Data Science in Earth and Environmental Systems: ES7023 - Assignment 2

Random numbers are fundamental to modeling and statistics. They allow us to represent variability or uncertainty in model inputs, parameters, equations, statistics, etc. As we covered in class, a random variable can be characterized by its probability distribution function or cumulative distribution function, which represents the probability that the variable takes on different values. Common distributions include the normal distribution (Gaussian) and uniform distribution, but there are many more.

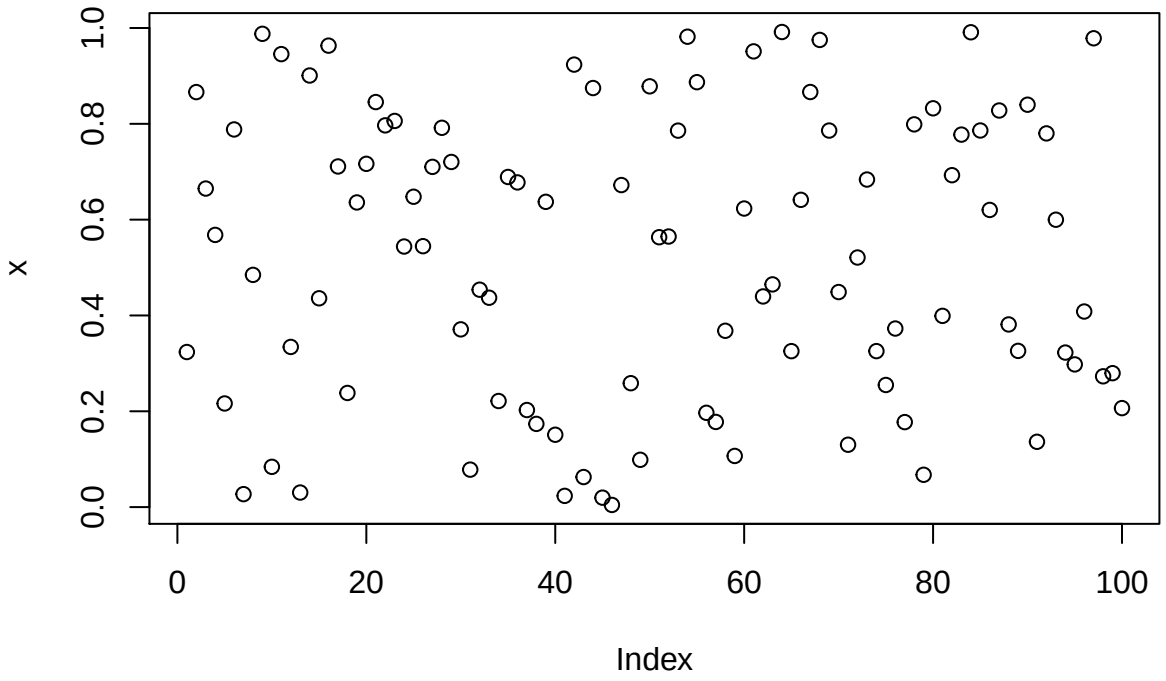
Computers are deterministic. One of the great strengths of computers is that they can be counted on to do the same thing over and over. But how can we generate something random from a deterministic machine? The short answer is that we can't, but we can generate something that for all practical purposes appears to be random, i.e., a pseudo-random number.

A common type of random variable is one with a uniform distribution. When we write $X \sim U(a, b)$, which means that X is a random variable with a uniform distribution between a and b. Most software packages have a built in function to generate $U(0, 1)$ variables. There are many techniques developed to generate pseudo-random numbers. A common one uses what is called a linear congruential generator (LCG) which uses the recursive formula:

