Quantum Random Number Generator

Caleb Noel
caleb.a.noel@colorado.edu

Keaton Harlan
keha1852@colorado.edu

April 2022

Abstract

The goal of this experiment was to generate truly random numbers. Truly random numbers are useful for many purposes, but most importantly encryption. We used a reverse biased transistor to generate avalanche noise, passing the noise through various amplification stages before connecting it to a DAQ. We counted the number of avalanche cascades that occurred in a time period, isolating it from other noise sources by using the characteristic RC time of our circuit. These counts were then converted to binary and debiased using the von Neumann extractor. Using this method, we were able to generate around 60,000 random bits over 13 trials. Our analysis using a variety of statistical tests found our generated bits to be sufficiently random.

1 Introduction

In this experiment, we sought to generate random numbers from a quantum system. More specifically, we were interested in the noise which is produced from an avalanche noise circuit, and the ways in which we could get that noise to generate random numbers. The important part of this project is ensuring that the process from which the noise is generated can be interpreted as a quantum process. Avalanche noise is not inherently quantum, but the events which start the avalanche process can be. Avalanching occurs when an electron in a sufficiently strong electric field travels across the depletion region of a PN junction, colliding with nearby atoms and creating new electron hole pairs. These pairs rapidly increase the reverse current across the junction, creating current noise across the P region. The source of the electron which begins the cascade is where quantum comes in; although many of the cascades can be due to thermal noise – a process which also has quantum contributions – some of the cascades can result from an electron tunneling from the N region into the depletion region, meaning that many of the events we are detecting are a result

of quantum processes. We use this fact later in converting the noise to numbers, seeking to utilize the unpredictability in the onset of avalanche cascades rather than the actual noise it outputs to better "couple" our random numbers to events that have quantum contributions.

2 Experimental Setup

Our initial setup used a circuit designed by Charles Platt, as seen in Figure 1. However, this design proved to be too complicated for us to detect much noise and we ended up simplifying the circuit.

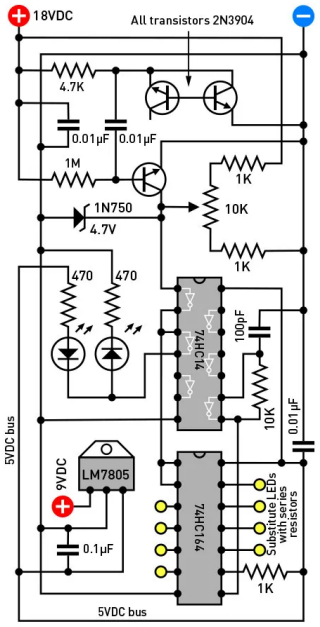


Figure 1: Initial Circuit (Figure From [1])

After several iterations we settled on a slightly modified version of a circuit by Rob Seward [2]. The avalanche noise is produced by transistor Q1 as seen in Figure 2. This noise controls the base for Q2, allowing us to amplify the avalanche breakdown. We then have a second amplification through a common emitter amplifier with a gain of 20db using Q3. This, in combination with our characteristic RC time t 0.47 ms, gives us a sufficiently strong and characteristically wide noise signal with which to generate random numbers. This analog signal then goes into the DAQ, where all subsequent digitization/signal analysis was done using the LabView environment. In order to capture the high frequency noise from the analog signal, it was important to find the highest possible sample rate at which the noise spectrum would still show randomness over a range of at least 100 Hz. We found this limit to be $f_{sample} = 20kHz$,

anything above that producing noticeable spikes in our power spectral density plot over the frequency range where our noise was random.

