True Random Number Generators

Jade Geng
Bard College

True Random Number Generators

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Jade Geng

Annandale-on-Hudson, New York
May, 2023

Abstract

Quantum Random Number Generators(QRNGs), or True Random Number Generators, generate random numbers based on naturally unpredictable(or hard-to-predict) sources. Their unpredictability results in a broad application in cryptography and technology. Their sources range from nuclear decay gamma rays to cosmic rays, then to quantum optics. This thesis aims to explore various randomness sources and compare their efficiency by running a series of randomness tests. The specific setup for each random number generator will also be presented.

Contents

Dedication

To my mom, for your love and wisdom.

And to my Bardian families—Silin, Lily, Hanyi, Sarah, and Rose, for supporting me over this difficult time.

Acknowledgments

I would like to express my deepest gratitude to Professor Antonios Kontos. Thank you for being so patient and for the enlightenment you gave me. I also want to thank all the physics professors at Bard, I could not have undertaken this journey without your support.

I would like to thank all my friends for always being there. With your presence, this place has been feeling like a home to me.

1
Introduction

Random numbers make up an important part of science, technology, and our everyday life. They can be applied to simulations, cryptography, and choosing which homework problem to go over first. The Random Number Generators presented in our life are mostly ”Pseudo-random Number Generators(PRNGs)”. PRNGs are made with computer algorithms. They often start with a smaller string called ”the seed”, and expand it to create a larger string. PRNGs usually are able to generate numbers that meet the ideal random number distribution. These numbers would be sufficient for everyday use, but may not fit for other purposes. If one knows which algorithm is used, the random numbers made by PRNGs can become predictable. Thus, PRNGs can not be applied in situations such as encrypting important information or making security passwords.

Quantum Random Number Generators(QRNGs), or True Random Number Generators, on the other hand, aim for true randomness. They generate numbers based on inherently unpredictable(or very hard to predict) physical phenomena. For example, atmospheric noise, theoretically can be predicted by calculating the motion of the whole atmosphere. However, this is extremely hard to achieve. Then, given an atom that might experience a nuclear, we can never predict the exact time for that to happen. Both atmospheric noise and radioactive decay can become suitable sources of randomness for QRNGs.

In the following chapters, I will elaborate on two of the significant ways of producing true random numbers and how I attempt to use them. In the second chapter, I will show how I used sources that follow the Poisson Distribution, such as radioactive decay and cosmic rays falling from outer space, to produce truly random numbers. I will go over each randomness source in detail and present the setup of my experiment. Then, In chapter 3, I will explain my setup for using spontaneous Parametric Down-converted photons as a random number generator, and how I shifted to use another data source when my setup wasn't sufficient. In chapter 4, I will briefly introduce the randomness tests I used, and compare between sources and methods. I will show two ways of visualization and an example of well made pseudo-random number generator. In the Appendix, I will present the code I used in getting the random string out of raw data.

2
Random events that follow the Poisson distribution

2.1 Poisson Distribution

Poisson distribution describes the probability distribution of events with a known occurrence rate. The Poisson distribution function output the probability for n number of events to happen with a given time period T and occurrence rate λ.

$$P_n = \frac{(\lambda T)^n}{n!} e^{-\lambda}$$

The Poisson distribution can only be applied to events where the occurrence of one event does not alter the probability for another to happen. In the rest of this chapter, I will show several randomness sources following Poisson distribution.

2.2 Methods to produce random numbers using Poisson Distributed source

There are two ways to produce random numbers based on random events: the slow clock method and the fast clock method.[7] Both methods can be applied to the same set of data.

The fast clock method focuses on the time difference between each pulse occurs. The recorded time differences between radiation pulses will be converted into 1's and 0's, depending on if the time is odd or even. By doing this we can have a string of random binary code. 2.2.1

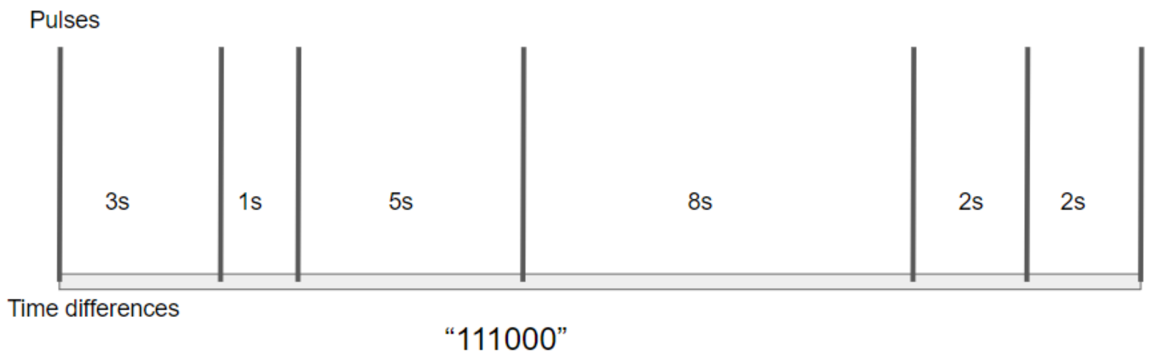


Figure 2.2.1: Fast Clock method

The slow clock method uses equal chunks of time. With the Poisson Distribution function, we can predict the time when the probability of zero decay is exactly 0.5.

$$0.5 = \frac{\lambda T^0}{0!} e^{-\lambda T} = \lambda e^{-\lambda T}$$

Then, T can be calculated:

$$T = \frac{ln\lambda - ln(0.5)}{\lambda}$$

Divide the measurement into small sets by time T, count the pulses occurs within T time. Let the 0 counts stay the same, and translate all other counts into "1". By doing this we can obtain a random binary string. 2.2.2

2.3 Radioactive sources

Radioactive decay, or nuclear decay, is the process by which an unstable atomic nucleus loses energy in the form of radiation. During this process, the nuclei could produce alpha particles, fast electrons or positrons, or electromagnetic waves. The probability of nuclear decay distributes evenly in time, which means the probability of the decay of an atom in a given period of time

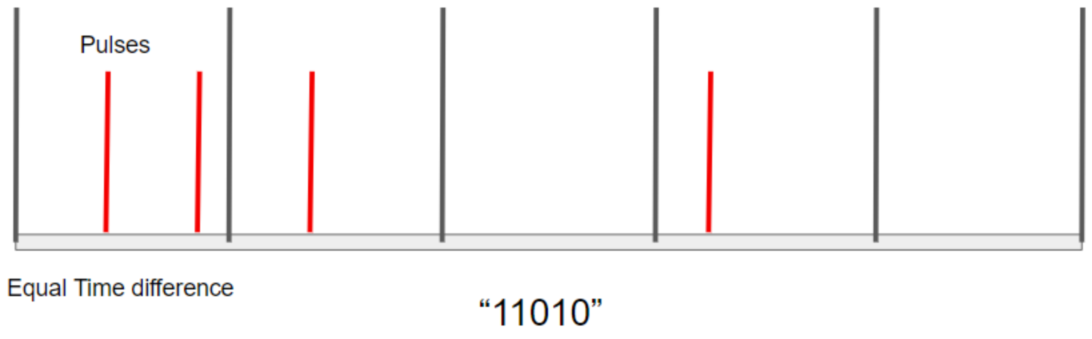


Figure 2.2.2: Slow Clock method

is constant. Besides, the decay of an atom is independent of the decay of other atoms[9]. So, nuclear decay is an ideal source of randomness. Since the individual decays are independent of each other, the probability of decay in a given time interval satisfies the Poisson Distribution:

$$P_m(T) = \frac{\lambda T^m}{m!} e^{-\lambda T}$$

Where $P_m(T)$ stands for the probability of having m decays in T time. λ stands for the average number of decays detected in 1 second.

