# Why is Generating Random Variables Important?

Being able to generate random variables is crucial for simulations due to the following reasons:

* Modeling Real-world Uncertainty: Many phenomena in the real world are inherently uncertain and stochastic. Simulations often incorporate random variables to emulate this inherent randomness and uncertainty.
* Stochastic Simulations: Stochastic simulations rely on random variables to model the variability and unpredictability in systems. This type of simulation is particularly important in fields like finance, biology, and operations research.
* Monte Carlo Methods: One of the most prominent simulation techniques, the Monte Carlo method, involves the use of random sampling to estimate outcomes, whether for integration, optimization, or modeling. The method's essence is based on generating random variables according to specific distributions.
* Exploring Different Scenarios: Simulations often aim to study how systems react under different circumstances. Random variables can be used to generate a multitude of scenarios, allowing for a broad exploration of potential system behaviors.
* Sampling from Distributions: Often, we need to simulate processes based on known probability distributions, like Gaussian, Poisson, or exponential. The ability to generate random variables from these distributions is fundamental to create realistic simulations.
* Stress Testing and Robustness: Random variables can be used to generate extreme scenarios or edge cases to stress-test systems, ensuring their robustness under various conditions.
* Validation & Calibration: When calibrating simulation models against real-world data, random variables can be used to simulate uncertainties in measurements or to replicate the variability seen in empirical data.
* Queueing Systems: In simulations of queueing systems, like those in telecommunications or customer service scenarios, random variables determine parameters like arrival times and service durations.
* Reliability Analysis: In engineering, random variables are used to model uncertainties in material properties, loads, and environmental conditions to predict the reliability and potential failure of structures or components.
* Probabilistic Risk Assessment: In fields like nuclear engineering or finance, simulations involving random variables help assess the risk associated with rare but high-impact events.
* Agent-based Modeling: In simulations where individual agents (like animals, vehicles, or people) exhibit random behaviors, generating random variables determines each agent's actions or states.
* Improving Computational Efficiency: In some simulations, instead of computing every possible scenario deterministically, a subset of scenarios is sampled using random variables, providing faster yet accurate approximations.
* Enhancing Realism: In graphics, gaming, and animation, random variables are employed to introduce elements like noise, texture, or natural variation, making visuals more lifelike.

The ability to generate random variables provides the foundation to introduce, manage, and explore uncertainty and variability in simulations. It allows for a richer, more realistic, and comprehensive study of systems and phenomena, bridging the gap between deterministic models and the inherently stochastic real world.

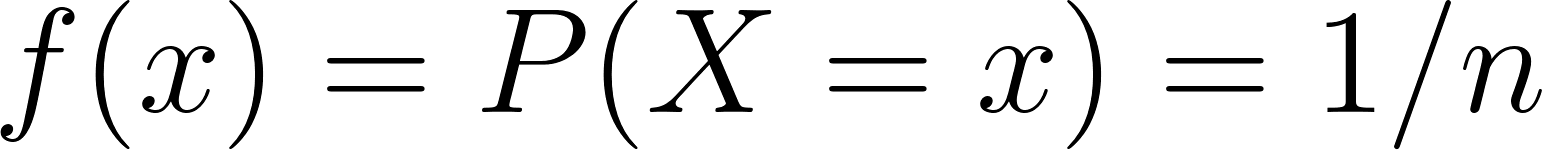
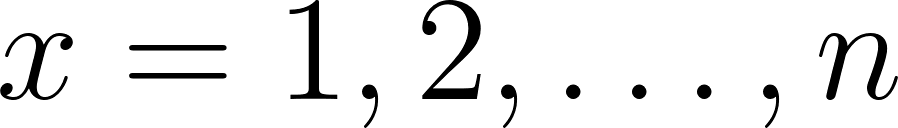
# Discrete Uniform Example

Utilize the [inverse transform method](https://en.wikipedia.org/wiki/Inverse_transform_sampling) to set:

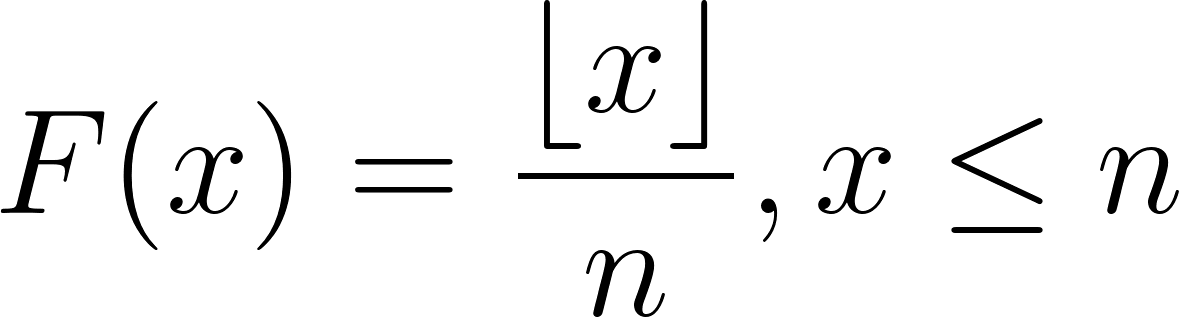
[A black background with a black square

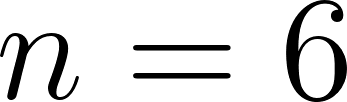
Description automatically generated with medium confidence](https://www.codecogs.com/eqnedit.php?latex=X%3D%5Clceil%7BnU%7D%20%5Crceil#0)