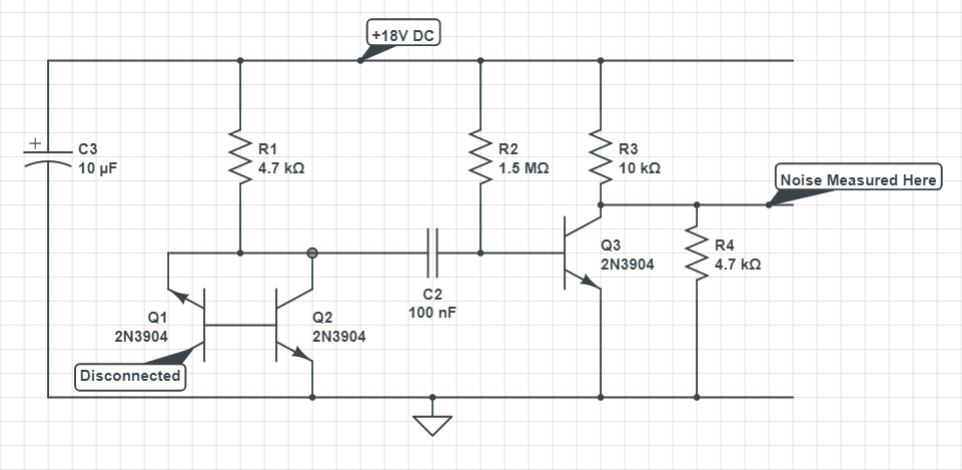


Figure 2: Final circuit design. Figure from [2]

Another important factor that we accounted for were stray sources of emf. Circuits which generate noise in this context are very sensitive to stray emf, so we opted to mount our circuit in a Faraday cage in order to shield it from outside interference.

3 Results

3.1 Data

Our circuit produced a waveform with characteristically wide voltage peaks corresponding to avalanche cascades. As shown in Figure 3, there are short pulses with a width of about 0.47ms, which suggested that this was the avalanche component of our noise.

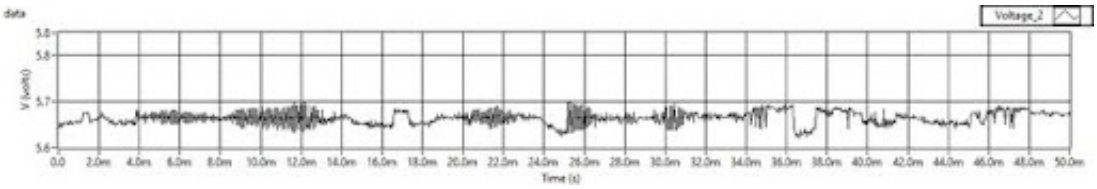


Figure 3: Noise signal acquired by DAQ

To confirm that our signal had random noise, we then plotted the average of >100 power spectral density plots, which plots $\frac{\sqrt{P^2}}{Frequency}$ vs $Frequency$.

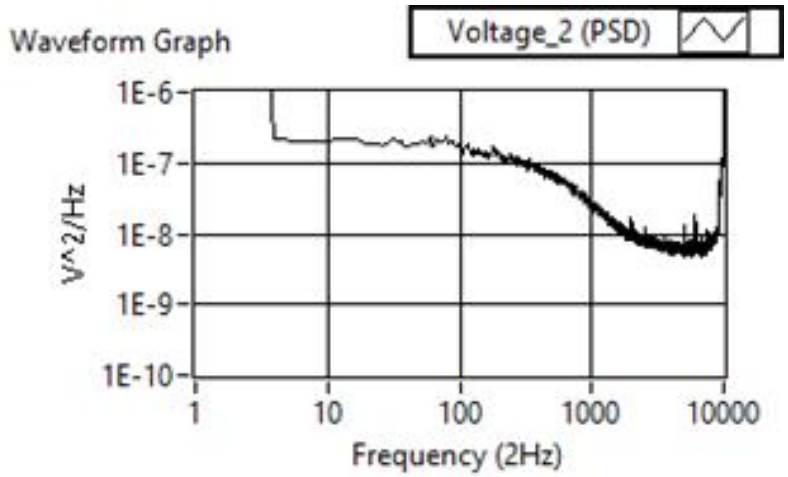


Figure 4: Averaged Power Spectral Density of the DAQ noise signal

As shown in Figure 4, the noise spectrum is flat between 10-200 Hz, which indicates truly random noise in this frequency range. The rolloff in the distribution after this point is typical of an RC network, rolling off higher frequencies according to its characteristic time, and the various spikes on the high end can be attributed to other sources of noise such as DC supply ripple, DAQ communications and thermal noise. Now that we had confirmed our signal to have truly random sources of noise, we just had to find a way to isolate these lower frequencies from the other noise sources.