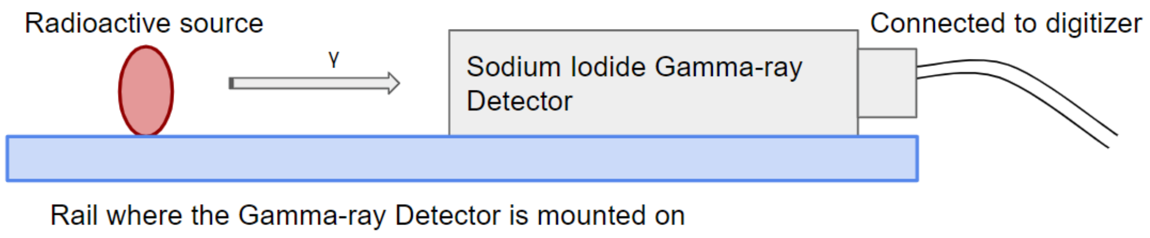

Radioactive source
γ
Sodium Iodide Gamma-ray Detector
Connected to digitizer
Rail where the Gamma-ray Detector is mounted on

Figure 2.3.1: This figure shows the basic setup for the nuclear method. Note that the gamma rays from the radioactive source actually shot in all directions.

The data acquisition process of the radioactive decay method includes:

- A computer with software CoMPASS(CAEN Multi-PArameter Spectroscopy Software) installed
- A Sodium Iodide detector

- A Waveform Digitizer(CAEN DT5790).
- various radioactive sources

we used takes down the time and energy of detected radiation pulses. We filter the data and only kept the pulse data within the predicted energy range. Dividing the total pulses by the total time taken we can have λ, the rate of decay detected. Doing this avoids calculating the decay rate and the dead time of the digitizer.

The radioactive sources we used are Cobalt-60, Cesium-137, and Sodium-22. These three radioactive sources are expected to experience beta decay, in which they emit an electron during the process. Start with Cesium-137. Cesium-137 is a radioactive isotope of cesium, commonly formed in the nuclear fission of uranium-235. It has a half-life of around 30.05 years. Cesium-137's decay produces Barium-137 and an electron(beta particle).

$$^{137}_{55}Cs \rightarrow ^{137}_{56} Ba + e^-$$

During the decay, there is a 94.6% chance for it to emit a gamma ray carrying around 661 KeV energy. Make a cut to the radiation spectrum detected using the software and only record pulses with energy peak centering 661KeV for Cesium-137.

Cobalt-60 is an artificial radioactive isotope of Cobalt. It has a half-life of about 5.27 years. Cobalt-60 undergoes beta decay and produces Nickel-60, an electron, an electron antineutrino, and two gamma rays of energy 1173 KeV and 1330 KeV. Here the cut was made around energy equal to 1173 KeV.

$$^{60}_{27}Co \rightarrow ^{60}_{28} Ni + e^- + \bar{V_e}$$

Sodium 22 is an isotope of sodium with a half-life of approximately 2.6 years. It decays into Neon-22, a positron, and a neutrino. The positron carries about 544KeV energy, and a cut has been made around energy 544KeV.

$$^{22}_{11}Na \rightarrow ^{22}_{10} Ne + e^+ + V_e$$

2.4 Cosmic rays and other entropy sources

Cosmic rays, on the other hand, can be utilized as a source of entropy. Cosmic rays are often presented in the form of relativistic ionized nuclei. The origin of most cosmic rays is believed to be not from our solar system since only a few of them show a relation to solar activity.[5] Because they originate in random places across the universe, cosmic rays can be a source of randomness. Cosmic rays carry a lot of energy due to their high speed and most of the energy were lost during the atmosphere interaction. Their interactions with atmospheric components, mostly nitrogen and oxygen, create muons. When a high energy proton hits an oxygen atom, it first creates neutral and charged pions; The neutral pions annihilate within a short time while the charged pions decay into muons and muon neutrinos. [2] Muon is a fundamental particle similar to an electron. It is a fermion and belongs to the lepton family, and carries a similar charge as an electron. The mass of a muon is around 1.884×10^{-28}kg, which is about 11% of the mass of a proton, and 200 times more massive than an electron. Muons are able to penetrate deep into the earth, which made it possible for us to detect muons inside a lab.

When muons travel through matter, it leaves a certain amount of energy. That is how we detect them using the same Sodium Iodide radiation detector. The energy loss of muon when it travels through matter is 1100 KeV 1800 KeV per g/cm^2. For detectors such as NaI scintillators, muons tend to leave greater energy due to their thickness. Thus, we need to set a lower bound on the spectrum to filter background noise.

Common background noise in the gamma-ray spectrum comes from the decay of certain atoms in building material or air. The common background noise sources include potassium-40, thorium, uranium, and products of thorium, uranium decays. Among these sources, P-40 produces a gamma ray with the highest energy since it emits electrons with 1314 KeV and gamma-rays with 1460 KeV energy[8]. So, we set the lower bound of the filter to be 1500 KeV to ensure what we counted are mostly real muons. The count rate of muons in our laboratory is relatively low We only acquired 190,367 entries over a week of time.

The background noise inside the laboratory can also be a source of randomness. These gamma rays could come from building materials, a piece of banana, or electronic devices in the lab. The range of gamma-ray spectrum we selected is from 10KeV to 1500KeV. Considering some of these gamma rays might be from electronic devices, their occurrence might have a uniform frequency. In other words, numbers generated with them could form repetitive patterns. Choosing this wide energy range can to some extent erase the repetitive patterns. These input pulses can be treated using the same methods used on radioactive decay and cosmic rays.

3
Random numbers generated from single photons

This chapter describes the procedure of producing random numbers using generated single photons. There could be multiple methods to produce random numbers using single photons. One is to use correlated single photons. The randomness of this method comes from the uncertainty of a single photon passing through a 50:50 beamsplitter. Firstly, we have pairs of correlated single photons, emitted in two paths. Set the beamsplitter on one of the paths so that half of the photons go through, and the other half is reflected in another direction. For each photon passed the beamsplitter, add a '1' to the result sequence; for every photon reflected, add a '0'.3.0.1 By doing this we can get a random sequence through single photons. However, this method did not work out. The randomness of our result sequence still comes from photons passing or not passing the beamsplitter, but we are not certain these photons are correlated.

3.1 Spontaneous Parametric Down-conversion

To understand the purpose of the correlated photon random number generator setup, it is necessary to introduce the Spontaneous Parametric Down-conversion of photons. This process takes a photon, converts the photon into a pair of photons with lower energy, then emits the photon pair at a small angle with respect to the propagation direction of the original photon.

