Evan Anderson
eanderson42
July 17th, 2021

Group 118 - Mini Project 2

Random Variate Generation Library

Prompt 13:
Make me a nice library of random variate generation routines. You can use your favorite high-level language like C++, Java, Python, Matlab, or even Excel. Include in your library routines for generating random variates from all of the usual discrete and continuous distributions, e.g., Bern(p), Geom(p), Exp(λ), Normal(μ,σ2), Gamma(α,β), Weibull(α,β), etc., etc. (Just one routine per distribution is fine.) Include in your write-up an easy user's guide, complete source code, and some appropriate examples.

For this project I've created a class that creates an RVG object, which can be used to generate random numbers based on a variety of distributions

The included .zip file contains 2 .py files; `random_variate_generation.py` is the importable class, used to create an object for RVG, and `rvg_test.py` are sample tests that I have used to confirm the distributions being created are as expected, and that the chi squared GOF test is passed.

Random Variate Generation
The importable class `rvg` contained within this file is initialized with a seed value (for reproducibility). After creation of the object (samples shown in rvg_test), various random number based on given probabilities can be created. The core of this class is the random number generator "self.di()" that uses the desert island linear congruent we discussed in class. To ensure the generated values are uniform, the routine `self.gof(n)` can be run on any number of samples, splitting the results into 5 bins and conducting a 99% confident chi squared GOF test.

Using these randomly generated numbers, the class can generate random numbers sampled form a wide variety of distributions. For a given class, any time a random number is generated, the `x_i` value is incremented. This is important to keep in mind for reproducibility. The x values are stored in `self.x_list` so can be values can be referenced more easily in the future. All variate generators for a given distribution have

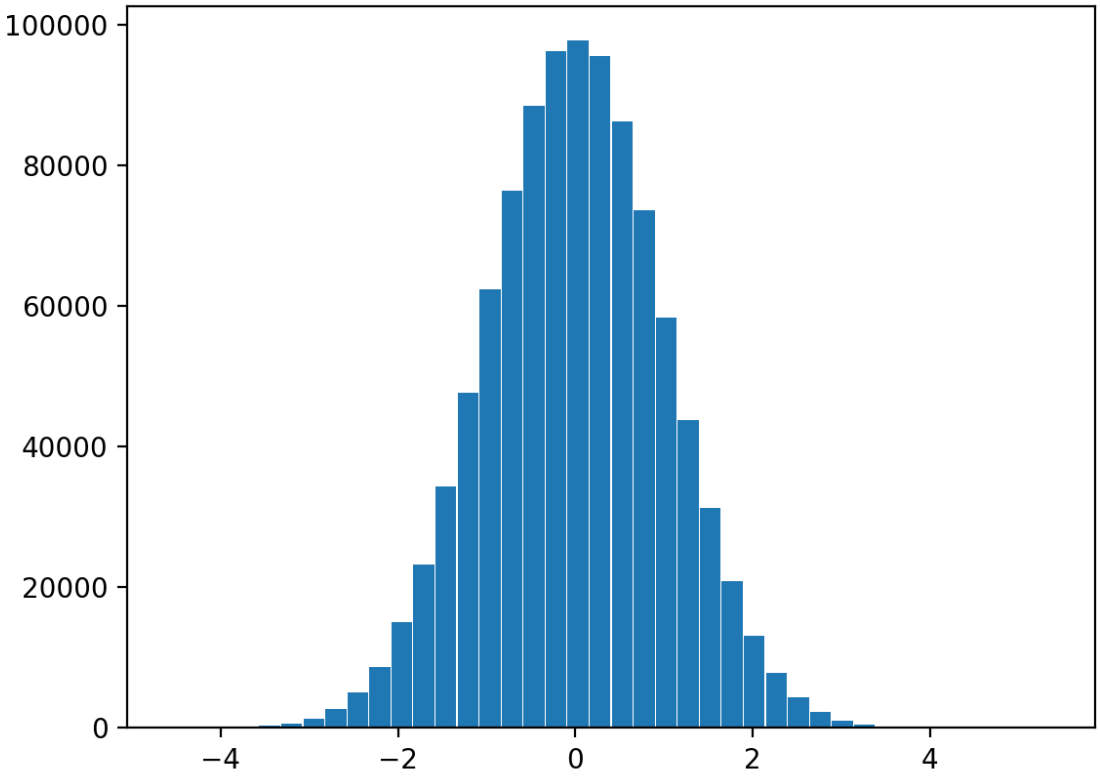
the option to be stored in the self.rn_storage dictionary. All routines can store these values by setting the optional parameter "store" to store=True.