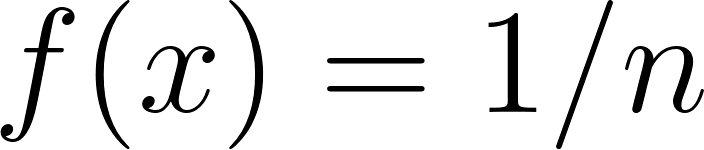
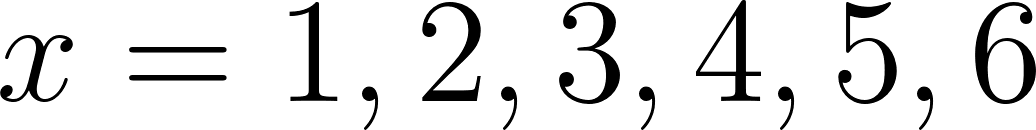
T[he probability mass function](https://en.wikipedia.org/wiki/Probability_mass_function) (PMF) for a [discrete uniform random variable](https://en.wikipedia.org/wiki/Discrete_uniform_distribution) is:

[](https://www.codecogs.com/eqnedit.php?latex=f(x)%3DP(X%3Dx)%3D1%2Fn#0) for [](https://www.codecogs.com/eqnedit.php?latex=x%3D1%2C2%2C%20%5Cdots%2C%20n#0)

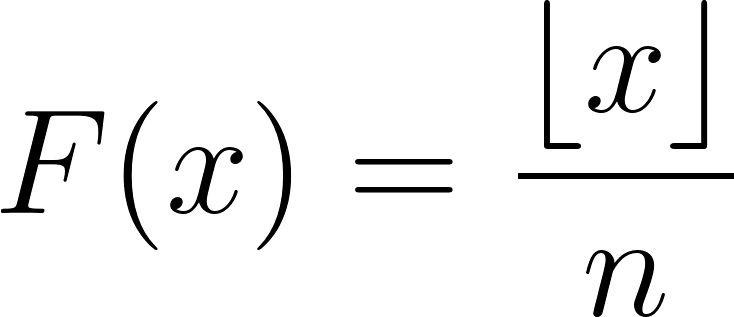
The [cumulative distribution function](https://en.wikipedia.org/wiki/Cumulative_distribution_function) is:

[](https://www.codecogs.com/eqnedit.php?latex=F(x)%3D%5Cdfrac%7B%5Clfloor%20x%20%5Crfloor%7D%7Bn%7D%2C%20x%20%5Cleq%20n#0)

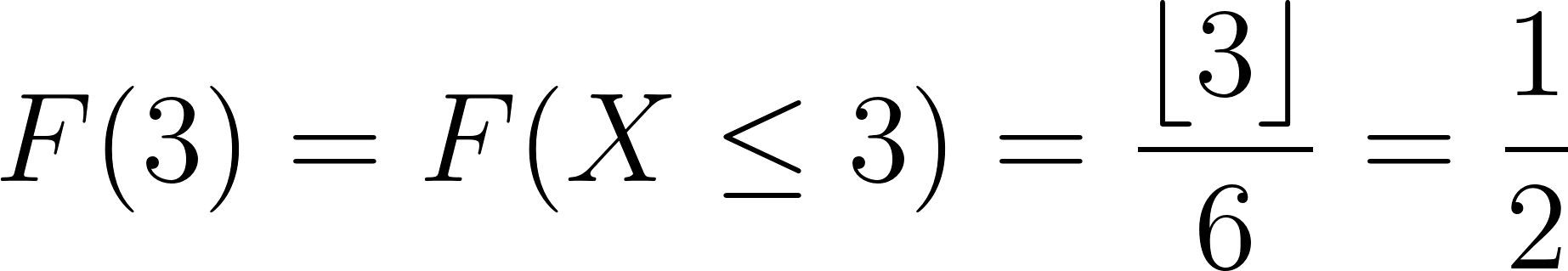
The [floor function](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions) is in the CDF because this allows any value to be input into the CDF and it will return the correct probability by using the floor function. An example is if we had a 6-sided die ([](https://www.codecogs.com/eqnedit.php?latex=n%3D6#0)) then the PMF is:

[](https://www.codecogs.com/eqnedit.php?latex=f(x)%3D1%2Fn#0) for [](https://www.codecogs.com/eqnedit.php?latex=x%3D1%2C2%2C3%2C4%2C5%2C6#0)

and the CDF is:

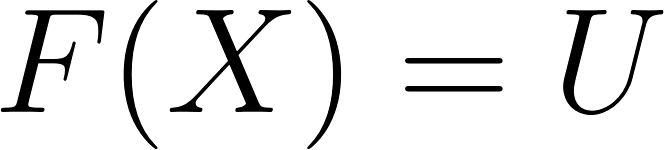
[](https://www.codecogs.com/eqnedit.php?latex=F(x)%3D%5Cdfrac%7B%5Clfloor%20x%20%5Crfloor%7D%7Bn%7D#0)

If we wanted to compute the probability that when we roll the die that a value less than or equal to 3 will be shown, we can use the CDF.

