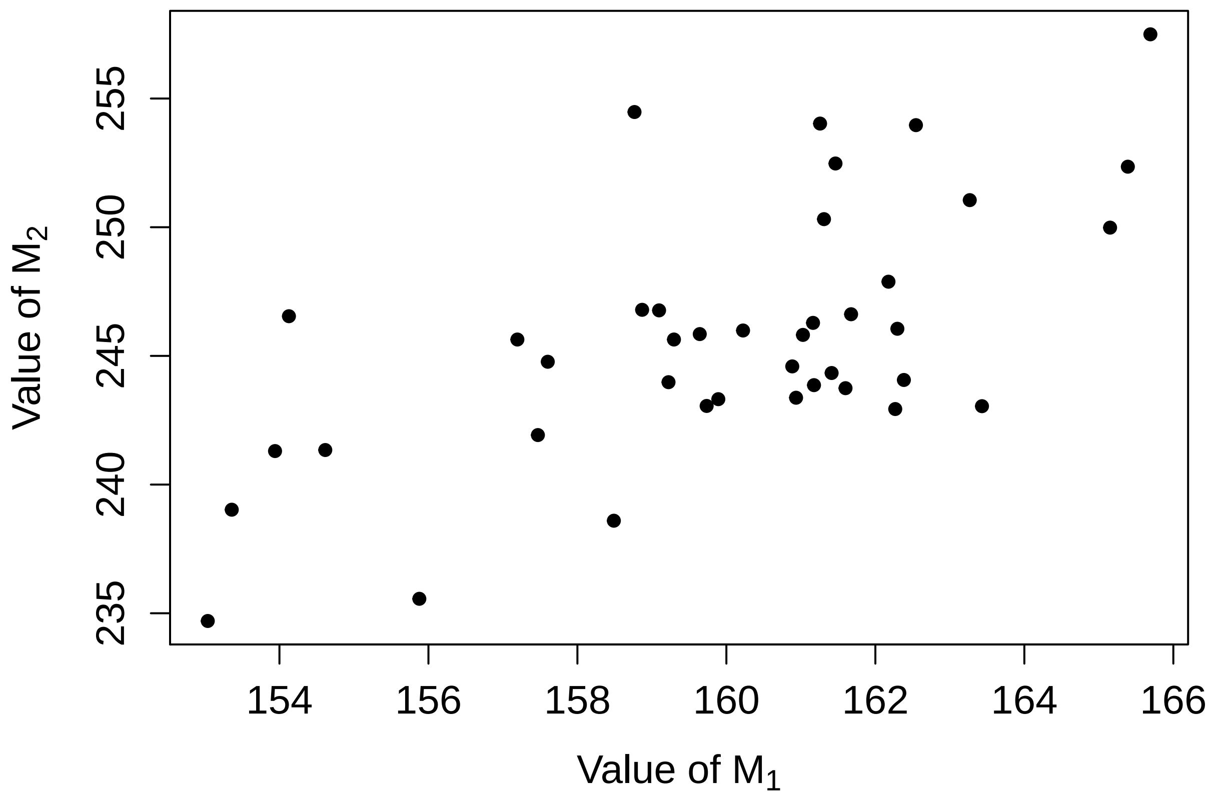
And we can check to confirm that the covariance structure of the data is correct using `cov`.

```
cov(sim_data);
```

```
##           M1        M2       M3
## M1 10.076346  9.953889 2.522992
## M2  9.953889 24.163529 4.784545
## M3  2.522992  4.784545 2.669000
```