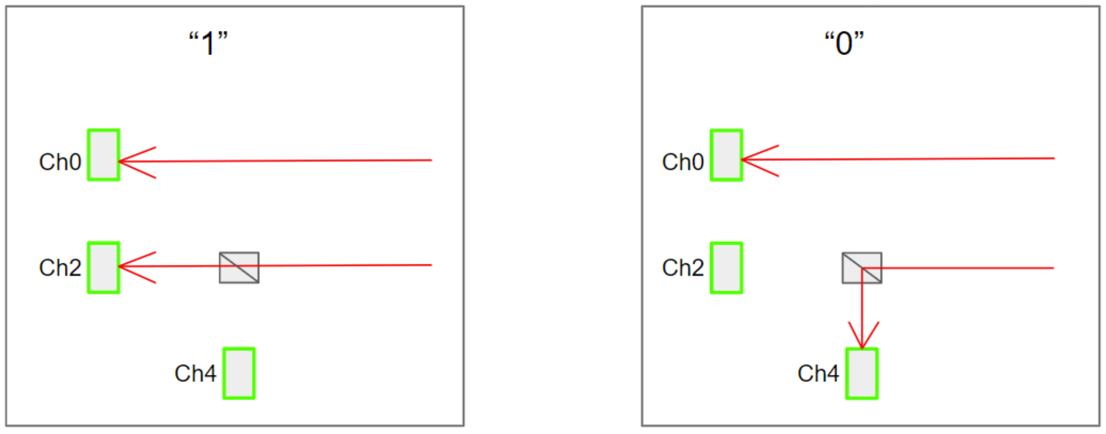


Figure 3.0.1: The method of using correlated photons as a randomness source. Each time Channel 0 and Channel 2 detect a coincidence, add a '1' to the sequence; Each time Channel 0 and Channel 4 detect a coincidence, add a '0'

This is due to the process that when light travel through a medium with different indices of refraction, its wavelength alters. For example, for light with wavelength λthat travels from vacuum to a medium with an index of refraction n, its wavelength changes into λ/n. Then, the momentum of the photon after entering this medium becomes $p = 2\pi n\hbar/\lambda$. By the conservation of momentum we have:

$$p_0 = p_1 + p_2 \tag{3.1.1}$$

Let the direction of propagation of the incoming light be the horizontal axis. By the conservation of momentum, the total vertical component of the momentum should add up to 0:

$$0 = \frac{n_1}{\lambda_1} sin\theta_1 - \frac{n_2}{\lambda_2} sin\theta_2 \tag{3.1.2}$$

Then, for the horizontal component, we have:

$$\frac{2\pi n_0 \hbar}{\lambda_0} = \frac{2\pi n_1 \hbar}{\lambda_1} cos\theta_1 + \frac{2\pi n_2 \hbar}{\lambda_2} cos\theta_2 \tag{3.1.3}$$

We know that the energy of converted photon pair must add up to the energy of the original photon

$$E_{original} = E_1 + E2 \tag{3.1.4}$$

Which, with $E = frachc\lambda$, can be written as

$$\frac{1}{\lambda_0} = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} \tag{3.1.5}$$

Assume the generated photon pair have similar energy/wavelength, and the photon pair transit through the same kind of medium. Equation(4.1.2) implies that angles θ_1 and θ_2 are equal. Thus, equation(4.1.3) can be reduced to

$$n_0 = n_1 cos\theta_1 \tag{3.1.6}$$

[6] The process above is enabled by using a non-linear crystal, for example, a BBO(beta Barium borate) crystal. BBO crystals are birefringent, meaning they have different indices of refraction for different polarization and propagation directions of light. The index of refraction for light polarized parallel to its optical axis is given by:

$$n_e = \left(2.3753 + \frac{0.01224}{(\lambda/1\mu m)^2 - 0.01667} - 0.01516\,(\lambda/1\mu m)^2\right)^{1/2} \tag{3.1.7}$$

Then, the index of refraction for light with polarization perpendicular to its optical axis is:

$$n_e = \left(2.7359 + \frac{0.01878}{(\lambda/1\mu m)^2 - 0.01822} - 0.01354\,(\lambda/1\mu m)^2\right)^{1/2} \tag{3.1.8}$$

The laser we use has wavelength $\lambda = 405$, For $\lambda = 405nm$, we get $n_{e405} = 1.568$, $n_{o405} = 1.6923$. For $\lambda = 810nm$, $n_{e810} = 1.5461$ and $n_{o810} = 1.6611$.

These are the indices of refraction when the crystal axis is parallel to the polarization of incident light(perpendicular to its propagation direction). The index of refraction also depends on the angle between the crystal's optical axis and the light's propagation direction, α

$$n_{e,\alpha} = \left(\frac{cos^2\alpha}{n_o^2} + \frac{sin^2\alpha}{n_e^2}\right)^{-\frac{1}{2}} \tag{3.1.9}$$

Equation (4.1.9) tells us that for a specific 29.1 °, the $n_{e,\alpha}$ for 405nm light is 1.6602. By equation(4.1.6) we have

$$\theta_1 = sin^{-1}\left(\frac{n_{ep,\alpha}}{n_{o810}}\right) \tag{3.1.10}$$

Use $n_{ep,alpha} = 1.6602$ and $n_{o810} = 1.6611$ we can have $\theta_1 = 1.886°$. When the 810nm light exits the crystal, its angle with respect to the horizontal axis, θ_2, is determined using Snell's law

$$\frac{sin\theta_1}{sin\theta_2} = \frac{n_{air}}{n_{o810}} \tag{3.1.11}$$

$$\theta_2 = sin^{-1}\left(\frac{sin(\theta_1)}{\frac{n_{air}}{n_{o810}}}\right) \tag{3.1.12}$$

Which gives θ_2 to be around 3°.

3.2 Setup and alignment

3.2.1 ideal setup

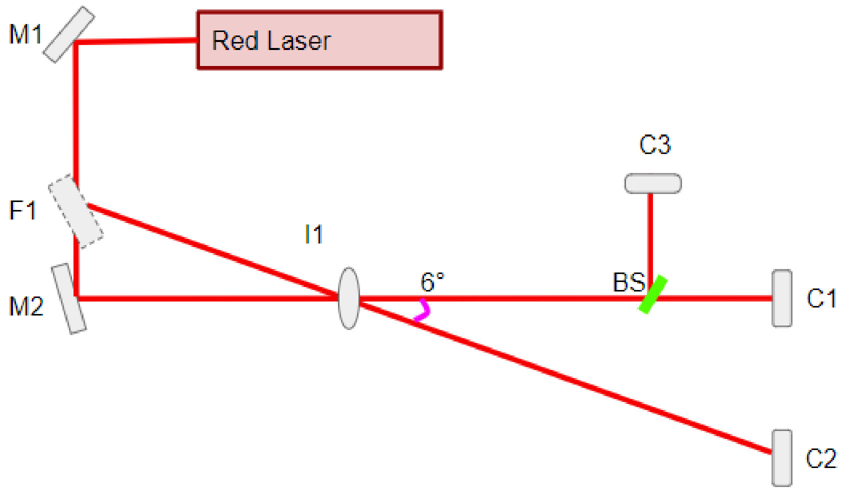


Figure 3.2.1: This figure shows the position of each component in the setup.

