Simulation of probabilities

Sagar Mukherjee

December 30, 2019

Abstract

A combination of Pseudo-random and Hardware-based random number generators is demonstrated using a Linear congruential generator and system-generated performance time counter of the order in nanoseconds. Thereby, a compilations of some standard algorithmic transformations is presented, with mathematical justifications and Python implementations. The idea is to first generate a Standard Uniform Random Variable and then transform it into another Random Variable which follows a specified probability law/distribution, by using an underlying mathematical algorithm.

1 Introduction

This report intends to mathematically justify the Module *"distributions.py"*, written on Python, discussing the derivations, implementations, and complexities. Firstly, a method of generating a Uniform Random Variable, between the interval [0, 1), is explained. Then, using this standard Uniform Random Variable, we generate other randomly distributed variables, following certain Probability laws. The analysis of each distribution is accompanied, with an appropriate algorithm, their mathematical derivations, their Python-implementation, etc., with complexities and other issues.

2 Random Number Generator

A Random number generator is any method of producing a sequence of "random numbers", that cannot be predicted with surety, beforehand, i.e., without the occurrence of an event, and can only be described with a *probability* or chance of occurrence.

There are mainly two classes of randomly generated variables. One, which generates Random numbers, using a physical entity, and thus its statistical randomness arises from microscopic deviations, such as temperature, time, pressure, etc. These are referred to, as the *"True random number generators"*. Two, which generates a deterministic sequence of random numbers, which, none-theless, exhibit complete randomness in appearance, and are referred to as the *"Pseudo-random number generators"*. Schematically, the algorithm flows as:

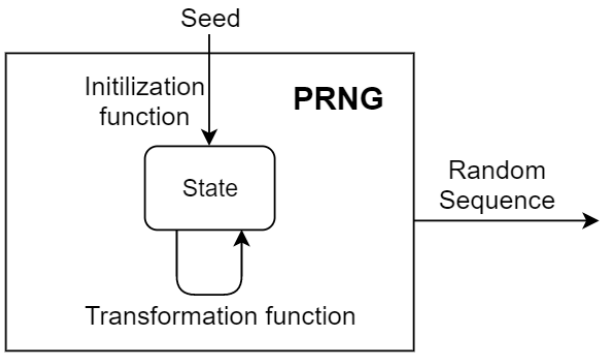


Figure 1: Pseudo-random number generator

Although hardware-based (physical) random sequences are closer to "true-random" numbers, pseudo-random number generators are much faster in practice and easily reproducible. This makes them *deterministic* in nature. For our purpose, we have combined both, in the sense, that we choose the seed for our pseudo-random number generator coming from the system time.

2.1 Linear Congruential Generator

One of the standard ways of obtaining a sequence of pseudo-random numbers is the multiplicative linear congruential method, which goes as follows:

Lemma 1 (Linear Congruential Generator)**.** *Let $a, m \in \mathbb{Z}_+$, and $X_0 \in \mathbb{Z}_+$ an initial seed. Define a sequence $(X_n)_{n\geq 1}$:*

$$X_{n+1} = aX_n \mod m \tag{1}$$

Then, X_n is a pseudo-random generated sequence.

```
def RNG():
        X = state                   # Restore previous state
        state = X = (a*X) % m       # Save the state
        yield X/m
```

Listing 1: Linear Congruential Generator

Remark. *X_n is a random integer from $\{0, 1, ..., m\}$ and hence X_n/m can be taken as an approximation/proxy for $X \sim$ Uniform$(0, 1)$, provided m is large enough for a given level of required precision/approximation.*

This follows from the simple reasoning that since X_n is a random integer between 0 and m, the sequence X_n/m are randomly generated fractions between 0 and 1, but are equally spaced.

Instead of a regular function, a python *generator* is used so as to be memory efficient, as none of the random numbers generated will have to be stored. Furthermore, it is also time-efficient, as we do not need to wait until all the numbers have been generated.

For pairs (a, m), it is noted that:

1. The sequence should "appear" random.
2. The values can be computed efficiently.
3. The constant, m, should be large enough, so that repetition doesn't occur soon enough.

A standard choice of the pair (a, m), which satisfies all the above 3 criteria is $a = 7^5 = 16807$ and $m = 2^{31} - 1 = 2147483647$. Modulus 'm' is chosen as a *Mersenne prime* here (2**31 - 1), so as to keep it coprime with any arbitrary seed, less than m, is computationally efficient, and also is large enough so as to not begin repitions.