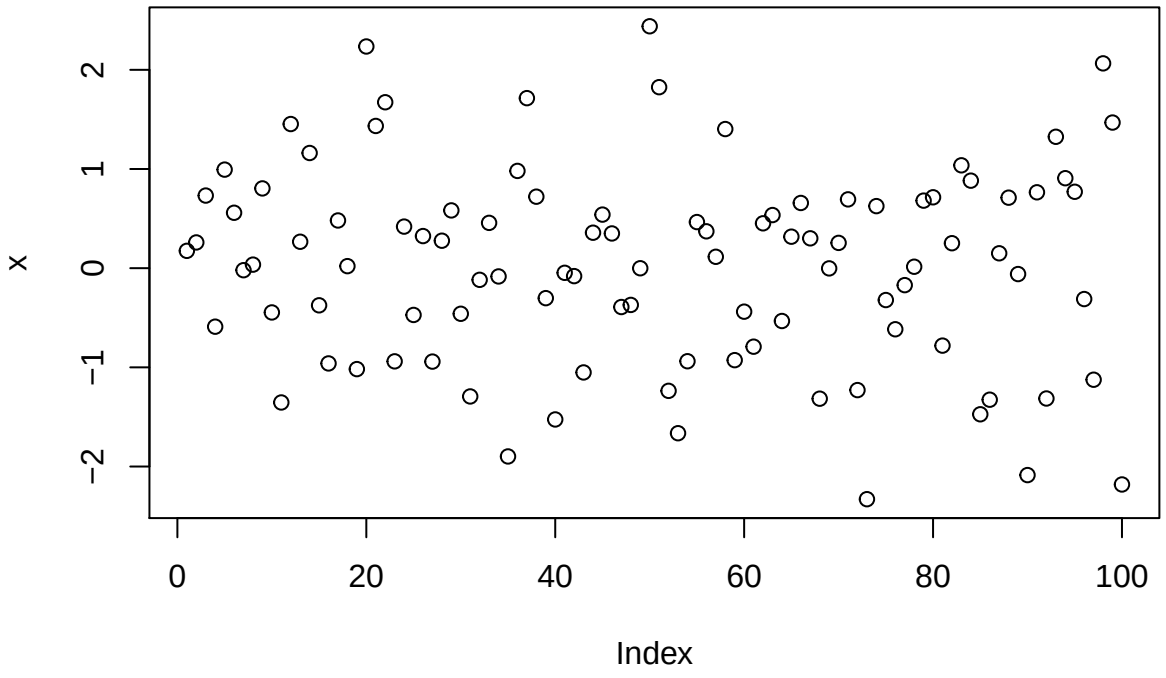
$$x_{n+1} = (ax_n + b) \pmod{m}$$

where the initial value, x_0, is called the seed, and 'mod' is the modulus operation. To obtain a value between 0 and 1, the computer usually further divides the answer by m. This formula relies on values of a, b, and m. Some values will result in very non-random series (i.e., a periodic sequence) while others produce series that appear in many respects random. Many random number generators (for example, that employed in the most widely-used version of C) uses values of $(a, b, m) = (1103515245, 12345, 2^{32})$. Another common distribution is the normal, or Gaussian distribution, which we write as $N(\mu, \sigma^2)$, where μ is the mean and σ^2 is the variance. Normal distributions can be generated from uniform distributions using something called the Box-Muller Method. Rather than getting into that, let's get started with the actual business of generating random numbers. Let's begin by generating and plotting some random variables.

```
#first uniform
n = 1e2 #define number of samples (i.e. draws) from distribution
x = runif(n, min=0, max=1)  #draw n values from U(0,1) distribution
plot(x) #plot values
```



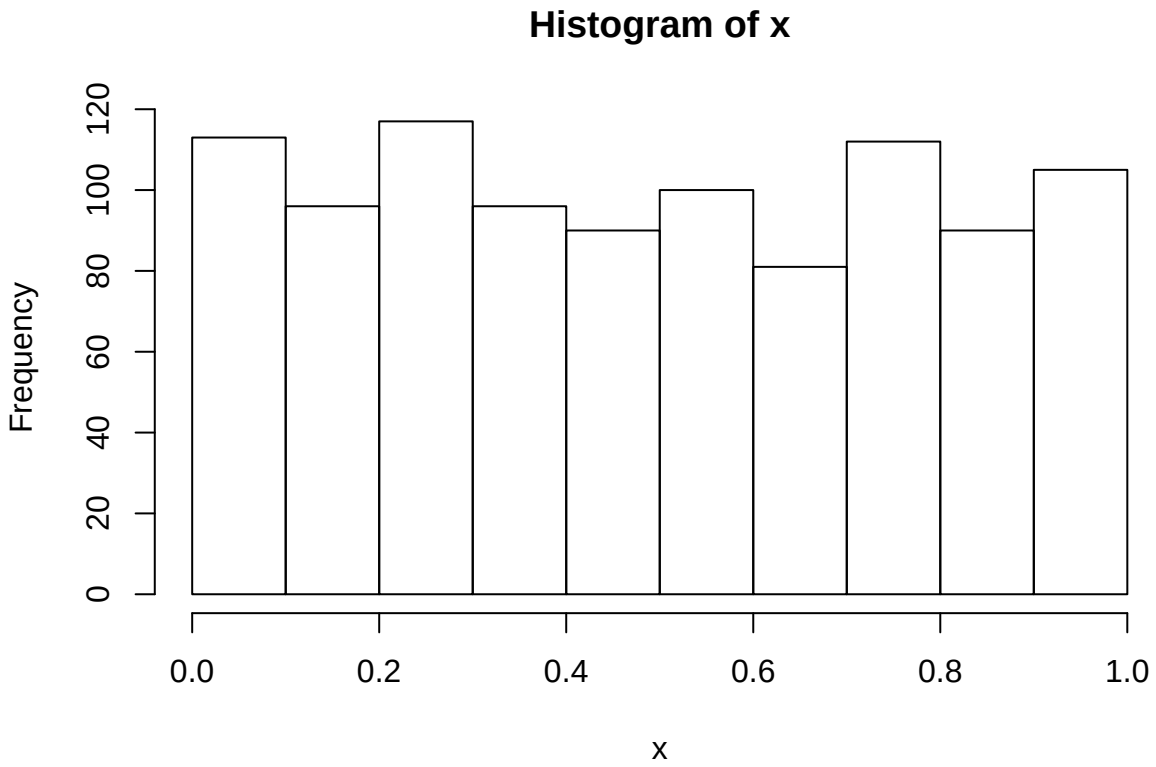

```r
#now normal
x = rnorm(n, mean=0, sd=1)
plot(x)
```




```r
#if you want to check the options for rnorm, help(rnorm) or ?rnorm
#   would do the same thing. example(rnorm) would run some examples
```