3.2 Digitization

As long as a noise contribution in the signal had a width on the order of our RC time, we knew that it would be the avalanche noise contribution to the circuit and thus occupied the lower frequency range of our random noise. Now that we had a method of isolating the avalanche noise, we – according to the concept of cascades having a random number of occurrences in a given time sample – chose to sample 50msec of the signal and count the number of times an avalanche signal occurred using the LabView Peak Counter, see Appendix A.1. We found that the avalanche occurrences generated noise up to 5.63 volts. From there, we input 5.63 volts as the amplitude threshold, and 4 blocks as the width threshold – since our characteristic RC time occupied 4 blocks – and passed the output for number of peaks into a measurement file over trial times of about 60 seconds. Due to the limitations of our sample rate, I had to decrease number of samples to 200 in order to get an amplitude acquisition of about 100Hz, meaning that each second, our file would acquire 100 different amplitude values. Each trial gave us 6,000 values and we did 13 trials, giving us 80,000 decimal values after we combined them. We then converted these decimal values into binary to make a 207,000 bit long bitstring using the code in Appendix A.2.

3.3 Debiasing

Initial analysis of our bits showed that we did not have an even distribution of ones and zeroes. Figure 5 is a histogram of the 207,000 collected bits.

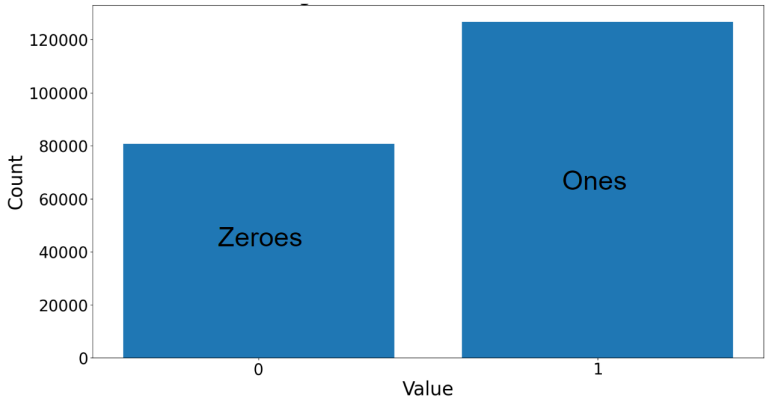


Figure 5: Histogram of raw bits

It is clear we do not have an even distribution of ones an zeroes. Thankfully this can be solve using debiasing. Mathematician and computer scientist John von Neumann created an algorithm to flatten this distribution while preserving randomness called the von Neumann extractor. The algorithm works by grouping the bits into groups of two, throwing away matching pairs, and outputting the first bit of non-matching pairs. This results in a large amount of data being thrown away, but removes the biasing issue. Using the code in Appendix A.2 resulting in this new histogram of about 54,000 bits. The average value of our bits is 0.499 ± 0.006 meaning they are now evenly distributed.

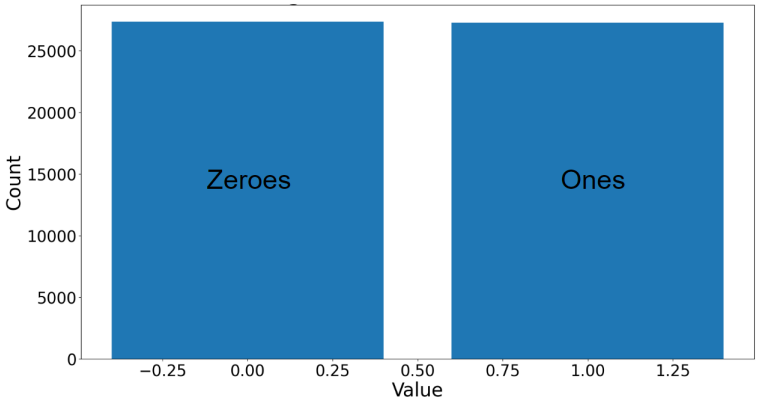


Figure 6: Histogram of debiased bits

3.4 Analysis

3.4.1 2d Correlation Test

When using the von Neumann extractor it is important that there is no correlation between successive bits. To show this we can make a 2d correlation plot. Converting our bits into 8-bit bytes, Figure 7 plots each value against the previous value.

