genDistribution()

A Random Variate Generation Library

Group 304: Teresa Dong & Henry Dong

Abstract

The goal of this project is to build a library of random variate generation routines using the Python programming language. Our innovative Python library contains routines for generating random variates from the most commonly encountered discrete and continuous distributions: Bern(p), Geom(p), Exp(λ), Normal(μ,σ2), Gamma(α,β), Weibull(α,β). The distributions generated by our library are benchmarked against those generated by the popular SciPy Python library via the Kolmogorov–Smirnov test.

Background

Commonly-used simulation techniques are often built on random variates that follow specific distributions. Thus, understanding how to generate these random variates is often the first step to building a simulation (Ross 2013). One of the simplest ways to do this is via the inverse-transform method (Sigman 2010). The versatile inverse-transform method can be used to generate both continuous and discrete random variables. Indeed, this simple and effective method serves as the basis of the popular Python library SciPy' Stats module's *rv_continous* and *rv_discrete* classes random variable classes, which utilizes the inverse-transform method to help users quickly generate most commonly-used distributions (The SciPy Community 2021).

Source Code Review

The source can be found in the attached **generateDistribution.py**. The library takes the form of a distGenerator class object. This class includes various functions to generate random variates from different distributions and generalized utility functions to plot and validate the generated variates against SciPy reference distributions.

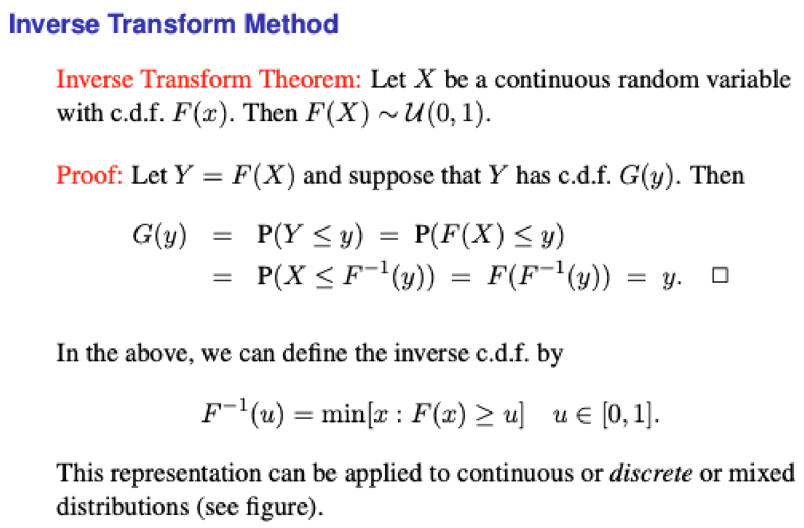
Generalized Utility Functions

Function	**Description**
plotDist()	Plot histogram of the generated random numbers
compPlotDist()	Side by Side Comparison of Distribution Histograms, usually the generated Random Variate (from the genDistribution.py Library) and the corresponding SciPy function
validateDist()	Validates if the generated distribution matches that of the SciPy reference distribution by conducting a Kolmogorov–Smirnov test (K-S) Test, usually the generated Random Variate (from the genDistribution.py Library) and the corresponding SciPy function generated Random Variate

Random Variate Generation Functions for Various Probability Distributions

Function	**Description**
generateExponential()	Generates size number of Exponential Random Variables based on Lambda parameter and random state seed **Function with Parameters:** *generateExponential(Lambda, size, random_state)*
generateWeibull()	Generates size number of Weibull Random Variables based on Alpha and Beta parameter and random state seed **Function with Parameters:** *generateWeibull(Alpha, Beta, size, random_state)*
generateBernoulli()	Generates size number of Bernoulli Random Variables based on p parameter and random state seed **Function with Parameters:** *generateBernoulli(p, size, random_state)*
generateGeometric()	Generates size number of Geometric Random Variables based on p parameter and random state seed **Function with Parameters:** *generateGeometric(p, size, random_state)*
generateNormal()	Generates size number of Normal Random Variables based on Mu, Sigma parameters and random state seeds using the Box-Mueller method **Function with Parameters:** *generateNormal(Mu, Sigma, size, random_state1, random_state2)*
generateGamma()	Generates size number of Gamma Random Variables based on Alpha and Beta parameter and random state seed **Function with Parameters:** *generateGamma(Alpha, Beta, size, random_state)*

The random variate generation functions are generated using the inverse transformation method.