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 M_1 and M_2 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 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 $\mathbf{V}$ (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 $N = 20$ 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	165.7233	245.3853	26.01012	4
species_1	155.6046	239.1075	24.18894	3
species_1	163.4912	249.5086	27.37282	2
species_1	161.8669	242.8096	25.92197	3
species_1	153.9133	236.6968	24.04611	1
species_1	156.7916	247.1902	26.68162	3
species_1	158.7524	246.2650	26.03986	2
species_1	155.6274	239.1490	25.50779	1
species_1	158.5183	237.6635	24.34570	2
species_1	161.0562	250.9858	25.12431	3
species_1	159.6725	244.9648	24.43121	4
species_1	160.3327	244.7104	28.64572	5
species_1	153.3150	239.8305	24.83063	3
species_1	162.0161	248.4019	27.73760	0
species_1	155.6808	242.6607	27.06744	1
species_1	163.5774	246.4818	26.12727	3
species_1	152.6626	239.6388	23.97588	4
species_1	160.2055	246.1951	25.53672	1
species_1	157.3337	242.9852	25.89966	5
species_1	157.1014	242.6270	24.58087	2

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.

```r
# 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	165.7233	245.3853	26.01012	4
species_1	155.6046	239.1075	24.18894	3
species_1	163.4912	249.5086	27.37282	2
species_1	161.8669	242.8096	25.92197	3
species_1	153.9133	236.6968	24.04611	1
species_1	156.7916	247.1902	26.68162	3
species_1	158.7524	246.2650	26.03986	2
species_1	155.6274	239.1490	25.50779	1
species_1	158.5183	237.6635	24.34570	2
species_1	161.0562	250.9858	25.12431	3
species_1	159.6725	244.9648	24.43121	4
species_1	160.3327	244.7104	28.64572	5
species_1	153.3150	239.8305	24.83063	3
species_1	162.0161	248.4019	27.73760	0
species_1	155.6808	242.6607	27.06744	1
species_1	163.5774	246.4818	26.12727	3
species_1	152.6626	239.6388	23.97588	4
species_1	160.2055	246.1951	25.53672	1
species_1	157.3337	242.9852	25.89966	5
species_1	157.1014	242.6270	24.58087	2
species_2	162.2139	252.1446	24.53397	1
species_2	153.0478	233.5664	22.53456	3
species_2	156.7220	241.5286	25.69019	4
species_2	156.8190	237.2602	25.43003	2
species_2	157.7582	249.9943	23.39491	0
species_2	156.4370	243.6116	25.18302	1
species_2	163.7550	249.5014	26.98091	4
species_2	158.3711	243.4549	23.41114	2
species_2	157.3587	241.9107	23.98068	3