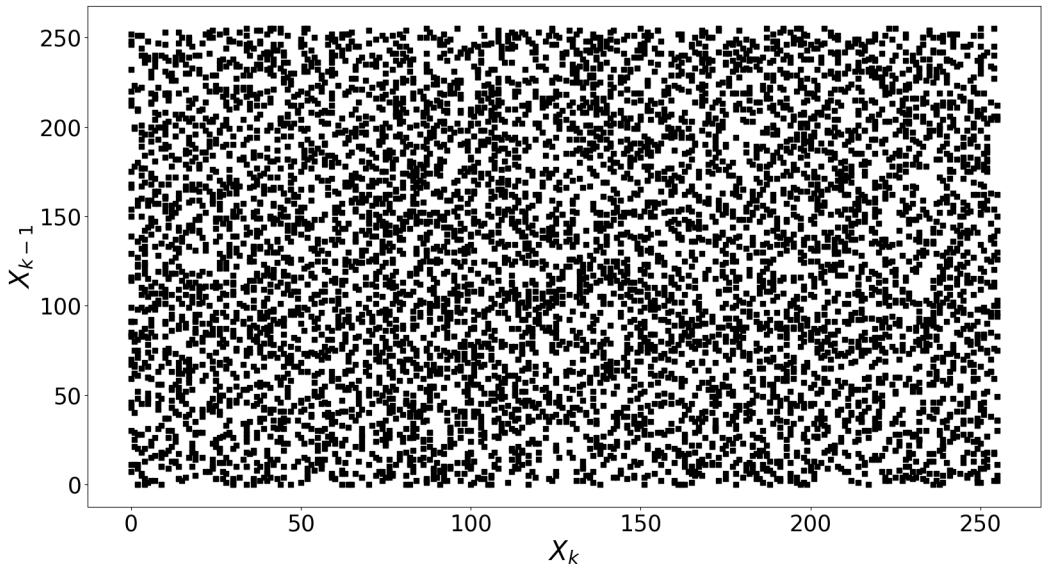


Figure 7: 2d correlation plot of 8 bit numbers.

A visual examination of Figure 7 suggests there is no correlation between successive bits. This means that the von Neumann extractor should work on our data, and is also evidence towards our generated numbers being random. Additional 2d correlation plots have been put in Appendix A.3.

3.4.2 Chi-Squared Test

The most basic test of randomness are the chi-squared test. We compute the chi-squared value of our numbers against an even distribution using the equation

$$\chi^2 = \sum \frac{(Observed - Expected)^2}{Expected} \tag{1}$$

However, random numbers would not be expected to perfectly follow an even distribution. The expected chi-squared values depend on the degrees of freedom. We can approximate the expected chi-squared as the number of degrees minus one [3].

We can do this analysis for a variety of different byte sizes, where changing the byte size changes the number of degrees of freedom. Doing the chi-squared test on the 4-bit byte distribution as is seen inFigure 8 gives us $\chi^2 = 14$. This agrees with the expected value for 16 degrees of freedom of 15 ± 6.

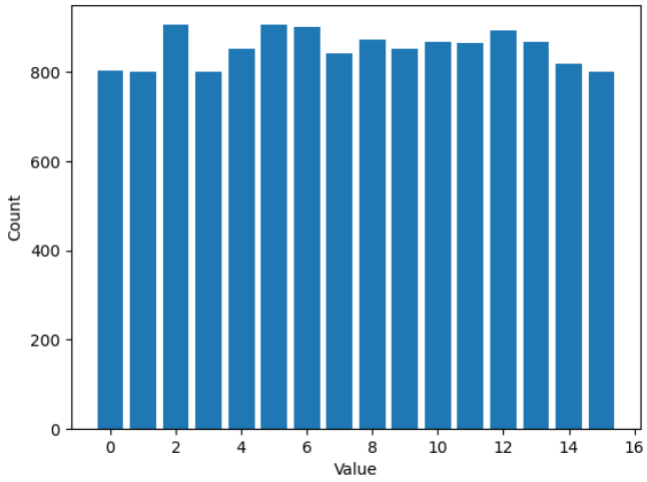


Figure 8: Histogram of 4-bit values.