Desert Island Generator - `self.di()`:
This is the core RNG based on the LCG described in class. This LCG is the core of the uniform generator, and thus the backbone of random variate generation for all other distributions. It does not require inputs, and increments the `x_i` value each time a random variate is generated

Uniform:
A uniform distribution with any given minimum and maximum value. The values default to the standard uniform(0,1), but any minimum and maximum can be utilized. It returns a single value.

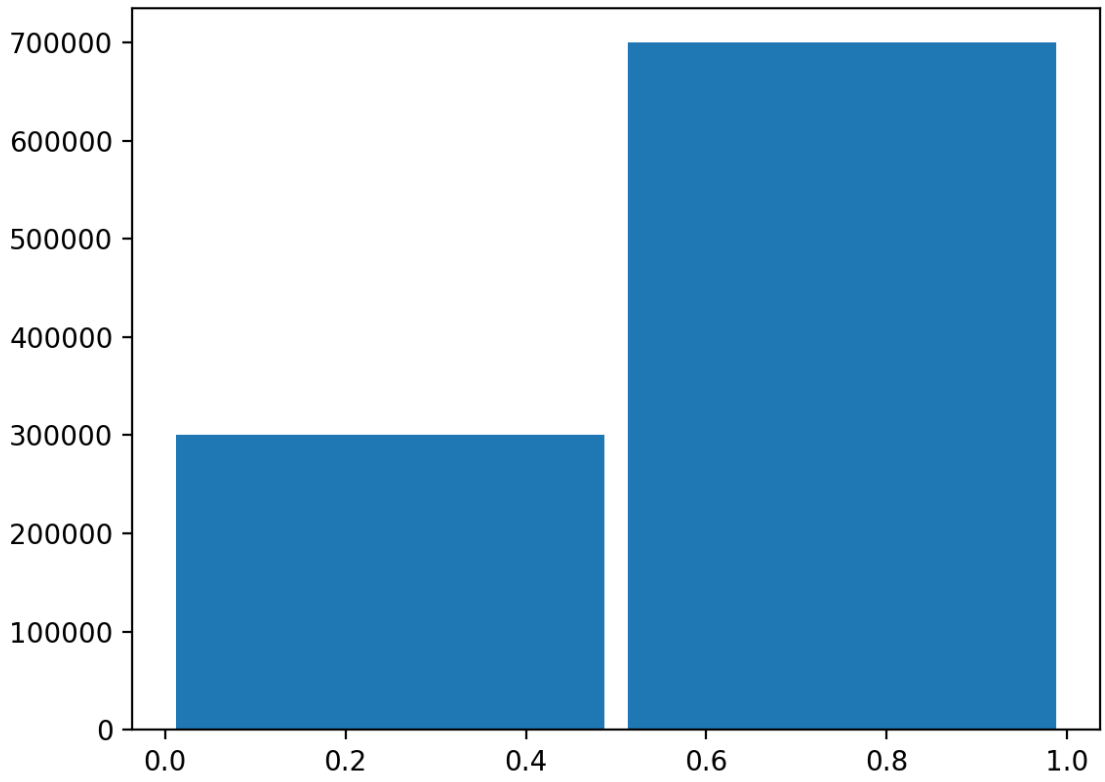
Normal - `self.norm(max, min, store)`:
A random number generator based on a normal distribution, this routine utilizes the Box-Muller method to produce a value from a normal distribution based on the input mew and sigma values. It uses the formula X = mew + SQRT(sigma)*Z to transform the standard normal distribution norm(0,1) into the desired distribution before returning a single value. This routine defaults to producing the standard normal(0,1) distribution if no inputs are provided.