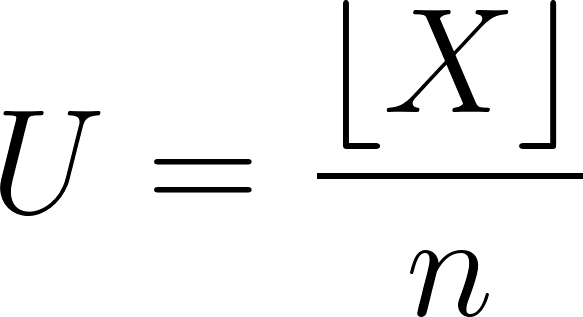
[](https://www.codecogs.com/eqnedit.php?latex=F(3)%3DF(X%20%5Cleq%203)%3D%5Cdfrac%7B%5Clfloor%203%20%5Crfloor%7D%7B6%7D%3D%5Cdfrac%7B1%7D%7B2%7D#0)

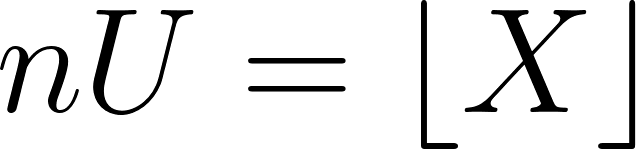
Similarly, we can input any value such as 3.3 and will obtain the same result as the CDF will account for this and return the correct probability.

The inverse transform method involves setting the random variable:

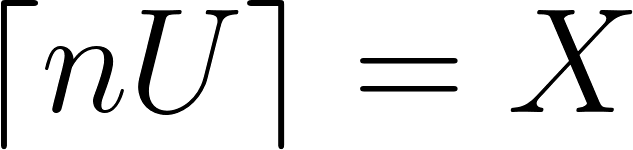
[](https://www.codecogs.com/eqnedit.php?latex=F(X)%3DU#0)

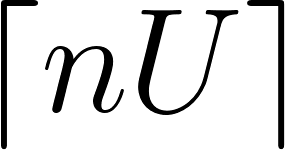
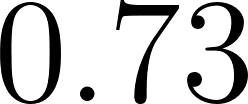
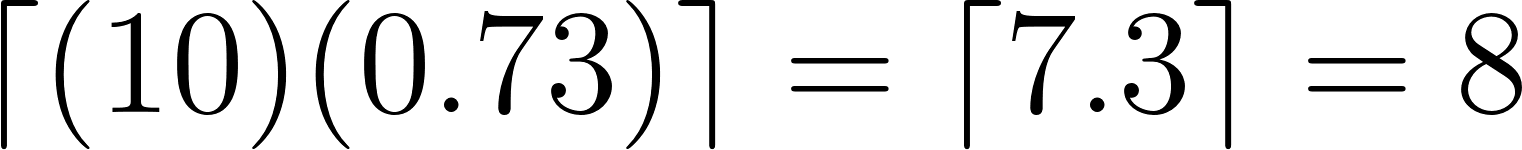
and solving for [](https://www.codecogs.com/eqnedit.php?latex=U#0). For a discrete uniform random variable here are the steps:

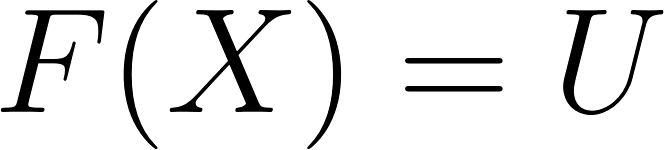
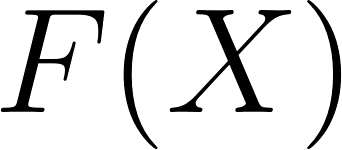
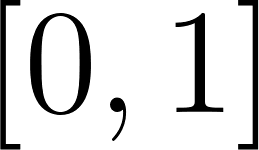
[](https://www.codecogs.com/eqnedit.php?latex=U%3D%5Cdfrac%7B%5Clfloor%20X%20%5Crfloor%7D%7Bn%7D#0)

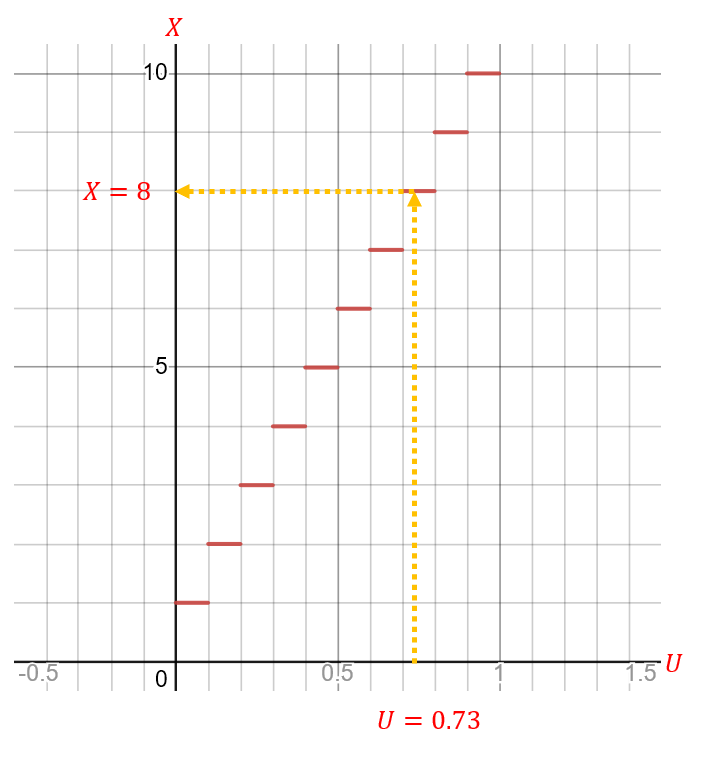
[](https://www.codecogs.com/eqnedit.php?latex=nU%3D%5Clfloor%20X%20%5Crfloor#0)

Apply the ceiling function to both sides of the equation.

