# Randomness

[Randomness](https://en.wikipedia.org/wiki/Randomness) is a fundamental concept in simulations as it allows for the modeling of uncertain events, variability, and complex behaviors. [Random variables (RVs)](https://en.wikipedia.org/wiki/Random_variable), such as interarrival times and service times, are used to imitate the stochastic nature of real-world systems. To generate these RVs, pseudorandom numbers (PRNs) are employed.

PRNs are numbers generated by deterministic algorithms that mimic the properties of true random numbers. They are not truly random but appear to be random in their distribution and behavior. In simulations, [Unif(0, 1) PRNs](https://en.wikipedia.org/wiki/Continuous_uniform_distribution) serve as the basis for generating other types of random variables. These PRNs are uniformly distributed between 0 and 1, providing a continuous range of values.

PRNs are generated using [deterministic algorithms](https://en.wikipedia.org/wiki/Deterministic_algorithm), such as [linear congruential generators (LCGs](https://en.wikipedia.org/wiki/Linear_congruential_generator)) and [Mersenne Twister](https://en.wikipedia.org/wiki/Mersenne_Twister). These algorithms use specific formulas, [initial values (seeds)](https://en.wikipedia.org/wiki/Random_seed), and parameters to produce sequences of numbers that, while not truly random, possess properties similar to random numbers.

To ensure the quality of PRNs, several tests are performed to verify their randomness. These tests assess properties such as uniformity, independence, and the absence of patterns or correlations. Common tests include the [chi-squared test](https://en.wikipedia.org/wiki/Chi-squared_test), the [Kolmogorov-Smirnov test](https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test), and [autocorrelation](https://en.wikipedia.org/wiki/Autocorrelation) tests.

To generate other types of random variables, transformations can be applied to Unif(0, 1) PRNs. These transformations involve mathematical functions that map the PRNs to the desired probability distribution. Common techniques include the [inverse transform method](https://en.wikipedia.org/wiki/Inverse_transform_sampling), the [acceptance-rejection method](https://en.wikipedia.org/wiki/Rejection_sampling), and the [Box-Muller transform](https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform).

# Why is Randomness and Pseudorandom Number Generation Important?

Understanding the concept of randomness and the generation of uniform pseudorandom numbers is foundational to many types of simulations, especially in areas like Monte Carlo methods, stochastic processes, and probabilistic modeling. Here's why these concepts are so important:

* Modeling Uncertainty: Many systems have inherent uncertainty, whether it's due to variability in inputs, inherent noise in the system, or future events that can't be precisely predicted. Simulations often incorporate randomness to capture and study these uncertainties.
* Basis for Other Distributions: Uniform pseudorandom numbers can be transformed into other distributions using various techniques. For instance, the Box-Muller transform can generate normally distributed numbers from uniformly distributed ones. This ability to generate a wide range of random variates from uniform pseudorandom numbers is vital for many simulations.
* Monte Carlo Methods: One of the most widely-used simulation techniques, Monte Carlo methods rely heavily on generating random numbers to explore a solution space. Applications range from financial option pricing to computational physics.
* Efficiency and Scalability: Modern algorithms for generating pseudorandom numbers are designed to be both fast and able to produce large sequences of numbers without repeating. This scalability is essential for large-scale simulations.
* Reproducibility: While true randomness is unpredictable, pseudorandom numbers are generated algorithmically and can be reproduced if the initial seed of the generator is known. Reproducibility is crucial in scientific simulations to validate results and allow others to replicate findings.
* Stochastic Processes and Queueing: In areas like operations research or systems modeling, understanding randomness is vital. For instance, the arrivals of customers in a queue might be modeled as a Poisson process, which is inherently random.
* Sampling and Integration: Random sampling techniques are used for integration in high-dimensional spaces, where traditional methods become computationally infeasible.
* Testing and Validation: In software development and system design, random inputs are often used to stress-test systems and identify vulnerabilities or potential points of failure.
* Statistical Validity: Understanding randomness is essential to ensure the statistical validity of simulation results. For instance, if one is using randomness to simulate 10,000 scenarios, understanding the nature of the randomness used can help in determining the confidence intervals for the results.
* Realism in Graphics and Gaming: In computer graphics and game design, randomness is used to generate realistic textures, terrains, and behaviors. Without a good source of randomness, many of the visual effects and behaviors would look artificial or too regular.
* Cryptographic Applications: While not strictly a simulation, cryptographic protocols require high-quality random numbers for tasks like key generation, making randomness vital in the domain of security.

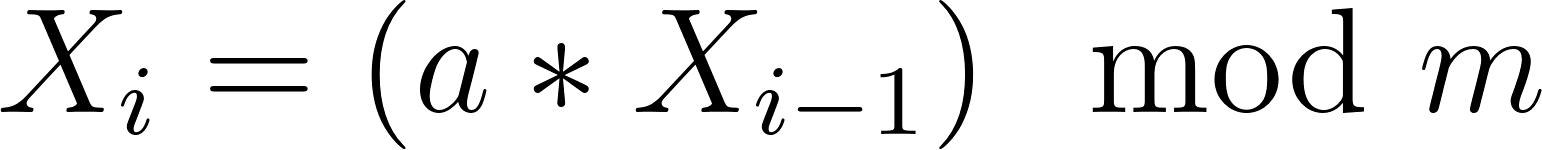
The ability to understand and harness randomness is a cornerstone in the realm of simulations. The right incorporation of randomness can provide a more realistic, expansive, and thorough exploration of various systems and scenarios.

# Unif(0,1) PRNs

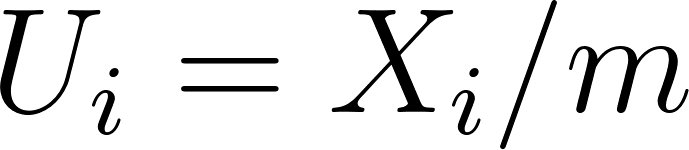
Unif(0, 1) PRNs are pseudorandom numbers that are uniformly distributed between 0 and 1. They are generated using deterministic algorithms, such as the Linear Congruential Generator (LCG), which is a popular algorithm for producing PRNs with specific properties. The LCG algorithm uses a seed, constants, and a modulus function to create sequences of pseudorandom numbers that resemble true random numbers.

