Poisson Random Number Generation in Julia

Jesse Anderson

Ramiro Roman

Dr. Jan Verschelde

February 12, 2023

Abstract

In this report, we have presented the results of a study that aimed to compare the runtimes of several algorithms in Julia, MATLAB, and Python for large samples. The study was conducted on a set of data with varying numbers of elements, and the goal was to determine which of the algorithms would perform better for large datasets. The results showed that the standard Julia implementation had a run time nearly thirty thousand times faster than MATLAB. The mathematical rationale behind these runtimes was also discussed in the report.

The results of this study provide valuable insights into the performance of algorithms and the importance of considering time complexity when choosing the right algorithm for a particular task. The results showed that while Julia's original algorithm had the fastest time, the implementation with the best standard deviation was actually a Log-Normal approximation to the Poisson distribution. The difference in the standard deviation was nearly thirty percent between the two implementations; however the run time was nearly eighteen hundred times as long.

In conclusion, this report highlights the importance of understanding the time complexity of algorithms when selecting the right algorithm for a particular task. The results of this study can help developers and researchers make informed decisions when selecting random number generator algorithms for their projects. The report provides a clear and concise discussion of the results and the mathematical rationale behind the runtimes of the algorithms, making it a valuable resource for anyone interested in the field of algorithms and random number generation.

Table of Contents

Abstract 2

Table of Contents 2

Nomenclature 3

Introduction 3

Analysis 4

Experimental Equipment and Procedure 6

Results 6

Discussion 7

Conclusions 7

References 7

Appendices 8

Nomenclature

$p(x)$	Probability Distribution Function
$F(x)$	Cumulative Distribution Function
λ	Mean number of events in a given time
μ	Mean number of successes that occur in a specified region
σ	Standard Deviation
X	Exponential Deviates
U	Uniform Deviates
$\sim Bin(A)$	Binomial Distribution
*~Gamma(***α**,**β***)*	Gamma Distribution with shape parameter **α**, inverse scale parameter **β**

Introduction

Random number generators are used to simulate random events in computer programs. They are used in a wide range of applications, such as simulations, modeling, cryptography, games, and statistical analysis. By generating sequences of random numbers, these algorithms can mimic the uncertainty and unpredictability of real-world phenomena. The use of random number generators can save time and resources compared to physically conducting experiments, and they can also allow one to explore scenarios that would be difficult or impossible to study otherwise. The generation of random numbers is a crucial aspect of many computational processes and has numerous applications in various fields.

The Poisson distribution is a discrete probability distribution that models the number of events that occur in a fixed interval of time or space, given the average rate at which these events occur. It is used to model the frequency of rare events, such as the number of telephone calls received by a call center in a given hour, the number of customers arriving at a store in a given day, or the number of radioactive decays in a given period of time.

The Poisson distribution is important because it provides a mathematical framework for understanding the occurrence of rare events. It is widely used in various fields, including engineering, finance, insurance, and quality control, to make predictions and decisions based on uncertain data. The Poisson distribution is also used as a model for testing statistical hypotheses and for fitting data to a theoretical distribution. The Poisson distribution is a widely used mathematical concept that has several practical applications in the real world. One of its earliest uses was by Siméon Denis Poisson to determine the probability of winning at the card game of baccarat in 1830. Another famous application was by Ladislaus Josephowitsch Bortkiewicz, who used the distribution to study horse kick deaths in the 1800s. The Poisson distribution can also be used to analyze the likelihood of enemies randomly bombing an area or specifically targeting certain structures. Additionally, it can be used to evaluate the financial viability of keeping a business open 24/7 based on probable sales. It is a simple yet powerful tool for modeling real-world phenomena and for making predictions about future events [5][6].

In Volume 2 of the *Art of Computer Programming*, Donald E. Knuth [1] provides 2 algorithms for generating numbers following this distribution. The first algorithm generates exponential deviates. The author states that to produce a Poisson deviate, we can generate independent exponential deviates with mean $1/\mu$ and stop as soon as the sum of the deviates exceeds 1. The resulting number is then equal to the number of exponential deviates minus 1, which follows a Poisson distribution with mean μ. This algorithm is sufficient for small enough values of μ.

The second method mentioned, referencing J.H. Ahrens and U. Dieter [4], involves generating a gamma distribution deviate, X, and then either setting N equal to m plus a Poisson deviate with mean μ-X if X is less than μ or setting N equal to a binomial distribution deviate if X is greater than or equal to μ. This is known as the Log-Normal-Poisson Algorithm, whose properties will be further explored.

The following describes the testing of several algorithms across different programming languages. The core language in which testing is done is with Julia, from there, the standard

"random" libraries across Matlab and Python are tested against the two algorithms described above, as well as different variations of a generator Poisson deviates in Julia.

