# PLSC 30600 - Lab 1 - Monte Carlo Simulation

06/01/2022

## Using simulation to understand properties of random variables

A central topic in statistics is determining the properties of different random variables. What are their expectations and variances? Does a sequence of random variables converge to some known distribution? A lot of this can be done theoretically, but sometimes it can be challenging to derive properties analytically.

A very common tool that is used to obtain numerical results in lieu of analytical ones is **monte carlo** simulation. The principle is simple: **we can approximate the properties of some random variable $X$ – its mean, variance, etc… – by taking a repeated number of *i.i.d.* draws from that random variable and computing them from the empirical distribution**. For example, if we wanted to know the expectation of some random variable, we can take a large number of independent, repeated draws from that random variable, store those draws, and compute the average. As we let the number of draws get arbitrarily large (our only limitation is computing resources and time), this will converge to the **true** mean.

In practice, you see simulations in statistical methods papers all the time, often to illustrate certain properties of an estimator where the intuition may not be clear just from the analytical result or to get some sense of properties that are difficult to derive. For example, in papers where much of the theory relies on asymptotic approximations, we may use simulations to get a sense of how good the approximation is in small samples or to compare the performance of different estimators across fixed sample sizes.

You also will find simulations useful as a way of checking analytical results – it's easy for a proof to go wrong or to be unsure of some of the steps, so it can help to use a simulation to understand what the correct answer *should* be at least for a particular set of parameter values. Simulations can be a great way of generating intuition when analytical results are hard to come by.

## Common tools of simulation

In conducting a simulation, you will typically rely on two sets of programming tools: **random number generator functions and loops**.

### Random number generators

- R has a suite of functions to allow you to generate (pseudo)-random numbers from particular distributions. These functions typically start with `r` and the name of the distribution. + For example, the function to generate random draws from the normal is `rnorm`.
- Built-in functions cover most of the common ones: uniform, bernoulli, binomial, normal, poisson, etc… and other packages exist for more esoteric distributions.
- Because computers are deterministic, the algorithms that generate these numbers are not truly "random" in the sense that the exact sequence of random numbers can be reproduced if one knows the initial "seed" for the algorithm.
- These random number generators create sequences of numbers that have the **properties of randomness** (e.g. independence), but they can be replicated for a fixed seed.

- This is actually great for us researchers since it ensures replicability – even if you are using a simulation, I can reproduce your exact numeric results if I fix the seed. However, this does mean that you need to **be aware of when you set the seed in your algorithm**. Set it once in your script or code fragment and don't change it.

```r
# Set the seed for a random number generator
set.seed(60615)

# Generate N=100 random normal variates
rand_norm <- rnorm(100, mean=2, sd=1)
rand_norm
```

```
##   [1]  3.177803828  2.699537486  3.452707642  0.951288136  1.771326693
##   [6]  2.347484686  3.403583177  1.921497102  0.344160841  0.758095289
##  [11]  1.362224509  1.782994464  3.675690639  0.897181172 -0.011361749
##  [16]  3.197864508  2.540413561  2.704224159  1.252209427  2.220283386
##  [21]  0.682560942  3.881767080  3.329664530  2.014444939  0.661520720
##  [26]  0.746601716  1.971236450  0.167340848  3.165509043  2.235777065
##  [31]  1.934975281  2.986871944  2.162361725  2.766149727  1.806696196
##  [36] -0.415526798  1.177105418  2.750761865  1.669299213  3.948017600
##  [41]  3.138411290  3.233340132  2.516623184  1.745224865  2.697083090
##  [46]  2.592984326  1.593352574  2.514009165  2.823842735  1.329125468
##  [51]  2.245305529 -0.387633262  1.067632328  1.074539886  1.764802060
##  [56]  1.760182087  2.654693025  1.053077481  3.079729246  2.248718754
##  [61]  3.402024465  1.212838810  2.331637116  0.704194393  1.986629426
##  [66]  1.605711895  3.267781792  2.063043621  2.660811911  1.328759942
##  [71]  3.209610223  3.311937812  0.920808525  1.998380962  2.250374366
##  [76]  1.102206893  1.643036635  1.404690501  2.789432103  2.315972390
##  [81]  1.995946177  1.659533996  2.368277409  2.831532737  1.766579986
##  [86]  2.154084555 -0.007013178  2.706771553  2.078045085  2.441507605
##  [91]  2.513138393  1.192162269  2.577472843  1.774624997  0.559602706
##  [96]  2.737428211  2.026011482  3.147517626  0.052643727  1.097017805
```

```r
# Calculate the mean - it's close to 2
print(mean(rand_norm))
```

```
## [1] 2.000202
```

```r
# Note that if I run this fragment again and again, I will get the exact
# same values in rand_norm - this is because the seed is fixed at the top of
# the fragment!
```

## Loops

- In a simulation, you will need to **generate draws from a random variable multiple times**. Often this will be some sort of function of other random variables – remember, all an estimator is a function of other random variables.
- There will not be an existing routine that does this for you, so you will need to write your own code that takes as **input** some random numbers and generates the desired **output** (e.g. a difference-in-means, a regression estimator, etc. . . ).

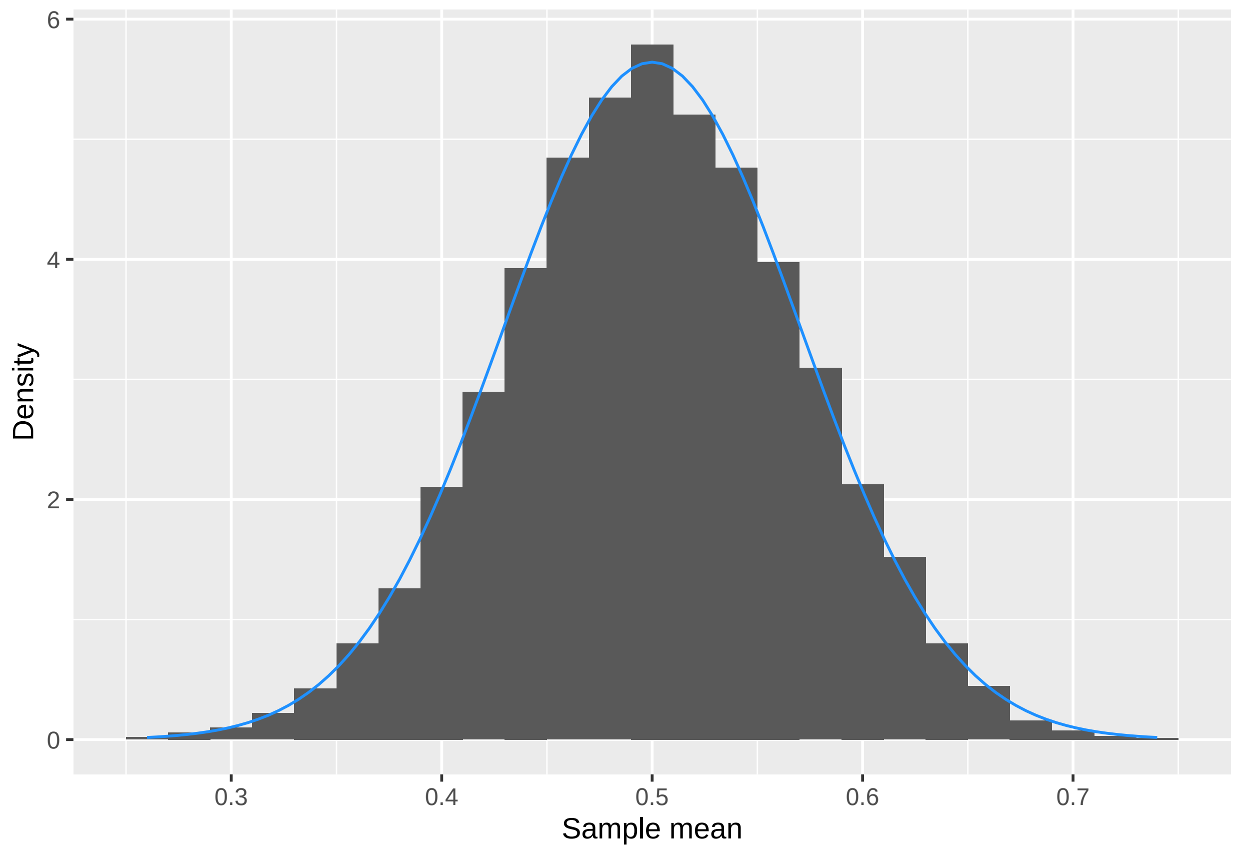
- You probably don't want to copy and paste this code 1 million times, so you'll need a way of repeating the same process with different random inputs and storing the results.