Linear Congruential Generator (LCG):

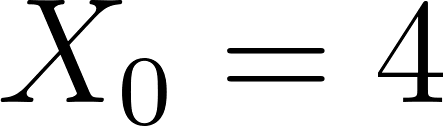
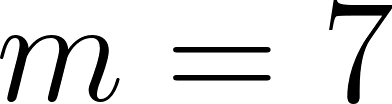
The LCG algorithm can be defined by the following recurrence relation:

[](https://www.codecogs.com/eqnedit.php?latex=X_i%20%3D%20(a%20*%20X_%7Bi-1%7D)%20%5Cmod%20m#0)

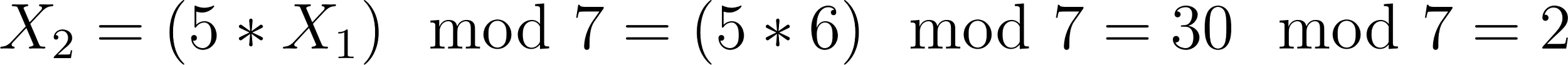
where [](https://www.codecogs.com/eqnedit.php?latex=X_i#0) represents the ith pseudorandom number, [](https://www.codecogs.com/eqnedit.php?latex=X_%7Bi-1%7D#0) is the previous intermedial value (these intermediate values are used in the calculations that generate the actual PRNS), [](https://www.codecogs.com/eqnedit.php?latex=a#0) and [](https://www.codecogs.com/eqnedit.php?latex=m#0) are carefully chosen constants, and [](https://www.codecogs.com/eqnedit.php?latex=%5Cmod#0) is the modulus function. The [](https://www.codecogs.com/eqnedit.php?latex=i#0)th PRN, [](https://www.codecogs.com/eqnedit.php?latex=U_i#0), is calculated by dividing [](https://www.codecogs.com/eqnedit.php?latex=X_i#0) by [](https://www.codecogs.com/eqnedit.php?latex=m#0):

[](https://www.codecogs.com/eqnedit.php?latex=U_i%20%3D%20X_i%20%2F%20m#0)

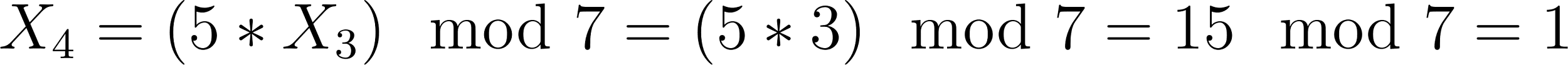
## Pretend Example

Let's work through an example using the following parameters: [](https://www.codecogs.com/eqnedit.php?latex=X_0%20%3D%204#0) (initial seed), [](https://www.codecogs.com/eqnedit.php?latex=a%20%3D%205#0), and [](https://www.codecogs.com/eqnedit.php?latex=m%20%3D%207#0).

[](https://www.codecogs.com/eqnedit.php?latex=X_1%20%3D%20(5%20*%20X_0)%20%5Cmod%207%20%3D%20(5%20*%204)%20%5Cmod%207%20%3D%2020%20%5Cmod%207%20%3D%206#0)

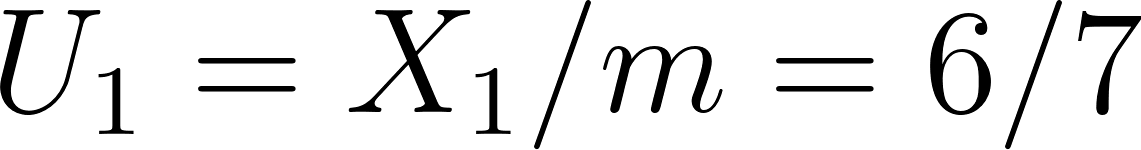
[](https://www.codecogs.com/eqnedit.php?latex=X_2%20%3D%20(5%20*%20X_1)%20%5Cmod%207%20%3D%20(5%20*%206)%20%5Cmod%207%20%3D%2030%20%5Cmod%207%20%3D%202#0)

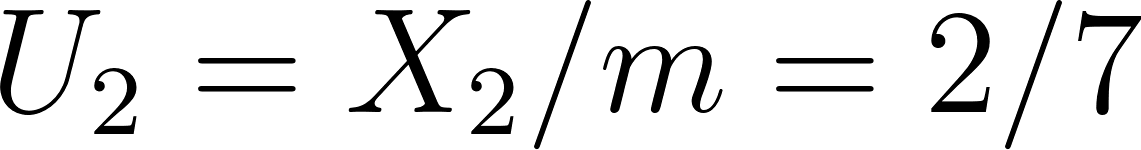
[](https://www.codecogs.com/eqnedit.php?latex=X_3%20%3D%20(5%20*%20X_2)%20%5Cmod%207%20%3D%20(5%20*%202)%20%5Cmod%207%20%3D%2010%20%5Cmod%207%20%3D%203#0)

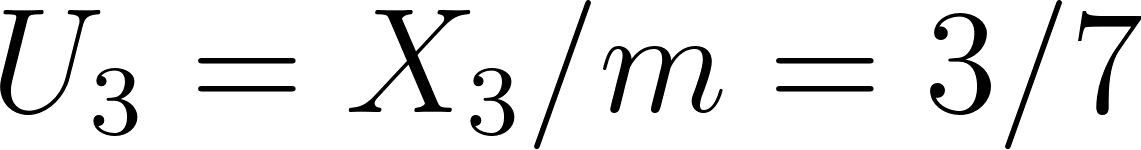
[](https://www.codecogs.com/eqnedit.php?latex=X_4%20%3D%20(5%20*%20X_3)%20%5Cmod%207%20%3D%20(5%20*%203)%20%5Cmod%207%20%3D%2015%20%5Cmod%207%20%3D%201#0)