Doing the chi-squared test on the 8-bit byte distribution in Figure 9 gives us $\chi^2 = 244$ which agrees with the expected value for 256 degrees of freedom of 255 ± 23.

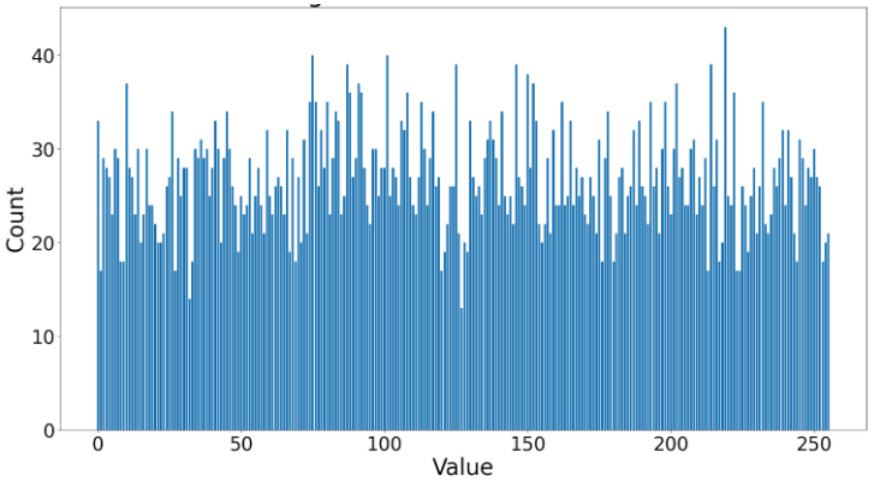


Figure 9: Histogram of 8-bit values

4 Conclusion

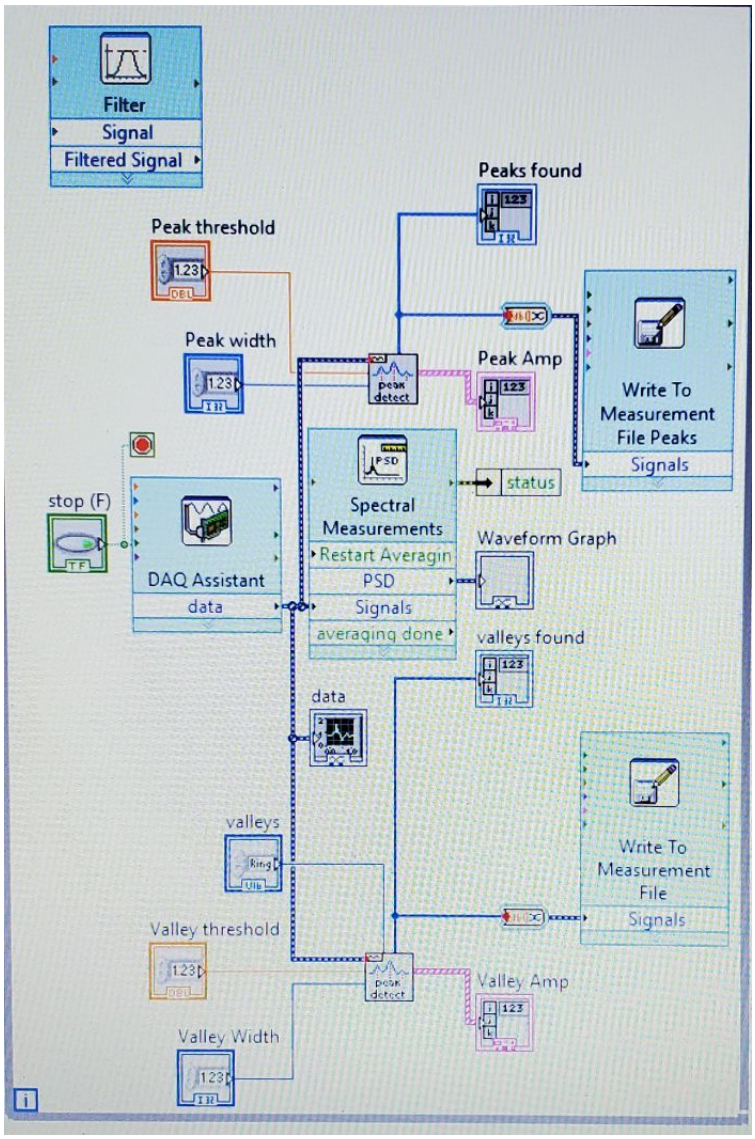
In the end, our circuit generated numbers that passed the 2d correlation test, the average value test and the χ^2 test. For the χ^2 tests; the 4 bit test yielded $\chi^2_{actual} = 14$ which agrees with $\chi^2_{expected}$=15±6, the 8 bit test yielded $\chi^2_{actual} = 244$ which agrees with $\chi^2_{expected}$=255±23. Since our data acquisition was limited to 100Hz as a result of the DAQ, this limited our ability to perform more rigorous testing due to our small sample size. Passing these tests suggests that the values we generated were truly random, and thus the circuit we created effectively generated random numbers. This further suggests that our theory about the number of avalanche cascades being random in a given sample is correct, and due to the quantum contributions to this process our circuit can be seen as using quantum systems to generate random numbers. Therefore, it is acceptable to call this a quantum random number generator, as the distribution which results is truly random much like any other quantum rng.