Now let's test how well the uniform random number generator in R works. First generate a vector of 1000 draws from a U(0,1) distribution and check out the histogram for this sample

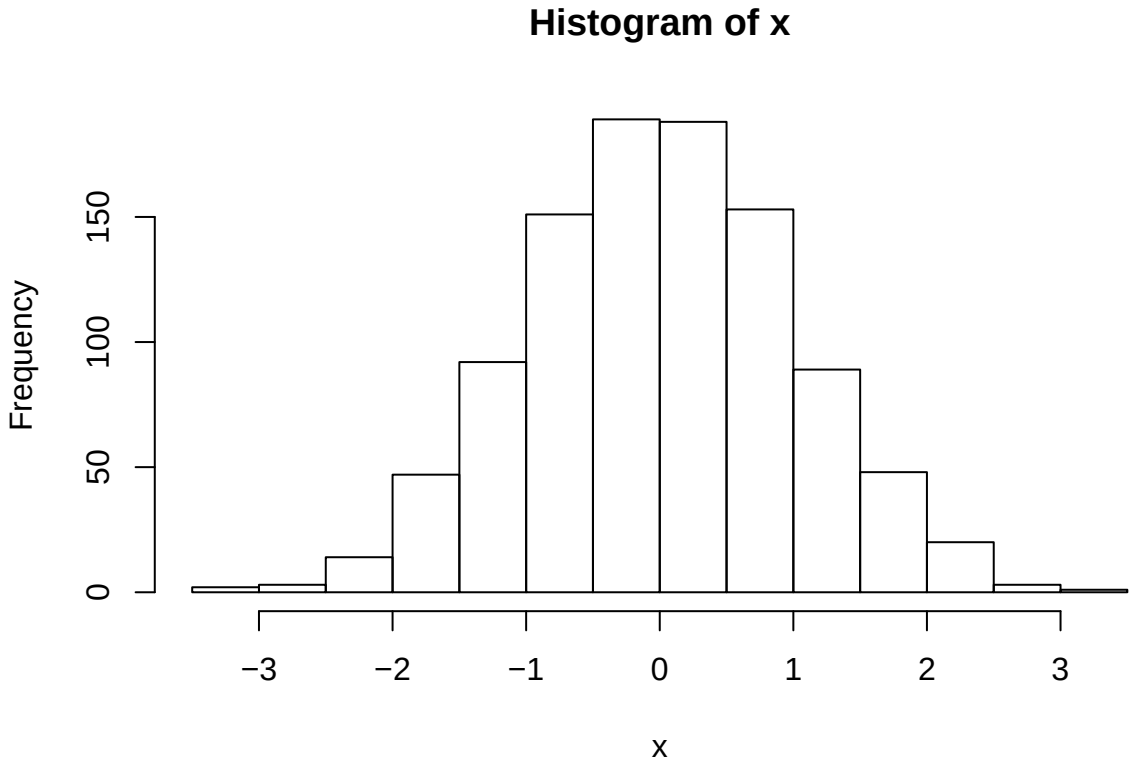
```r
x = runif(1e3)  #this will generate with default, min=0, max=1
hist(x)     #plots histogram
```



The histogram should look fairly flat, since our expectation is that each bin should have 1/10th of the values. But for any given sample the exact number in each bin will vary. Try it a few times and see how the histogram shape changes. What happens if you increase the size of the sample?