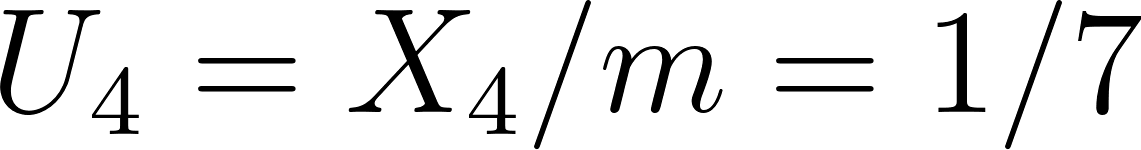
[](https://www.codecogs.com/eqnedit.php?latex=X_5%20%3D%20(5%20*%20X_4)%20%5Cmod%207%20%3D%20(5%20*%201)%20%5Cmod%207%20%3D%205%20%5Cmod%207%20%3D%205#0)

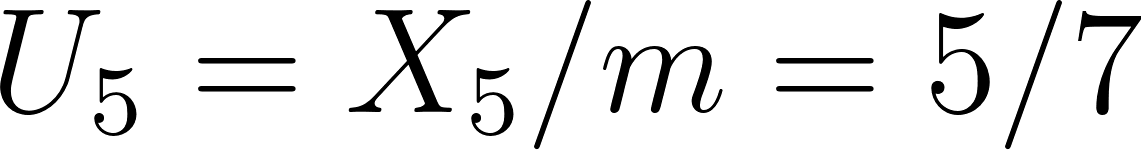
Now, we can calculate the corresponding PRNs:

[](https://www.codecogs.com/eqnedit.php?latex=U_1%20%3D%20X_1%20%2F%20m%20%3D%206%20%2F%207#0)

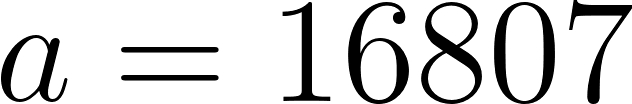
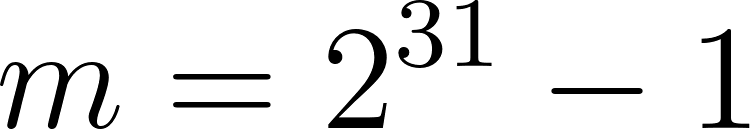
[](https://www.codecogs.com/eqnedit.php?latex=U_2%20%3D%20X_2%20%2F%20m%20%3D%202%20%2F%207#0)

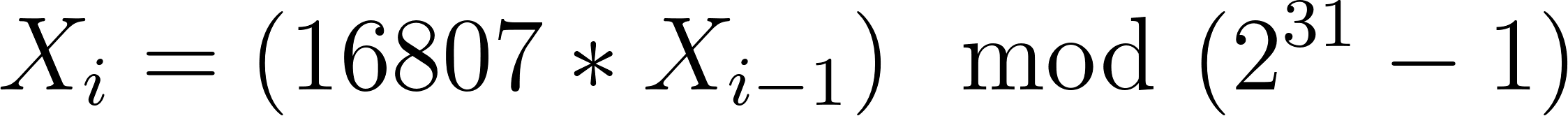
[](https://www.codecogs.com/eqnedit.php?latex=U_3%20%3D%20X_3%20%2F%20m%20%3D%203%20%2F%207#0)

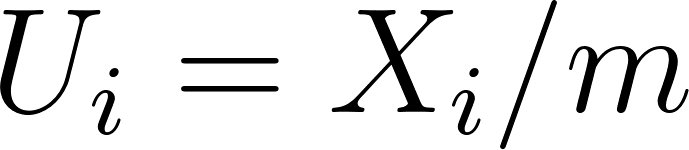
[](https://www.codecogs.com/eqnedit.php?latex=U_4%20%3D%20X_4%20%2F%20m%20%3D%201%20%2F%207#0)

[](https://www.codecogs.com/eqnedit.php?latex=U_5%20%3D%20X_5%20%2F%20m%20%3D%205%20%2F%207#0)

## Real Example (Desert Island Generator)

The Desert Island Generator, also known as the [Park-Miller generator or MINSTD](https://en.wikipedia.org/wiki/Lehmer_random_number_generator), is an LCG with the following parameters: [](https://www.codecogs.com/eqnedit.php?latex=a%20%3D%2016807#0) and [](https://www.codecogs.com/eqnedit.php?latex=m%20%3D%202%5E%7B31%7D%20-%201#0) (prime number). It calculates [](https://www.codecogs.com/eqnedit.php?latex=X_i#0) and [](https://www.codecogs.com/eqnedit.php?latex=U_i#0) as follows:

[](https://www.codecogs.com/eqnedit.php?latex=X_i%20%3D%20(16807%20*%20X_%7Bi-1%7D)%20%5Cmod%20(2%5E%7B31%7D%20-%201)#0)

[](https://www.codecogs.com/eqnedit.php?latex=U_i%20%3D%20X_i%20%2F%20m#0)

Note that the name “Desert Island” is more general and means any method that works reasonably well and is fairly simple. It is a play on the idea that if you were stuck on a desert island what is something you would take with you. In this case, the Park-Miller generator is nicknamed as a desert island method because it’s something you could use on a desert island to generate fairly well-behaved Unif(0, 1) PRNs.

This generator is used in several simulation languages and has desirable properties such as long periods (cycle times) and good statistical properties. While it is a widely used generator, better generators with improved properties have been developed, such as the Mersenne Twister.