Analysis

In order to define the Poisson distribution, the probability density function that is known as the exponential distribution is defined as

$$p(x) = \{0,\ x < 0\ \lambda e^{-\lambda x},\ x \geq 0$$

Whose cumulative distribution function is defined as

$$F(x) = \{0,\ x < 0\ 1 - e^{-\lambda x},\ x \geq 0$$

With mean

$$\mu = \frac{1}{\lambda}$$

And standard deviation

$$\sigma = \mu = \frac{1}{\lambda}$$

The Poisson point process is defined as a random object with points located in some space with each point or state occurring independently of each other. The Poisson distribution is the occurrence of a random variable following the exponential distribution in a Poisson point process in a discrete manner. If we partition the time interval $[0, T]$ into equal n subintervals, then the probability that a spike occurred in k subintervals, but not in the remaining $n - k$ other subintervals is:

$$\left(\frac{n}{k}\right)\left(\frac{\lambda T}{k}\right)^k\left(1 - \frac{\lambda T}{n}\right)^{n-k} = \frac{n(n-1)\ldots(n-k+1)}{(n-\lambda T)^k}\ \frac{(\lambda T)^k}{k!}\left(1 - \frac{\lambda T}{n}\right)^n$$

So as the number of partitions grows, the actual probability that k spikes occur is

$$p(k) = \frac{(\lambda T)^k}{k!}e^{-\lambda T}$$

This is the discrete Poisson distribution with cumulative distribution function as

$$F(x) = e^{-\lambda T}\sum_{0 \leq k \leq x}\frac{(\lambda T)^k}{k!}$$

With mean

$$\mu = \lambda T$$

And standard deviation

$$\sigma = \sqrt{\lambda T}$$

A Poisson deviate N can be generated from exponential deviates $X_1, X_2, \ldots$ with mean $\frac{1}{\mu}$ and stopping as soon as $X_1 + X_2 + \ldots + X_m \geq 1$. At this moment $N \leftarrow m - 1$. The probability that $X_1 + X_2 + \ldots + X_m \geq 1$ is the probability that a gamma deviate of order m is $\geq \mu$. This comes to

$$(\int_{\mu}^{\infty} t^{m-1} e^{-t} dt)/(m-1)!$$

Therefore, the probability that $N = n$ is

$$\frac{1}{n!}\int_{\mu}^{\infty} t^n e^{-t} dt - \frac{1}{(n-1)!}\int_{\mu}^{\infty} t^{n-1} e^{-t} dt = e^{-\mu}\frac{\mu^n}{n!}, \qquad n \geq 0$$

The expression of the algorithm above is suitable for smaller values of μ. Knuth describes the Log-Normal Poisson distribution as a method for generating Poisson deviates of a large μ while utilizing the gamma and binomial distributions. First, gamma deviate X is generated of order $m = [\alpha\mu]$, where α is a suitable constant. Note that since X is equivalent to $-ln(U_1 \ldots U_m)$, we bypass m number of steps from the previous method. If $X < \mu$, set $N \leftarrow m + N_1$, where N_1 is a Poisson deviate with mean $\mu - X$. If $X \geq \mu$ set $N \leftarrow N_1$, where N_1 has the binomial distribution $(m - 1, \mu/X)$. In the research of J.H. Ahrens and U. Dieter, $\alpha = 7/8$ is a suitable constant.

This transformation of $N_1 \sim Bin(N_1)$ is valid due to the following: "Let $X_1, \ldots, X_m$ be independent exponential deviates with the same mean; let $S_j = X_1 + \ldots + X_j$ and let $V_j = S_j/S_m$ for $1 \leq j \leq m$. Then the distribution of $V_1, V_2, \ldots, V_{m-1}$ is the same as the distribution of $m - 1$ independent uniform deviates sorted into increasing order"[7] . The (m-1) fold integral is then

$$\frac{f(v_1, v_2, \ldots, v_{m-1})}{f(1,1,\ldots,1)} = \frac{\int_0^{v_1} du_1 \int_{u_1}^{v_2} du_2 \ldots \int_{u_{m-2}}^{u_{m-1}} du_{m-1}}{\int_0^{1} du_1 \int_{u_1}^{2} du_2 \ldots \int_{u_{m-2}}^{1} du_{m-1}}$$

With making the substitution $t_1 = su_1,\ t_1 + t_2 = su_2,\ \ldots,\ t_i + \ldots + t_{m-1} = su_{m-1}$. This ratio is the corresponding probability that the uniform deviates $U_1, \ldots, U_{m-1}$ satisfy $U_1 \leq v_1, \ldots, U_{m-1} \leq v_{m-1}$, given that they also satisfy $U_1 \leq \ldots \leq U_{m-1}$.

Experimental Equipment and Procedure

The code was run on an Acer Aspire A515-54G-70TZ with an Intel Core i7-8565U, 20GB of 2400 Mhz RAM, and a 512GB PCIe 3.0 SSD, on Windows 10. From a sample of 10,000 Poisson Random Numbers a histogram was plotted; the time to sample the numbers was measured and this measurement was repeated 1,000 times for a total random number generation of 10,000,000 numbers. This methodology was repeated for each Julia, MATLAB, and Python code.