Normal Distribution - 1000k Samples for norm(0,1)



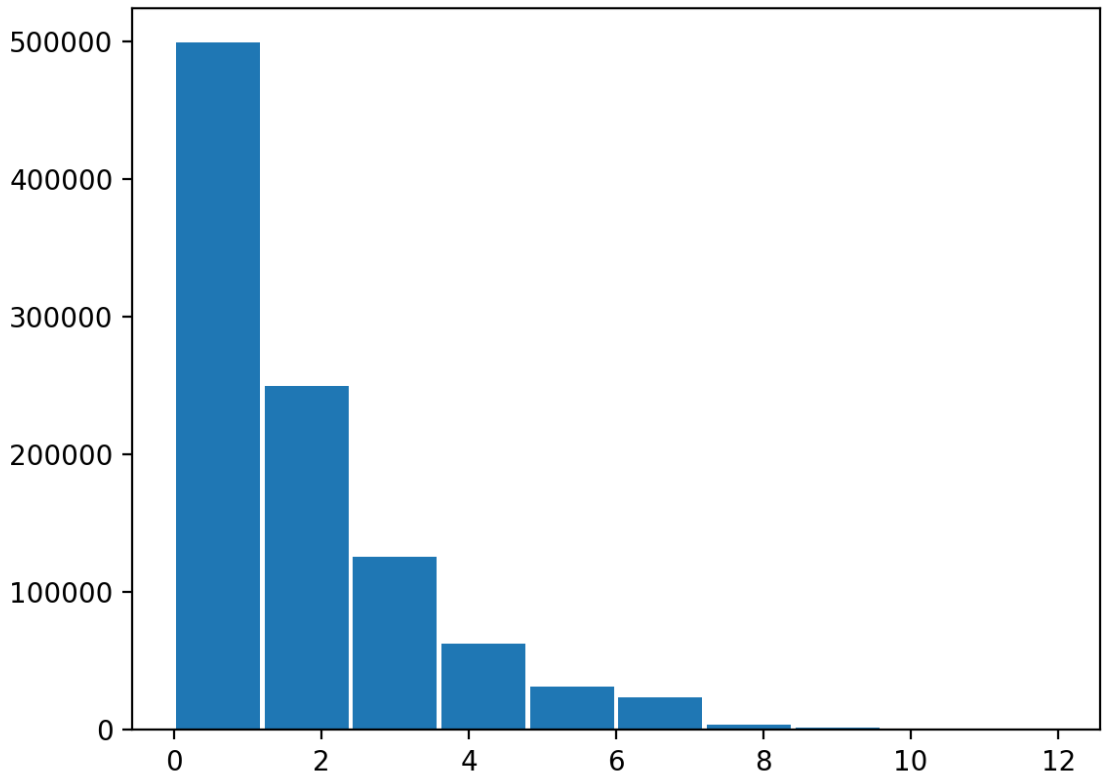
Bernoulli - `self.bern(p)`:
A number randomly generated based on the bernoulli distribution, returning a single value (0 or 1) based on the required input p value. This routine returns 0 if the uniform generated is less than 1 - p, otherwise returns 0.

Bernoulli Distribution - 1000k Samples for bern(p=0.7)