## Unif(0,1) PRNs Example R Code

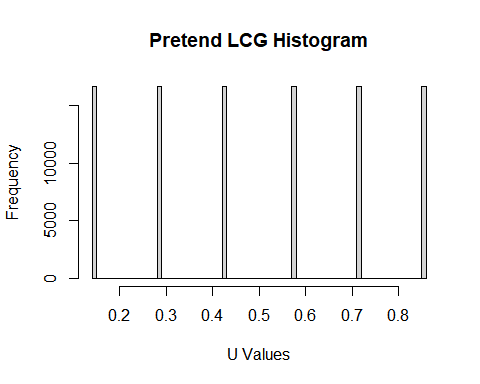
*# Pretend LCG example*  
pretend\_lcg <- **function**(seed, a, m, n) {  
 X <- numeric(n)  
 U <- numeric(n)  
   
 X[1] <- seed  
   
 **for** (i **in** 2:n) {  
 X[i] <- (a \* X[i - 1]) %% m  
 }  
   
 U <- X / m  
 list(X\_values = X, U\_values = U)  
}  
  
*# Desert Island LCG (Park-Miller)*  
desert\_island\_lcg <- **function**(seed, n) {  
 a <- 16807  
 m <- 2^31 - 1  
   
 X <- numeric(n)  
 U <- numeric(n)  
   
 X[1] <- seed  
   
 **for** (i **in** 2:n) {  
 X[i] <- (a \* X[i - 1]) %% m  
 }  
   
 U <- X / m  
 list(X\_values = X, U\_values = U)  
}  
  
*# Usage examples*  
pretend\_result <- pretend\_lcg(seed = 4, a = 5, m = 7, n = 6)  
print(paste("Pretend LCG X values:", toString(pretend\_result$X\_values)))

## [1] "Pretend LCG X values: 4, 6, 2, 3, 1, 5"

print(paste("Pretend LCG U values:", toString(pretend\_result$U\_values)))

## [1] "Pretend LCG U values: 0.571428571428571, 0.857142857142857, 0.285714285714286, 0.428571428571429, 0.142857142857143, 0.714285714285714"

hist(pretend\_lcg(seed = 4, a = 5, m = 7, n = 1e5)$U\_values, 100,  
 main = "Pretend LCG Histogram",  
 xlab = "U Values",  
 ylab = "Frequency")



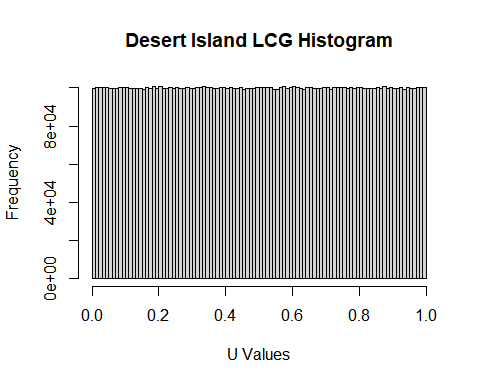
desert\_island\_result <- desert\_island\_lcg(seed = 12345, n = 6)  
print(paste("Desert Island LCG X values:",   
 toString(desert\_island\_result$X\_values)))

## [1] "Desert Island LCG X values: 12345, 207482415, 1790989824, 2035175616, 77048696, 24794531"

print(paste("Desert Island LCG U values:",   
 toString(desert\_island\_result$U\_values)))

## [1] "Desert Island LCG U values: 5.74858859449094e-06, 0.0966165285076092, 0.83399462738726, 0.94770249768519, 0.0358785949814499, 0.0115458532290281"

hist(desert\_island\_lcg(seed = 12345, n = 1e7)$U\_values, 100,  
 main = "Desert Island LCG Histogram",  
 xlab = "U Values",  
 ylab = "Frequency")



## Unif(0,1) PRNs Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1UDCdHwD8znSNm5drSzept97-b2d7K7FF#scrollTo=KExXkcbaRVGo)

# Generating Other RVs

Uniform pseudorandom numbers (PRNs) can be used as the basis for generating other types of random variables with different probability distributions. To do this, we apply appropriate transformations to the uniformly distributed PRNs.

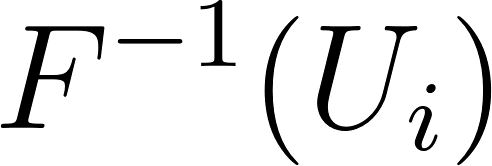
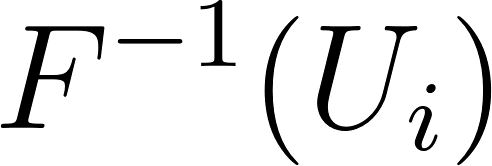
## Starting Point:

The starting point for generating other random variables is to obtain a sequence of uniformly distributed PRNs, [](https://www.codecogs.com/eqnedit.php?latex=U_i#0), in the range (0, 1). These values can be generated using methods such as the Linear Congruential Generator or other algorithms.

Various transformations can be applied to the uniform PRNs to obtain random variables with different probability distributions. Some examples include the exponential distribution, normal distribution, and Poisson distribution. One popular method for achieving this is the inverse transform method.

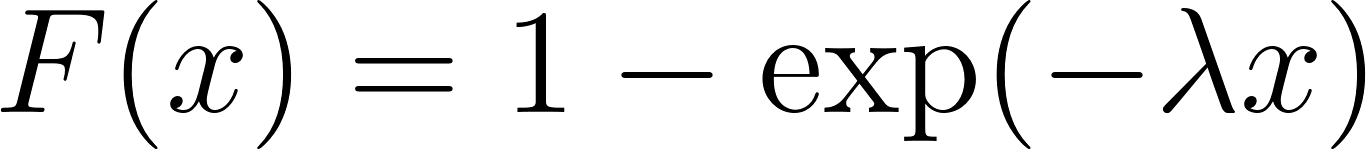
The inverse transform method is a technique used to generate random variables with a specified probability distribution by transforming uniform PRNs. The basic idea is to use the cumulative distribution function (CDF) of the desired distribution and its inverse.

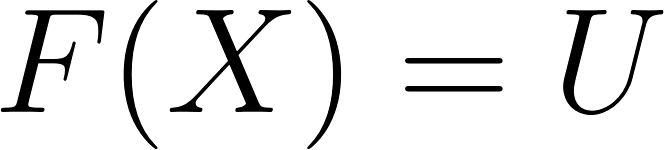
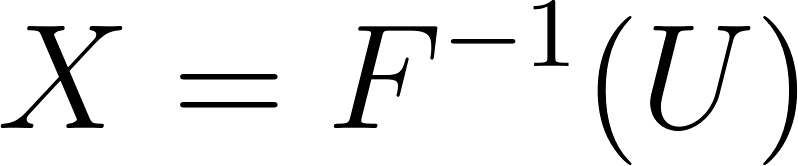
The general steps for the inverse transform method are:

* Generate a uniform PRN, [](https://www.codecogs.com/eqnedit.php?latex=U_i#0), in the range (0, 1).
* Determine the inverse of the CDF, [](https://www.codecogs.com/eqnedit.php?latex=F%5E%7B-1%7D(U_i)#0), for the desired probability distribution.
* The value obtained in previous step, [](https://www.codecogs.com/eqnedit.php?latex=F%5E%7B-1%7D(U_i)#0), is a random variable with the desired distribution.