species_2	156.1618	238.4674	23.00494	3
species_2	156.2991	243.7892	25.75436	4
species_2	159.8829	246.8511	27.61171	3
species_2	159.4107	244.1014	25.45222	2

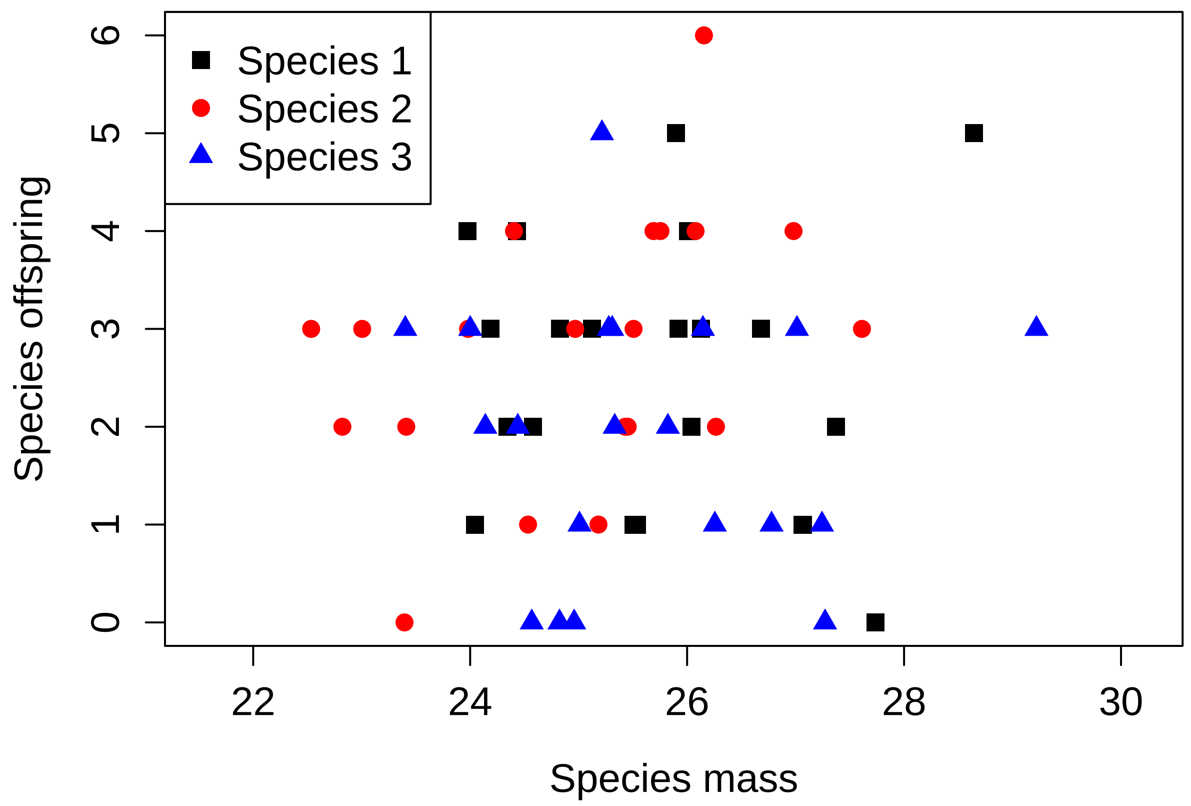
species	Length	Width	Mass	offspring
species_2	161.8764	255.3517	26.26608	2
species_2	160.7349	244.3218	24.96840	3
species_2	160.4128	245.2242	25.50661	3
species_2	159.0240	247.5599	26.15541	6
species_2	164.9370	247.6954	26.07797	4
species_2	154.3556	237.4766	22.82251	2
species_2	157.9366	241.1513	24.40430	4
species_3	161.6646	246.2174	25.82250	2
species_3	161.8532	241.3525	26.14582	3
species_3	161.9597	252.6404	25.33255	2
species_3	155.4538	239.6360	25.21504	5
species_3	156.9286	245.3507	26.77871	1
species_3	161.9480	248.9367	27.01266	3
species_3	160.5676	248.5774	27.24341	1
species_3	157.8673	243.2748	24.56773	0
species_3	149.0598	240.2099	24.00048	3
species_3	167.2622	254.7384	27.27187	0
species_3	159.1154	251.6606	25.00836	1
species_3	153.9687	240.0937	24.82369	0
species_3	162.4386	245.8779	26.25580	1
species_3	162.7365	245.7652	24.14015	2
species_3	155.3541	238.9556	23.40205	3
species_3	165.1910	249.8727	29.22020	3
species_3	161.5354	248.6516	24.95895	0
species_3	158.9878	251.0800	25.27713	3
species_3	158.5102	250.1244	24.43942	2
species_3	158.3598	244.5510	25.30946	3

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 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.

```r
aov_result <- aov(Mass ~ species, data = dat);
summary(aov_result);
```

```
##             Df Sum Sq Mean Sq F value Pr(>F)
## species      2    6.6   3.302   1.754  0.182
## Residuals   57  107.3   1.882
```

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");
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
##    216.8    223.1   -105.4    210.8       57
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -1.6282 -0.8495  0.2157  0.4750  2.1212
##
## Random effects:
##  Groups  Name        Variance Std.Dev.
##  species (Intercept) 0.00687  0.08289
## Number of obs: 60, groups:  species, 3
##
```

```
## Fixed effects:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.07017    1.56273   0.045    0.964
## Mass         0.03205    0.06114   0.524    0.600
##
## Correlation of Fixed Effects:
##      (Intr)
## Mass -0.998
```

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.