[](https://www.codecogs.com/eqnedit.php?latex=%5Clceil%20nU%20%5Crceil%3D%20X#0)

Thus, a method for generating a discrete uniform random variable is to generate a uniform random variate and substitute into the expression [](https://www.codecogs.com/eqnedit.php?latex=%5Clceil%20nU%20%5Crceil#0). A graph of the function is below for [](https://www.codecogs.com/eqnedit.php?latex=n%3D10#0). From the graph, one should notice that the discrete values on the y-axis are split into uniform pieces. A uniform random variate that falls inside one of these intervals will map to one of the discrete values. For example if we generate a uniform random variate of [](https://www.codecogs.com/eqnedit.php?latex=0.73#0), using the formula derived earlier this would result in an outcome of [](https://www.codecogs.com/eqnedit.php?latex=%5Clceil%20(10)(0.73)%20%5Crceil%3D%5Clceil%207.3%20%5Crceil%3D8#0).

Note that this is a graph of the inverse of the CDF. Recall that we set [](https://www.codecogs.com/eqnedit.php?latex=F(X)%3DU#0), so the horizontal axis is analogous to the random variable [](https://www.codecogs.com/eqnedit.php?latex=F(X)#0) which is in [](https://www.codecogs.com/eqnedit.php?latex=%5B0%2C%201%5D#0). .



See the graph here: <https://www.desmos.com/calculator/xcvgvgozpl>

## Discrete Uniform Example R Code

*# Function to generate a discrete uniform random variable from 1 to n\_sides*  
generate\_discrete\_uniform <- **function**(n, n\_sides, seed = 1) {  
 *# Set the seed for the random number generator*  
 set.seed(seed)  
   
 *# Generate n uniformly distributed random numbers between 0 and 1*  
 U <- runif(n)  
   
 *# Apply the inverse CDF to transform the uniform random numbers to discrete*   
 *# uniform random variables*  
 X <- ceiling(n\_sides \* U)  
   
 *# Return both the uniformly distributed random numbers (U) and the generated*   
 *# discrete uniform random variables (X)*  
 return(list(U = U, X = X))  
}  
  
*# Example: Generate 5 discrete uniform random variables from 1 to 10*  
n\_sides <- 10  
discrete\_uniforms <- generate\_discrete\_uniform(5, n\_sides, 123)  
  
*# Print the generated uniformly distributed random numbers (U)*  
discrete\_uniforms$U

## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673

*# Print the generated discrete uniform random variables (X)*  
discrete\_uniforms$X

## [1] 3 8 5 9 10

*# Generate 1e7 (= 10,000,000) discrete uniform random variables from 1 to 10*  
discrete\_uniforms <- generate\_discrete\_uniform(1e7, n\_sides, 123)  
  
*# Calculate and print the relative frequency of each outcome in the generated*   
*# discrete uniform random variables (X)*  
table(discrete\_uniforms$X) / length(discrete\_uniforms$X)

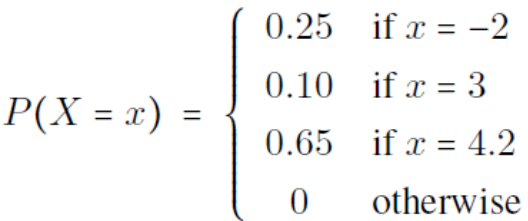
##   
## 1 2 3 4 5 6 7 8   
## 0.0999284 0.1001580 0.0999748 0.1001370 0.0998557 0.1000297 0.1000179 0.0999284   
## 9 10   
## 0.0999793 0.0999908

## Discrete Uniform Example Python Code

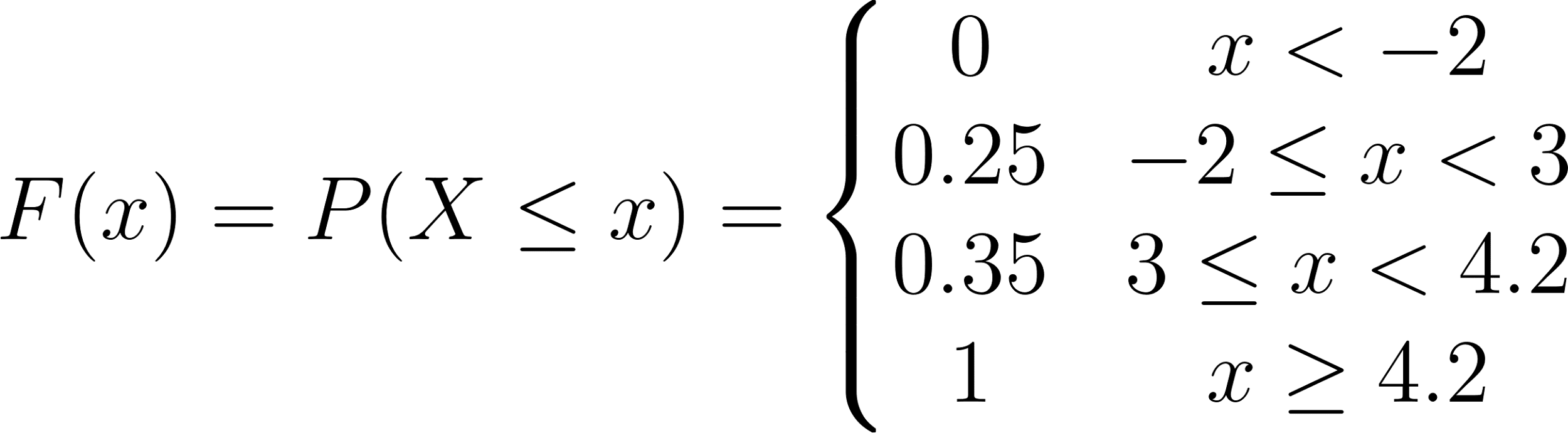
[Python Notebook](https://colab.research.google.com/drive/1gN04hmsbdBEvmkJU6XKBVUJvdG_6ORl1)