- The easiest way of doing this is a `for` loop. Here we will use monte carlo simulation to approximate the mean of the sampling distribution of the sample mean with $N = 30$, $E[X_i] = .5$, $X_i \sim$ Bernoulli i.i.d.

```r
set.seed(60615) # Set random seed
niter <- 10000 # Number of for loop iterations - this can be as big as we need for precision
sample_means = rep(NA, niter) # Placeholder vector to store loop results
for (i in 1:niter){ # The loop will execute niter times, each time i will increment by 1 but everything
  sample_means[i] <- mean(rbinom(n=50, size=1, prob=.5))
}

# What's the mean of the sampling distribution? The population mean E[X_i]
mean(sample_means)
```

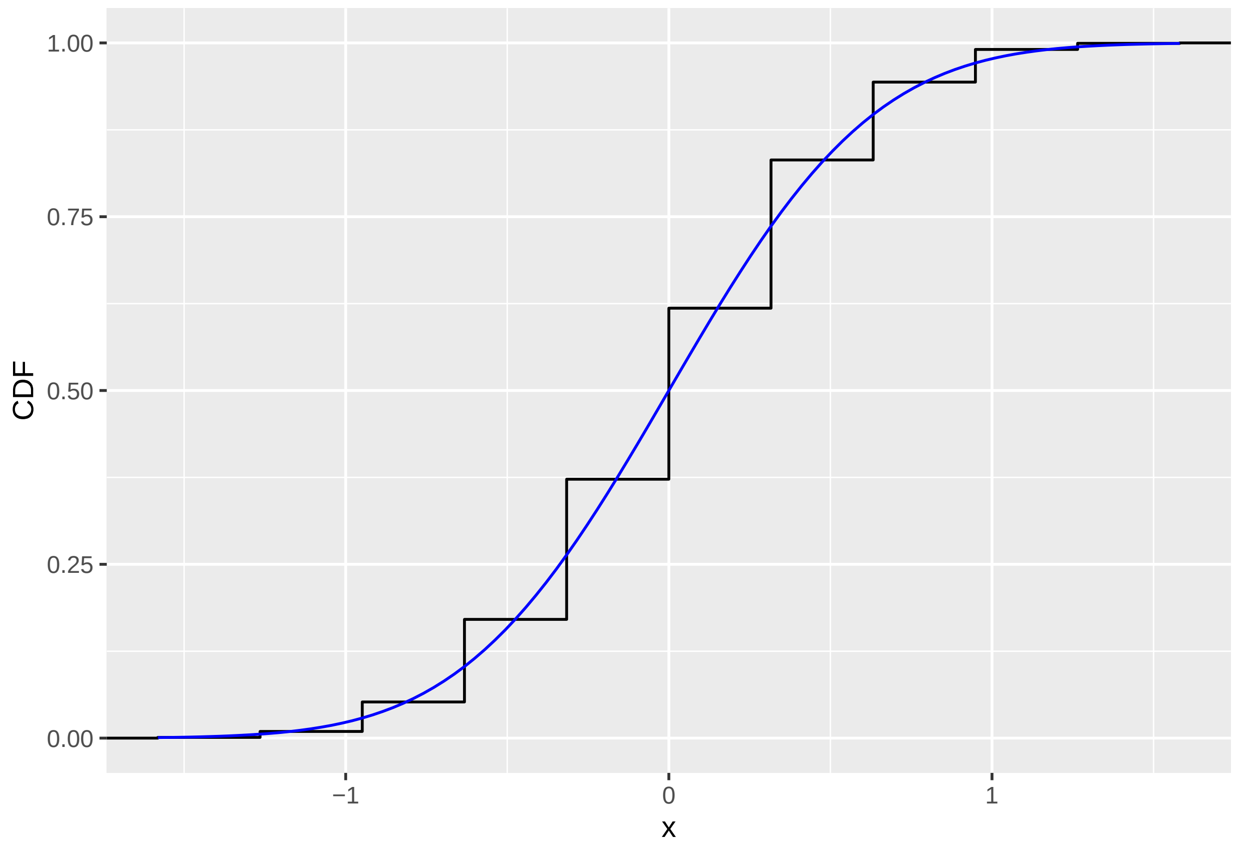
```
## [1] 0.500516
```

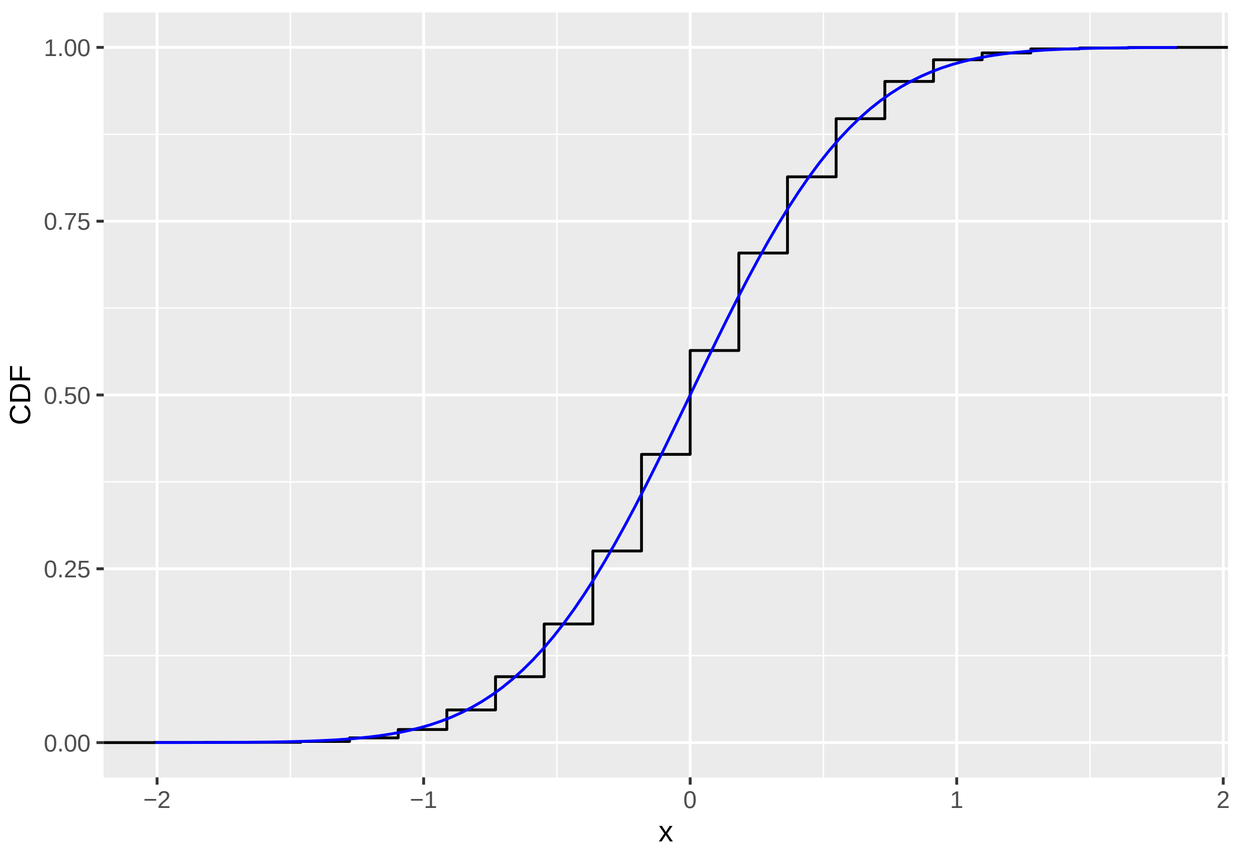
```r
# What's its distribution? It's not bernoulli! It's (roughly) normal (by the CLT)
tibble(xbar = sample_means) %>%
  ggplot(aes(x=xbar)) +
  geom_histogram(aes(y = ..density..), binwidth=.02) +  # Do this to get the histogram density and not
  xlab("Sample mean") +
  ylab("Density") +
  stat_function(fun=dnorm, args=list(mean=.5, sd=sqrt(.25/50)),
                col="dodgerblue") # Overlay a normal curve w/ asymptotic mean/variance
```