Results

Overall, the results were in line with what was expected. As n grew the Poisson distribution became more normalized until it reached the normal distribution. For small n, the distribution appeared skewed to the right. In the table below the run times of the code in various languages is outlined:

Language	Algorithm	Mean	Standard Deviation	Run Time[s](n=10,000 averaged over 1,000 runs)
Julia	Slide 26, L3	100.0543	9.8699	0.0021
	Poisson()	100.1924	9.9746	0.0003
	Ahrens & Dieter[6]	100.0283	9.8273	0.0026
	Normal Approx. to Poisson[3]	99.6980	7.0247	5.5606
MATLAB	poissrnd()	100.0010	9.9956	8.5051
Python	numpy.random.poisson()	99.9997	10.0009	0.5246

Table 1: Run times of Poisson Random Number Generation in Julia, Matlab, and Python

As one can see in the table above, Julia's run time was the fastest, with MATLAB being ~28,000 times slower. Python fared better with a time which was ~1,750 times slower. Notably the standard deviation and mean of every implementation was roughly the same; however with a Normal Approximation to the Poisson distribution it was found that the standard deviation was reduced by ~30%. The run time suffered drastically, being thousands of times slower than the fastest implementation, but in scientific and high precision computing it will be optimal to use this implementation.

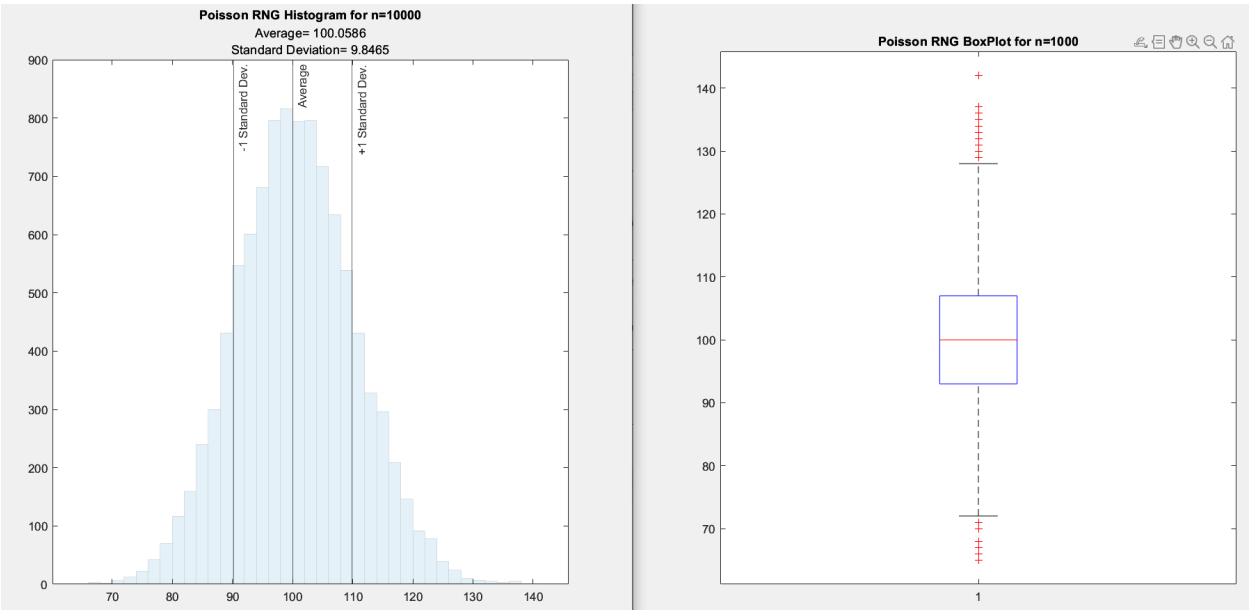

Figure 1: Histogram and Boxplot of Poisson RNG for n =1,000

Discussion

In this report, we tested several algorithms for generating Poisson random numbers across different programming languages. The core language used for testing was Julia, with additional tests conducted on the standard random libraries of Matlab and Python. The results of the runtime tests showed that the Log-Normal-Poisson Algorithm was faster in generating Poisson deviates compared to the exponential deviates method, especially for larger values of λ.

Conclusions

In conclusion, the Poisson distribution is a crucial concept in understanding the occurrence of rare events, and the Log-Normal-Poisson Algorithm is a reliable and efficient method for generating Poisson random numbers. The results of the runtime tests demonstrate that this algorithm is faster in generating Poisson random numbers compared to the exponential deviates method, especially for larger values of λ. This makes the Log-Normal-Poisson algorithm a useful tool in various applications that require the generation of Poisson random numbers.

References

[1] "Numerical Distributions," in *The Art of Computer Programming*, Stanford, CA: Computer Science Dept. School of Humanities and Sciences, Stanford University, 1976.

[2] W. H. Press, “Random Numbers,” in *Numerical Recipes in C++: The Art of Scientific Computing*, New Delhi: Cambridge Univ. Press India, 2007.

[3] A. C. Atkinson. "The Computer Generation of Poisson Random Variables." IEE Proceedings, vol. 131, no. 4, pp. 362-365, Aug. 1984.

[4] J. H. Ahrens and U. Dieter. "Computer Methods for Sampling from Gamma, Beta, Poisson and Binomial Distributions." Computing, vol. 12, pp. 223-246, 1974. Springer-Verlag, 1974.

[5] Corporate Finance Institute. "Poisson Distribution." Poisson Distribution, December 15, 2022. https://corporatefinanceinstitute.com/resources/data-science/poisson-distribution/.

[6] O'Connor, J. J. and Robertson, E. F. "Ladislaus Bortkiewicz - biography." Maths History, July 2000. https://mathshistory.st-andrews.ac.uk/Biographies/Bortkiewicz/.