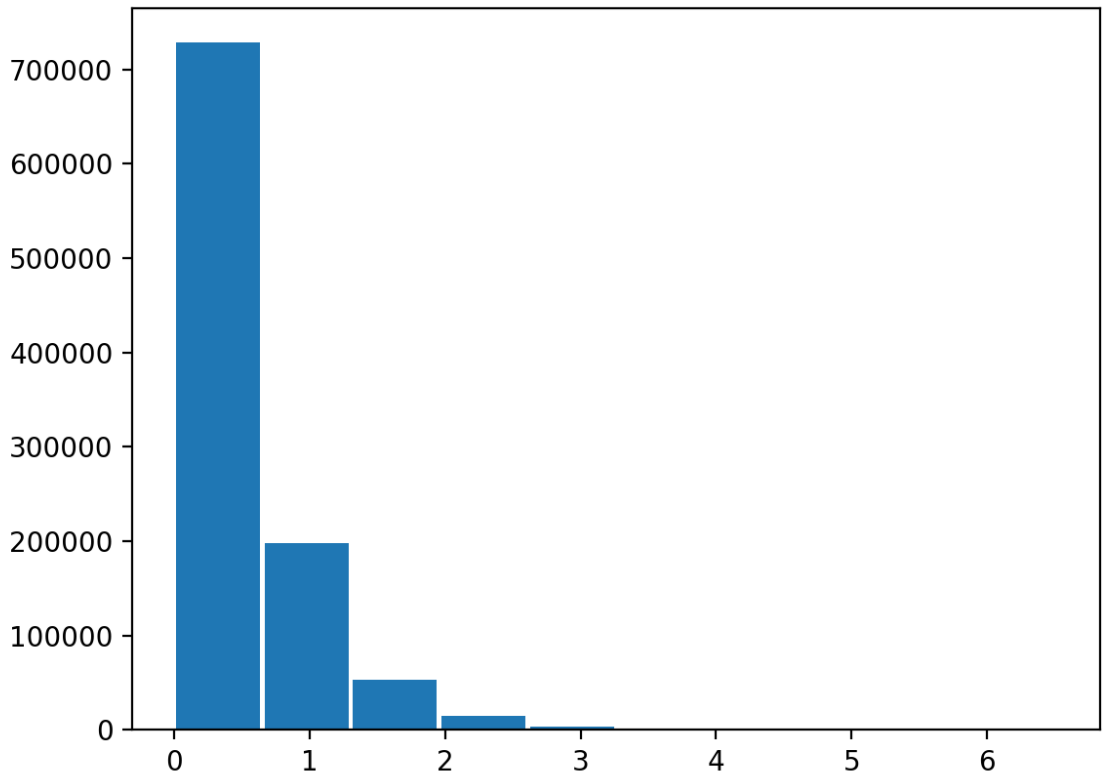
Geometric - `self.geo(p)`:
A number randomly generated based on the inverse transform of the geometric distribution described in class. The required p value input is used to determine the output value.

Geometric Distribution - 1000k Samples for geo(0.5)



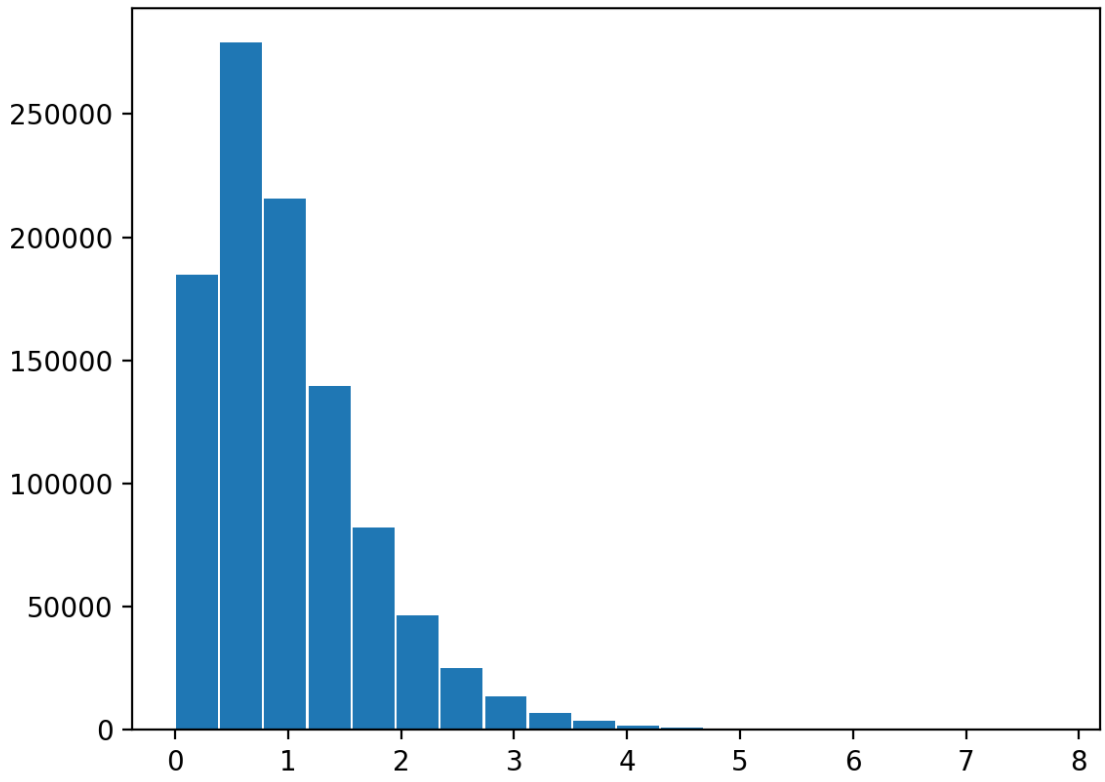
Exponential - `self.exp(lm)`:
A randomly generated number based on the exponential distribution, derived from a uniform using the inverse transform described in class, with the required input `lambda`. This routine returns a single value.