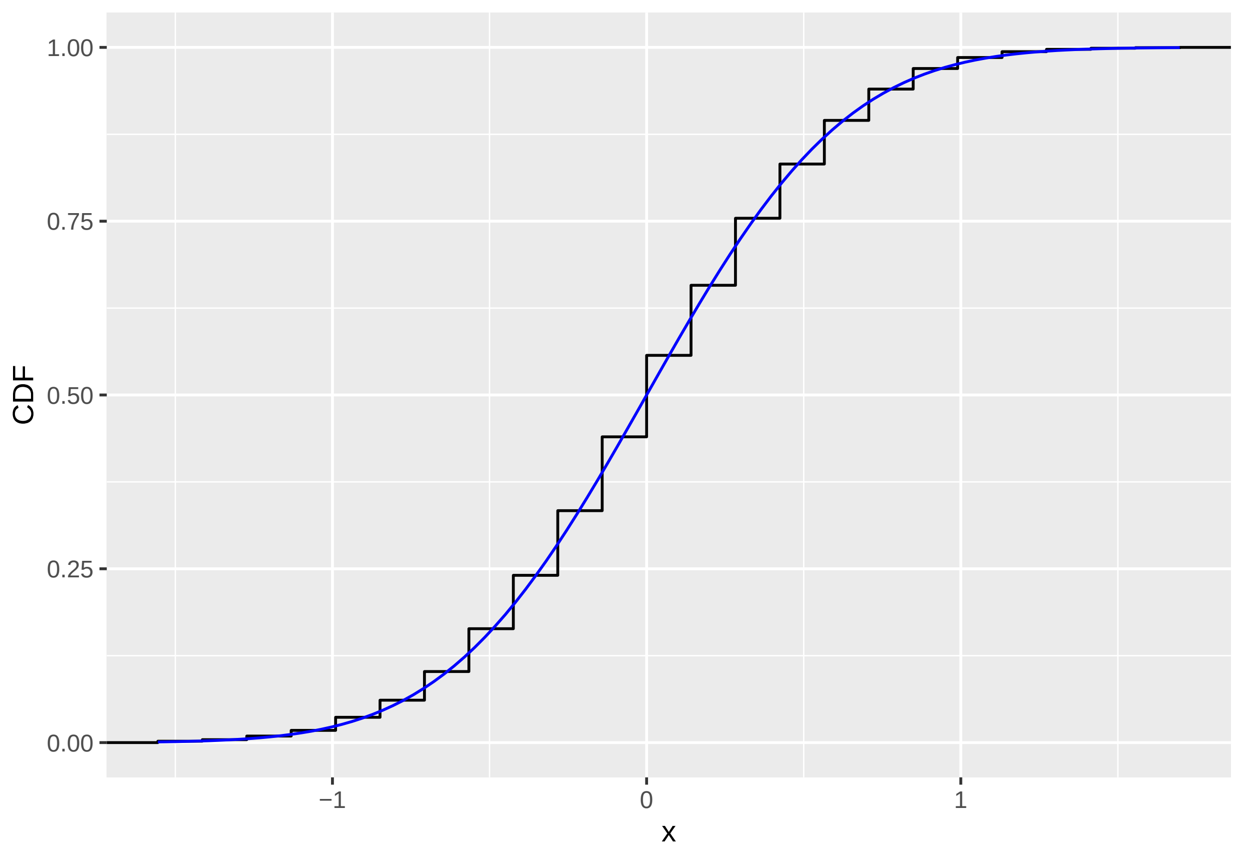


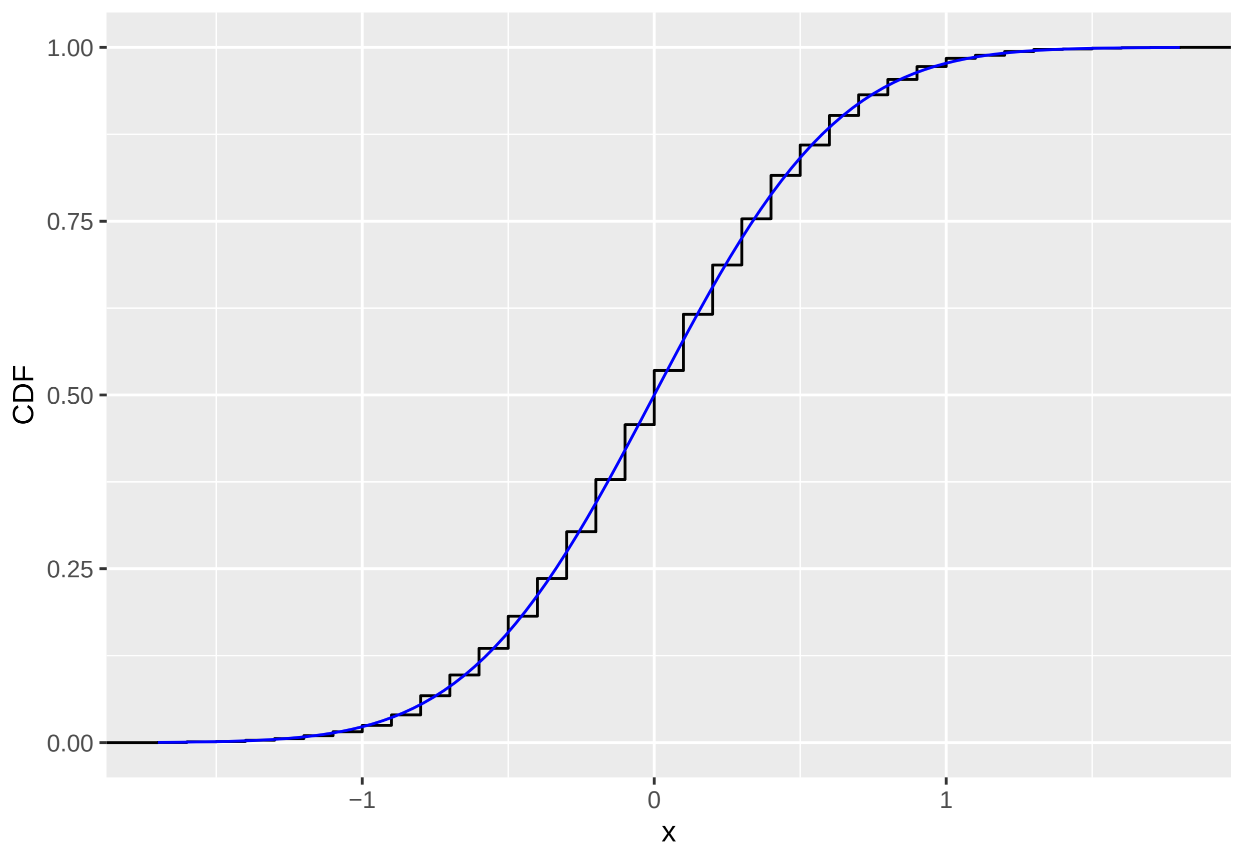
Recall the statement of the CLT: $X_1, X_2, ..., X_n$ is a sequence of independent and identically distributed (i.i.d.) random variables with $V(X_i) = \sigma^2 < +\infty$, $\sqrt{n}(\frac{1}{n}\sum_{i=1}^{n} - \mathbb{E}[X_i]) \xrightarrow{D} N(0, \sigma^2)$. In the simulation below, we illustrate the convergence of the CDF. Again, we use i.i.d. Bernoulli random variable with $\mathbb{P}[X_i = 1] = 0.5$. Thus, $\mathbb{E}[X_i] = 0.5$ and $\sigma^2 = 0.25$.

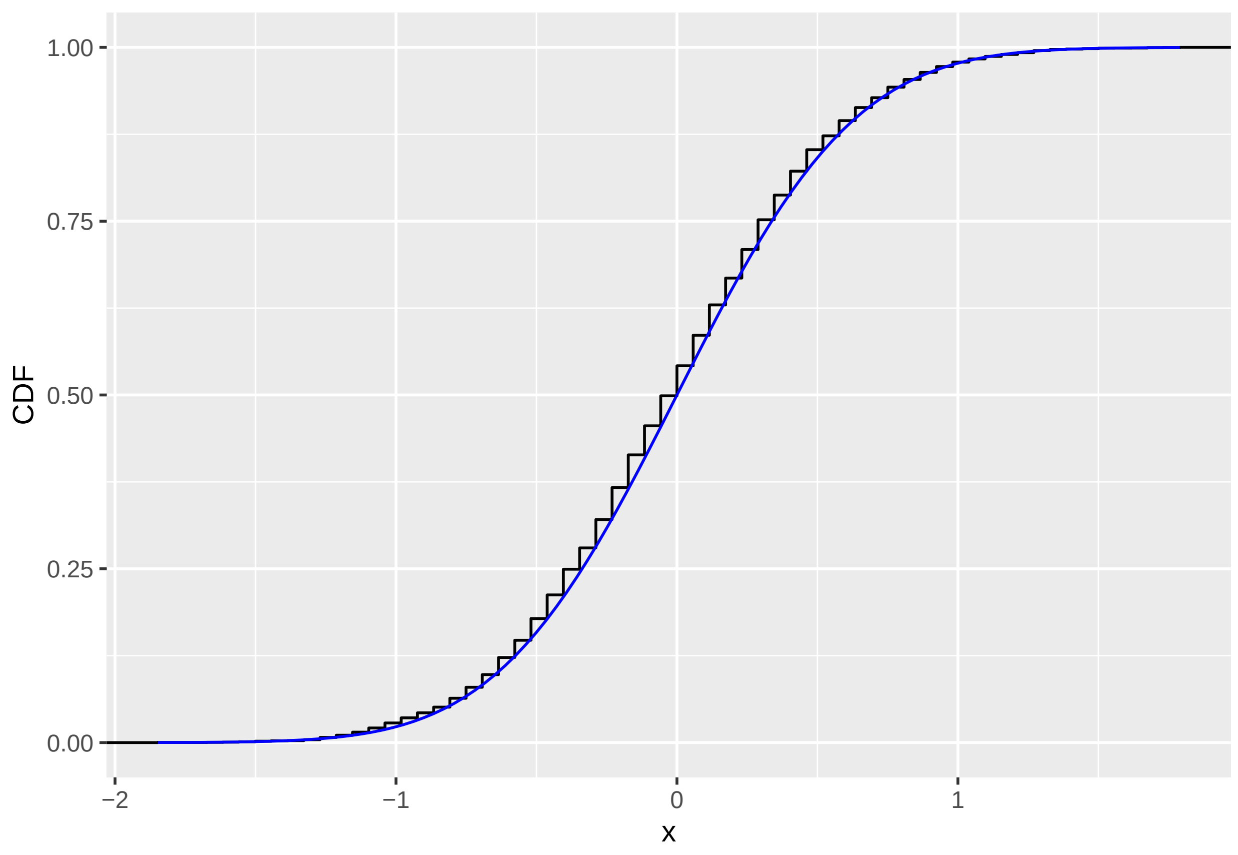
```
# set seed to make results replicable
set.seed(60615)

# for each sample size, iterate 5000 times
niter <- 5000

# create a vector to store the results
clt_seq <- rep(NA, niter)

# loop over different n, i.e., sample size: 10, 30, 50, 100, 300, 500, 5000
for(j in c(10, 30, 50, 100, 300, 500, 5000)){

  samp_size <- j

  for(i in 1:niter){

    # compute the quantity in the CLT: \sqrt{n} (\frac{1}{n} \sum_{i = 1}^{n} - \mathbb{E}[X_{i}])
    clt_seq[i] <- sqrt(samp_size) * (mean(rbinom(n = samp_size,
                                                 size = 1,
                                                 prob = .5)) - 0.5)
  }

  # save plot to compare the empirical CDF with normal CDF
  plot <- tibble(clt_sim = clt_seq) %>% ggplot(aes(x = clt_sim)) +
          stat_ecdf(geom = "step") +
          xlab("x") +
          ylab("CDF") +
          stat_function(fun = pnorm, args =  list(mean = 0, sd = 0.5),
                        col = "blue")

  # plot figures
  print(plot)

}
```