[7] Knuth, D. E. (2016). Chapter 3 - Random Numbers. In Seminumerical algorithms (3rd ed., Vol. 2, pp. 1–140). essay, Addison-Wesley.

Appendices

Julia Code

```
# Function from slide 26, L3
using Random
using Plots
using Statistics

function poisson_rng(lambda)
   k = 0
   p = 1.0
   L = exp(-lambda)
   while p > L
      k += 1
      p *= rand()
   end
   return k-1
end

c = zeros(1000,1)
global x #Julia has some major issues with using variables outside the loop, scope.
x = Nothing
x=zeros(10000,1)
for ii in 1:1000
   a = time()
   for i in 1:10000
      x[i] = poisson_rng(100)
   end
   b = time()
   c[ii]=b-a
end
meanC = mean(c)
meanX = mean(x)
stdX = std(x)
print(meanC, " seconds to run Julia default Poisson code 10,000 times.")
print("Mean = ",meanX," Standard Deviation = ", stdX)

histogram(x)
```

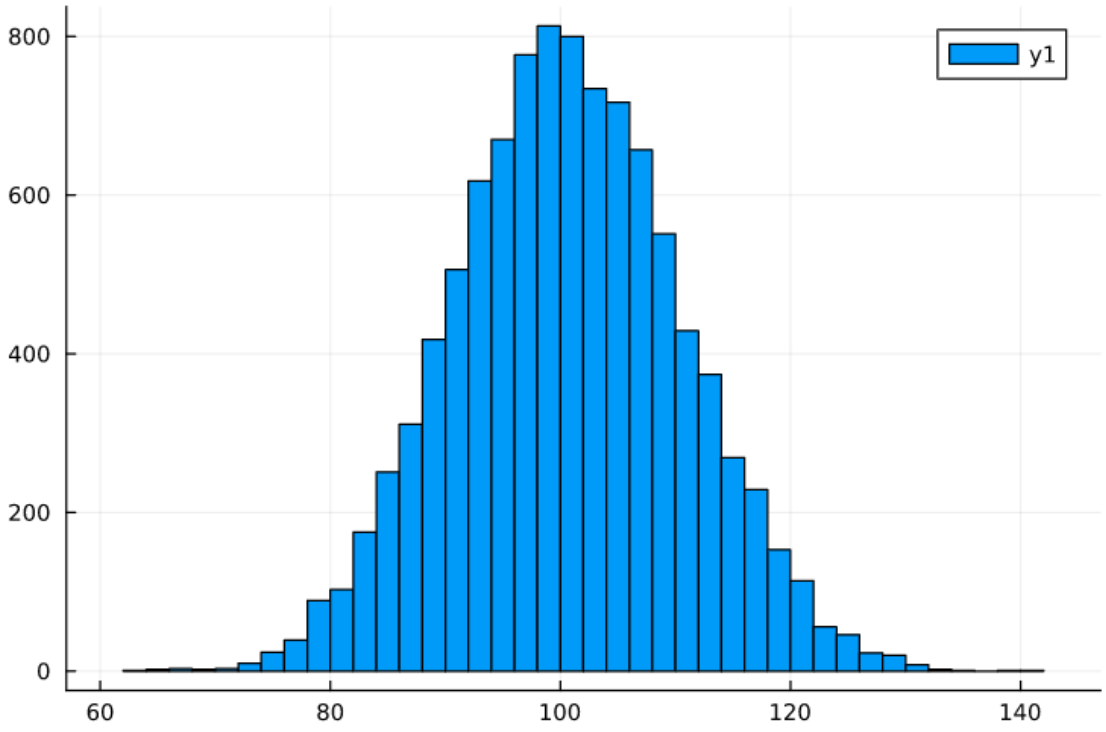

Figure 2: Histogram of the results of the above simulation

```
# Function from slide 27, L3
# import Pkg;
# Pkg.add("StatsKit")
using StatsKit
c = zeros(1000,1)
global x
x = Nothing
for i in 1:1000
    a = time()
    p = Poisson(100)
    x = rand(p,10000)
    b = time()
    c[i]=b-a
end
meanC = mean(c)
meanX = mean(x)
stdX = std(x)
print(meanC, " seconds to run Julia default Poisson code 10,000 times.")
```

```
print("Mean = ",meanX," Standard Deviation = ", stdX)

histogram(x)
```