Exponential Distribution - 1000k Samples for exp(lm=2)



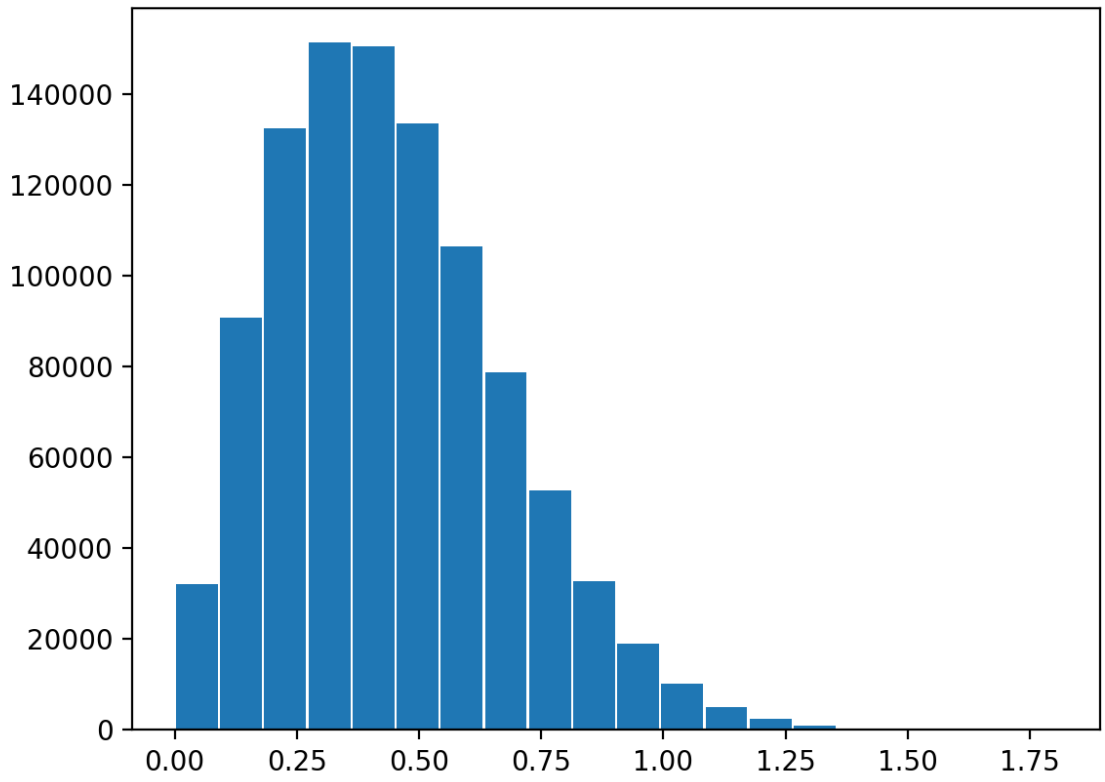
Gamma - `self.gamma(alpha, beta)`:
A randomly generated number based on the gamma distribution, derived from a uniform using the inverse transform described in class, with the required inputs `alpha` and `beta` to describe the shape and scale respectively. This routine returns a single value.