Now try a normal distribution:

```
x = rnorm(1e3, mean=0, sd=1)
hist(x)
```



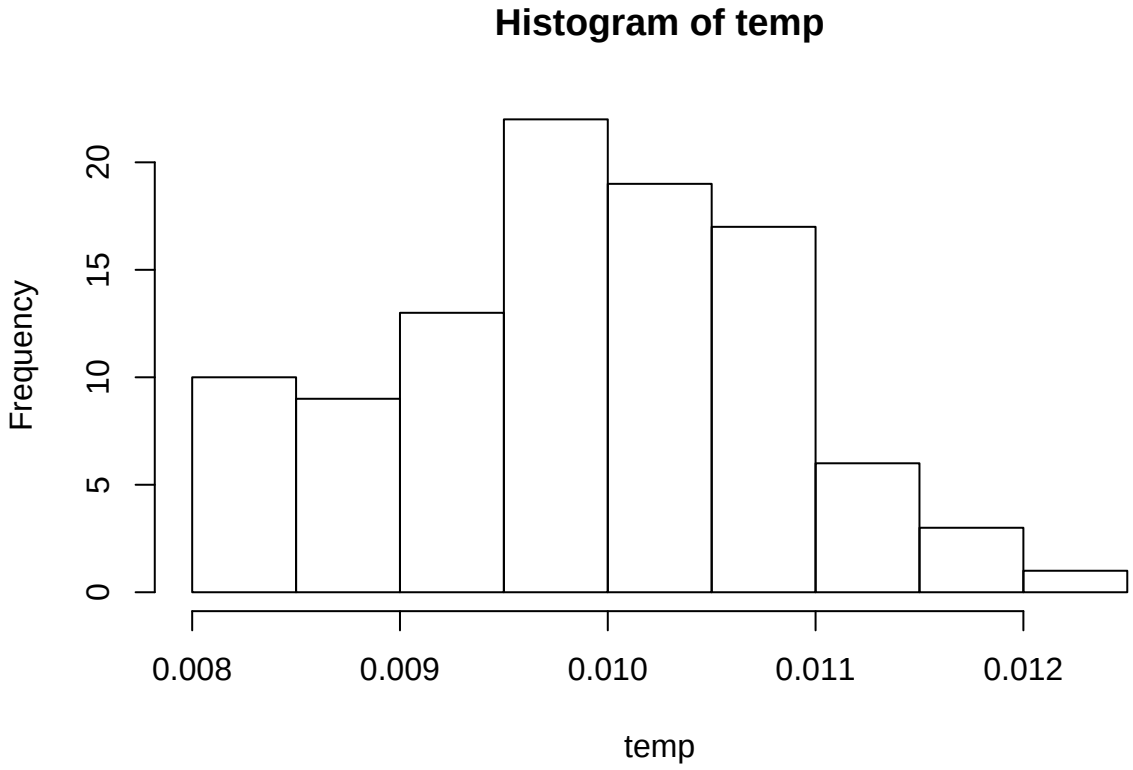
That was a visual check. Now let's compute some values and compare them to our expectation. For example, we can count how many of the values for the uniform draw are above 0.99. We would expect each value to have a 1% chance of exceeding this value. First do it once:

```
n = 1e4
sum(runif(n) > .99)/n
```

```
## [1] 0.0105
```

Now let's see whether, if we do this several times, our expectation that our statistic equals 0.01 is consistent with R.

```
n = 1e4 #length of random sequence
nrun = 1e2  #number of times to generate sequence and compute statistic
temp = numeric(nrun)    #vector of length nrun to save results
for (i in 1:nrun) {
   x = runif(n)
   temp[i] = sum ( x > 0.99) / n #save statistic
}
hist(temp)
```




```
#print the mean +/- 2 standard deviations (use ?paste to see command syntax)
paste("Confidence interval for true value", mean(temp)-2*sd(temp)/sqrt(nrun),
      mean(temp)+2*sd(temp)/sqrt(nrun))
```

```
## [1] "Confidence interval for true value 0.00973758612468561 0.0101164138753144"
```

It looks like the confidence interval for the true value of the statistic for our random number generator includes 0.01, which means we can't reject the generator based on this test. Obviously, many more checks of the random number generation could be done, but let's move on. Let's demonstrate some of those basic formulas and principles you learned in statistics class. For example, the standard error of the sample mean for a normally distributed variable is supposed to be the standard deviation divided by the square root of n (see: https://en.wikipedia.org/wiki/Standard_error): $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$

```
n = 1e2
nrun = 1e3
temp = numeric(length = nrun)
sd = 1
for (i in 1: nrun) temp[i] = mean(rnorm(n, mean = 0, sd = sd))
#print the stdev of the mean (i.e. standard error) and our expectation
paste(sd(temp), sd/sqrt(n))
```

```
## [1] "0.0986622608541199 0.1"
```