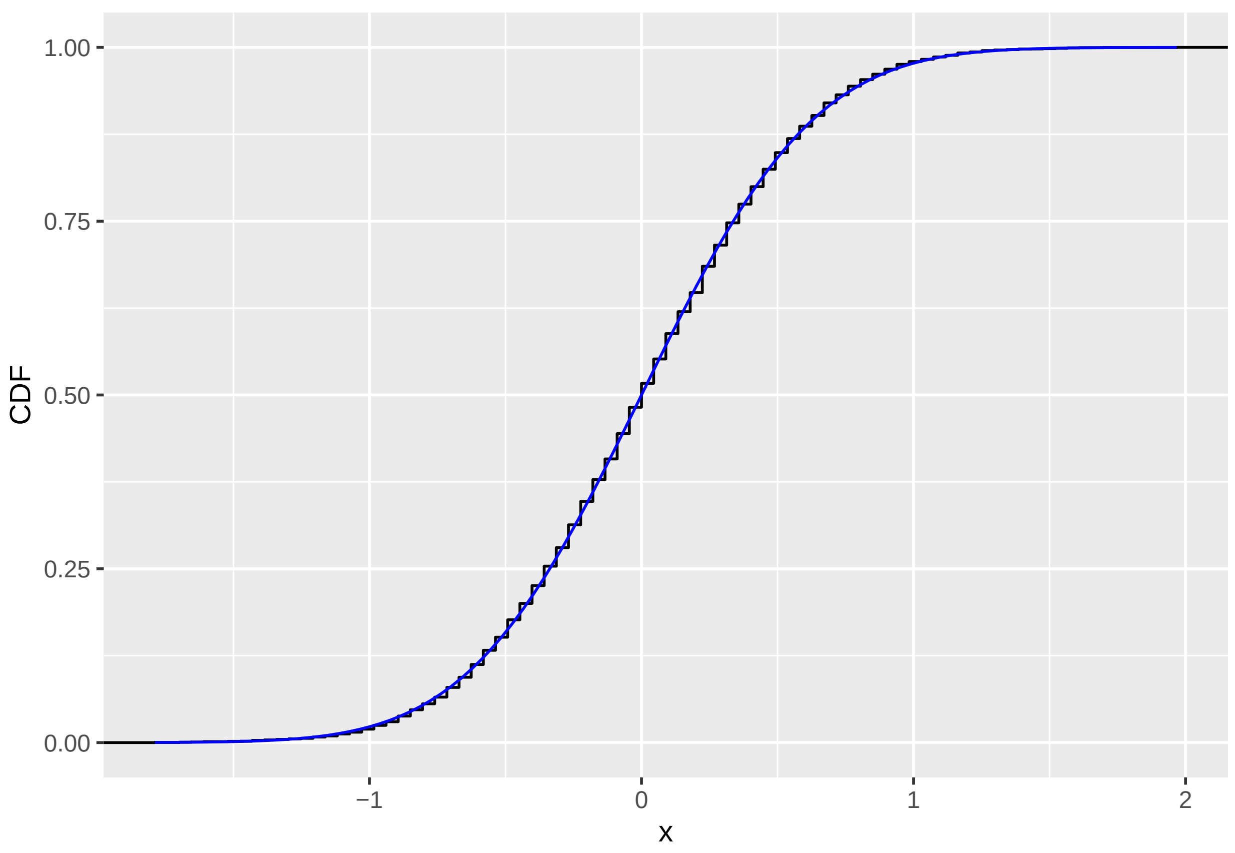


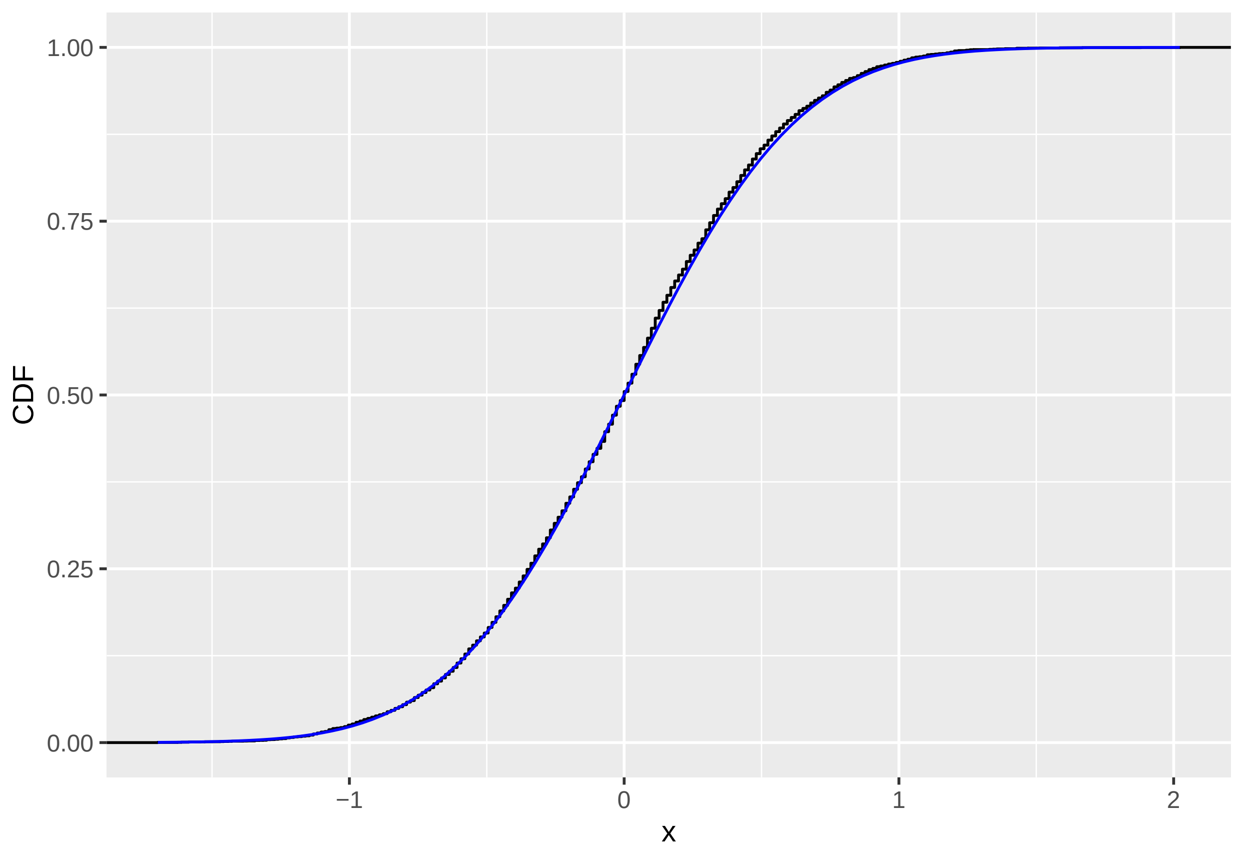

1.00
0.75
0.50
0.25
0.00
CDF
−2
−1
0
1
2
x


1.00
0.75
0.50
0.25
0.00
CDF
−1
0
1
x


1.00
0.75
0.50
0.25
0.00
CDF
−1
0
1
x


1.00
0.75
0.50
0.25
0.00
CDF
−2
−1
0
1
x


1.00
0.75
0.50
0.25
0.00
CDF
−1
0
1
2
x



- R also has some built-in functions that can replace the `for` loop. These are not necessarily time-saving, but can clean up your code significantly.
- The `lapply`, `apply`, `tapply`, `sapply` set of functions are built in to base-R and will apply a function to each element of a vector or matrix, returning the results in another vector. This is what's known in R as **vectorization**.
- Vectorized code is not guaranteed to be faster, though some vectorized functions are written in C instead of R (like `rowSums()` for taking the sum of rows of matrices). However, it is often **immensely cleaner** and helps you simplify problems by thinking about the inputs and outputs in terms of whole vectors rather than scalars.

## Map

- A `tidyverse` version of the apply functions is the `map` series of functions located in the `purrr` library (loaded by default with `tidyverse`).
- These take as an input a vector followed by a function and any arguments that should be passed to that function. Re-doing the above loop using `map`.
- `map` by default returns a list. We'd like to return a vector of a particular type - here `map_dbl` returns a vector of the `double` type (floating point) - which is what we want.

```
set.seed(60615) # Set random seed
niter <- 100000 # Number of for loop iterations - this can be as big as we need for precision
# Pipe %>% makes whatever is piped the first argument in the next function - in this case,
# just a vector 1:100000 - it ends up not mattering since the actual values of the function # aren't us
sample_means = 1:niter %>% map_dbl(function(x) mean(rbinom(n=50, size=1, prob=.5)))
```

```r
# We get exactly the same value!
mean(sample_means)
```

```
## [1] 0.499744
```

With these two tools, you can generate a simulation for nearly anything you want as long as you know the data-generating process.