The ideal experiment looks forward to generating Spontaneous Parametric Down-converted photons. Figure3.2.2 illustrates the position of each component. (insert pic here)

Here is the general setup procedure for the ideal experiment:

First of all in order to make all light travel in the horizontal plane, all the optical parts need to be on the same level, for this experiment, we chose 10.5cm.

Then, set up the alignment red laser. Because the generated single photons are outside the visible light spectrum, they cannot be directly observed. The alignment laser simulates the pathway of down-converted photons. We know that there is the ±3°angle between down converted pairs and the original beam. Set the laser and two mirrors(labeled M1 and M2) as shown. Then, put an iris(I1) at the position shown. Set a fiber collimator(C1) mounted on an adjustable mount at 1 meter straight away from the iris, as shown in the picture3.2.1.

Then, put a flipper mirror(F1) at the position shown in the figure(No.undecided). The light directed by the flip mirror should go through I1, then reach another collimator(C2). The angle between the flip mirror path and the previous path should make be 6°.

Set a third collimator(C3) at the place shown in figure(). Set the 50:50 beam splitter on the path of M1-C1 to reflect half of the beam into C3. Attach bandpass filters to the collimators. Adjust the mount to reflect the red beam back to the laser. Then, remove the filters. Adjust the mount so that the light coming out from the other side of the fiber gets maximized. Do the above for all three collimators.

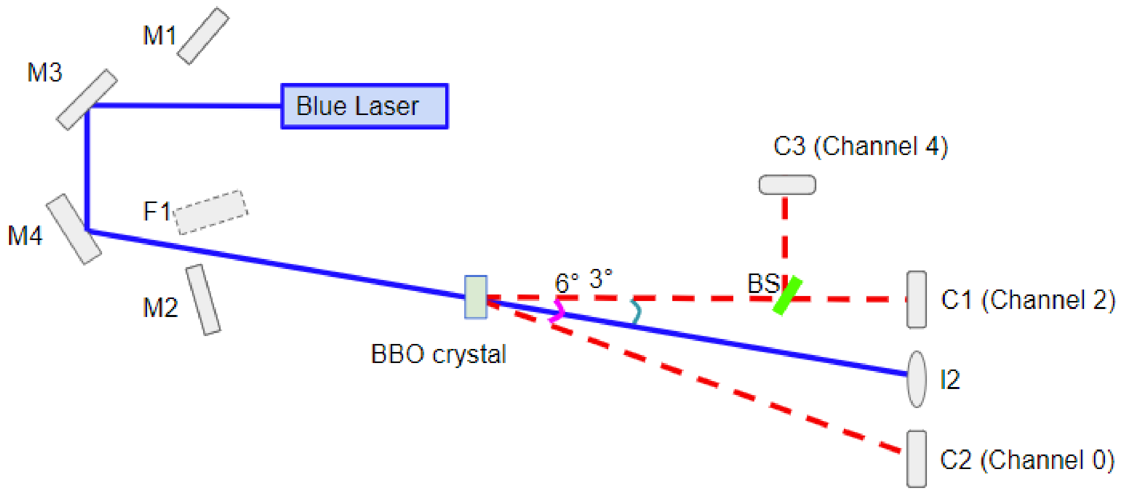


Figure 3.2.2: This figure shows the final setup of the single photon random number generator

Next, put an iris(labeled as I2) between the two collimators. Make sure the angle C1-I1-I2 and angle I2-I1-C2 are equal to 3°. Put the blue laser at the place shown. Add two mirrors to reflect the blue laser to go across both irises.

Finally, replace iris1 with the BBO crystal. The crystal we used has been cut by its manufacturer so there is already a 29°angle between the optical axis and the input propagation direction. When mounting the crystal, we chose to use a rotational mount. Adjust the mount so that the reflection from the crystal goes back to the blue laser.

The randomness of this experiment will come from the 50% probability for single photons to pass through the beam splitter. A coincidence between C1 and C2 should indicate a photon has passed the beamsplitter, and a coincidence between C1 and C3 should indicate a photon gets reflected by the beam splitter. In order to generate a random string from the setup, mark a 0 every time a coincidence between C1 and C2 occurs, and mark a 1 when a coincidence between C1 and C3 occurs.

3.2.2 What really happened

At this point, we are ready to begin data acquisition. The process should be the following: attach the band pass filter to filter out any non-810nm light, then connect the collimators to single photon counters. The single photon counters are super sensitive to light. So, for the purpose of not causing damage to them, never have them connected to power while the room light is on. The detectors should be connected to the digitizer with BNC terminators to eliminate signal reflections. A computer with CoMPASS software installed can read the count rate of single photons from the digitizer. The channel number each collimator is connected to is labeled in Figure3.2.2.

Ideally, we can see a count per second for all three channels to be around 10K 20K cps. However, only the dark count occurs, which is around 1k counts per second. Redoing the alignment or rotating the crystal does not work in changing the situation. This failure could be due to misalignment or the crystal working insufficiently. Therefore, we changed to using another randomness source: the random coincidences between random photon inputs. Firstly we need to remove the bandpass filter from the collimators, we can see the reading of some random single photons. The count rate of these photons is sensitive to the rotation angle of the crystal, so we speculate they are also photons converted by the crystal, but with various wavelengths. These photons will be our source of randomness.

To begin with this process, adjust the mounts so that the count rate of channels 2 and 4 are approximately equal–around 150K cps. Then, set the "time selection" of the software to "Channel_ref and other". Select channel 0 to be the reference channel. The channel number and the collimator they refer to are shown in Figure3.2.2. Next, set the coincidence window to 16ns. Hence, once channel 0 detects 1 signal at time t, it will look for signals from the other two channels that appear no later than t+16ns.

Since the photon inputs are not really correlated, we need to consider the accidental coincidence rate between Ch0&Ch2 or Ch0&Ch4. This random coincidence happens when two photons are detected within the time window. The time window is the greatest time gap allowed between two input signals for them to be determined in coincidence. The selection of the time window, together with the count rate of the two detectors, determines the rate of random coincidence:

$$R_{acc} = R_1 R_2 \Delta T \tag{3.2.1}$$

Here, R_1 and R_2 are the rates of singles occurring through each channel. ΔT is time window, and R_{acc} is the random coincidence rate. By equation 4.2.1, given $R_{channel0} = 100K$, $R_{channel2=150K}$, time window $\delta T = 16ns$, the rate of random coincidence equals to 240 cps. This number matches the actual count rate.