# Another Discrete Uniform Example

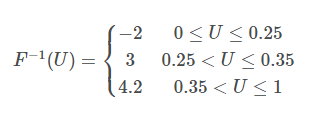
Any arbitrary [discrete random variable](https://en.wikipedia.org/wiki/Random_variable#Standard_case) can be generated using a similar method as described above. Given the following discrete random variable with PMF:

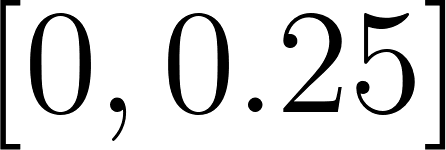
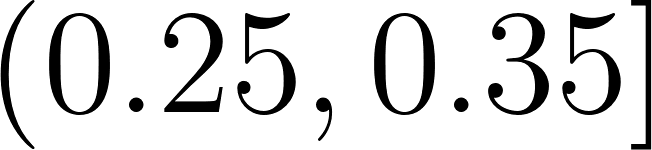
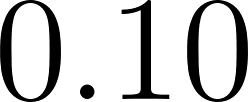
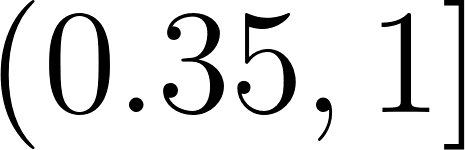
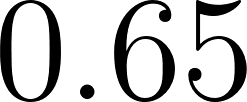
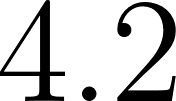


The CDF of this discrete random variable is:

[](https://www.codecogs.com/eqnedit.php?latex=F(x)%3DP(X%20%5Cleq%20x)%3D%5Cleft%5C%7B%5Cbegin%7Bmatrix%7D%200%20%26%20x%3C-2%5C%5C%5C%5C%200.25%20%26%20-2%20%5Cleq%20x%20%3C%203%20%5C%5C0.35%20%26%203%20%5Cleq%20x%20%3C%204.2%5C%5C1%20%20%20%26%20x%20%5Cgeq%204.2%20%5Cend%7Bmatrix%7D%5Cright.#0)

“Inverting” this CDF would result in something like the following:



One should notice that when [](https://www.codecogs.com/eqnedit.php?latex=U#0) falls in the interval [](https://www.codecogs.com/eqnedit.php?latex=%5B0%2C%200.25%5D#0) (which it will with probability [](https://www.codecogs.com/eqnedit.php?latex=0.25#0)), a [](https://www.codecogs.com/eqnedit.php?latex=-2#0) will be returned. When it falls in the interval [](https://www.codecogs.com/eqnedit.php?latex=(0.25%2C%200.35%5D#0) (which it will with probability [](https://www.codecogs.com/eqnedit.php?latex=0.10#0)), a [](https://www.codecogs.com/eqnedit.php?latex=3#0) will be returned. When it falls in the interval [](https://www.codecogs.com/eqnedit.php?latex=(0.35%2C1%5D#0) (which it will with probability [](https://www.codecogs.com/eqnedit.php?latex=0.65#0)), a [](https://www.codecogs.com/eqnedit.php?latex=4.2#0) will be returned. These are the probabilities we wish the random variable to have according to its PMF.

## Another Discrete Uniform Example R Code

*# Function to generate a non-uniform discrete random variable with three*   
*# possible outcomes: -2, 3, and 4.2*  
another\_discrete\_uniform <- **function**(n, seed = 1) {  
 *# Set the seed for the random number generator*  
 set.seed(seed)  
   
 *# Generate n uniformly distributed random numbers between 0 and 1*  
 U <- runif(n)  
   
 *# Initialize an empty numeric vector of length n to store the generated*   
 *#discrete random variables*  
 X <- numeric(n)  
   
 *# Assign the values of the discrete random variable based on the value of U*  
 X[U <= 0.25] <- -2 *# Probability 0.25*  
 X[U > 0.25 & U <= 0.35] <- 3 *# Probability 0.10*  
 X[U > 0.35] <- 4.2 *# Probability 0.65*  
   
 *# Return both the uniformly distributed random numbers (U) and the generated*   
 *# discrete random variables (X)*  
 return(list(U = U, X = X))  
}  
  
*# Generate 5 discrete random variables*  
discrete\_uniforms <- another\_discrete\_uniform(5)  
  
*# Print the generated uniformly distributed random numbers (U)*  
discrete\_uniforms$U

## [1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819

*# Print the generated discrete random variables (X)*  
discrete\_uniforms$X

## [1] 3.0 4.2 4.2 4.2 -2.0

*# Generate 1e7 (= 10,000,000) discrete random variables*  
discrete\_uniforms <- another\_discrete\_uniform(1e7)  
  
*# Calculate and print the relative frequency of each outcome in the generated*   
*# discrete random variables (X)*  
table(discrete\_uniforms$X) / length(discrete\_uniforms$X)

##   
## -2 3 4.2   
## 0.2499508 0.1000913 0.6499579

## Another Discrete Uniform Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1gN04hmsbdBEvmkJU6XKBVUJvdG_6ORl1#scrollTo=2x1uwP_-ZoF1)