Gamma Distribution - 1000k Samples for gamma(alpha=2, beta=2)



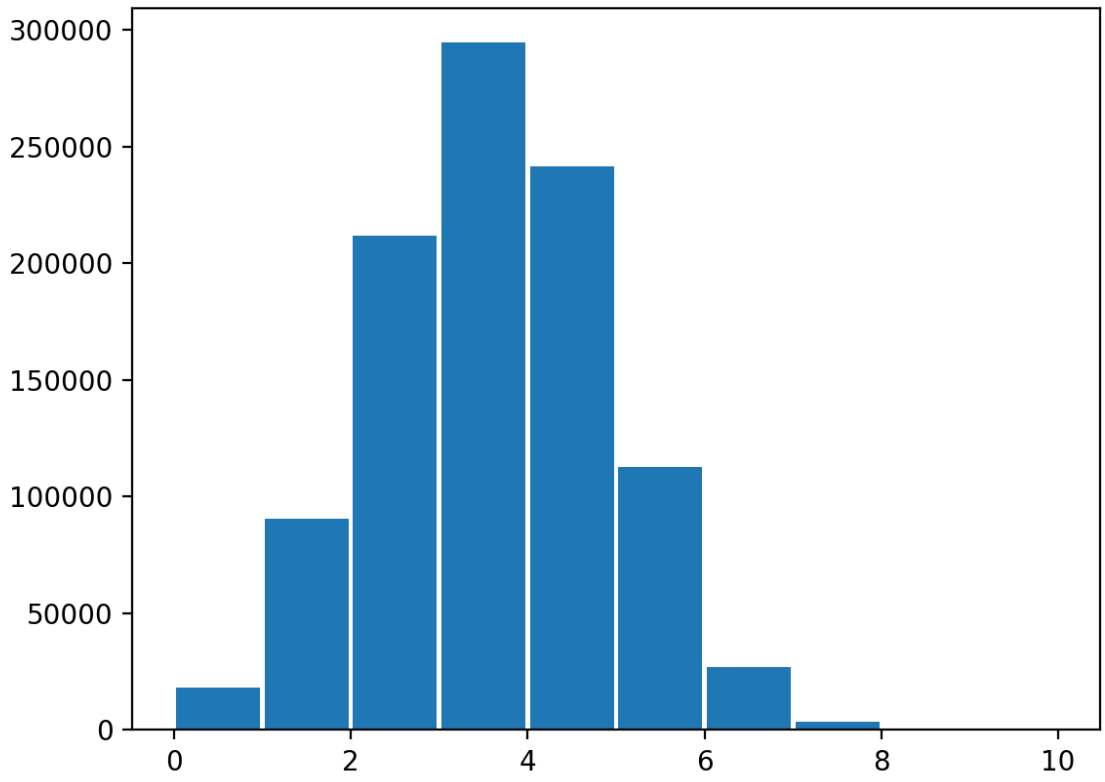
Weibull - `self.weibull(alpha, beta)`:
A randomly generated number based on the weibull distribution, derived from a uniform using the inverse transform described in class, with the required inputs `alpha` and `beta` to describe the shape and scale respectively. This routine returns a single value.

Weibull Distribution - 1000k Samples for weibull(alpha=2, beta=2)