4
randomness tests, results, and random number visualization

4.1 Brief Introduction on Randomness Tests

The tools we used in determining the randomness of the binary strings are NIST's randomness tests. Each test focuses on a different aspect of "being random". For each test, it takes a binary string, and from the string, it generates a "P-value" range from 0 to 1. A larger P-value means the string is more "random" and a smaller P-value suggests it's less random. The standard of generating the P-value depends on how close certain aspect of the string is to the standard random sources. We will briefly introduce the tests in the following sections[3]:

1. **Frequency Test** focuses on the ratio between 1's and 0's in a binary string. It compares the number of 1's and 0's inside a string. All the other randomness tests are dependent on this test. This test is based on the null hypothesis that in a sequence of independent identically distributed Bernoulli random variables, the ratio between 1's and 0's should be 1:1. The test evaluates how close the ratio is to 1:1 with respect to the length of the sequence. For example, the test gives $P = 0.6547$ for 11100 and $P = 4.1533^{-5}$ for 252 "1" and 168 "0". The rule of decision is if the P-value drops under 0.01, the string will be considered "non-random". For both sequences, the ratio between 1's and 0's are 3:2, but

for a sequence with 420 elements, the possibility of having this ratio is way too small. So, it is determined as ”non-random”. The frequency test needs to be performed before doing any other tests in the test suite.

2. **The frequency Test within a Block** first divide the input string into several smaller blocks, then checks if the ratio is close to 1:1 in these blocks. A small P-value(less than 0.01) suggests that the ratio is too far away from 1:1 in at least one of the blocks. So, the string ”111111111111111111111111111111111111111 1111111111100000000000000000000000000000 000000000000000000000...” will pass the frequency test because it has a perfect ratio in general, but has too many 1’s or 0’s in localized blocks.

3. **Runs Test** defines a run as a string of length k composed with similar bits. The Runs Test checks if the number of runs is too much or too few for the sequence to be random. It also looks for if the sequence is switching between 1’s and 0’s too frequently.

4. **The Longest Run of Ones in a Block Test** is based on the longest run of 1’s in localized blocks. It divides the input sequence into smaller blocks, then check if the length of the longest run of 1’s is within the expectation of being random. A small P-value suggests there are too many 1’s in a cluster.

5. **The Binary Matrix Rank Test** divides the input sequence into matrices and tests if there exists linear dependence among the matrices. The size of matrices made by the test suite we use is 32×32. For this size of matrices, the recommended size of the input sequence is at least 38,912 bits.

6. **Discrete Fourier Transform (Spectral) Test** serves the purpose of checking repetitive patterns in the input sequence. The test performs discrete Fourier transform on the input sequence. Then, it calculates if the number of peaks in the resulting plot is too far away from the expectation value-below the 95% or above the 105% of the expectation.

7. **The Non-overlapping Matching Test** pays attention to repetitive elements in a random string. The tests take templates from a library, record the number of occurrences of the templates, then check if they are presented too frequently or too few. When the test compares the input sequence with an m-bit template, it creates an m-bit "window" that moves through the sequence. Once the part of the input in the window matches the template, the window moves the m-bit forward. Then, The Non-overlapping Template Matching Test checks if the number of occurrence of the template obey the normal distribution.

8. **The Overlapping Template Matching Test** focuses on the same aspect of being random as the Non-overlapping Test. However, the window of the Overlapping Template Matching test only moves 1 bit forward it finds the template. Hence, the template it uses is, by default, m-bit of 1's.

9. **The Maurer's "Universal Statistical" Test** focus on the bits in between repetitive patterns. Then check how small the sequence can be compressed into without losing information. If the size of the compressed sequence is too small, the sequence must have too many repetitive parts and hence will be considered non-random. The test compares the compressed length with the computed expected length. The recommended length of input for this test is no less than 387,840.

10. **Linear Complexity Test** looks at how complex the sequence is. It does this by checking the length of the linear feedback shift register (LFSR). The length of the shortest LFSR is determined using the Berlekamp-Massey algorithm.

11. **Serial Test** focuses on the number of occurrences of all existing overlapping patterns in the input sequence. For an ideal random sequence, the chances of occurrence of all m-bit strings are equal. The test checks the frequency of occurrence to see if they are close to being equal. If set m to 1, the test will be the same as the Frequency Test.

12. **Approximate Entropy Test** also looks at the frequency of all existing overlapping patterns. But, it compares the frequency of adjacent patterns with the expected frequency in a random string. A small P-value indicates that repetition occurs too frequently.

13. **Cumulative Sums (Cusum) Test** looks at the displacement of random walk based on the input sequence. Every 1 in the input would be a step forward and every 0 becomes a step backward. For an ideal random list, the final displacement should be close to 0.

14. **Random Excursions Test** looks at the number of cycles that have K visits. A cycle of random walks is defined here as the list of steps it takes to start from the origin and eventually come back. The number of visits is determined by how many times the walk passed the same spot within a cycle. The test compares the number of visitors to an expectation value of a random sequence.

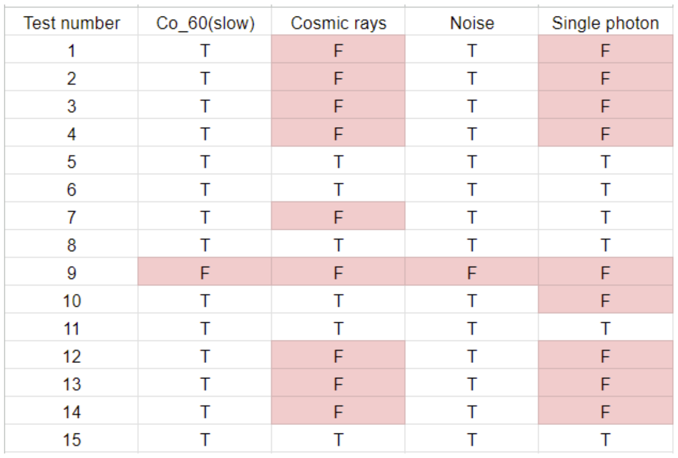
15. **Random Excursions Variant Test** focuses on the number of times a certain state is reached in a random walk based on the input sequence. The test compares this number to the number of times expected for a random string.

4.2 Test results and conclusion

Figure4.2.1 is a table showing the test results of random numbers generated from different sources. In terms of the number of tests passed, we can see that numbers from Co_60 nuclear decay and background noise performed well and are mostly random. The reason why they failed Maurer's "Universal Statistical" Test is this test requires more entries than we can provide. The input's size is limited by that of the cosmic ray sequence.

The test results suggest that for cosmic ray source, there's an imbalance between 1's and 0's, and an excessive amount of the same bit clusters together frequently. As cosmic rays, by definition, should be a valid source of randomness, these defects could come from the data set being too small. A longer data acquisition time could give a more accurate average count rate, thus generating better random numbers.