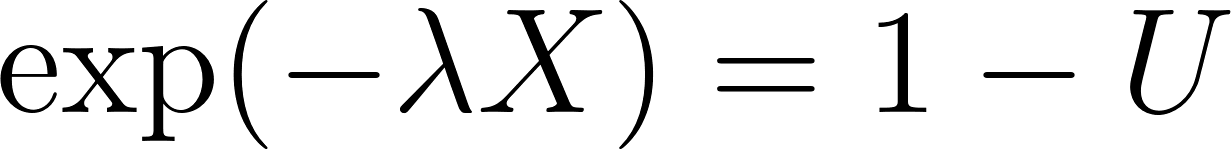
## Example: Exponential Distribution

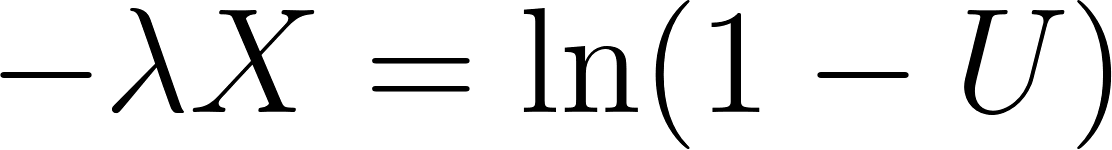
Suppose we want to generate random variables with an exponential distribution, [](https://www.codecogs.com/eqnedit.php?latex=%5Ctext%7BExp%7D(%5Clambda)#0). The CDF of the exponential distribution is given by:

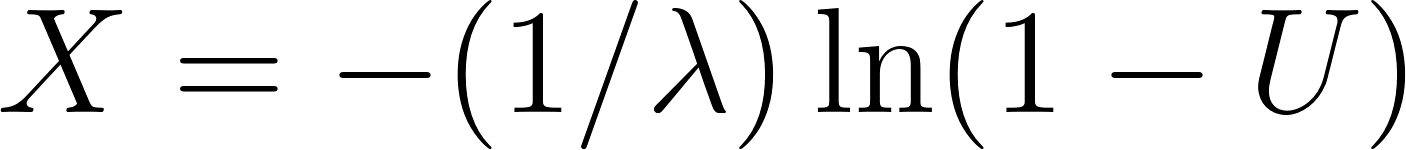
[](https://www.codecogs.com/eqnedit.php?latex=F(x)%20%3D%201%20-%20%5Cexp(-%5Clambda%20x)#0)

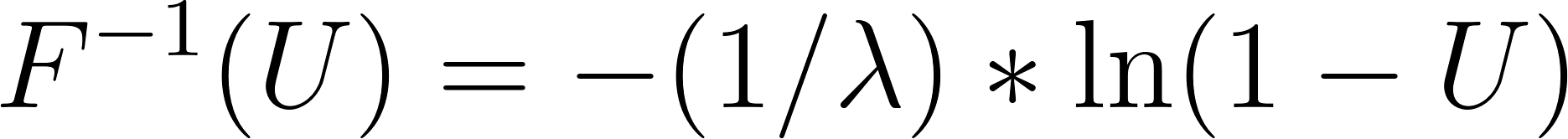
To apply the inverse transform method, we first set the random variable [](https://www.codecogs.com/eqnedit.php?latex=F(X)%3DU#0) and solve for [](https://www.codecogs.com/eqnedit.php?latex=X#0) which is represented as [](https://www.codecogs.com/eqnedit.php?latex=X%3DF%5E%7B-1%7D(U)#0):

[](https://www.codecogs.com/eqnedit.php?latex=U%20%3D%201%20-%20%5Cexp(-%5Clambda%20X)#0)

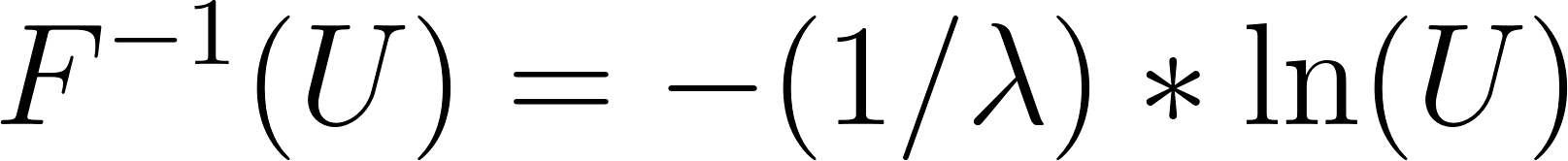
[](https://www.codecogs.com/eqnedit.php?latex=%5Cexp(-%5Clambda%20X)%20%3D%201%20-%20U#0)

[](https://www.codecogs.com/eqnedit.php?latex=-%5Clambda%20X%20%3D%20%5Cln(1-U)#0)

[](https://www.codecogs.com/eqnedit.php?latex=X%20%3D%20-(1%20%2F%20%5Clambda)%20%5Cln%20(1-U)#0)

[](https://www.codecogs.com/eqnedit.php?latex=F%5E%7B-1%7D(U)%20%3D%20-(1%20%2F%20%5Clambda)%20*%20%5Cln(1%20-%20U)#0)

Since [](https://www.codecogs.com/eqnedit.php?latex=U#0) is uniformly distributed in the range (0, 1), [](https://www.codecogs.com/eqnedit.php?latex=1%20-%20U#0) is also uniformly distributed in the range (0, 1) (see [proof at end of document](#a9t1f5nndjz4); also see [this section in the inverse transform notes](https://docs.google.com/document/d/1fpQrzpObFTpGE3vNHpLuG6UfQnVj31TiMPvaZcBwPpk/edit#bookmark=id.fkuachqvbe6m)). Therefore, we can simplify the expression:

[](https://www.codecogs.com/eqnedit.php?latex=F%5E%7B-1%7D(U)%20%3D%20-(1%20%2F%20%5Clambda)%20*%20%5Cln(U)#0)

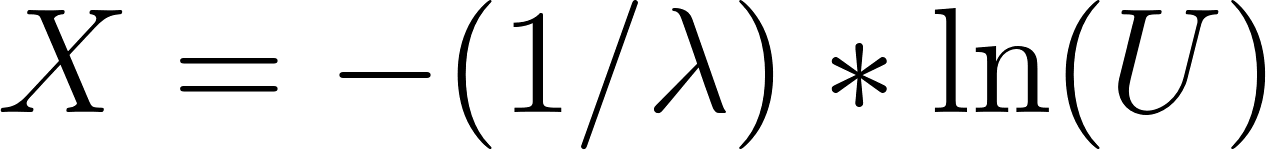
By plugging in the uniform PRN, [](https://www.codecogs.com/eqnedit.php?latex=U_i#0), into the above expression, we obtain a random variable that follows the exponential distribution with parameter [](https://www.codecogs.com/eqnedit.php?latex=%5Clambda#0).

The inverse transform method can be used to generate random variables with various important distributions. However, there are also many other sophisticated methods available for generating random variables, such as rejection sampling and the Box-Muller transform for normal distributions.

## Generating Other RVs Example R Code

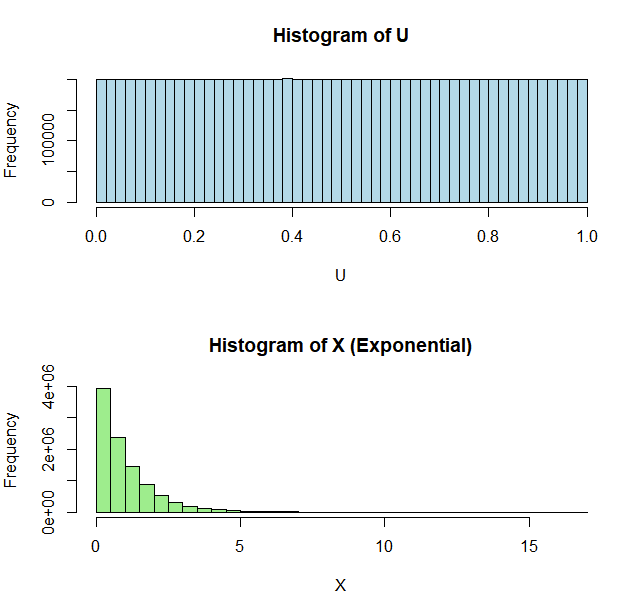
Using set.seed(123) sets the random seed to a fixed value (123 in this case) to ensure that the same sequence of random numbers is generated each time the script is run. This is useful for demonstration purposes and for making results reproducible.

The code generates a sequence of 1e7 (10 million) uniform random variables in the range (0, 1) using the runif() function and stores them in the variable [](https://www.codecogs.com/eqnedit.php?latex=U#0).

The code sets the rate parameter ([](https://www.codecogs.com/eqnedit.php?latex=%5Clambda#0)) for the exponential distribution to 1. Then, using the inverse transform method, it generates exponential random variables by applying the transformation [](https://www.codecogs.com/eqnedit.php?latex=X%20%3D%20-(1%20%2F%20%5Clambda)%20*%20%5Cln(U)#0) and stores them in the variable [](https://www.codecogs.com/eqnedit.php?latex=X#0).

By comparing the histograms of [](https://www.codecogs.com/eqnedit.php?latex=U#0) and [](https://www.codecogs.com/eqnedit.php?latex=X#0), we can visually confirm that [](https://www.codecogs.com/eqnedit.php?latex=U#0) is uniformly distributed and [](https://www.codecogs.com/eqnedit.php?latex=X#0) follows an exponential distribution

*# Set seed for reproducibility*  
set.seed(123)  
  
*# Generate uniform random variables*  
U <- runif(1e7)  
  
*# Inverse transform method for exponential distribution*  
lambda <- 1 *# Rate parameter for the exponential distribution*  
X <- - (1 / lambda) \* log(U)  
  
*# Plot histograms of U and X*  
par(mfrow = c(2, 1))  
hist(U, main = "Histogram of U", xlab = "U", col = "lightblue",   
 border = "black", breaks = 50)  
hist(X, main = "Histogram of X (Exponential)", xlab = "X", col = "lightgreen",   
 border = "black", breaks = 50)



## Generating Other RVs Example Python Code

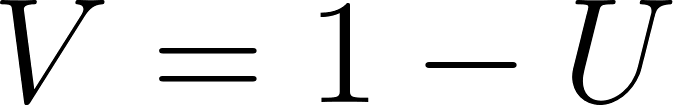
[Python Notebook](https://colab.research.google.com/drive/1UDCdHwD8znSNm5drSzept97-b2d7K7FF#scrollTo=YIrW7CGxRc6B)

# Proof of the Uniformity of and

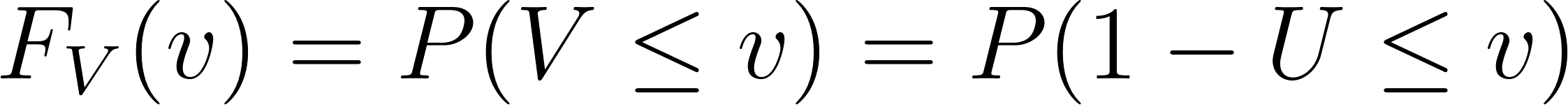
To prove that if [](https://www.codecogs.com/eqnedit.php?latex=U#0) is uniformly distributed in the range (0, 1), [](https://www.codecogs.com/eqnedit.php?latex=1%20-%20U#0) is also uniformly distributed in the range (0, 1), we can examine the cumulative distribution function (CDF) and probability density function (PDF) of the transformed random variable ([](https://www.codecogs.com/eqnedit.php?latex=1%20-%20U#0)).