2.2 System-time based *seed*

There are two ways, we can seed our random-number generator algorithm. One way is to have a user-provided seed. Another way is to combine hardware-based random number generation, which provides us with another level of randomness. For our implementation in Python we used the "time" module and used the performance counter method (fastest ticks among all methods), to give a time stamp in nanoseconds. It was found that this time stamp has last 2 digits always as *zero*. Hence we take the best significant digit, which is the 100^{th} digit of this timestamp.

```
rand_digit = lambda : int((int(perf_counter_ns()) / 100) % 10)
```

Listing 2: Random digit from $perf\_counter\_ns()$

We can use this to generate random digits from time and use them as our initial seed for the *state* of the pseudo-random number generator. One way to do this, is to take random digits, 4 times, and make that as our seed.

```
def seed(n = None):
        """
        Method to initialize the starting state of the RNG() method
        """
        if n != None:
            pass
        else:
            n = 0
            for i in range(4):
                n *= 10
                n += rand_digit()
        state = n
```

Listing 3: Seed

3 Transformation Analysis

Now, we will use this Standard Normal Variable approximation to derive other random variables with specified probability laws.

3.1 Discrete Uniform Random Variable

A Random Variable X, is said to follow a Discrete-uniform distribution, between integers a and b, if there's an equal probability of getting any random integer, in the set $\{a, a+1, a+2, ..., b\}$. Mathematically:

$$X \sim \mathcal{U}\{a, b\} \tag{2}$$

Theorem. *Let $U \sim Uniform(0,1)$, and $X = a + \lfloor (b-a)U \rfloor$ (where, $\lfloor . \rfloor$ denotes the greatest integer function). Then, $X \sim \mathcal{U}\{a, b-1\}$.*

Proof. Let $U \sim Uniform(0,1)$. We'll sequentially, transform this random variable into X, and prove its distribution.

$$\begin{aligned} & U \sim Uniform(0,1) \\ \implies & (b-a)U \sim Uniform(0, b-a) \\ \implies & \lfloor (b-a)U \rfloor \sim \mathcal{U}\{0, b-a-1\} \\ \implies & a + \lfloor (b-a)U \rfloor \sim \mathcal{U}\{a, b-1\} \\ \implies & X \sim \mathcal{U}\{a, b-1\} \end{aligned}$$

Hence, we have shown that the prescribed transformation, from U to X, produces a discrete-uniform random variable. □

Following is the Python implementation for this transformation:

```
def discreteUniform(a, b):
        U = next(RNG())
        X = a + int((b-a)*U)
        return X
```

Listing 4: Discrete Uniform Random Variable

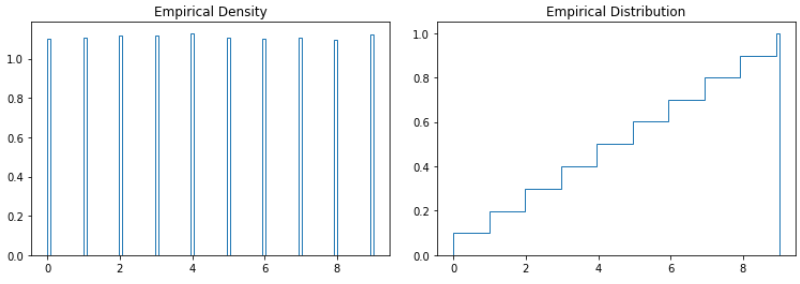


Figure 2: Discrete Uniform Distribution

3.2 Bernoulli Random Variable

A Random Variable X, is said to follow a *Bernoulli distribution*, with parameter p, if the probability measure for X satisfies:

$$\mathbb{P}(X) = \begin{cases} p & \text{if } X = 1 \\ 1-p & \text{if } X = 0 \end{cases}$$

Theorem (Inverse Transform Algorithm)**.** *Let $U \sim Uniform(0,1)$, and let $F(x), x \in \mathbb{R}$, be any cumulative distribution function. Define $X = F^{-1}(U)$. Then, $X \sim F$.*

Proof. To prove this, it is equivalent to show that:
$\mathbb{P}(X \leq x) = F^{-1}(U) = F(x), \forall x \in \mathbb{R}$. We have:

$$\begin{aligned} F_X(x) &= \mathbb{P}(X \leq x) \\ &= \mathbb{P}(F^{-1}(U) \leq x) \\ &= \mathbb{P}(F(F^{-1}(U)) \leq F(x)) \end{aligned}$$