Inverse Transform Method

Inverse Transform Theorem: Let X be a continuous random variable with c.d.f. $F(x)$. Then $F(X) \sim \mathcal{U}(0,1)$.

Proof: Let $Y = F(X)$ and suppose that Y has c.d.f. $G(y)$. Then

$$\begin{aligned} G(y) &= \mathrm{P}(Y \le y) = \mathrm{P}(F(X) \le y) \\ &= \mathrm{P}(X \le F^{-1}(y)) = F(F^{-1}(y)) = y. \quad \square \end{aligned}$$

In the above, we can define the inverse c.d.f. by

$$F^{-1}(u) = \min[x : F(x) \ge u] \quad u \in [0,1].$$

This representation can be applied to continuous or *discrete* or mixed distributions (see figure).

First, the respective function generates uniform variates of the user-specified size. Then, the inverse transformation method is applied to obtain the variates for the corresponding distribution.

Implementations and formulas used are shown below:

Exponential(λ)

```
def generateExponential(self, Lambda, size, random_state):
    """Generates size number of Exponential Random Variables based on Lambda parameter and random seed seed

    Args:
        Lambda (float): Rate parameter (lambda) for exponential distribution
        size (int): Number of random variates to generate
        random_state (int): Random seed for the exponential variable

    Returns:
        list<float>: List of Random Variates from the Exponential Distribution
    """

    U = stats.uniform.rvs(size=size,random_state=random_state)

    R = -(1/Lambda) * (np.log(1 - U))

    return R
```

Breaking the block down, we see that first, uniform random variates are generated using SciPy.

```
U = stats.uniform.rvs(size=size,random_state=random_state)
```

Then, the inverse transformation formula for Exponential distribution is applied to the uniform variates U to create the random variable R:

$$R = \frac{ln\,(1-U)}{-\lambda}$$

```
R = -(1/Lambda) * (np.log(1 - U))
```

Weibull(α, β)

```
def generateWeibull(self, Alpha, Beta, size, random_state):
    """Generates size number of Weibull Random Variables based on Alpha and Beta parameter and random state seed

    Args:
        Alpha (float): exponentiation parameter, where alpha=1 is the non-exponentiated Weibull distribution
        Beta (float): shape parameter for the non-exponentiated Weibull law
        size (int): Number of random variates to generate
        random_state (int): Random seed for the weibull variable

    Returns:
        list<float>: List of Random Variates from the Weibull(Alpha, Beta) Distribution
    """

    U = stats.uniform.rvs(size=size,random_state=random_state)

    R = pow((-1 * np.log (1 - U)),(1/Beta))/Alpha

    return R
```

Similarly to above, we first generate uniform random variates using SciPy. Then, we apply the inverse transformation formula for Weibull distribution to the uniform variates U to create the random variable R:

$$R = \frac{-ln(1-U)^{\,1/\beta}}{\alpha}$$

Bernoulli(p)

```
def generateBernoulli(self, p, size, random_state):
    """Generates size number of Bernoulli Random Variables based on p parameter and random state seed

    Args:
        p (float): p is probability of single success, 1-p is probability of single failure
        size (int): Number of random variates to generate
        random_state (int): Random seed for the bernoulli variable

    Returns:
        list<float>: List of Random Variates from the Bernoulli(p) Distribution
    """

    U = stats.uniform.rvs(size=size,random_state=random_state)

    B = (U<=p).astype(int)

    return B
```

Similar to above, uniform random variates are generated first. However, since the Bernoulli distribution is a discrete random variable distribution, we generate the random variates B using the logic:

B=1 when U <= p and B=0 when U > P

Geometric

```
def generateGeometric(self, p, size, random_state):
    """Generates size number of Geometric Random Variables based on p parameter and random state seed

    Args:
        p (float): p is probability of single success, 1-p is probability of single failure
        size (int): Number of random variates to generate
        random_state (int): Random seed for the geometric variable

    Returns:
        list<float>: List of Random Variates from the Geometric(p) Distribution
    """

    U = stats.uniform.rvs(size=size,random_state=random_state)

    R = np.log(1-U)/np.log(1-p)

    R_discrete = []
    for r in R:
        R_discrete.append(math.ceil(r))

    return R_discrete
```