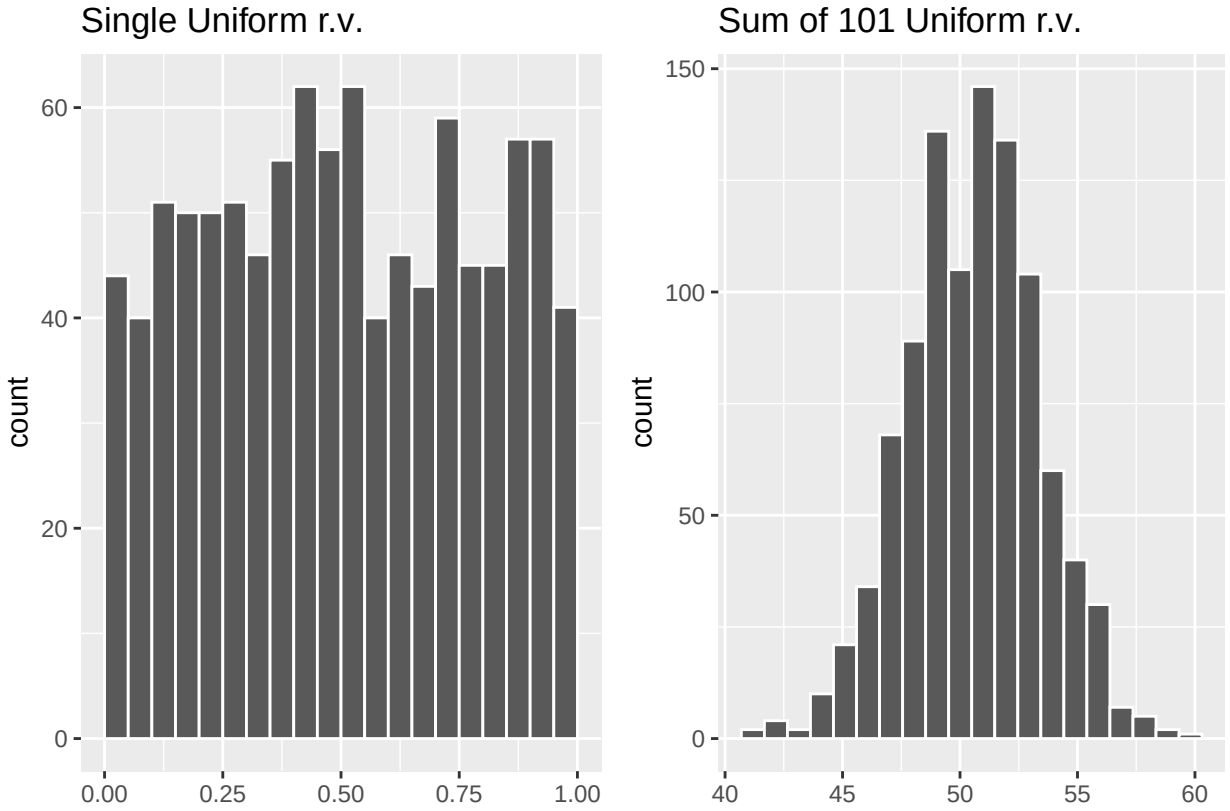
Something else you probably learned is the central limit theorem, which says that the sum of independent random variables tends toward a normal distribution, even if the random variables are not themselves normally distributed (see https://en.wikipedia.org/wiki/Central_limit_theorem). This is why errors are usually represented by normal distributions, since it is assumed that errors result from a lot of different variables that are unknown.

```
temp1=temp2 = runif(1e3)    #start with a sample from a uniform distribution
#add 100 other independent samples from uniform distributions
for (i in 1:1e2) {
```

```
  temp2= temp2 + runif(1e3)}
par(mfrow=c(1,2))
hist(temp1, main = "Single Uniform r.v.")
hist(temp2, main="Sum of 101 Uniform r.v.")    #plot histogram (not shown here)
```

```
# here we do the same thing ggplot()
df=data.frame(unif=temp1,sum.unif=temp2)

g1=ggplot(df)+geom_histogram(aes(unif), breaks = seq(0,1,0.05), col="white")+
  labs(title="Single Uniform r.v.",x="")
g2=ggplot(df)+geom_histogram(aes(sum.unif), bins = 20, col="white")+
  labs(title="Sum of 101 Uniform r.v.",x="")
grid.arrange(g1,g2,nrow=1)
```



Note that this could also be done in one line:

```
hist(apply(array(runif(1e3*1e2),dim=c(1e3,1e2)), 1, sum))
```