The last equality is due to the fact that F is monotonic increasing, due to being a cumulative function. Now, since $F(F^{-1}(U)) = U$, we get:

$$\begin{aligned} F_X(x) &= \mathbb{P}(F(F^{-1}(U)) \leq F(x)) \\ &= \mathbb{P}(U \leq F(x)) \\ &= F(x) \end{aligned}$$

Hence, for any cdf F, $F^{-1}(U) \sim F$. □

Remark. *If F is not continuous, then we use the generalized inverse function: $F^{-1}(u) = min(x : F(x) \geq u)$. Thus, for obtaining Bernoulli Random Variable, $F^{-1}(U) = 1$, if $U \leq p$, else 0.*

Following is the Python implementation for this transformation:

```
def bernoulli(p):
        U = next(RNG())
        if U <= p: X = 1
        else: X = 0
        return X
```

Listing 5: Berboulli Random Variable

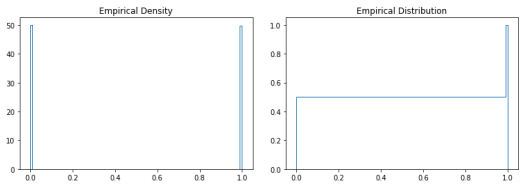


Figure 3: Bernoulli Distribution

3.3 Binomial Random Variable

A Random Variable X, is said to follow a *Binomial distribution*, with parameters n and p, if its a sum of successes from n independent Bernoulli trials, each with parameter probability, p, and follows the Probability law:

$$\mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, 2, ..., n. \tag{3}$$

Theorem. *Let $B_i \sim Bernoulli(p)$, $\forall i \in \{1, 2, ..., n\}$. Then, $X \sim Binomial(n, p)$, where X is the sum of each Bernoulli Random Variables:*

$$X = \sum_{i=1}^{n} B_i \tag{4}$$

Proof. Let $X = \sum_{i=1}^{n} B_i$. From a combinatorial argument, this means, that X is the sum of n independent Bernoulli trials, each with the success parameter p. Let $X = k$.

$\implies X$ is the number of ways of choosing k successes, and $(n-k)$ failures

$$\implies \mathbb{P}(X = k) = \binom{n}{k} \prod_{i=1}^{k} \mathbb{P}(B_i = 1) \prod_{i=1}^{n-k} \mathbb{P}(B_i = 0) = \binom{n}{k} p^k (1-p)^{n-k}$$

Hence, we have shown that the sum of n independent Bernoulli trials gives a Binomial Random Variable. □

Following is the Python implementation for this transformation:

```
def binomial(n, p):
        X = 0
        for i in range(n):
            U = next(RNG())
            if U <= p:                    # Success Case
                X += 1
        return X
```

Listing 6: Binomial Random Variable

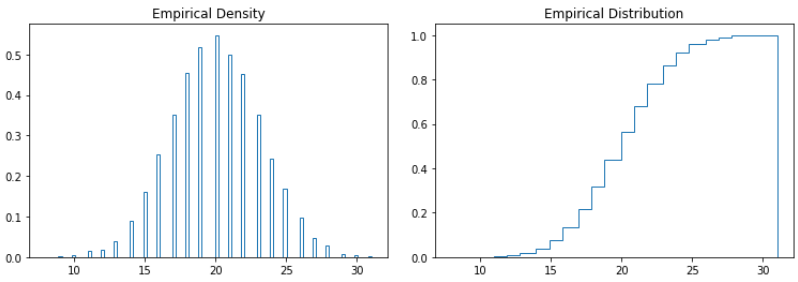


Figure 4: Binomial Distribution

3.4 Geometric Random Variable

A Random Variable X, is said to follow a *Geometric distribution*, with parameter p, if its the number of independent Bernoulli trials required for first success, each with parameter p, and follows the Probability law:

$$\mathbb{P}(X = k) = p(1-p)^{k-1}, \quad , \forall k \in \mathbb{N}. \tag{5}$$

Theorem. *Let $B_i \sim Bernoulli(p)$, $\forall i \in \{1, 2, ..., n\}$. and X is the number of Bernoulli trials required to obtain first success. Then $X \sim Geometric(p)$.*

Proof. From a combinatorial argument, $\mathbb{P}(X = k)$ is the product of $(k-1)$ failure probabilities and one success probability.

$$\begin{aligned} \implies \mathbb{P}(X = k) &= (\mathbb{P}(B_i = 1)) \prod_{i=1}^{k-1} \mathbb{P}(B_i = 0) \\ &= p \prod_{i=1}^{k-1} (1-p) \\ &= p(1-p)^{k-1} \end{aligned}$$