This is another example of a discrete random variable. However, here we use a ceiling function to convert the uniform random variates to discrete geometric random variates with the formula:

$$R_{Discrete} = CEIL\left[\frac{ln(1-U)}{ln(1-p)}\right]$$

Normal

```
def generateNormal(self, Mu, Sigma, size, random_state1, random_state2):
    """Generates size number of Normal Random Variables based on Mu, Sigma parameters and random state seeds using the Box-Mueller method

    Args:
        Mu (float): mean of the normal distribution
        Sigma (float): standard deviation of the normal distribution
        size (int): Number of random variates to generate
        random_state1 (int): Random seed for the normal variable
        random_state2 (int): Random seed for the normal variable

    Returns:
        list<float>: List of Random Variates from the Normal(Mu, Sigma) Distribution
    """

    U1 = stats.uniform.rvs(size=size,random_state=random_state1)
    U2 = stats.uniform.rvs(size=size,random_state=random_state2)
    # Standard Normal pair
    Z0 = np.sqrt(-2*np.log(U1))*np.cos(2*np.pi*U2)
    Z1 = np.sqrt(-2*np.log(U1))*np.sin(2*np.pi*U2)

    # Scaling
    Z2 = Z0*Sigma+Mu

    return Z2
```

First, we generate two sets of uniform random variates of specified size and seed

```
U1 = stats.uniform.rvs(size=size,random_state=random_state1)
U2 = stats.uniform.rvs(size=size,random_state=random_state2)
```

Then, we create standard normal random variates using the Box-Muller method which states that:

If U_1, U_2 are iid $(0,1)$, then the following quantities are iid Nor(0,1):

$$Z_1 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$$
$$Z_2 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2)$$

In Python, this method is coded as follows:

```
# Standard Normal pair
Z0 = np.sqrt(-2*np.log(U1))*np.cos(2*np.pi*U2)
Z1 = np.sqrt(-2*np.log(U1))*np.sin(2*np.pi*U2)
```

Afterwards, we scale up to a normal distribution with mean, μ, and variance, $\boldsymbol{\sigma}^2$, using the transformation:

Now, suppose we have $Z \sim$ Nor$(0,1)$, and we want $X \sim$ Nor(μ, σ^2). We can apply the following transformation:

$$X \leftarrow \mu + \sigma Z$$

```
# Scaling
Z2 = Z0*Sigma+Mu
```

Gamma

```
def generateGamma(self, Alpha, Beta, size, random_state):
    """Generates size number of Gamma Random Variables based on Alpha and Beta parameter and random state seed

    Args:
        Alpha (float): shape parameter for alpha, when its an integer, gamma becomes an erlang distribution, when its 1 gamma becomes the exponential
distribution
        Beta (float): parameter for shifting the distribution
        size (int): Number of random variates to generate
        random_state (int): Random seed for the gamma variable

    Returns:
        list<float>: List of Random Variates from the Gamma(Alpha, Beta) Distribution
    """


    G = np.zeros(size)

    for i in range(1,size):
        U = stats.uniform.rvs(size=Alpha, random_state=random_state+i)

        # Based on Formula: https://arxiv.org/pdf/1304.3800.pdf
        G[i] = -1/Beta*np.log(self.multiplyList(U))


    return G
```

The implementation of the Gamma distribution is slightly more complicated (Martino, L., & Luengo, D. 2013). Each random variable G, is created by using the product of Alpha uniform variates along with a Beta factor according to the formula:

$$X = -\frac{1}{\beta}\sum_{i=1}^{\alpha}\ln\left(U_i\right) = -\frac{1}{\beta}\ln\left(\prod_{i=1}^{\alpha}U_i\right),$$

Implemented in Python, we need a method to take the product of an array of uniform variates, $\prod$, whichwe implemented in the function, multiplyList():

```
def multiplyList(self,myList) :
    # Multiply elements one by one
    result = 1
    for x in myList:
        result = result * x
    return result
```

User Guide with Examples

1. Initializing the Library

To use the source code, first we need to import our Python library using the code block below:

```
# Import the generateDistribution Library
import generateDistribution as genDist
```

Once the library is imported, we need to create an object of the distGenerator class:

```
# Create an object of the distGenerator class
dist = genDist.distGenerator()
```

Once the object is created, we can now use the various variable generation and utility functions outlined above. Usage of all the functions can be seen in the attached demo.ipynb.