Let [](https://www.codecogs.com/eqnedit.php?latex=U#0) be a random variable uniformly distributed in the range (0, 1) with PDF:

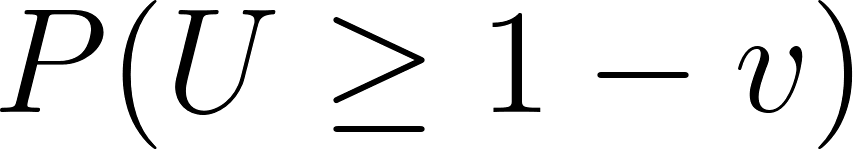
[](https://www.codecogs.com/eqnedit.php?latex=f_U(u)%20%3D%201#0) for [](https://www.codecogs.com/eqnedit.php?latex=0%20%5Cleq%20u%20%5Cleq%201#0) and [](https://www.codecogs.com/eqnedit.php?latex=0#0) otherwise

Now, let [](https://www.codecogs.com/eqnedit.php?latex=V%20%3D%201%20-%20U#0). We want to show that [](https://www.codecogs.com/eqnedit.php?latex=V#0) is also uniformly distributed in the range (0, 1).

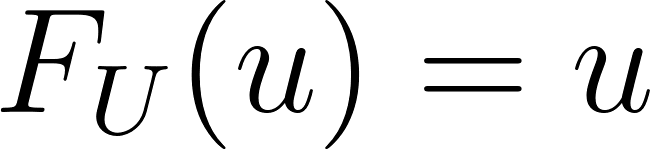
To do this, let's find the CDF of [](https://www.codecogs.com/eqnedit.php?latex=V#0), [](https://www.codecogs.com/eqnedit.php?latex=F_V(v)#0). By definition:

[](https://www.codecogs.com/eqnedit.php?latex=F_V(v)%20%3D%20P(V%20%5Cleq%20v)%20%3D%20P(1%20-%20U%20%5Cleq%20v)#0)

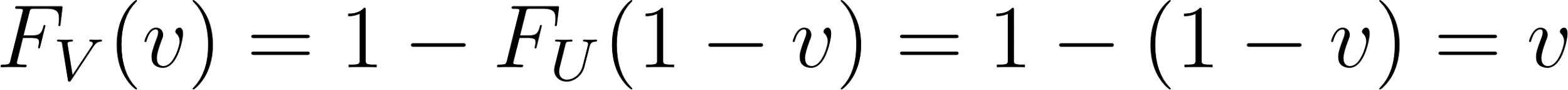
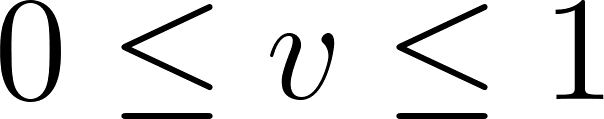
Rearranging the inequality to express it in terms of [](https://www.codecogs.com/eqnedit.php?latex=U#0):

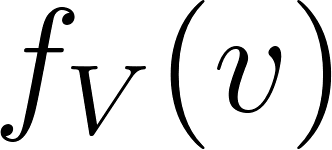
[](https://www.codecogs.com/eqnedit.php?latex=P(U%20%5Cgeq%201%20-%20v)#0)

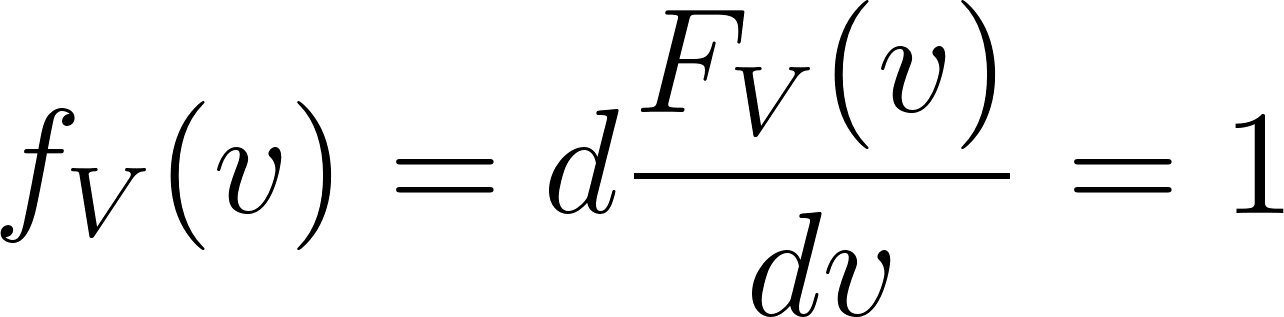
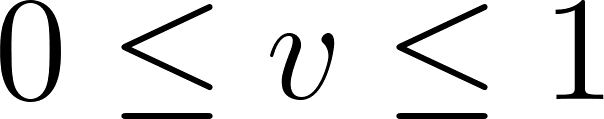
Since [](https://www.codecogs.com/eqnedit.php?latex=U#0) is uniformly distributed, we can express the CDF of [](https://www.codecogs.com/eqnedit.php?latex=U#0), [](https://www.codecogs.com/eqnedit.php?latex=F_U(u)#0), as:

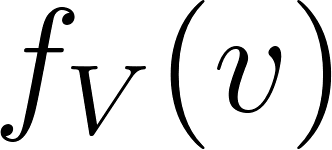
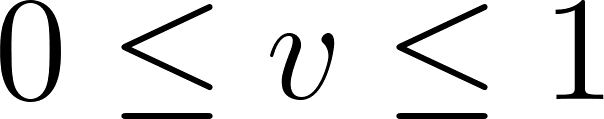
[](https://www.codecogs.com/eqnedit.php?latex=F_U(u)%20%3D%20u#0) for [](https://www.codecogs.com/eqnedit.php?latex=0%20%5Cleq%20u%20%5Cleq%201#0)

So, the CDF of [](https://www.codecogs.com/eqnedit.php?latex=V#0) can be expressed as:

[](https://www.codecogs.com/eqnedit.php?latex=F_V(v)%20%3D%201%20-%20F_U(1%20-%20v)%20%3D%201%20-%20(1%20-%20v)%20%3D%20v#0) for[](https://www.codecogs.com/eqnedit.php?latex=%200%20%5Cleq%20v%20%5Cleq%201#0)

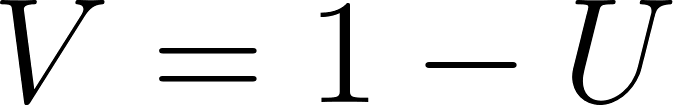
Now, let's find the PDF of [](https://www.codecogs.com/eqnedit.php?latex=V#0), [](https://www.codecogs.com/eqnedit.php?latex=f_V(v)#0), by taking the derivative of [](https://www.codecogs.com/eqnedit.php?latex=F_V(v)#0) with respect to [](https://www.codecogs.com/eqnedit.php?latex=v#0):

[](https://www.codecogs.com/eqnedit.php?latex=f_V(v)%20%3D%20d%5Cdfrac%7BF_V(v)%7D%7Bdv%7D%3D%201#0) for [](https://www.codecogs.com/eqnedit.php?latex=0%20%5Cleq%20v%20%5Cleq%201#0) and [](https://www.codecogs.com/eqnedit.php?latex=0#0) otherwise

Since the PDF of [](https://www.codecogs.com/eqnedit.php?latex=V#0), [](https://www.codecogs.com/eqnedit.php?latex=f_V(v)#0), is equal to [](https://www.codecogs.com/eqnedit.php?latex=1#0) for [](https://www.codecogs.com/eqnedit.php?latex=0%20%5Cleq%20v%20%5Cleq%201#0) and [](https://www.codecogs.com/eqnedit.php?latex=0#0) otherwise, [](https://www.codecogs.com/eqnedit.php?latex=V#0) is also uniformly distributed in the range (0, 1). Therefore, we have shown that if [](https://www.codecogs.com/eqnedit.php?latex=U#0) is uniformly distributed in the range (0, 1), [](https://www.codecogs.com/eqnedit.php?latex=1%20-%20U#0) is also uniformly distributed in the range (0, 1)

## U and 1-U Example R Code

Using set.seed(123) sets the random seed to a fixed value (123 in this case) to ensure that the same sequence of random numbers is generated each time the script is run. This is useful for demonstration purposes and for making results reproducible.

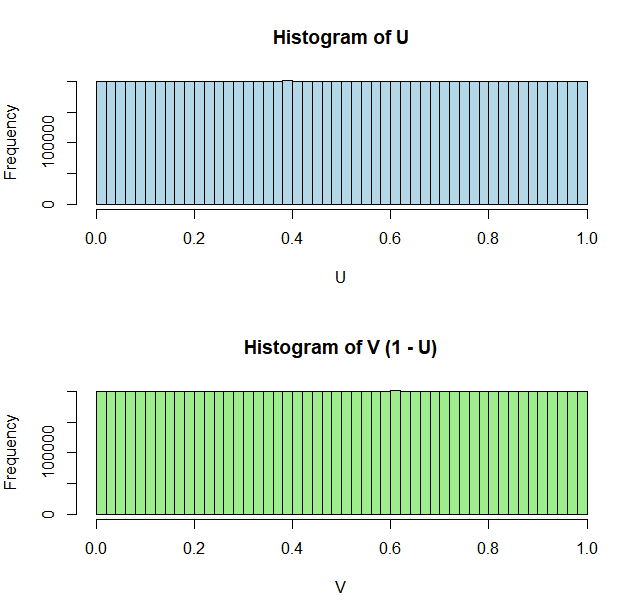
The code generates a sequence of 1e7 (10 million) uniform random variables in the range (0, 1) using the runif() function and stores them in the variable U. It then creates a new variable [](https://www.codecogs.com/eqnedit.php?latex=V#0) by subtracting each value in [](https://www.codecogs.com/eqnedit.php?latex=U#0) from [](https://www.codecogs.com/eqnedit.php?latex=1#0) (i.e., [](https://www.codecogs.com/eqnedit.php?latex=V%20%3D%201%20-%20U#0)).

The code sets up a layout for two plots in one column. It then plots histograms for both [](https://www.codecogs.com/eqnedit.php?latex=U#0) and [](https://www.codecogs.com/eqnedit.php?latex=V#0), with appropriate titles, x-axis labels, and colors.

The code calculates summary statistics for [](https://www.codecogs.com/eqnedit.php?latex=U#0) and [](https://www.codecogs.com/eqnedit.php?latex=V#0) using the summary() function, which provides the minimum, 1st quartile, median, mean, 3rd quartile, and maximum values for each variable.

By comparing the histograms and summary statistics of [](https://www.codecogs.com/eqnedit.php?latex=U#0) and [](https://www.codecogs.com/eqnedit.php?latex=V#0), we can visually confirm that both variables are uniformly distributed.

*# Set seed for reproducibility*  
set.seed(123)  
  
U <- runif(1e7)  
V <- 1 - U  
  
*# Plot histograms of U and V*  
par(mfrow = c(2, 1))  
hist(U, main = "Histogram of U", xlab = "U", col = "lightblue",   
 border = "black", breaks = 50)  
hist(V, main = "Histogram of V (1 - U)", xlab = "V", col = "lightgreen",   
 border = "black", breaks = 50)



*# Calculate summary statistics for U and V*  
summary\_U <- summary(U)  
summary\_V <- summary(V)  
  
*# Print summary statistics*  
cat("Summary statistics for U:\n")

## Summary statistics for U:

print(summary\_U)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.0000 0.2499 0.4999 0.4999 0.7499 1.0000

cat("\nSummary statistics for V (1 - U):\n")

##   
## Summary statistics for V (1 - U):

print(summary\_V)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.0000 0.2501 0.5001 0.5001 0.7501 1.0000

## U and 1-U Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1UDCdHwD8znSNm5drSzept97-b2d7K7FF#scrollTo=kZ0oCVtGRgAv)