Hence, $X \sim$ Geometric(p). □

Following is the Python implementation for this transformation:

```
def geometric(p):
        U = 1
        X = 0
        while U > p:                    # True, if Bernoulli trial
    fails
            U = next(RNG())
            X += 1
        return X
```

Listing 7: Geometric Random Variable

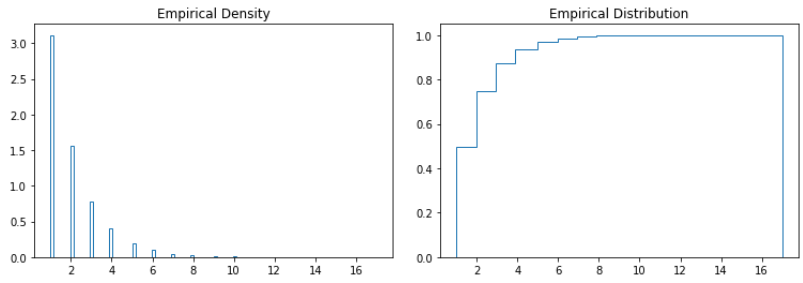


Figure 5: Geometric Distribution

3.5 Negative Binomial Random Variable

A Random Variable X, is said to follow a *Negative Binomial distribution*, with parameters r and p, which gives the total number of successes out of independent Bernoulli trials, each with parameter probability, p, before getting r failures, and follows the Probability law:

$$\mathbb{P}(X = k) = \binom{r+k-1}{k} p^r (1-p)^k, \quad k \in \mathbb{Z}_+. \tag{6}$$

Theorem. *Let* $G_i \sim Geometric(p)$, $\forall i \in \{1, 2, ..., r\}$. *and let* $X = \sum_{k=1}^{r} G_k$. *Then* $X \sim NB(r, p)$.

Proof. From a combinatorial argument, $\mathbb{P}(X = k)$ is the product of k failure probabilities and r success probabilities; whole thing multiplied by the number of ways of choosing k failures out of $(r + k - 1)$ trials.

$$\begin{aligned}
\implies \mathbb{P}(X = k) &= \binom{r+k-1}{k} \prod_{i=1}^{r} \mathbb{P}(G_i = 1) \prod_{i=1}^{k} \mathbb{P}(G_i = 0) \\
&= \binom{r+k-1}{k} \prod_{i=1}^{r} p \prod_{i=1}^{k} (1-p) \\
&= \binom{r+k-1}{k} p^r (1-p)^k
\end{aligned}$$

Hence, $X \sim \text{NB}(r, p)$. □

Following is the Python implementation for this transformation:

```
def negativeBinomial(r, p):
        X = 0
        failures = 0
        while failures < r:
            U = next(RNG())
            if U <= p:                    # Success Case
                X += 1
            else:                         # Failure Case
                failures += 1
        return X
```

Listing 8: Negative Binomial Random Variable

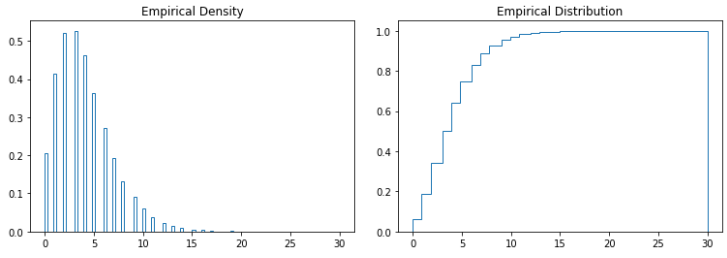


Figure 6: Negative Binomial Distribution

3.6 Poisson Random Variable

A Random Variable X, is said to follow a *Poisson distribution*, with *mean* λ, which gives the number of times an event occurs in an interval of time or space, and follows the Probability law:

$$\mathbb{P}(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k \in \mathbb{Z}_+. \tag{7}$$

Theorem. *Let* $X \sim Poisson(\lambda)$. *Then* $\mathbb{P}(X = i+1) = \dfrac{\lambda}{i+1}\mathbb{P}(X = i)$.