## Illustration: Bias of the uncorrected sampling variance estimator

- Recall our classic setup for estimating a population mean $\mu$ using $N$ i.i.d. samples $X_i$ with $E[X_i] = \mu$, $Var(X_i) = \sigma^2$ (no assumptions on the *distribution* of $X_i$, just its moments). Our estimator $\hat{\mu} = \bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$ has some known properties.
- We know $E[\hat{\mu}] = \mu$ and that it is therefore **unbiased**.
- We know that its true sampling variance is $Var(\hat{\mu}) = \frac{\sigma^2}{N}$ and by extension the standard error is $\frac{\sigma}{\sqrt{N}}$.
- However, if we want to construct confidence intervals and do inference, we need $Var(\hat{\mu})$, but we do not know $\sigma^2$.
- **We therefore have to estimate using some estimator**.
- One straightforward approach simply plugs in an estimate of $\sigma^2$, $\hat{\sigma^2}$ giving $\widehat{Var(\hat{\mu})} = \frac{\hat{\sigma^2}}{N}$. But how should we construct our estimator $\hat{\sigma^2}$? Well, the variance is defined as $E[(X - E[X])^2]$, the average of the squared deviations from the mean. + We can use the sample analogue $\hat{\sigma^2}_{\text{uncorrected}} = \frac{1}{N}\sum_{i=1}^{N}(X_i - \bar{X})^2$. However, this estimator turns out to be *biased* although it is *consistent*.
- It turns out that using $N-1$ instead of $N$ addreses this bias $\hat{\sigma^2}_{\text{corrected}} = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \bar{X})^2$.

Let's show this via simulation! Start by defining the data-generating process for a fixed value of $N = 30$, $\mu = 0$, $\sigma^2 = 4$, We'll assume $X_i$ is normal, though this isn't strictly necessary.

```r
# Set seed
set.seed(60615)

# Define the parameters of the DGP
N = 30
mu = 0
sigma = sqrt(4)

# Define a function to compute the unadjusted variance
var_unadj = function(x){
  return((1/length(x))*sum((x - mean(x))^2))
}

# Number of iterations
niter = 10000

# Run the simulation and get evaluations of the unadjusted variance
sim_results = 1:niter %>% map_dbl(function(x) var_unadj(rnorm(n=N, mean=mu, sd=sigma)))
```

```r
# Calculate bias
mean(sim_results) - sigma^2
```

```
## [1] -0.1355836
```

Now, we can show that this bias goes away when we use the $N-1$ correction (sometimes called “Bessel’s Correction”).