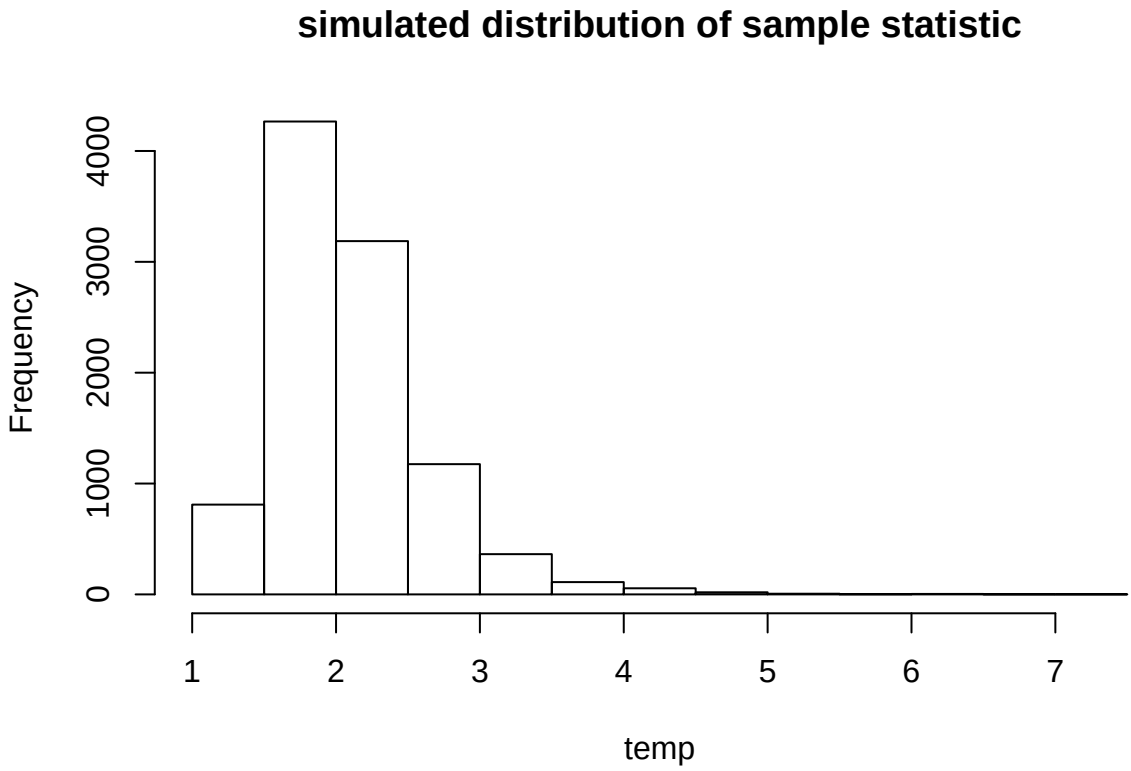
The `apply` command in R is a useful space and time saver, as it allows you to apply functions over margins of an array instead of using a for loop to do the same. The first term is the input array, the second is the margin over which to apply the function, and the third is the function to be applied. Type `?apply` in the console for details.

Note : The `purrr` library (from tidyverse) provides `map` functions to replace the apply family that are more coherent in their use and more easy to remember. See : https://r4ds.had.co.nz/iteration.html#the-map-functions

Finally, let's say that we have forgotten the formula for the variance of a statistic we want to compute, such as the ratio of two normally distributed variables. Or, alternatively, let's say that no such formula exists because the statistic does not follow a standard distribution. Now you have the power to simulate this

statistic and see what its distribution is. For example, let's look at the distribution for a ratio between two normally distributed variables.

```
nrun=1e4
#vector to save computed statistic for each sample
temp = numeric(length= nrun)
for (i in 1: nrun)
   temp[i] = rnorm(1, mean=10, sd = 1) / rnorm(1, mean = 5, sd=1)
hist(temp, main="simulated distribution of sample statistic")
```




```
paste(mean(temp), sd(temp))
```

```
## [1] "2.08636539429238 0.534615475229954"
```

What if I want to generate a distribution other than the uniform or normal? Good question! There are a few more built-in distributions to R (see help), but in general you will need to design your own random number generator. These are usually based on first drawing from the uniform distribution, and then transforming the value drawn using the inverse of the CDF for the desired distribution. You can look up "inverse transform sampling" if you are interested to learn more.

Assignment (Include all code and plots)

From this assignment forward you will submit a mini-paper. In your mini-paper you should present your results for the problems below and discuss what you think are the important features. *In this assignment, focus on what the results suggest about modeling.*