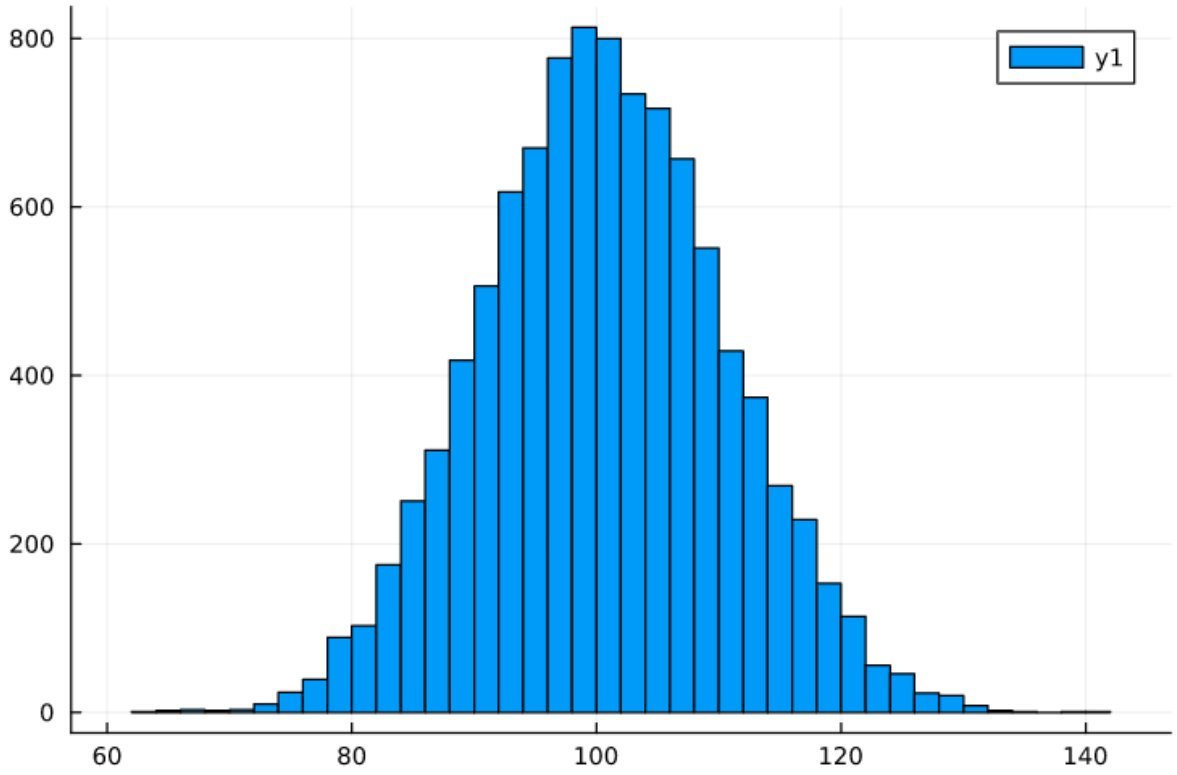

Figure 3: Histogram for the poisson() implementation in Julia

```
#The below code is our actual assignment. We were able to successfully code this part, but the m
ore involved
#and thankfully not assigned implementation on page 140 confused us.
# import Pkg;
# Pkg.add("Distributions")
using Distributions
using Statistics
# import Pkg;
# Pkg.add("StatsPlots")
using StatsPlots #boxplots
using Plots
using StatsKit

function poisson_rng(mu::Float64) #20
    alpha = 0.00000000001 #7/8 over estimates. Choosing a new alpha
```

```
    m = abs(alpha * mu)
    x = rand(Gamma(m, 1.0/mu))#~Gamma(α,β)
    if x < mu
        n = m+rand(Poisson(mu - x))
    else
        n = rand(Binomial(m - 1, mu/x))
    end
    return n
end
global x
x = Nothing
c = zeros(1000,1)
for ii in 1:1000
    x=zeros(10000,1)
    a = time()
    for i in 1:10000
        x[i] = poisson_rng(100.0)
    end
    b = time()
    c[ii] = b-a
end
meanC = mean(c)
print(meanC," seconds to run Log-Normal-Poisson Algorithm 10000 times\n")

meanX = mean(x)
stdX = std(x)
print("Mean = ",meanX," Standard Deviation = ", stdX)
#p1 = histogram(x, label="Experimental",color=:blue, title="Log-Normal-Poisson Algorithm \n
n = 10000", xlabel = "x",ylabel = "Frequency")
p2 = boxplot(x, label="Experimental",color=:blue, title="Log-Normal-Poisson Algorithm \n n =
10000")
savefig("Plots1.png")
```