# Inverse Transform Method

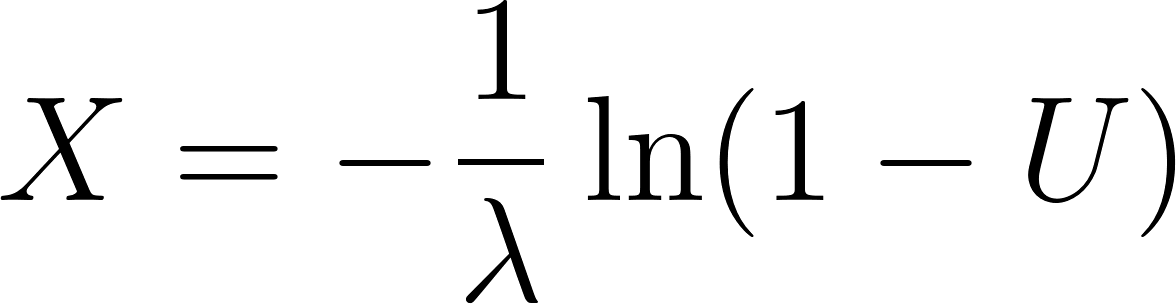
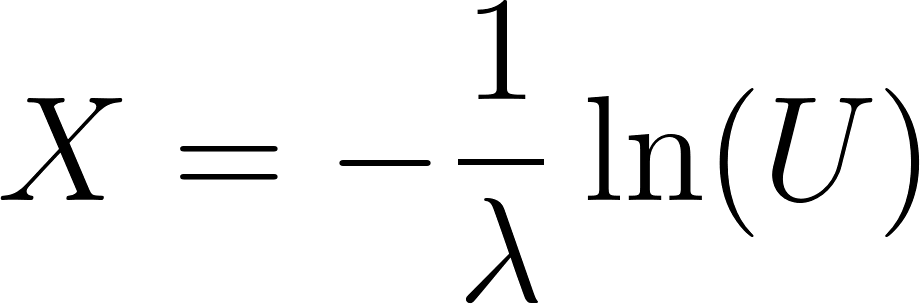
Also see [Inverse Transform Method Supplemental Notes](https://docs.google.com/document/u/0/d/1fpQrzpObFTpGE3vNHpLuG6UfQnVj31TiMPvaZcBwPpk/edit)

A proof of the [inverse transform method](https://en.wikipedia.org/wiki/Inverse_transform_sampling) will not be presented yet but will be shown in later modules. For now, a continuous random variable will be generated using this method. The discrete examples show that the method does in fact work for discrete random variables, so we will extend it to work with continuous random variables.

Refer back to M1L8 Generating Randomness, specifically the section on Generating Other RVs. From that section, we saw that we can generate an exponential random variable:

[](https://www.codecogs.com/eqnedit.php?latex=X%20%5Csim%20%5Ctext%7BExp%7D(%5Clambda)#0)

using the formulas:

[](https://www.codecogs.com/eqnedit.php?latex=X%3D-%5Cdfrac%7B1%7D%7B%5Clambda%7D%5Cln(1-U)#0) or [](https://www.codecogs.com/eqnedit.php?latex=X%3D-%5Cdfrac%7B1%7D%7B%5Clambda%7D%5Cln(U)#0)

## Inverse Transform Method Example R Code

This code compares R's built-in uniform random number generator with the inverse transform method using U and 1-U. The goal is to demonstrate that both methods produce equivalent results when generating random variables from an exponential distribution.

The code generates three sets of random variables from an exponential distribution with a rate of 5: x\_1 using the inverse transform method with U, x\_2 using the inverse transform method with 1-U, and x\_3 using R's built-in rexp() function. It then calculates summary statistics and histograms for each set of random variables.

*# Set the random seed*  
set.seed(1)  
  
*# Number of random variables to generate*  
n <- 5e6  
  
*# Generate uniform random variables U*  
U <- runif(n)  
  
*# Inverse transform method with U*  
x\_1 <- -(1/5) \* log(U)  
  
*# Inverse transform method with 1 - U*  
x\_2 <- -(1/5) \* log(1 - U)  
  
*# R's built-in random exponential generator*  
x\_3 <- rexp(n, rate = 5)  
  
*# Calculate and display mean of each set of random variables*  
cat("Mean of x\_1:", mean(x\_1), "\n")

## Mean of x\_1: 0.2001137

cat("Mean of x\_2:", mean(x\_2), "\n")

## Mean of x\_2: 0.1999099

cat("Mean of x\_3:", mean(x\_3), "\n")

## Mean of x\_3: 0.1999908

*# Calculate and display variance of each set of random variables*  
cat("\nVariance of x\_1:", var(x\_1), "\n")

##   
## Variance of x\_1: 0.04003881

cat("Variance of x\_2:", var(x\_2), "\n")

## Variance of x\_2: 0.03995771

cat("Variance of x\_3:", var(x\_3), "\n")

## Variance of x\_3: 0.03999593

*# Display summary statistics of each set of random variables*  
cat("\nSummary of x\_1:\n")

##   
## Summary of x\_1:

print(summary(x\_1))

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.00000 0.05756 0.13868 0.20011 0.27728 3.49244

cat("\nSummary of x\_2:\n")

##   
## Summary of x\_2:

print(summary(x\_2))

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.00000 0.05753 0.13858 0.19991 0.27718 3.41739