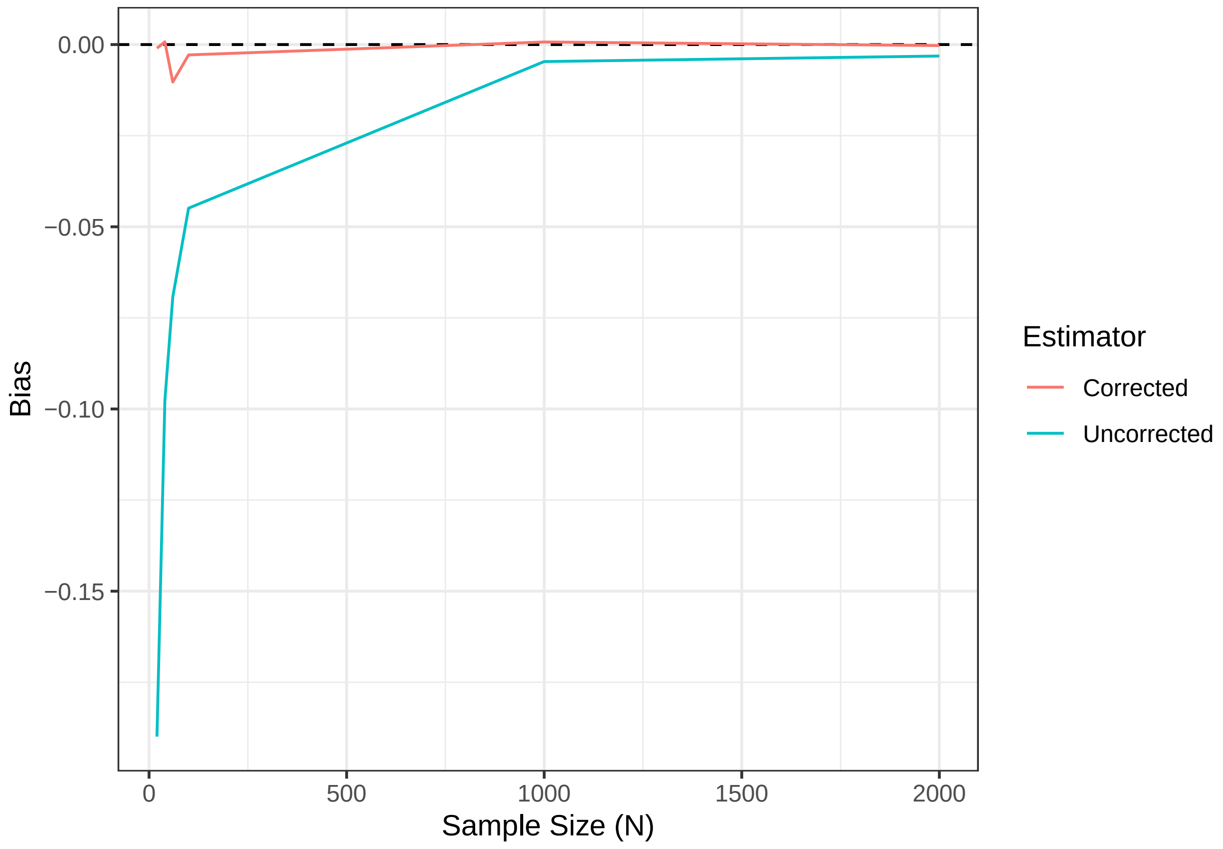
```r
# Set seed
set.seed(60615)

# Define the parameters of the DGP
N = 30
mu = 0
sigma = sqrt(4)

# Define a function to compute the unadjusted variance
# This function is equivalent to var() in R
var_adj = function(x){
  return((1/(length(x)-1))*sum((x - mean(x))^2))
}

# Number of iterations
niter = 10000

# Run the simulation and get evaluations of the unadjusted variance
sim_results_adj = 1:niter %>% map_dbl(function(x) var_adj(rnorm(n=N, mean=mu, sd=sigma)))

# Calculate bias
mean(sim_results_adj) - sigma^2
```

```
## [1] -0.002327903
```

- Let’s see how these two estimators compare across different values of N. There are lots of ways to do this.
- You could do a nested for loop – the outer loop varying N and the inner loop running all `niter` iterations. This will work but the code can look messy and confusing.
- We’ll stick with `map`, but nest it. `map` applies a function to each element of a list.
- We’ll pass the vectors in each list element to *another* call to map to do what we did above but for each sample size.

```r
# Set seed
set.seed(60615)

# Parameters
mu = 0
sigma = sqrt(4)

# Set range of Ns - this can be whatever you want, but more levels = more sims
N_vec = c(20, 40, 60, 100, 1000, 2000)
niter = 10000
```

```
# Make a list with N_vec elements each containing niter copies of the relevant sample size
iter_list <- N_vec %>% map(function(x) rep(x, niter))

# Use Map twice - first to split by list then apply map_dbl to each vector (this is equivalent to a nes
# Then summarize each simulation run by taking the mean
sim_results_unadj <- iter_list %>% map(. %>% map_dbl(function(x) var_unadj(rnorm(n=x, mean=mu, sd=sigma
sim_results_adj <- iter_list %>% map(. %>% map_dbl(function(x) var_adj(rnorm(n=x, mean=mu, sd=sigma)))

# Put into a dataframe for plotting
plot_frame = data.frame(n = N_vec, Uncorrected = sim_results_unadj - sigma^2, Corrected = sim_results_ad

# GGplot likes faceting by factors - let's go from wide to long here!
plot_frame <- plot_frame %>% pivot_longer(cols = c(Uncorrected, Corrected), names_to="Estimator", value

# Plot the results
plot_frame %>% ggplot(aes(x=n, y=Bias, colour=Estimator)) + geom_hline(yintercept = 0, lty=2) + geom_li
```