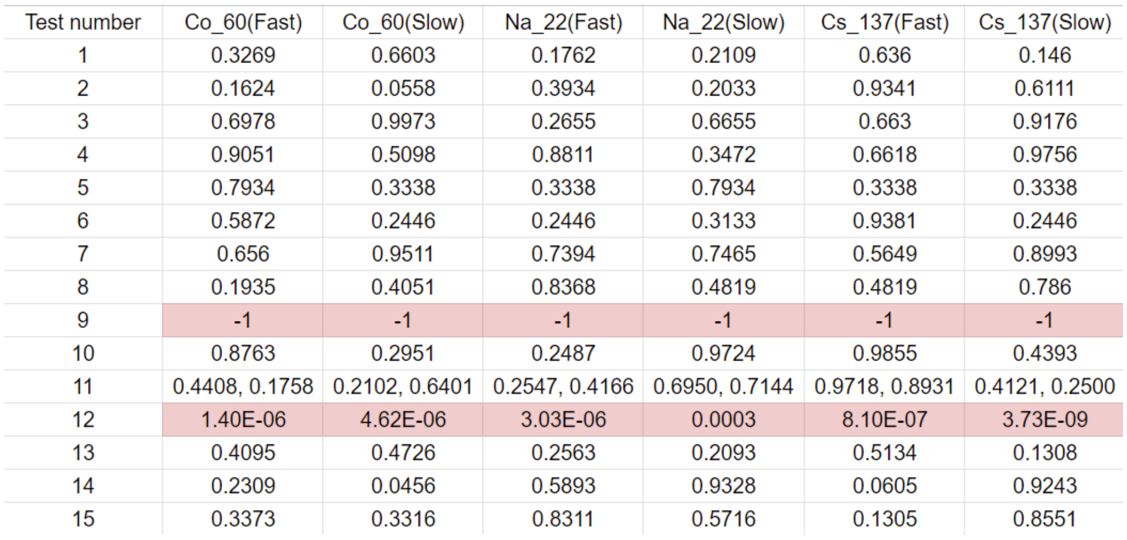
The random numbers generated from single photons also have an imbalance between 1's and 0's. Meanwhile, it also lacks linear complexity. The interpretations of such results are, firstly, because these numbers are not made based on the correlated photons as the experiment previously aims for. The count rates from the two channels are not exactly equal since the photons might have different angles than the correlated ones. Then, since the data was collected without the bandpass filter, the data might be contaminated by background light sources, which might be emitted with a constant frequency. This frequency could destroy the linear complexity of the random string.



Test number	Co_60(slow)	Cosmic rays	Noise	Single photon
1	T	F	T	F
2	T	F	T	F
3	T	F	T	F
4	T	F	T	F
5	T	T	T	T
6	T	T	T	T
7	T	F	T	T
8	T	T	T	T
9	F	F	F	F
10	T	T	T	F
11	T	T	T	T
12	T	F	T	F
13	T	F	T	F
14	T	F	T	F
15	T	T	T	T

Figure 4.2.1: This table lists how random numbers generated from different sources behave under the randomness tests. Here T means the string has passed the test and F means failure. For the first 4 sources, the numbers were made using the slow clock method. All sequences have the same length of 20,000 bits.

Figure4.2.2 compares the P-Value of numbers generated with the fast clock method and slow clock method. These sequences fail to pass Approximate Entropy Test because their lengths reduce to 3500 bits. Between 3 pairs of random number sequences. Each pair are made from the same raw data, but using different methods. From the table, we can see no significant difference or trend between the P-Values from the two methods. So, we can conclude that in terms of randomness, both methods are valid. However, in terms of efficiency, the fast clock method takes at least twice the amount of raw entries than the slow clock method, which is why the length of sequences used here has decreased. So we conclude the slow clock method is more efficient.



Test number	Co_60(Fast)	Co_60(Slow)	Na_22(Fast)	Na_22(Slow)	Cs_137(Fast)	Cs_137(Slow)
1	0.3269	0.6603	0.1762	0.2109	0.636	0.146
2	0.1624	0.0558	0.3934	0.2033	0.9341	0.6111
3	0.6978	0.9973	0.2655	0.6655	0.663	0.9176
4	0.9051	0.5098	0.8811	0.3472	0.6618	0.9756
5	0.7934	0.3338	0.3338	0.7934	0.3338	0.3338
6	0.5872	0.2446	0.2446	0.3133	0.9381	0.2446
7	0.656	0.9511	0.7394	0.7465	0.5649	0.8993
8	0.1935	0.4051	0.8368	0.4819	0.4819	0.786
9	-1	-1	-1	-1	-1	-1
10	0.8763	0.2951	0.2487	0.9724	0.9855	0.4393
11	0.4408, 0.1758	0.2102, 0.6401	0.2547, 0.4166	0.6950, 0.7144	0.9718, 0.8931	0.4121, 0.2500
12	1.40E-06	4.62E-06	3.03E-06	0.0003	8.10E-07	3.73E-09
13	0.4095	0.4726	0.2563	0.2093	0.5134	0.1308
14	0.2309	0.0456	0.5893	0.9328	0.0605	0.9243
15	0.3373	0.3316	0.8311	0.5716	0.1305	0.8551

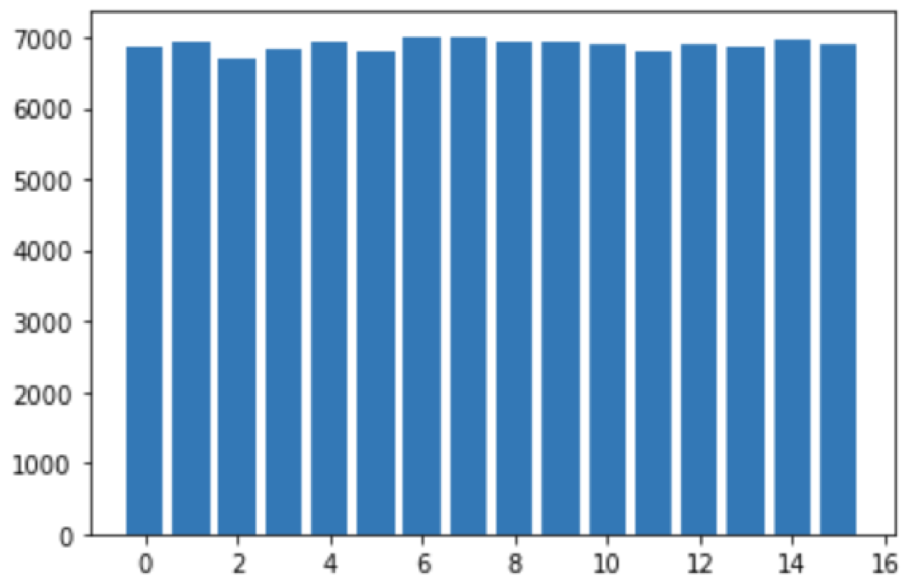
Figure 4.2.2: This table aims to compare the randomness same raw data processed by the fast clock method and slow clock method. The P-Values generated by each test are shown in this table. All the sequences tested were cut into 3500 bits. The light pink cell filling suggests a failure of the randomness test.

4.3 Visualizations of Random Numbers

Aside from the randomness tests, the random numbers generated can be visualized so that the audience can judge its randomness by simply looking at it.

Figure 4.3.1 is the bar plot of numbers versus their number of occurrences in a random sequence. From this bar plot, we can clearly tell that the frequency of each decimal number is evenly distributed, thus this sequence is likely to be random.