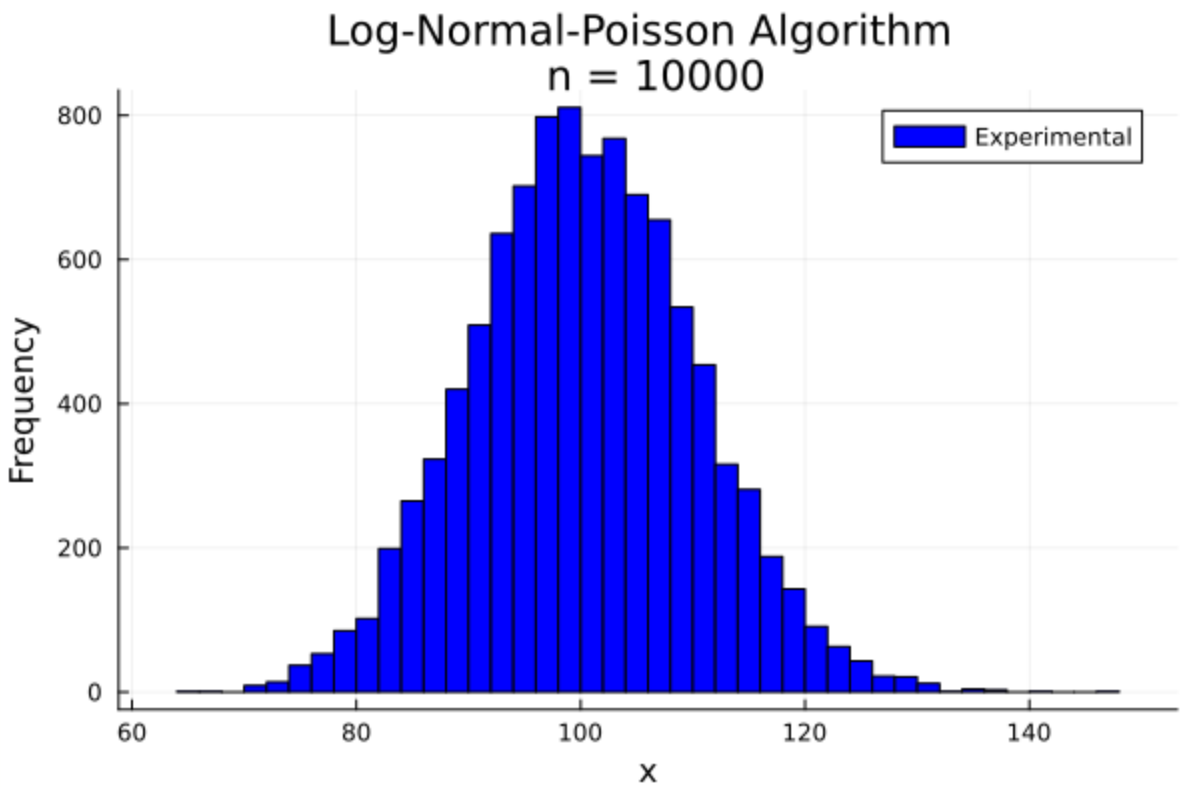

Figure 4: Histogram for the Log-Normal-Poisson Algorithm

```
#22. [HM40 ] Can the exact Poisson distribution for large μ be obtained by generating
# an appropriate normal deviate, converting it to an integer in some convenient way, and
# applying a (possibly complicated) correction a small percent of the time?

# http://www.it.uom.gr/teaching/linearalgebra/NumericalRecipiesInC/c7-3.pdf

using Distributions
using Plots
function poisson_approx(mu::Float64) #Julia sensitive to data types
   normal = Normal(mu, sqrt(mu)) #normal distribution, mean mu and variance sqrt(mu)
   while true #infinity
      xxx = rand(normal) #gen random, keep going until x greater than 0.
      if xxx >= 0
         k = convert(Int64, floor(xxx)) #necessary to avoid error in factorial
         u = rand()
         if log(u) <= -(xxx - k) + k * log(mu) - mu - log(factorial(big(k))) #comparison, acceptanc
e/rejection, note need big() or else fails factorial for some cursed reason.
            #if statement == SLOW
            return k
```

```
            end
        end
    end
 end
poisson_approx(100.0)
global x
x = Nothing
c = zeros(10,1)
for ii in 1:10
    x = zeros(10000,1)
    a = time()
    for i in 1:10000
        x[i] = poisson_approx(100.0)
    end
    b = time()
    c[ii] = b-a
end
meanC = mean(c)
print(meanC," seconds to run Log-Normal-Poisson Algorithm 10000 times\n")
meanX = mean(x)
stdX = std(x)
print("Mean = ",meanX," Standard Deviation = ", stdX)
p1 = histogram(x, label="Experimental",color=:gray, title="Log-Normal-Poisson Algorithm \n n
= 10000", xlabel = "x",ylabel = "Frequency")
#p2 = boxplot(x, label="Experimental",color=:gray, title="Log-Normal-Poisson Algorithm \n n =
 10000")
```