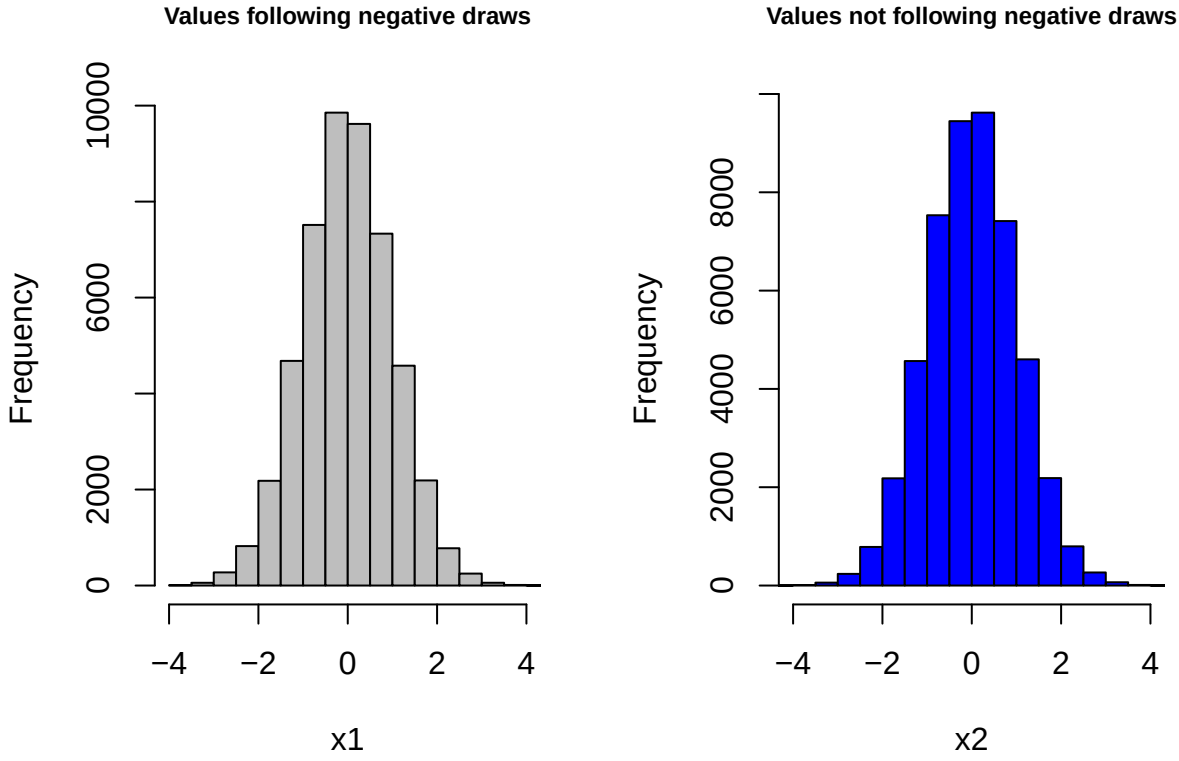
For problems 2 and 3, start your R code with the command `set.seed(42)` before running the rest of the code. The seed number is the starting point used in the generation of a sequence of pseudorandom numbers. This will ensure that you will get the same result every time you run your pseudorandom number generator.

Note that you'll need to run the command anytime you run the code, not just once at the beginning of a session.

Problem 1:

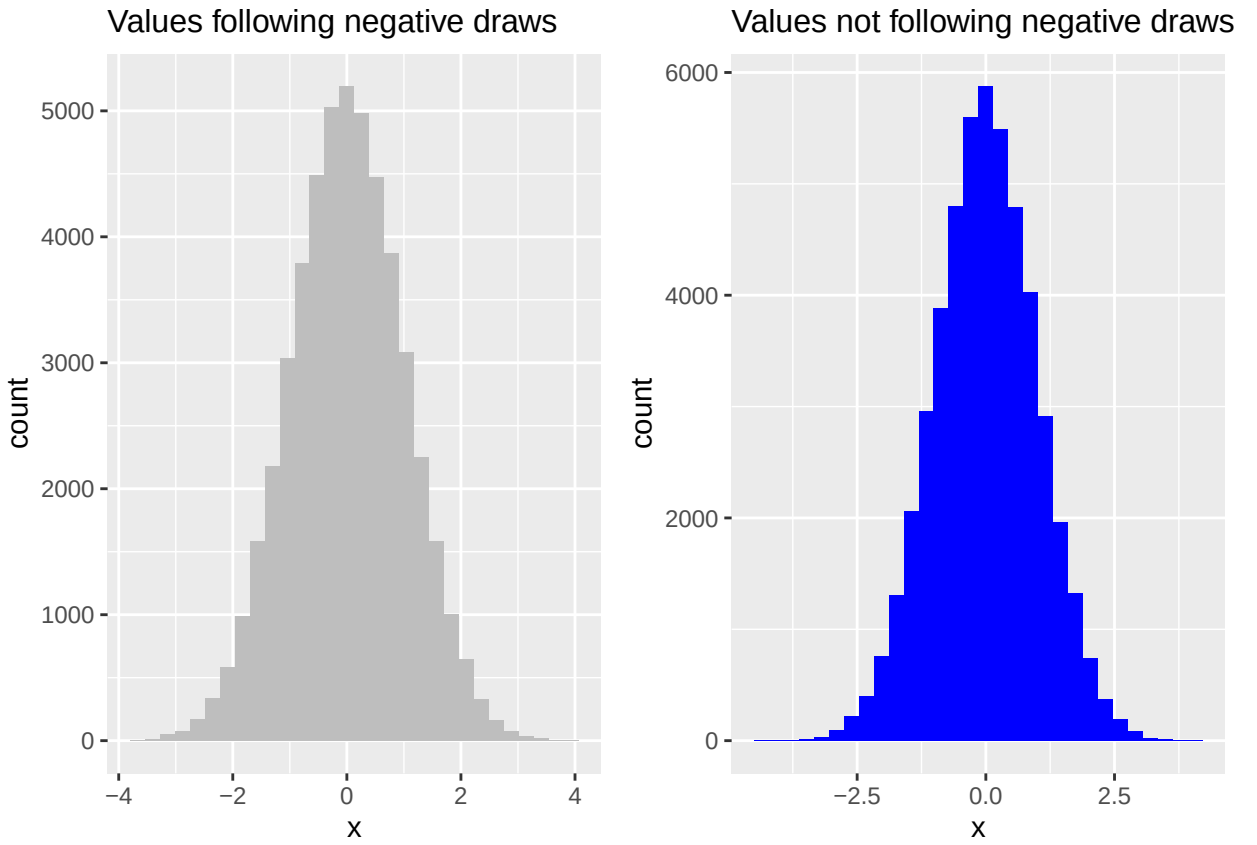
Let's look at the normal distribution again. We may want to check that successive draws are statistically independent from the previous draw. Let's compare the distribution of values following negative and positive draws.

```
nsamp = 1e5
x = rnorm(nsamp)
index = which(x[1:(nsamp-1)] <= 0)   #find which values are negative
#take a subset of x, equal to all those values following negative values
x1 = x[index+1]
#take the other values, note in r a negative index means to omit these terms
x2 = x[-(index+1)]
#now let's plot them one on top of the other
par(mfrow=c(1,2))   #split window into two rows and one column
hist(x1, col="gray", main = "Values following negative draws",xlim=c(-4,4),cex.main=0.75)
hist(x2, col="blue", main = "Values not following negative draws",xlim=c(-4,4),cex.main=0.75)
```




```
# Rewrite using dplyr and ggplot (a bit more complicated...)
nsamp = 1e5
# creates a data from with two columns index and random sample
df <- data.frame(i=1:nsamp, x=rnorm(nsamp))
#filter for negative values and adds a column to the output with the index+1
df_neg <- filter(df,  x <= 0) %>% mutate(i_next = i+1)
#filter for the values following the negative values
df_neg_next <- filter(df, i %in% df_neg$i_next)
#filter for the rest using (! in the 'not' operator)
```

```
df_not_neg_next <- filter(df, !(i %in% df_neg$i_next))
#plot using ggplot
g1 <- ggplot(df_neg_next, aes(x=x)) + geom_histogram( fill = "grey") +
  ggtitle("Values following negative draws") +theme(plot.title=element_text(size=12))
g2 <- ggplot(df_not_neg_next, aes(x=x)) + geom_histogram( fill = "blue") +
  ggtitle("Values not following negative draws")+theme(plot.title=element_text(size=12))
grid.arrange(g1,g2,nrow=1)
```