2. Generating a Random Distribution

For the purposes of this report, I will show how to generate and validate a variable from the exponential distribution below:

For our example, we will be generating 1000 random variates for the exponential distribution with lambda = 2.

The first step is to define parameters by creating the variates to be passed into the library and SciPy reference functions. We want to define them separately to make sure both our custom library and SciPy are using the same parameters so that when we validate later, we can confirm that we are comparing apple to apples.

2.1.1. Defining Parameters

In this example, the parameters to be defined are:

- **Lambda (defined above)**: Rate parameter
- **Dist_Name**: String parameter to use as the heading for the plots, usually takes the form of Distribution(Parameter=Parameter Value), where Parameter Value is passed in using string substitution
- **Size**: Number of random variates to generate
- **Seed**: Random seed that is used to make sure all the generators are using the same seed and therefore is consistent

```
# Distribution specific Parameter
Lambda = 2
Dist_Name = f'Exponential(Lambda={Lambda})'
# Generated Parameters used by both Library and Scipy Reference functions
seed = 123
size = 1000
```

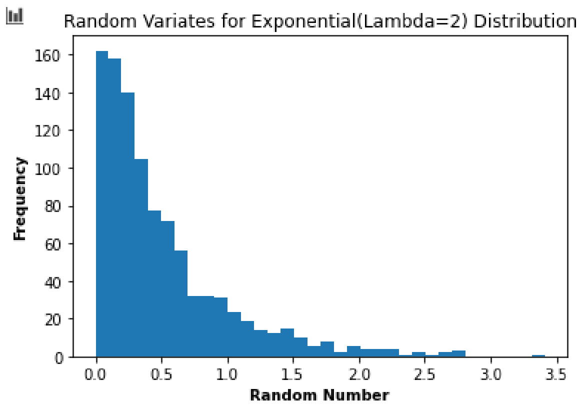
2.1.2. Plot Generated Distribution

We will first generate and plot these variates using our custom library using the code below:

```
# Generate Random Variables using the Library
exp = dist.generateExponential(Lambda=Lambda, size=size, random_state=seed)

# Plot the Generated Distribution
dist.plotDist(Dist_Name, exp)
```

This creates the following plot:



3. Validating Generated Distribution

Next, we will want to validate the random variates created using our library actually follow an Exponential distribution. To do so, we will compare our random variates generated, exp, with random variates generated using SciPy. To do this let's use SciPy to generate the reference exponential variates, exp_ref, using the same parameters:

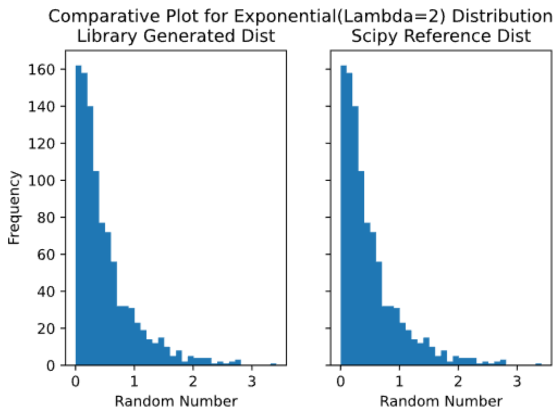
3.1. Create Reference Distribution

```
# Reference Distribution is from Scipy
exp_ref = stats.expon.rvs(scale=(1/Lambda), size=size, random_state=seed)
```

3.2. Plot Generated Distribution against Reference Distribution

Then let's plot these reference variates against our library generated variates using the library utility function compPlotDist, which takes both the library generated and SciPy reference random variates and plots them side by side:

```
# Comparative Plot of Generated vs Scipy Reference Distribution
dist.compPlotDist(Dist_Name,exp,exp_ref)
```



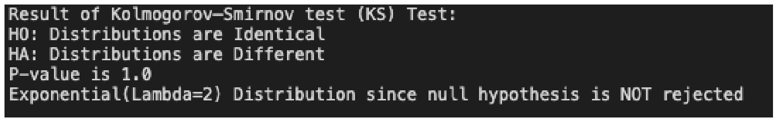
Visually, the two plots do look very similar.