Figure 4.3.2 represent another way to visualize randomness. A bit map is a picture in which every pixel represents a bit[1]. Bit maps are great tools to discover repetitive patterns or imbalances between 1's and 0's. The left figure in Figure 5.3.2 is from our nuclear random number group-the Cs_137, while the right figure is made from Python's pseudo-random number generator. There is a wide difference between the existing pseudo-random number generators. The pseudo-random produced by Python belongs to the high-quality ones-they look exactly random to our eyes. However, this sequence did not pass the Random Excursions Test, which suggests it is not truly random. Therefore, although these visualization methods provide straightforward ways to see the random sequences, we can not tell if the sequence is truly random or generated by algorithms.



Figure

Figure 4.3.1: This figure shows the number of occurrences of each number after a binary sequence is converted to a decimal sequence. Then the range of the decimal numbers goes from 0 to 15. The source of this visualization is Cs_137 energy peak centering at 661kV, taken within 10 minutes. These numbers are generated using the slow clock method

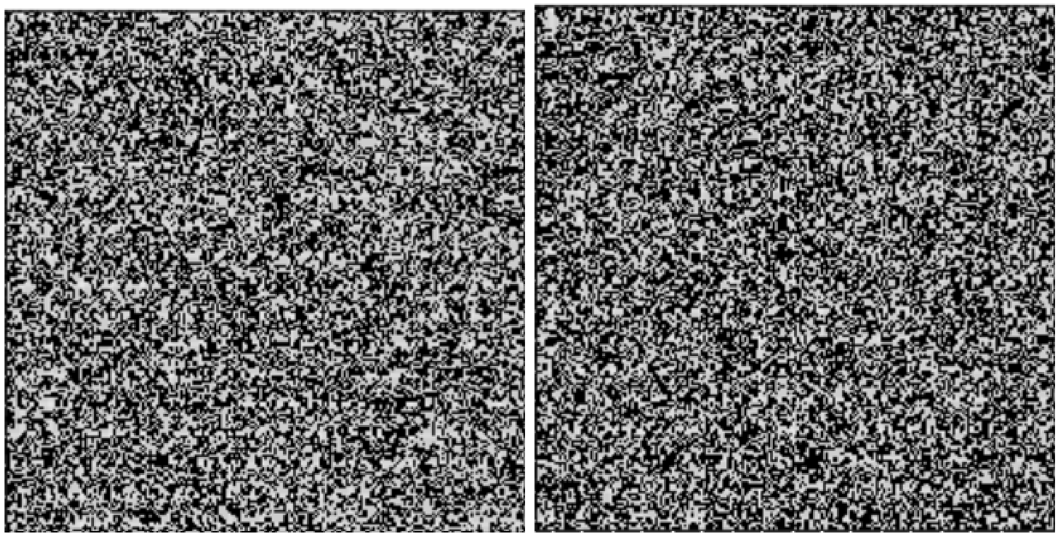


Figure 4.3.2: The left figure is a visualization of a random sequence from Cs_137 using the slow clock method; The right figure shows the visualization of a pseudo-random sequence generated with the Python Random module. Each black cell represents a 1 and each white cell represents a 0. Both are 200×200

Appendix A
Fragments that may enhance understanding this thesis

A.1 Some Tests that I want to describe a little bit in-depth

Frequency test focuses on the ratio between 1's and 0's in a binary string. It compares the number of 1's and 0's inside a string. All the other randomness tests are dependent on this test.

To begin with, the input binary code was converted from 1's and 0's into 1's and -1's, then was added together. Let the new variable be S_n

$$11100000 \rightarrow S_n = 1 + 1 + 1 + (-1) + (-1) + (-1) + (-1) + (-1) = -2 \tag{A.1.1}$$

Then calculate the test statistics, where n is the length of the binary string.

$$s_{obs} = \frac{|S_n|}{\sqrt{n}} = 0.707 \tag{A.1.2}$$

Then, apply the erfc(Complementary Error Function) function to the test statistics:

$$P = erfc(\frac{S_{obs}}{\sqrt{2}}) = 1 - \frac{2}{\sqrt{\pi}} \int_0^{\frac{S_{obs}}{\sqrt{2}}} e^{-u^2} = 0.4551 \tag{A.1.3}$$

This function is originally used in digital transmission and describes the probability of error during transmission. It decreases when S_{obs} increases.

The Non-overlapping and Overlapping Template Matching Tests process their data in different ways. The Non-overlapping Template Matching Test checks if the number of occurrence of the template obey the normal distribution. The expected mean and variance are shown below:

$$\mu = \frac{n - m + 1}{2^m} \tag{A.1.4}$$

$$\sigma^2 = n\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right) \tag{A.1.5}$$

While the Overlapping Template Matching Test is based on the discovery that, for a random string of numbers, the number of repeated occurrences of certain patterns obeys the Pó 1ya-Aeppli distribution, which is also known as the Geometric Poisson distribution.

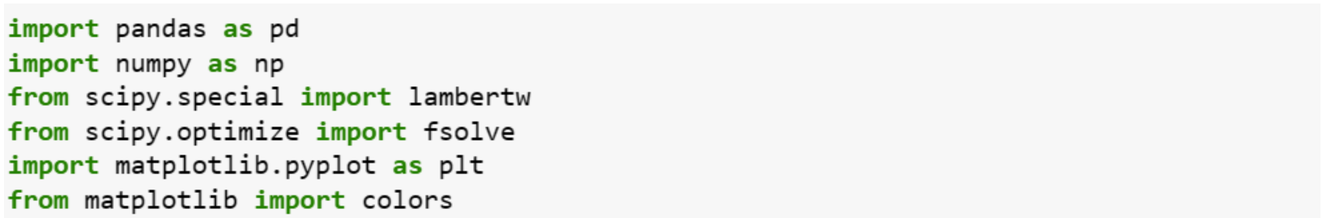
$$f_N(n) = Pr(N = n) = \begin{cases} \sum_{k=1}^{n} e^{-\lambda}\frac{\lambda^k}{k!}(1-\theta)^{n-k}\theta^k \binom{n-1}{k-1}, n > 0 \\ e^{-\lambda}, n = 0 \end{cases} \tag{A.1.6}$$

A.2 Randomness Test Suite

The test suite code I used to check the sequences I produced is Steven Kho Ang's Python adaptation.[4]

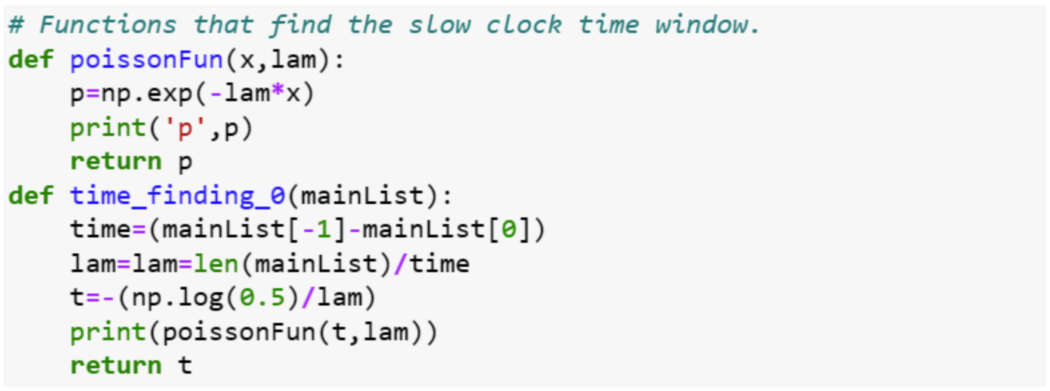
A.3 Code used

This section presents the code we used in composing the random sequences. All code used is written in Python.