What do you think? We can test if x1 and x2 have statistically different distributions using the Kolmogorov-Smirnov test.

```
ks.test(x1,x2)
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  x1 and x2
## D = 0.0083893, p-value = 0.05926
## alternative hypothesis: two-sided
```

```
# dplyr
ks.test(df_neg_next$x, df_not_neg_next$x)
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  df_neg_next$x and df_not_neg_next$x
## D = 0.008129, p-value = 0.07346
## alternative hypothesis: two-sided
```

Explain what we just did here. What is x1? x2? what do the plots show? Look up the Kolmogorov-Smirnov test and explain the results of the test. What can you conclude? We only expect a few senstences here.

Problem 2:

Often when comparing two variables we want to know whether their correlation is "significant." Plot the distribution for the correlation between two independent, normally distributed random variables of length n=50. To compute the distribution, you will want to compute the statistic (correlation) many times (say, 5000) and save the result each time, as we did in the example of the ratio between two normally distributed variables. What is the likelihood that a correlation with absolute value greater than 0.3 is computed even though the two variables are independent? Answer these questions also for n=20. To do this you will need the function to find the correlation, which is `cor()`, and to find the absolute value, which is `abs()`. *Remember what we said earlier:* You should present your results for the problems and discuss what you think are the important features. Focus on what the results suggest about modeling.

Problem 3:

Let's say that we now have one variable v of length n we wish to correlate with m other variables. For all m between 1 and 100, compute the distribution for the maximum absolute correlation coefficient between v and each of the other m variables, assuming the variables are independent. (This is what we would typically call the null distribution, meaning the distribution assuming the null hypothesis.) For both n=20 and n = 50, plot m on the x-axis and the probability of getting a maximum absolute correlation above 0.4 on the y-axis. To compute the maximum you will need to use the function `max()`. Include both sets of values (n=20 and n=50) on the same plot and differentiate by color. Don't forget to include a legend.

Problem 4 (For Graduate Students Only):

For the dataset(s) you have chosen for your project:

- What are the variables and are they discrete or continuous, nominal or ordinal, binary or count?
- If applicable, what research questions have been previously asked using this dataset? What methods/models were used?
- What question(s) would you like to address using this dataset? Note that this can evolve over the course of the class.
- Do you need additional data to answer your question? If so, please find the relevant datasets before the next assignment. Otherwise, try to refine your research question and/or note this limitation.