Proof. Let $X \sim \text{Poisson}(\lambda)$. Then X follows the probability law given by (7).

$$\begin{aligned} \implies \forall i \in \mathbb{Z}_+ : \frac{\mathbb{P}(X = i+1)}{\mathbb{P}(X = i)} &= \frac{\dfrac{\lambda^{i+1}e^{-\lambda}}{(i+1)!}}{\dfrac{\lambda^i e^{-\lambda}}{i!}} \\ &= \lambda \frac{i!}{(i+1)!} \\ &= \frac{\lambda}{i+1} \end{aligned}$$

□

Hence, applying the inverse transform algorithm with this property gives the following Python implementation:

```
def poisson(mean):
        U = next(RNG())
        i = 0
        p = exp(-mean)
        F = p
        while F <= U:
            #                           mean * P(X=i)
            # Property:  P(X=i+1) =   -------------
            #                             (i+1)
            p = mean*p/(i+1)
            F += p
            i += 1
        X = i
        return X
```

Listing 9: Poisson Random Variable

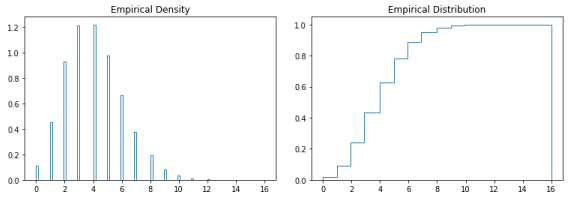


Figure 7: Poisson Distribution

3.7 Exponential and Gamma Random Variables

A Random Variable X, is said to follow an *Exponential distribution*, with *mean* λ, which gives the time between two events in a process, where events occur continuously and independently at constant rate, and follows the Probability law:

$$F(X;\lambda) = \begin{cases} 1 - e^{-\lambda x} & \text{if } X \geq 0 \\ 0 & \text{if } X < 0 \end{cases}$$

A Random Variable X, is said to follow a *Gamma distribution*, if its a sum of n independent exponential random variables. Mathematically:

$$\text{Gamma}(n, \lambda) \sim \sum_{i=1}^{n} \text{Exponential}(\lambda) \tag{8}$$

Hence, let us first derive a transformation from Uniform$(0, 1)$ to Exponential(p). Then, For Gamma random variable, we can take the sum of n exponentials.

Theorem. *Let* $U \sim$ *Uniform*$(0, 1)$*, and* $X = \dfrac{-\log U}{\lambda}$*. Then* $X \sim$ *Exponential*(λ)*.*

Proof. Let us apply the Inverse transform Algorithm, to the distribution function of an Exponential random variable. From the theorem, we know:

$$F^{-1}(U) \sim F,$$

where $F(x) = 1 - e^{-\lambda x}$. Let $x = F^{-1}(u)$.

$$\begin{aligned} &\implies u = F(x) = 1 - e^{-\lambda x} \\ &\implies 1 - u = e^{-\lambda x} \\ &\implies x = \frac{-\log(1-u)}{\lambda} \\ &\implies \frac{-\log(1-U)}{\lambda} \sim \text{Exponential}(\lambda) \end{aligned}$$

Now, notice that if $U \sim$ Uniform$(0, 1)$, then $(1 - U) \sim$ Uniform$(0, 1)$.

$$\implies X = \frac{-\log(U)}{\lambda} \sim \text{Exponential}(\lambda)$$

Hence, we have shown that the prescribed transformation, from U to X, produces an exponential random variable. □

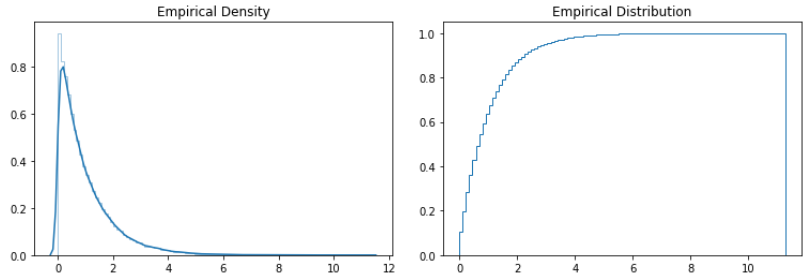
Remark. *Let* $U_1, U_2, ..., U_n \sim$ *Uniform*$(0, 1)$ *and let* $E_i = \dfrac{-\log(U_i)}{\lambda}$*. Then, as we know that sum of exponentials is gamma-distributed,*

$$\sum_{i=1}^{n} E_i = -\sum_{i=1}^{n} \frac{\log(U_i)}{\lambda} = -\frac{\log(\prod_{i=1}^{n} U_i)}{\lambda}.$$

Following are the Python implementations for both the random variables:

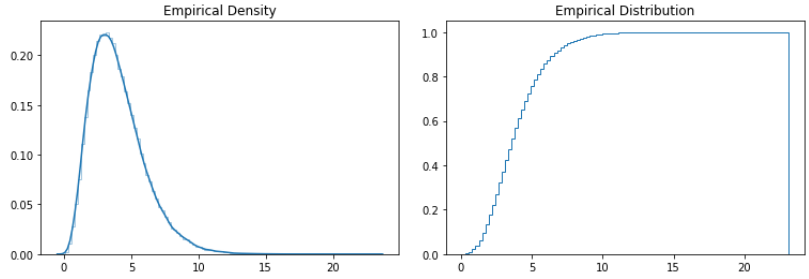
```
def exponential(rate):
        U = next(RNG())
        X = -log(U)/rate
        return X
```

Listing 10: Exponential Random Variable




```
def gamma(n, rate):
        Y = 1
        for i in range(n):
            U = next(RNG())
            Y *= U
        # Property: log(U1)+log(U2)+... = log(U1*U2...) = log(Y)
        X = -log(Y)/rate
        return X
```

Listing 11: Gamma Random Variable



3.8 Normal (Gaussian) Random Variable

A Normal Random Variable $X \sim \mathcal{N}(\mu, \sigma^2)$, is a random variable that follows the *Gaussian distribution*:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Since the integral of e^{-x^2} has no closed form, there's no way to apply the Inverse-transform Algorithm directly for the Gaussian Random Variable. Hence, we employ an alternative method (proposed by George Box and Mervin Muller).

This transformation produces a Bi-variate standard normal random variable, from where we can get any general normal random variable, applying the series of transformations:

$$U_1, U_2 \sim \text{Uniform}(0,1) \mapsto X, Y \sim \mathcal{N}(0,1) \mapsto \mu X + \sigma \sim \mathcal{N}(\mu, \sigma^2)$$

Theorem (Box-Muller Transformation)**.** *Let* $U_1, U_2 \sim$ *Uniform*$(0,1)$*. Then,* $X, Y \sim \mathcal{N}(0,1)$*, where,* X *and* Y *are given by:*

$$X = \sqrt{-2\log(U_1)}\cos(2\pi U_2)$$
$$Y = \sqrt{-2\log(U_1)}\sin(2\pi U_2)$$

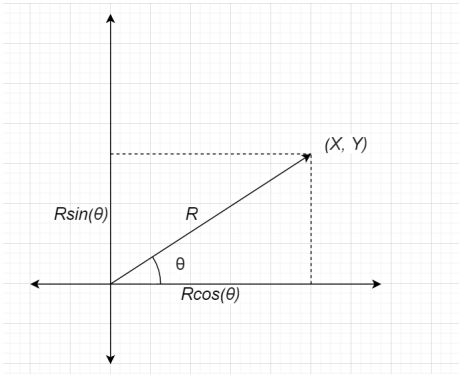


Figure 8: Box-Muller Polar Coordinates

Proof. Let $X, Y \sim \mathcal{N}(0,1)$ be independent ($X \perp\!\!\!\perp Y$). Representing them as polar coordinates, as shown in the Figure 2, we get:

$$R = \sqrt{X^2 + Y^2} \in \mathbb{R}_+$$
$$\tan\theta = \frac{Y}{X} \in \mathbb{R}.$$

$$\begin{aligned} \because X \perp\!\!\!\perp Y \implies f(x,y) &= f(x)f(y) \\ &= \frac{1}{\sqrt{2\pi}} e^{\frac{-X^2}{2}} \frac{1}{\sqrt{2\pi}} e^{\frac{-Y^2}{2}} \\ &= \frac{1}{2\pi} e^{\frac{-(X^2+Y^2)}{2}} \end{aligned}$$

Substituting $D := R^2$, $\theta := \tan^{-1}(\frac{Y}{X})$, we get:

$$\implies f(D,\theta) = \frac{c}{2\pi}e^{-D/2}, \text{ such that } \int_0^{2\pi}\int_0^{\infty} f(D,\theta)dDd\theta = 1$$

$$\implies \frac{c}{2\pi}\int_0^{2\pi}\int_0^{\infty} e^{-D/2}dDd\theta = 1$$

$$\implies \frac{c}{2\pi}\int_0^{2\pi} d\theta \int_0^{\infty} e^{-D/2}dD = 1 \qquad (\because D \perp\!\!\!\perp \theta)$$

$$\implies \frac{c}{2\pi}(2\pi)\left(\frac{e^{-D/2}}{-1/2}\right)\Bigg|_0^{\infty} = 1$$

$$\implies c = (-2e^{-\infty} + 2e^{0})^{-1} = 1/2$$

$$\therefore f(d,\theta) = \frac{1}{4\pi}e^{-D/2} = \left(\frac{1}{2\pi}\right)\left(\frac{e^{-D/2}}{2}\right)$$