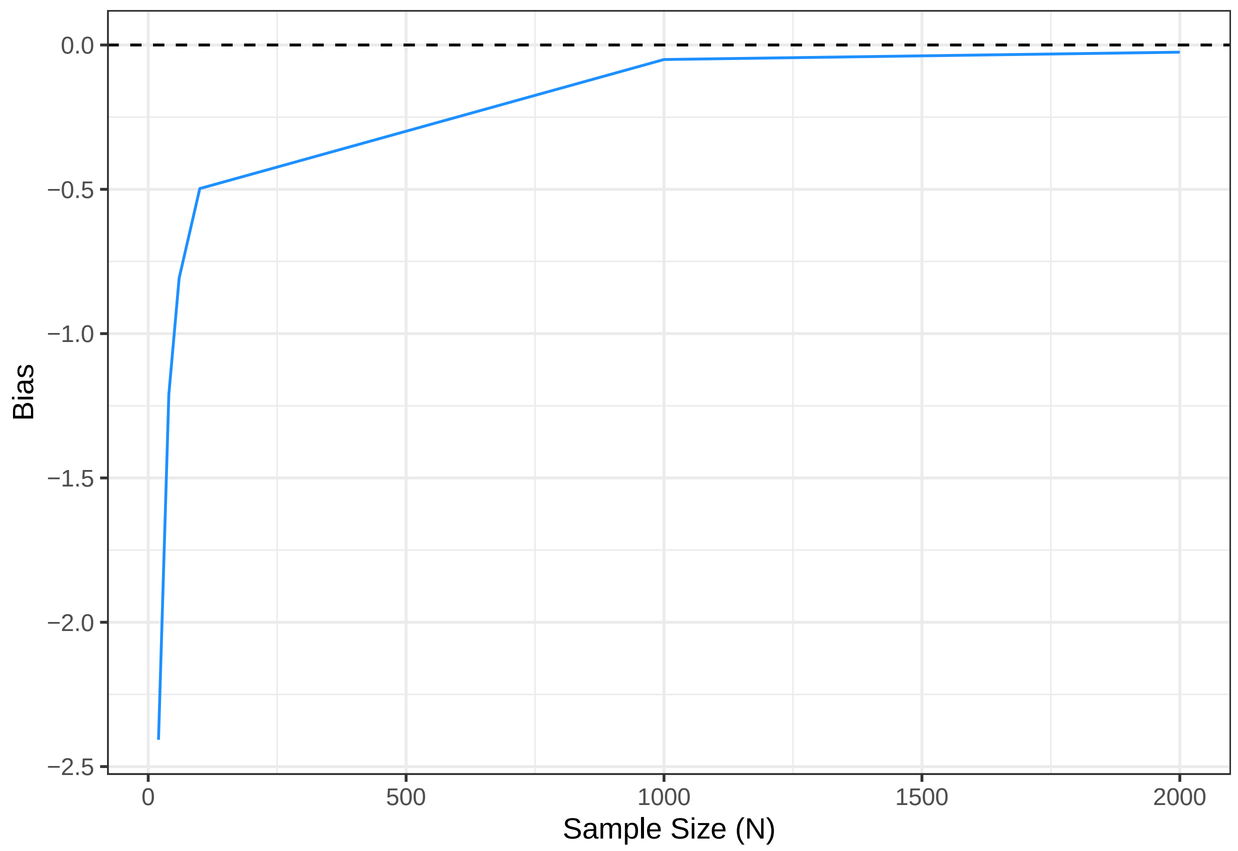


## Challenge problem: Bias of the maximum of a uniform.

Suppose we observe a sample of $N$ i.i.d. observations $X_i \sim \text{Uniform}(0, K)$. Construct a simulation to show that the maximum of $X_1, X_2, \ldots, X_n$ is a biased estimator of $K$. How does the bias change as $N$ gets large? As $K$ gets large?

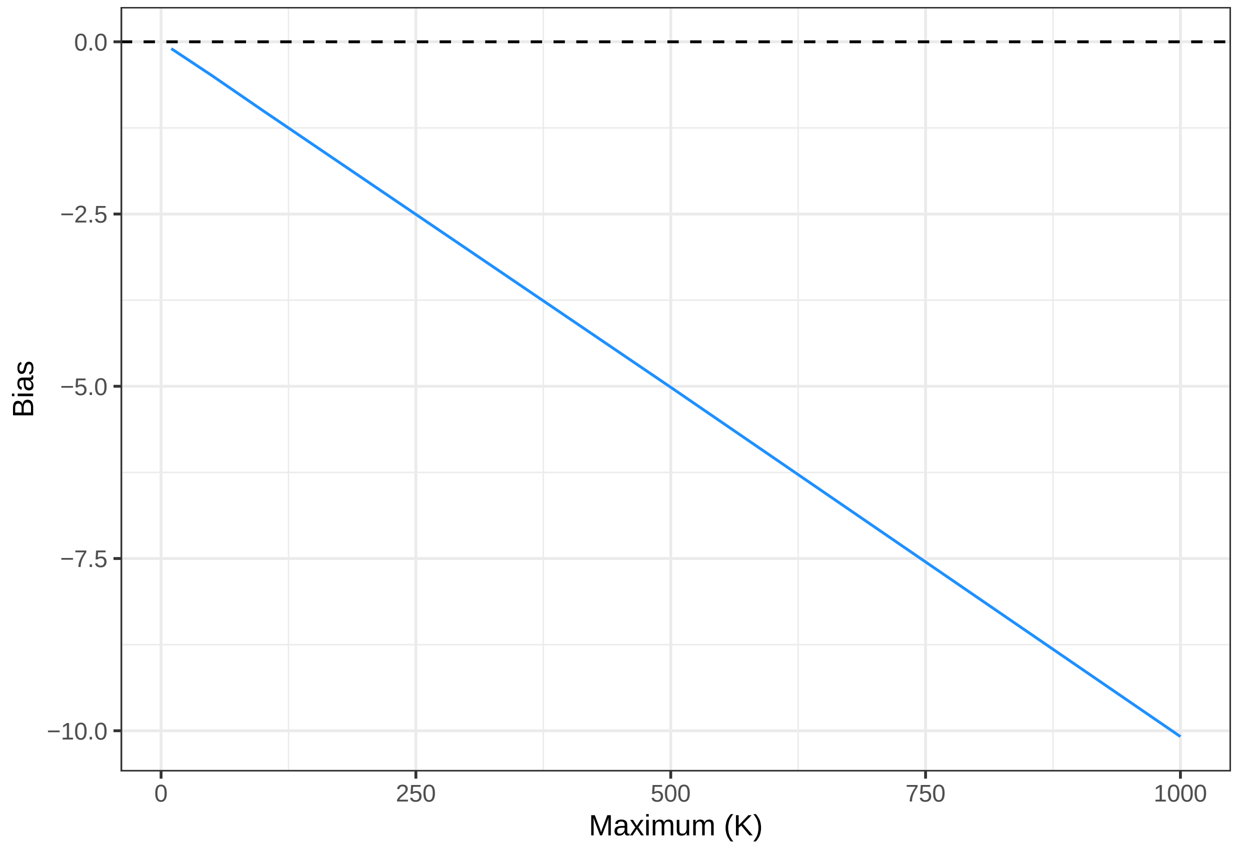
Hint: `?Uniform` will give you the documentation for the functions related to the uniform distribution in R.

```
# Set seed
set.seed(60615)

# Parameters
minimum = 0

# Set range of Ns - this can be whatever you want, but more levels = more sims
# Set range of Ks
N_vec = c(20, 40, 60, 100, 1000, 2000)
K_vec = c(10, 20, 50, 100, 500, 1000)
niter = 10000

# First simulation - let's fix K at 50, vary N
# Make a list with N_vec elements each containing niter copies of the relevant sample size
iter_list_N <- N_vec %>% map(function(x) rep(x, niter))

# Do the simulation (maximum of the vector)
sim_results_N_fixK <- iter_list_N %>% map(. %>% map_dbl(function(x) max(runif(n=x, min=minimum, max=50))

# Plot the results
plot_frame_N = data.frame(n = N_vec, Bias = sim_results_N_fixK -50)
plot_frame_N %>% ggplot(aes(x=n, y=Bias)) + geom_hline(yintercept = 0, lty=2) +  geom_line(colour = "do
```



```
# Second simulation - let's fix N at 100, vary K
# Make a list with N_vec elements each containing niter copies of the relevant sample size
```

```
iter_list_K <- K_vec %>% map(function(x) rep(x, niter))

# Do the simulation (maximum of the vector)
sim_results_K_fixN <- iter_list_K %>% map(. %>% map_dbl(function(x) max(runif(n=100, min=minimum, max=x)

# Plot the results
plot_frame_K = data.frame(n = K_vec, Bias = sim_results_K_fixN - K_vec)
plot_frame_K %>% ggplot(aes(x=n, y=Bias)) + geom_hline(yintercept = 0, lty=2) +  geom_line(colour = "do
```


Bias
Maximum (K)