3.3. Confirm Distribution with K-S Test

But let's confirm using a Kolmogorov–Smirnov test (K-S) Test, by passing in the library generated random variable, exp, and the SciPy generated random variable, exp_ref, into the validateDist() function:

```
# Validate Generated Distribution vs the Scipy Distribution
dist.validateDist(Dist_Name,exp, exp_ref,alpha=0.05)
```

The results of the test are as follows:



```
Result of Kolmogorov–Smirnov test (KS) Test:
H0: Distributions are Identical
HA: Distributions are Different
P-value is 1.0
Exponential(Lambda=2) Distribution since null hypothesis is NOT rejected
```

Since the Distributions are identical, we have confirmed that our library generated distribution IS an exponential distribution with parameter Lambda

Note: *The same analysis is re-run for all other distributions in the attached file* ***demo.ipynb***.

Findings

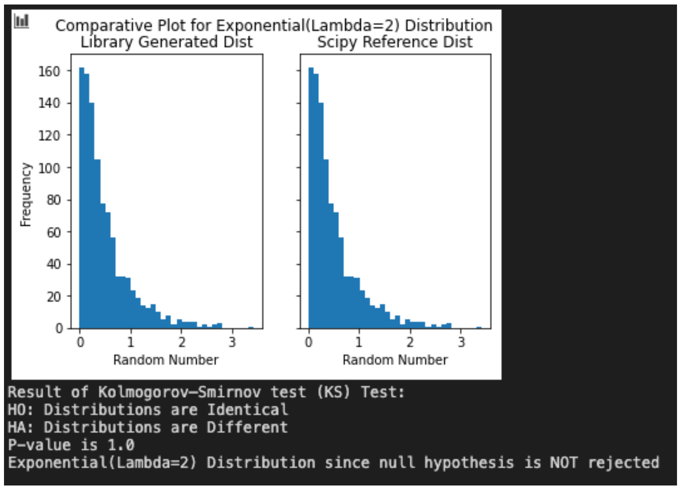
For all our generated library distributions, we have both plotted and conducted Kologomorov-Smirnov (K-S) Tests to confirm that they are the same as the SciPy reference distributions. For the visual inspection, we would like to confirm that the histograms of our library generated random variates are the same as the SciPy generated random variates using the same parameters. For the Kologomorov-Smirnov (K-S) test, we are looking for a p-value above our set threshold. Specifically, we are seeking to NOT reject the null hypothesis that the library distribution and the SciPy generated distribution are the same:

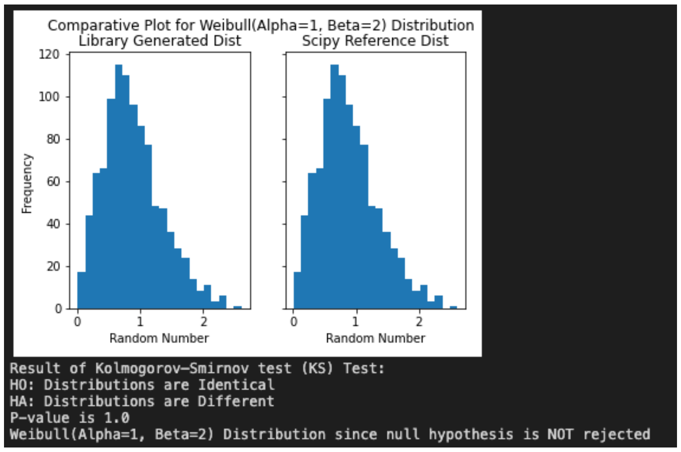
Kologomorov-Smirnov (K-S) Test Hypotheses:

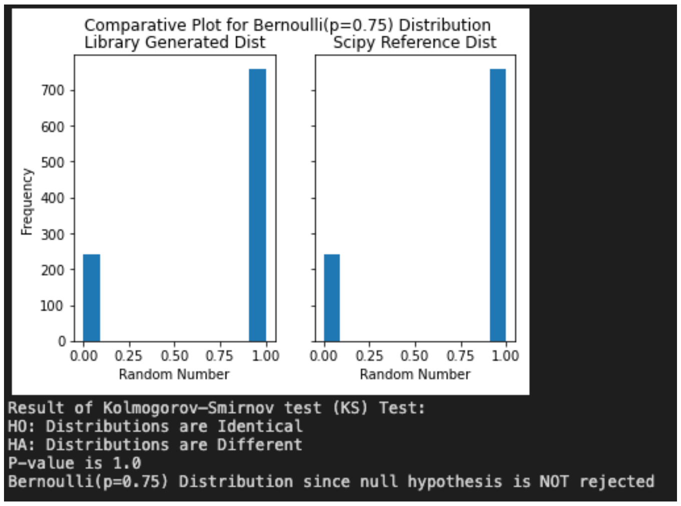
- H0: Distributions are identical
- HA: Distributions are different

The analyses and findings for each of our probability distributions are below:

Probability Distribution	**K-S Test Results**

Exponential
 	
Visually, we see that the left-hand-side chart of our library generated variates is the same as the right-hand chart of the SciPy generated random variable using the same parameters.
 	