However, this is a product of an exponential density (mean = 2), and a uniform density (on $]0, 2\pi[$).

$$\implies R^2 \perp\!\!\!\perp \theta, R^2 \sim \text{Exponential}(2), \theta \sim \text{Uniform}(0, 2\pi)$$

$$\therefore X = R\cos\theta = \sqrt{-2\log(U_1)}\cos(2\pi U_2),$$

$$Y = R\sin\theta = \sqrt{-2\log(U_1)}\sin(2\pi U_2)$$

Hence, we have proved the Box-Muller Transformation. □

Following is the Python implementation for the Box-Muller Transformation, and plots for uni-variate, as well as, bi-variate normal random variables:

```
def normal(m = 0, s = 1):
    """
    Method for obtaining a Normal/Gaussian random variable, with
    paramters: mean (mu) = 'm', standard deviation (sigma) = 's',
    which follows the Normal/Gaussian distribution.
    """
        # Box-Muller Transformation:
        #(Sheldon Ross, "Simulation-4th-Edition")
        U1 = next(RNG())
        U2 = next(RNG())
        X = ((-2*log(U1))**0.5)*cos(2*PI*U2)
        # Y = ((-2*log(U1))**0.5)*sin(2*PI*U2)
        return m + s*X
        # return m + s*X, m + s*Y
```

Listing 12: Box-Muller (Normal Random Variables)

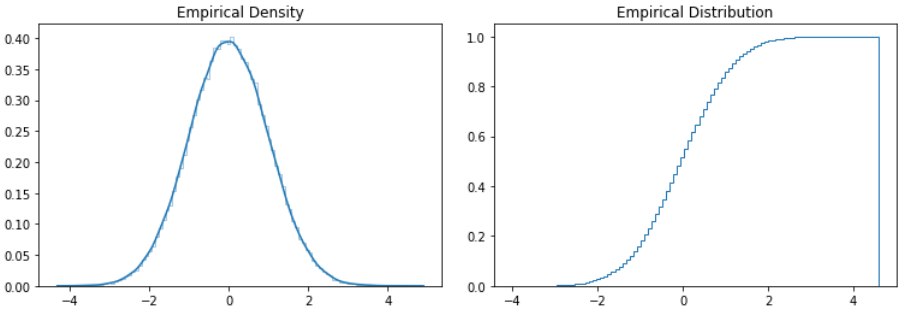
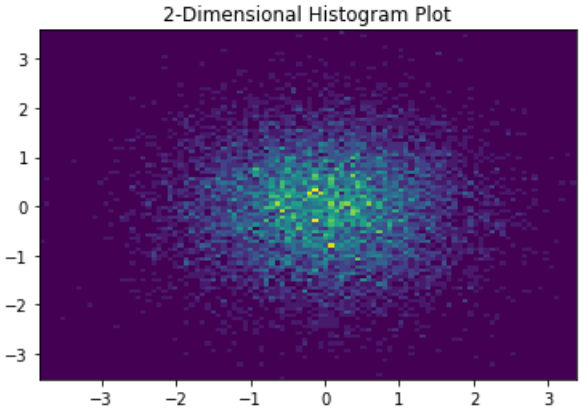


Figure 9: Uni-variate Normal Random Variable



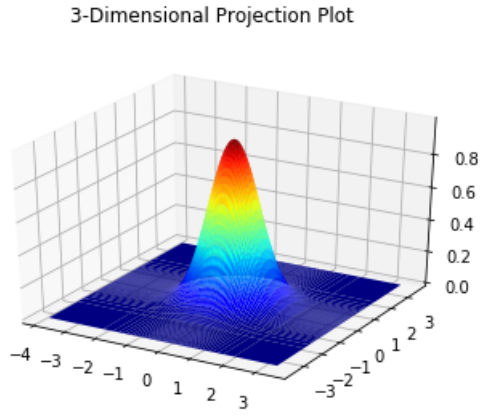


Figure 10: Bi-variate Normal Random Variable

3.9 Log-normal Random Variable

A Random Variable X, is said to follow a *Log-normal distribution*, with the parameters μ, σ, if its *logarithm* if normally distributed:

$$\log X \sim \mathcal{N}(\mu, \sigma^2) \tag{9}$$

Hence, if $X \sim \mathcal{N}(\mu, \sigma^2)$, then $e^X \sim$ Lognormal (μ, σ^2).