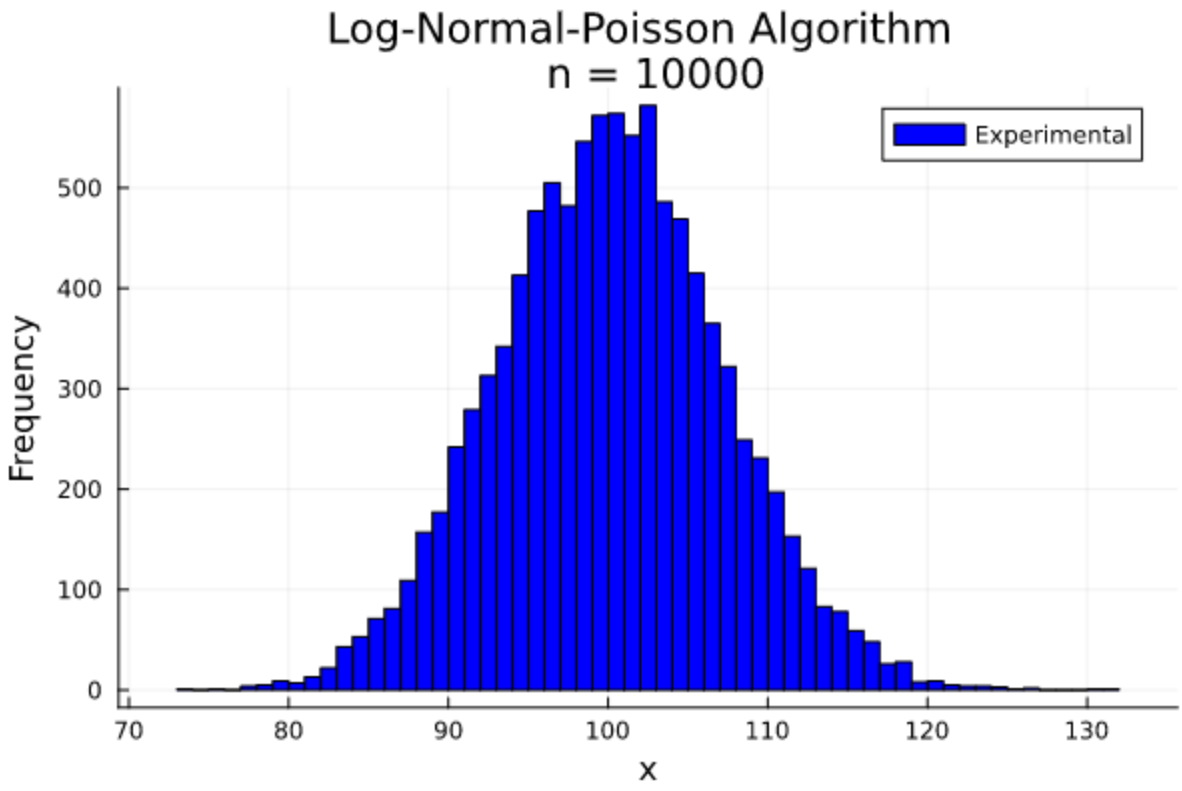

Figure 5: Histogram plot in Julia for the Normal Approximation to the Poisson Distribution as described by Knuth

MATLAB Code

```
%%
clc
clear all
close all
%%
holdholdPoisson = zeros(10000,1000);
r = poissrnd(10.0); %initialize r, test
holdPoisson = poissrnd(100.0,10000,1); %lambda, row size, col size
tic
holdholdPoisson = poissrnd(100.0,10000,1000);
%%
holdtime = toc; %timer to see code execution for presentation, note parfor is slower!
meanSmall = mean(holdPoisson); %% the mean approaches lambda, ~10.0840
meanLarge = mean(mean(holdholdPoisson));
stdLarge = mean(std(holdholdPoisson)); %call to standard deviation
stdSmall = std(holdPoisson);
%%
%For small RNG we should see left skew
```

```
holdPoisson2 = poissrnd(10.0,100,1);
meanSmall2 = mean(holdPoisson2);
stdSmall2 = std(holdPoisson2);
%%
%
txt1a = ["Average= "+ num2str(meanSmall)];
txt1b= ["Standard Deviation= "+ num2str(stdSmall)];
txt1 = [txt1a,txt1b];
txt2a = ["Average= "+ num2str(meanSmall2)];
txt2b= ["Standard Deviation= "+ num2str(stdSmall2)];
txt2 = [txt2a,txt2b];
%
fig1a = figure(); %assign figure object to variable to manipulate later
histogram(holdPoisson,'FaceAlpha',0.1,'EdgeAlpha',0.1) %roughly normal for one 1000 set, set
transparent for analysis.
title("Poisson RNG Histogram for n=10000") %plot title
subtitle(txt1)
movegui('northwest')%moves plot object to the upper left quadrant of the user's screen
hold on %allows multiple plotting
xline([meanSmall-stdSmall meanSmall  meanSmall+stdSmall],'-',{'-1 Standard
Dev.','Average','+1 Standard Dev.'})
%Above line creates three vertical lines at +/-1 standard deviation away
%from the mean and the mean itself.
%
fig1b = figure();
boxplot(holdPoisson) %Native MATLAB Whisker Plot
title("Poisson RNG BoxPlot for n=1000")
movegui('north')
%
fig2a = figure();
histogram(holdPoisson2,'FaceAlpha',0.1,'EdgeAlpha',0.1)
title("Poisson RNG Histogram for n=100")
subtitle(txt2)
movegui('west')
hold on
xline([meanSmall2-stdSmall2 meanSmall2  meanSmall2+stdSmall2],'-',{'-1 Standard
Dev.','Average','+1 Standard Dev.'})
%
fig2b = figure();
boxplot(holdPoisson2)
title("Poisson RNG BoxPlot for n=100")
movegui('center')
```