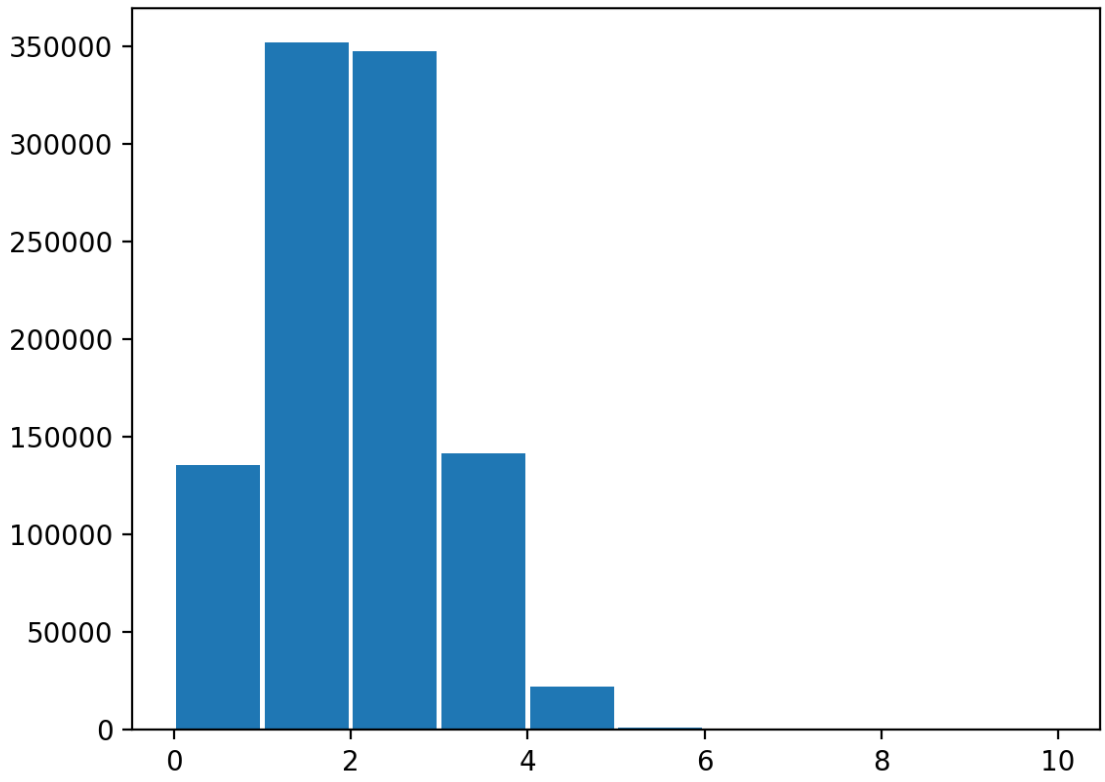


Poisson - `self.pois(lm)`:
A randomly generated number based on the poisson distribution, derived using multiple uniforms and a variation of the acceptance-rejection method. The `lambda` is a required input, and this routine returns a single value.

Poisson Distribution - 1000k Samples for pois(lm=4)

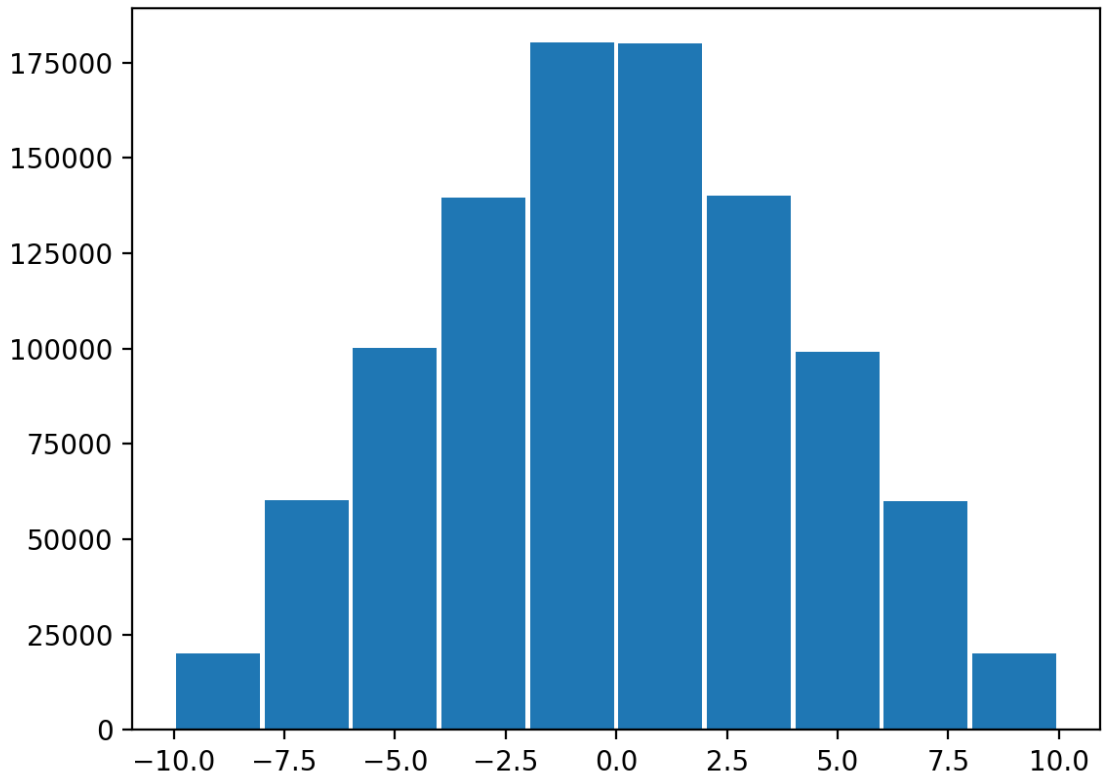


Poisson Distribution - 1000k Samples for pois(lm=2)