5 Acknowledgements

Huge thanks to Professor Paul Beale for his help. He helped with our own testing, as well as using his own testing suite on our data and sending us those results.

A Appendix

A.1 Block Diagram



A.2 Code

```
import os
import sys
import itertools
import matplotlib.pyplot as plt
from textwrap import wrap

#-----------------------------------------------------------------------
#Imports bits from .lvm file. Returns a bitstring
#-----------------------------------------------------------------------
def importFromLVM(filename):
    with open(filename) as file:
        rawfile = file.readlines()
        bits=""
        for line in rawfile[22:]:
            bits += bin(int(float(line.replace('\t','').replace('\n', ''))))[2:]
    return bits

#-----------------------------------------------------------------------
#Uses von Nuemann algorithm to debiass bits. Returns them as 0-255 decimal
#-----------------------------------------------------------------------
def vonNuemann(filename, byteSize):
    bits = importFromLVM(filename)
    pairs = wrap(bits, 2)
    biased=""
    for pair in pairs:
        if(len(pair)==2):
            if(pair[0]!=pair[1]):
                biased += pair[0]
    bytes = wrap(biased, int(byteSize)) #Change this to change the byte size
    nums=[]
    for byte in bytes:
        dec = int(byte, 2)
        nums.append(dec)
    return nums,bytes,bits

#-----------------------------------------------------------------------
#Plot decimal values for a given file
#-----------------------------------------------------------------------
def individualPlot(filename, byteSize):
    test1=vonNuemann(filename, byteSize)
    nums = test1[0]
    nums.sort()
    xAxis=[g[0] for g in itertools.groupby(nums)]
    yAxis=[len(list(g[1])) for g in itertools.groupby(nums)]
```

```
    plt.bar(xAxis, yAxis)
    plt.show()

#-------------------------------------------------------------------------
#Plot decimal values for a given file
#-------------------------------------------------------------------------
def combinedPlot(directory, byteSize):
    allnums=[]
    for dataset in os.listdir(directory):
        f = os.path.join(directory, dataset)
        allnums += vonNuemann(f, byteSize)[0]
    allnums.sort()
    #print([(g[0], len(list(g[1]))) for g in itertools.groupby(allnums)])
    xAxis=[g[0] for g in itertools.groupby(allnums)]
    yAxis=[len(list(g[1])) for g in itertools.groupby(allnums)]
    plt.bar(xAxis, yAxis)
    plt.xlabel("Value", fontsize=30)
    plt.ylabel("Count", fontsize=30)
    plt.title("Histogram of Generated Numbers", fontsize=35)
    plt.xticks(fontsize=25)
    plt.yticks(fontsize=25)
    plt.show()

#-------------------------------------------------------------------------
#2dPlot
#-------------------------------------------------------------------------
def noisePlot(directory, byteSize):
    allnums=[]
    for dataset in os.listdir(directory):
        f = os.path.join(directory, dataset)
        allnums += vonNuemann(f, byteSize)[0]

    #print([(g[0], len(list(g[1]))) for g in itertools.groupby(allnums)])
    xPoints=[]
    for point in allnums:
        xPoints.append(point)
    yPoints=[]
    for point in allnums:
        yPoints.append(point)
        yPoints.append(yPoints.pop(0))

    plt.scatter(xPoints, yPoints, color='black', marker='s')
    plt.title("2d Correlation Plot", fontsize = 35)
    plt.xlabel("$X_{k}$", fontsize=30)
    plt.ylabel("$X_{k-1}$", fontsize=30)
    plt.xticks(fontsize=25)
```

```
    plt.yticks(fontsize=25)
    plt.show()

#-------------------------------------------------------------------------------
#Serial Test
#-------------------------------------------------------------------------------
def serialTest(directory, byteSize):
    allnums=[]
    for dataset in os.listdir(directory):
        f = os.path.join(directory, dataset)
        allnums += vonNuemann(f, byteSize)[0]

    pairs = [tuple(allnums[i:i+2]) for i in range(0,len(allnums), 2)]

    xAxis=[g[0] for g in itertools.groupby(pairs)]
    yAxis=[len(list(g[1])) for g in itertools.groupby(pairs)]

    plt.bar(xAxis, yAxis)
    plt.xlabel("Value", fontsize=30)
    plt.ylabel("Count", fontsize=30)
    plt.title("Histogram of Generated Numbers", fontsize=35)
    plt.xticks(fontsize=25)
    plt.yticks(fontsize=25)
    plt.show()
#-------------------------------------------------------------------------------
#Main
#-------------------------------------------------------------------------------
if __name__ == "__main__":
    args = sys.argv
    globals()[args[1]](*args[2:])
```