This is confirmed by the K-S Test result. After applying the test, we see that the p-value is 1. Since the p-value is high, we cannot reject the null hypothesis that the distributions are identical. Or specifically, our library generated exponential function is identical to the SciPy generated exponential function created with the same parameters. Therefore, we can confirm that we have successfully generated an Exponential random variable.
	
	`Result of Kolmogorov-Smirnov test (KS) Test:`
	`H0: Distributions are Identical`
	`HA: Distributions are Different`
	`P-value is 1.0`
	`Exponential(Lambda=2) Distribution since null hypothesis is NOT rejected`

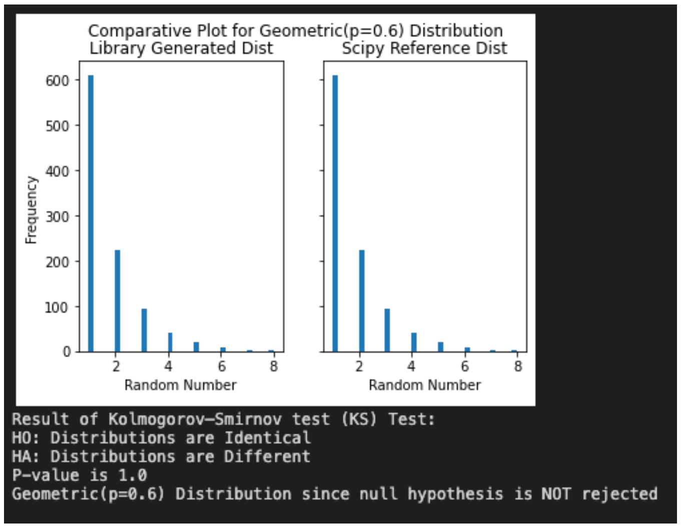
Weibull
 	
The Weibull analysis is similar to the exponential analysis. A visual inspection shows that the Weibull generated random variable is similar to the distribution generated using the SciPy reference function.
 	
We further confirm this by conducting a K-S Test where we see that the p-value is 1. Therefore, we cannot reject the null hypothesis that the distributions are identical and have confirmed that we successfully generated a Weibull random variable.
	
	`Result of Kolmogorov-Smirnov test (KS) Test:`
	`H0: Distributions are Identical`
	`HA: Distributions are Different`
	`P-value is 1.0`
	`Weibull(Alpha=1, Beta=2) Distribution since null hypothesis is NOT rejected`

Bernoulli
 	
A visual inspection shows that the Bernoulli generated random variable is similar to the distribution generated using the SciPy reference function.
 	
We further confirm this by conducting a K-S Test where we see that the p-value is 1. Therefore, we cannot reject the null hypothesis that the distributions are identical and have confirmed that we successfully generated a Bernoulli random variable.
	
	`Result of Kolmogorov-Smirnov test (KS) Test:`
	`H0: Distributions are Identical`
	`HA: Distributions are Different`
	`P-value is 1.0`
	`Bernoulli(p=0.75) Distribution since null hypothesis is NOT rejected`

Geometric

A visual inspection shows that the Geometric generated random variable is similar to the distribution generated using the SciPy reference function.

We further confirm this by conducting a K-S Test where we see that the p-value is 1. Therefore, we cannot reject the null hypothesis that the distributions are identical and have confirmed that we successfully generated a Geometric random variable.