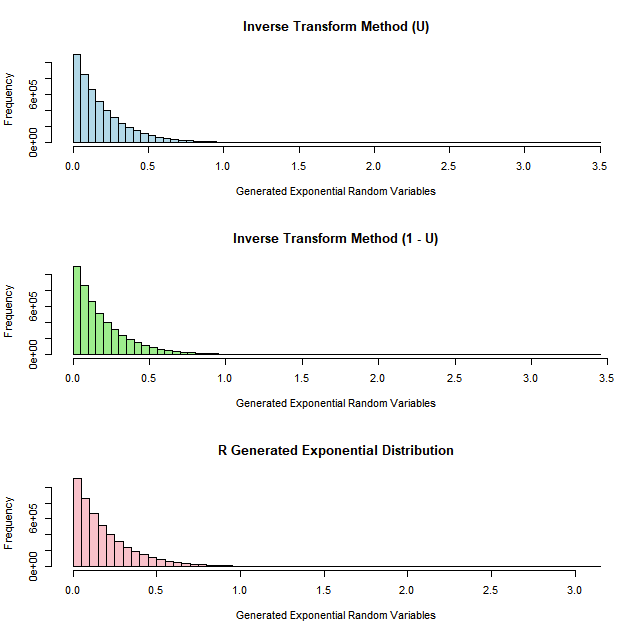
cat("\nSummary of x\_3:\n")

##   
## Summary of x\_3:

print(summary(x\_3))

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.00000 0.05744 0.13864 0.19999 0.27729 3.11313

*# Plot histograms of each set of random variables*  
par(mfrow = c(3, 1))  
hist(x\_1, 100, main = "Inverse Transform Method (U)",   
 xlab = "Generated Exponential Random Variables", col = "lightblue")  
hist(x\_2, 100, main = "Inverse Transform Method (1 - U)",   
 xlab = "Generated Exponential Random Variables", col = "lightgreen")  
hist(x\_3, 100, main = "R Generated Exponential Distribution",   
 xlab = "Generated Exponential Random Variables", col = "pink")



## Inverse Transform Method Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1gN04hmsbdBEvmkJU6XKBVUJvdG_6ORl1#scrollTo=FXfMShelZoF1)

# Generating Uniforms Example

## “Desert Island” Generator

Refer back to M1L8 Generating Randomness, specifically the section on Unif(0, 1) PRNs and M1L7 More Baby Examples, specifically the section Evil Random Numbers.

## FORTRAN Implementation Example R Code

The [FORTRAN](https://en.wikipedia.org/wiki/Fortran) implementation is provided as an example to show how it can be done to provide more numerical stability. The implementation is from Bratley, P., Fox, B., & Schrage, L. (1987). A guide to simulation (2nd ed.). Springer Science+Business.

Also see a short description of [Schrage’s algorithm](https://mathworld.wolfram.com/SchragesAlgorithm.html) as well as this description of [Schrage’s method](https://en.wikipedia.org/wiki/Lehmer_random_number_generator#Schrage's_method)

FUNCTION UNIF( IX)

C PORTABLE RANDOM NUMBER GENERATOR IMPLEMENTING THE RECURSION:

C IX = 16807 \* IX MOD (2\*\*(31) - 1)

C USING ONLY 32 BITS, INCLUDING SIGN.

C SOME COMPILERS REQUIRE THE DECLARATION :

C INTEGER\*4 IX , Kl

C INPUT:

C IX = INTEGER GREATER THAN 0 AND LESS THAN 2147483647

C OUTPUTS:

C IX = NEW PSEUDORANDOM VALUE,

C UNIF = A UNIFORM FRACTION BETWEEN 0 AND 1.

Kl = IX/127773

IX = 16807\*( IX - Kl\*127773) - Kl \* 2836

IF ( IX .LT. 0) IX = IX + 2147483647

UNIF = IX\*4.6566I2875E-I0

RETURN

END

RETURN

END

The FORTRAN implementation provided above is an implementation of the [Lehmer random number generator](https://en.wikipedia.org/wiki/Lehmer_random_number_generator), also known as the Park-Miller generator, which is a particular type of linear congruential generator (LCG). This implementation uses the following constants:

a = 16807 (the multiplier)

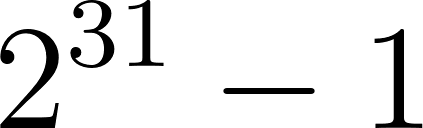
m = 2147483647 (the modulus)

c = 0 (the increment)

The function UNIF takes a seed value (IX) as input and returns a random number between 0 and 1. Here's a step-by-step explanation of the code:

* Calculate K1, the quotient of IX divided by 127773: K1 = IX/127773
* Update the seed value IX: IX = 16807\*(IX - K1\*127773) - K1\*2836
* If IX is negative, add the modulus 2147483647 to make it positive: IF(IX.LT.0) IX = IX + 2147483647
* Scale IX by multiplying with 4.656612875E-10 to obtain a random number between 0 and 1: UNIF = IX \* 4.656612875E-10

The constants 127773, 2836, 2147483647, and 4.656612875E-10 were chosen for specific reasons:

* 127773 and 2836: The modulus 2147483647 can be factored as 127773 \* 16807 + 2836. This factorization is used to implement Schrage's method, which helps avoid integer overflow when calculating a \* IX by breaking the computation into smaller parts.
* 2147483647 ([](https://www.codecogs.com/eqnedit.php?latex=2%5E%7B31%7D-1#0)): This is the largest prime number that can be represented as a signed 32-bit integer. Using a prime number as the modulus results in a full-period generator.
* 4.656612875E-10: This is the reciprocal of the modulus (1/2147483647). Multiplying IX by this constant scales the random number from the range [1, 2147483646] to the range (0, 1). It is equivalent to dividing by 2147483647.