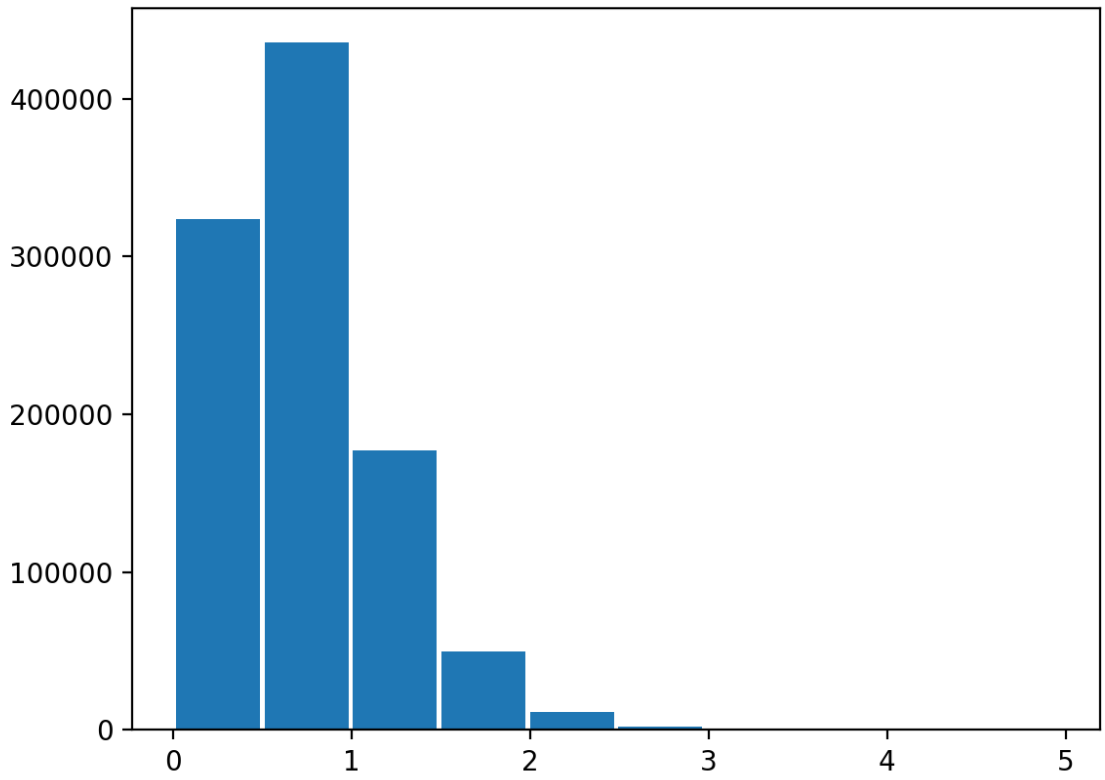
Triangle - `self.tri(mint, maxt)`:
A randomly generated number based on the triangular distribution. The required input maxt and mint are the tri(max, min) values respectively to generate the distribution. This distribution assumes a center value as (mint + maxt)/2. The values are derived by generating two uniforms and adding them (the composition method) to generate the number. This routine returns a single value.

Triangular Distribution - 1000k Samples for tri(mint=-10, mint=10)



Erlang - `self.erlang(lm, n)`:
A randomly generated number based on the erlang_n distribution, derived from a uniform using the inverse transform described in class, with the required inputs `lambda` and `n`. This routine returns a single value.

Erlang Distribution - 1000k Samples for erlang(lm=4,n=3)



Chi Squared - Goodness of Fit

For user confidence, a routine to test the uniformity of the random numbers, `self.chi_squared_gof(n)` is included in the class as well. This routine samples n uniform values and uses the chi squared goodness of fit test split into 5 bins to test the overall uniformity of the randomly generated uniforms.

Chi Squared GOF - 10k Observations, 5 bins

```
With 10000 observation, a chi squared value of 4.096 was observed.

This is less than the test statistic for alpha = 0.01 and k-1 observations, 10330.917
Therefore we fail to reject the null hypothesis, implying that the RNGs are uniform
```