A.3 2d Correlation Plots

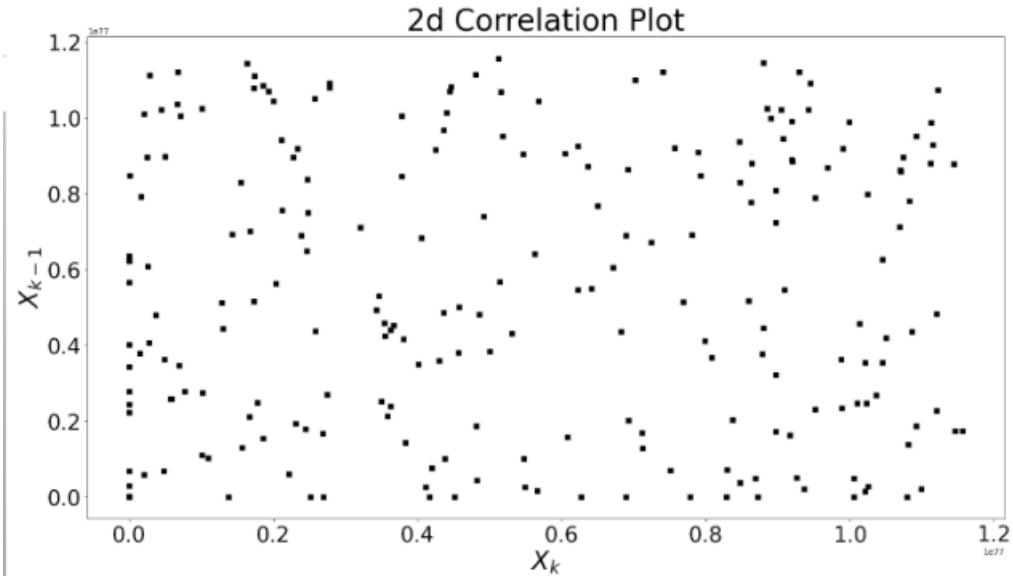


Figure 10: Caption

6 Contribution

Caleb worked on: General Formatting, Abstract, Debiasing, 2dcorrelation, Acknowledgements.
Keaton worked on: Introduction, Bibliography formatting, Digitization, Conclusion.
Both worked on: Exp setup, Results, Data, Editing and Reviewing, chi-squared test, Appendix

References

[1] Charles Platt (2015). Really Really Random Number Generator `https://makezine.com/projects/really-really-random-number-generator`.

[2] Rob Seward (Date N/A). Make your own True Random Number Generator 2 `http://robseward.com/misc/RNG2/`.

[3] Donald E. Knuth. *The Art of Computer Programming, Vol.2*. Addison-Wesley, 1998.