These constants were chosen to satisfy the requirements of a good linear congruential generator, including a full-period generator with minimal correlations and a computationally efficient implementation.

The following is an R implementation using the FORTRAN implementation’s algorithm.

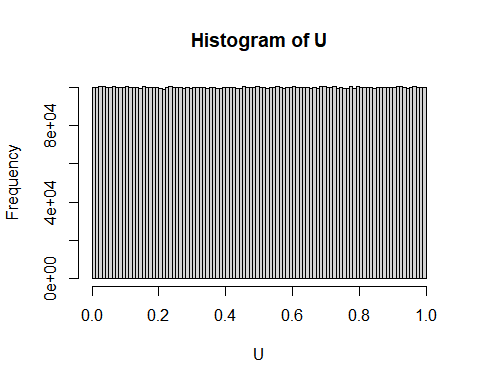
*# Initialize a seed value*  
seed <- 12345  
  
*# Define the Lehmer random number generator function*  
unif\_lehmer <- **function**(IX, n) {  
   
 *# Initialize an empty vector to store the generated random numbers*  
 random\_numbers <- numeric(n)  
   
 **for** (i **in** 1:n) {  
 *# Calculate K1*  
 K1 <- floor(IX / 127773)  
   
 *# Update the seed value*  
 IX <- 16807 \* (IX - K1 \* 127773) - K1 \* 2836  
   
 *# Make seed positive if it's negative*  
 **if** (IX < 0) {  
 IX <- IX + 2147483647  
 }  
   
 *# Scale the seed value to obtain a random number between 0 and 1*  
 random\_numbers[i] <- IX \* (1 / 2147483647)  
 }  
   
 return(random\_numbers)  
}  
  
*# Generate n uniform random numbers using the Lehmer random number generator*  
n <- 10  
random\_numbers <- unif\_lehmer(seed, n)  
  
*# Print the generated random numbers*  
random\_numbers

## [1] 0.09661653 0.83399463 0.94770250 0.03587859 0.01154585 0.05115522  
## [7] 0.76578717 0.58492974 0.91413005 0.78380039

U <- unif\_lehmer(123, 1e7)  
summary(U)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.0000 0.2499 0.5000 0.5000 0.7500 1.0000

hist(U, 100)



The following code compares the execution time of three methods: 1) a previously implemented LCG algorithm in R (see [M1L7 More Baby Examples](https://docs.google.com/document/u/0/d/1fn0n3x8Hrdr4fzwIpKbRXh0JFT6Wiz9E8kE3GoyFrUk/edit)), 2) the FORTRAN algorithm implemented in R, and 3) the built-in uniform random variate generation function in R.

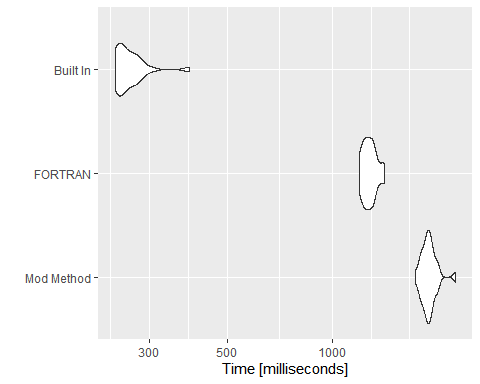
The built-in R function is the clear winner; however, the FORTRAN algorithm implemented in R shows that it is faster than the previous algorithm that used standard modular arithmetic. It would probably be even faster if it was run as a native FORTRAN program.

*# Define the Lehmer random number generator function*  
unif\_lehmer <- **function**(IX, n) {  
   
 *# Initialize an empty vector to store the generated random numbers*  
 random\_numbers <- numeric(n)  
   
 **for** (i **in** 1:n) {  
 *# Calculate K1*  
 K1 <- floor(IX / 127773)  
   
 *# Update the seed value*  
 IX <- 16807 \* (IX - K1 \* 127773) - K1 \* 2836  
   
 *# Make seed positive if it's negative*  
 **if** (IX < 0) {  
 IX <- IX + 2147483647  
 }  
   
 *# Scale the seed value to obtain a random number between 0 and 1*  
 random\_numbers[i] <- IX \* (1 / 2147483647)  
 }  
   
 return(random\_numbers)  
}  
  
*# LCG function*  
lcg <- **function**(n, seed, lcg\_a, lcg\_c, lcg\_m) {  
 random\_numbers <- numeric(n)  
 x <- seed  
 **for** (i **in** 1:n) {  
 x <- (lcg\_a \* x + lcg\_c) %% lcg\_m  
 random\_numbers[i] <- x / lcg\_m  
 }  
 return(random\_numbers)  
}  
  
library(microbenchmark)

seed <- 123456789  
n <- 1e7  
mb <- microbenchmark("Mod Method" = lcg(n, seed, 16807, 0, 2^31 - 1),  
 "FORTRAN" = unif\_lehmer(seed, n),  
 "Built In" = runif(n),  
 times= 25)  
  
mb

## Unit: milliseconds  
## expr min lq mean median uq max neval  
## Mod Method 1727.9072 1840.2676 1891.1261 1878.6094 1922.7445 2252.1253 25  
## FORTRAN 1195.3617 1235.5723 1278.9605 1265.3031 1308.2982 1409.7488 25  
## Built In 239.6412 245.8785 263.7588 255.3492 272.3532 388.9429 25

autoplot(mb)



## FORTRAN Implementation Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1gN04hmsbdBEvmkJU6XKBVUJvdG_6ORl1#scrollTo=Z11E93QyZoF2)