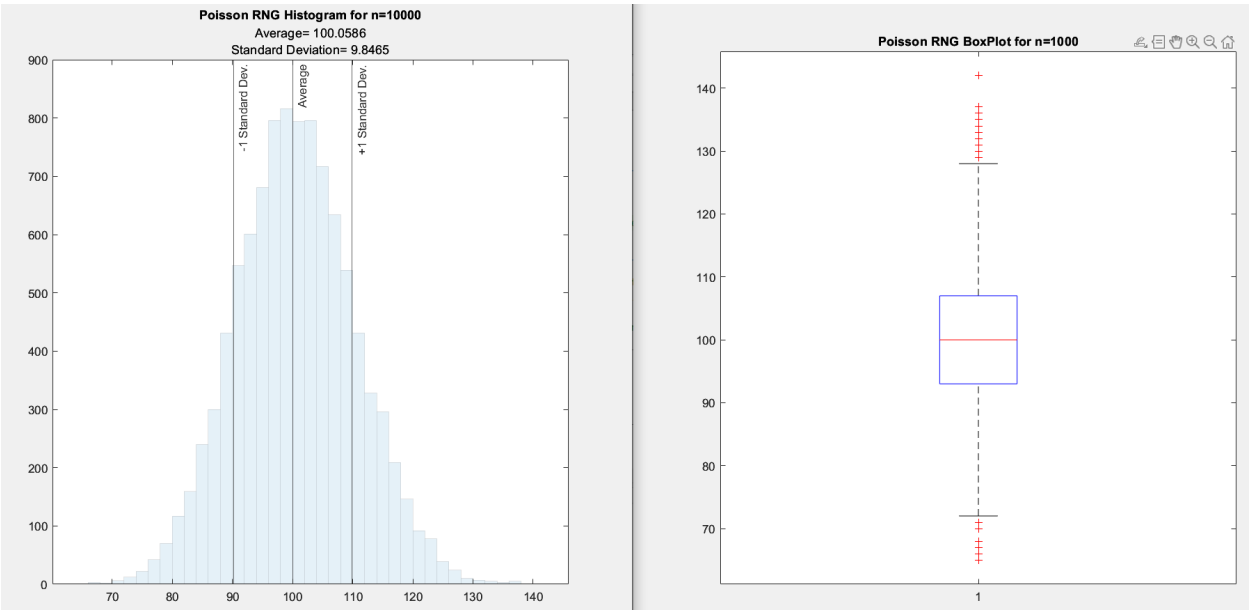

Figure 6: MATLAB histogram and boxplot for the above code

Python

```
import numpy as np
import time
import statistics
import matplotlib.pyplot as plt

sec1 = time.time()

holdPoisson = np.random.poisson(10,size = [1000,1])
holdholdPoisson = np.random.poisson(10,size = [1000,1000])
sec2 = time.time()
#timer to see code execution for presentation

holdPoisson2 = np.random.poisson(10,size = [100,1])

meanSmall = np.mean(holdPoisson) ## the mean approaches lambda, ~10.0840
```

```
meanLarge = np.mean(np.mean(holdholdPoisson))
stdLarge = np.mean(np.std(holdholdPoisson)) #call to standard deviation
stdSmall = np.std(holdPoisson)
meanSmall2 = np.mean(holdPoisson2)
stdSmall2 = np.std(holdPoisson2)
print(meanSmall)
print(meanSmall2)
print(meanLarge)
print(stdSmall)
print(stdSmall2)
print(stdLarge)



fig,axs = plt.subplots(2,2)

axs[0,0].hist(holdPoisson, alpha =0.5)
axs[0,0].vlines(x = [meanSmall-stdSmall, meanSmall,  meanSmall+stdSmall],
ymin = 0, ymax = 250, colors = 'black')
axs[0,0].title.set_text("Poisson RNG Histogram for \n n =1000")
axs[0,1].boxplot(holdPoisson)
axs[0,1].title.set_text("Poisson RNG BoxPlot for \n n =1000")

axs[1,0].hist(holdPoisson2, alpha =0.5)
axs[1,0].vlines(x = [meanSmall2-stdSmall2, meanSmall2,  meanSmall2+stdSmall2],
ymin = 0, ymax = 25, colors = 'black')
axs[1,0].title.set_text("Poisson RNG Histogram for \n n =100")
axs[1,1].boxplot(holdPoisson2)
axs[1,1].title.set_text("Poisson RNG BoxPlot for \n n =100")
```

```
plt.show()
plt.savefig("save.png")
```

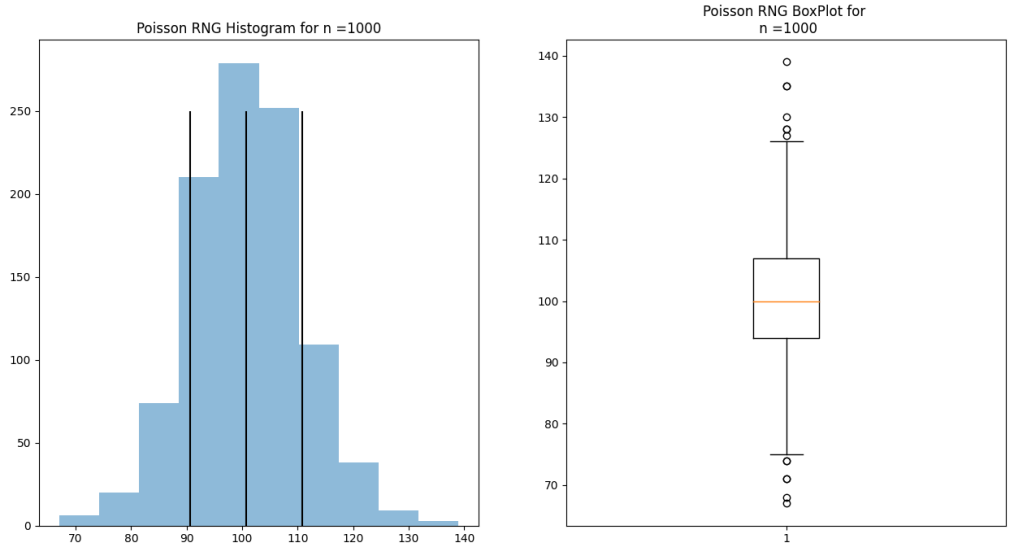

Figure 7: Code in Python to test numpy.random.poisson() function