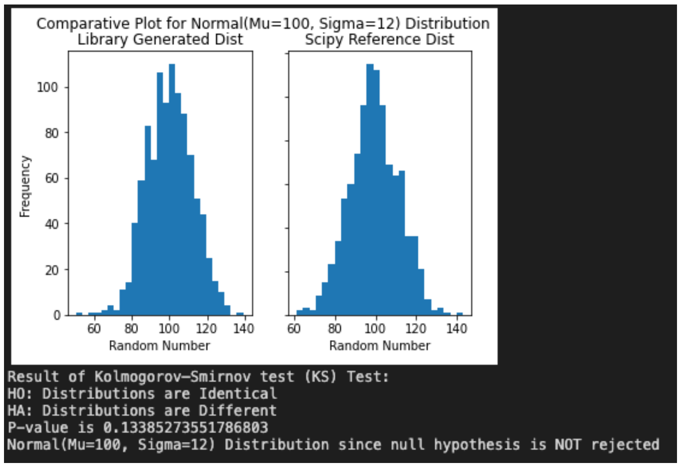


```
Result of Kolmogorov–Smirnov test (KS) Test:
H0: Distributions are Identical
HA: Distributions are Different
P-value is 1.0
Geometric(p=0.6) Distribution since null hypothesis is NOT rejected
```

Normal

A visual inspection shows that the distributions are slightly different. This is likely because the Box-Muller method is a normal APPROXIMATION method and we have multiple seeds generating the uniforms underlying the approximation, while the SciPy reference function only has one random seed. The multiple seeds were necessary for the library function because we didn't want to use the same two uniform variates in the Box-Muller equation.

Regardless, the K-S test has a high enough p-value at 0.133 which is above our alpha threshold of 0.05, so we can still say that our library generated distribution is normal.

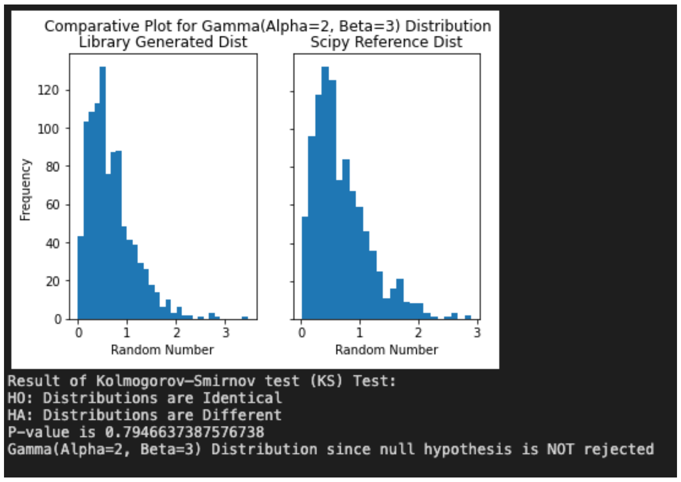


```
Result of Kolmogorov–Smirnov test (KS) Test:
H0: Distributions are Identical
HA: Distributions are Different
P-value is 0.13385273551786803
Normal(Mu=100, Sigma=12) Distribution since null hypothesis is NOT rejected
```

Gamma

A visual inspection shows that the distributions are slightly different. This is likely because we have multiple seeds generating the uniforms underlying the our library approximation, while the SciPy reference function only has one random seed. The multiple seeds were necessary for the library function because we needed a different set of uniforms to multiply for each Gamma variable.

Regardless, the K-S test has a high enough p-value at 0.794 which is above our alpha threshold of 0.05, so we can still say that our library generated distribution is Gamma.



```
Result of Kolmogorov–Smirnov test (KS) Test:
H0: Distributions are Identical
HA: Distributions are Different
P-value is 0.7946637387576738
Gamma(Alpha=2, Beta=3) Distribution since null hypothesis is NOT rejected
```

References

1. Ross, S. (2013). *Simulation* (Fifth ed.). Academic Press. Retrieved May 03, 2021, from https://www.sciencedirect.com/book/9780124158252/simulation.
2. Sigman, K. (2010). Inverse Transform Method. Retrieved from http://www.columbia.edu/~ks20/4404-Sigman/4404-Notes-ITM.pdf
3. The SciPy Community. Scipy.stats.rv_continuous¶. Retrieved May 03, 2021, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.html#scipy.stats.rv_continuous
4. Martino, L., & Luengo, D. (2013). *Extremely efficient generation of Gamma random variables for $a \geq 1$*(Unpublished master's thesis). Universidad Carlos III de Madrid, Av. Universidad 30, 28911 Leganes, Spain ´ ; Dep. Ing. de Circuitos y Sistemas, Univ. Politecnica de Madrid, Ctra. de Valencia km. 7, ´ 28031 Madrid, Spain. Retrieved May 3, 2021, from https://arxiv.org/pdf/1304.3800.pdf