Following is the Python implementation for this transformation, and plot for the same:

```
def logNormal(m = 0, s = 1):
        Y = normal(m, s)
        X = exp(Y)
        return X
```

Listing 13: Log-normal Random Variable

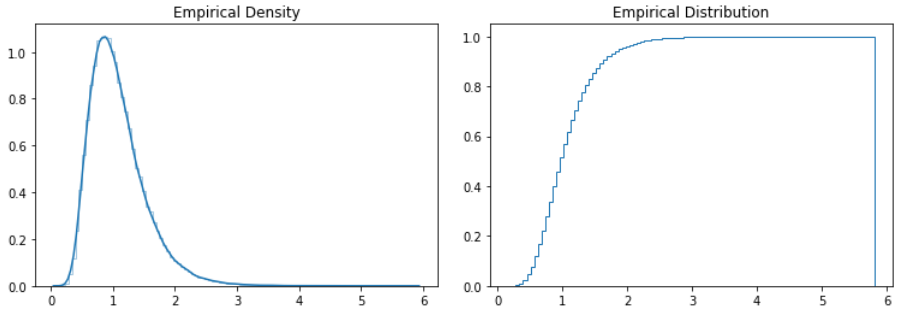


Figure 11: Log-normal Random Variable

3.10 Uniform Random Variable Plot

Following is the distribution plot for the Uniform Random Variable from the Linear Congruential Generator. Larger, the set of random variables we use, more accurate empirical plots will be obtained.

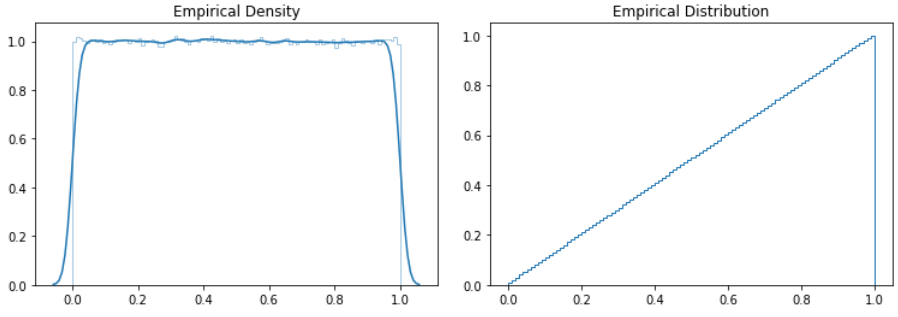


Figure 12: Uniform Random Variable

4 Limitations

Although this module shall work for most of the practical cases, especially for Statistical Analysis, there are certain limitations to using it. Following are some of specific concerns, regarding the usage:

1. Even though the Linear Congruential Generator is one of the fastest Pseudo-random number generators and requires minimal memory, it has a drawback of having a small period (modulus m), costing us with precision. Even if we increase the period to a larger number, this problem will persist, making computations even costlier.

2. Another flaw of using an LCG is that, due to a serial correlation between successive values of X_n, the points can only lie at a certain maximum number of hyperplanes, thereby resulting a small finite number of possible values.

3. Our random number generator is not *cryptographically-secure*, thereby rendering it completely useless for Security-purposes. This arises, due to the deterministic nature of pseudorandom number generators.

4. Some of the distributions (for e.g., the Geometric distribution) were transformed directly, instead of applying the Inverse-Transform Algorithm to them. This might result in slower time complexities.

5. The 3-dimensional projection plot, currently only plots for the Bi-variate normal random variables. For applying the same to other distributions, the code can be expanded, provided, individual *numpy* calculations and transformations are also added in the code. However the histogram-2D plot works well for any bi-variate random variable, from any arbitrary distribution.

5 Conclusions

The purpose of this project was to find a way of simulating any random variable with a specified probability law. We were able to generate a Standard Uniform Variable, with high precision (decimal places). Thereby, we were able to transform this random variable into another randomly-distributed variable, using a suitable transformation. Although not cryptographically-secure, these work extremely well for statistical purposes. We were also also successful in verifying the plots of theoretic cumulative distributions and densities, against the empirical distributions and densities obtained programmatically.