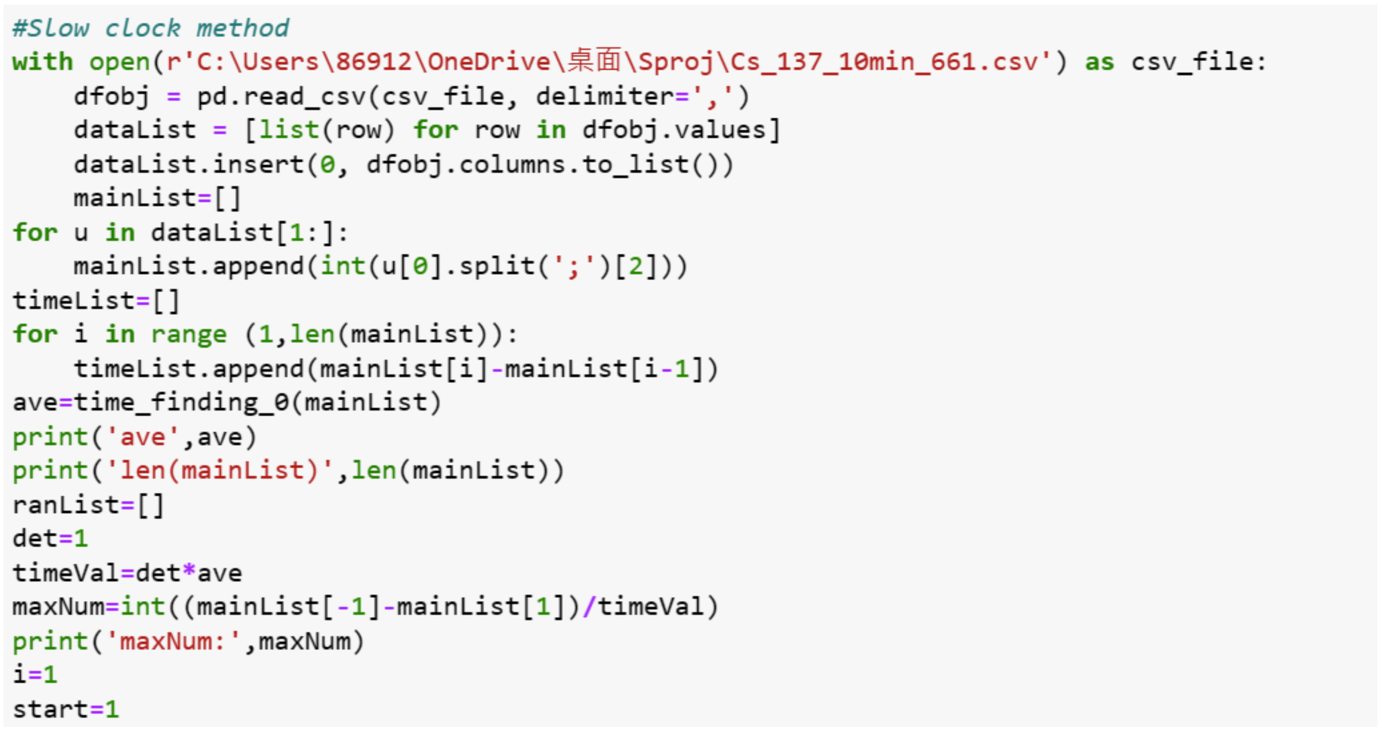
```
import pandas as pd
import numpy as np
from scipy.special import lambertw
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
from matplotlib import colors
```

Figure A.3.1: This is a picture showing the libraries imported for the code used in generating random numbers following Poisson distribution



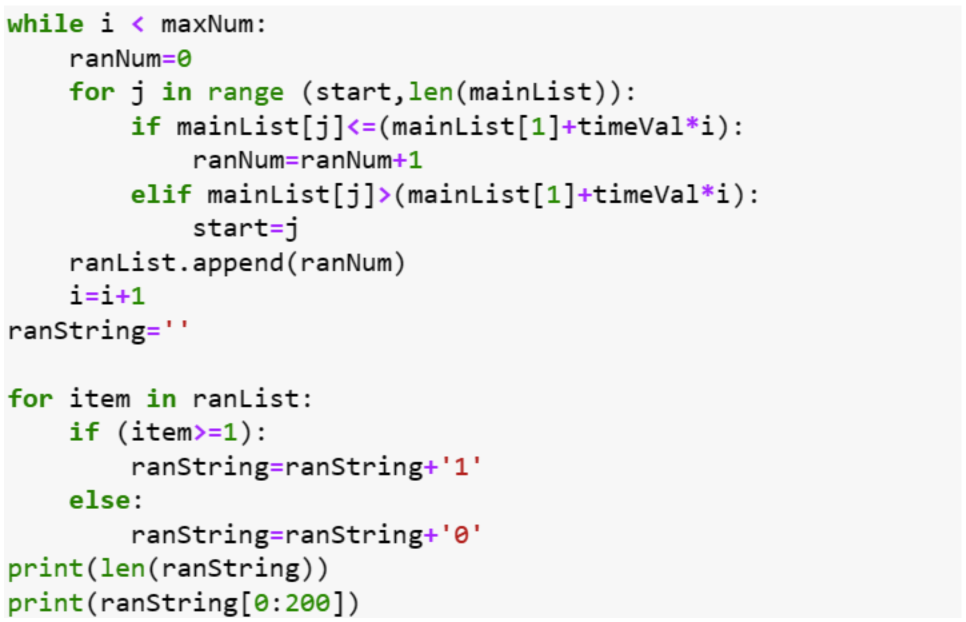
```python
# Functions that find the slow clock time window.
def poissonFun(x,lam):
    p=np.exp(-lam*x)
    print('p',p)
    return p
def time_finding_0(mainList):
    time=(mainList[-1]-mainList[0])
    lam=lam=len(mainList)/time
    t=-(np.log(0.5)/lam)
    print(poissonFun(t,lam))
    return t
```

Figure A.3.2: These pictures show the code used for the slow clock method



```python
#Slow clock method
with open(r'C:\Users\86912\OneDrive\桌面\Sproj\Cs_137_10min_661.csv') as csv_file:
    dfobj = pd.read_csv(csv_file, delimiter=',')
    dataList = [list(row) for row in dfobj.values]
    dataList.insert(0, dfobj.columns.to_list())
    mainList=[]
for u in dataList[1:]:
    mainList.append(int(u[0].split(';')[2]))
timeList=[]
for i in range (1,len(mainList)):
    timeList.append(mainList[i]-mainList[i-1])
ave=time_finding_0(mainList)
print('ave',ave)
print('len(mainList)',len(mainList))
ranList=[]
det=1
timeVal=det*ave
maxNum=int((mainList[-1]-mainList[1])/timeVal)
print('maxNum:',maxNum)
i=1
start=1
```

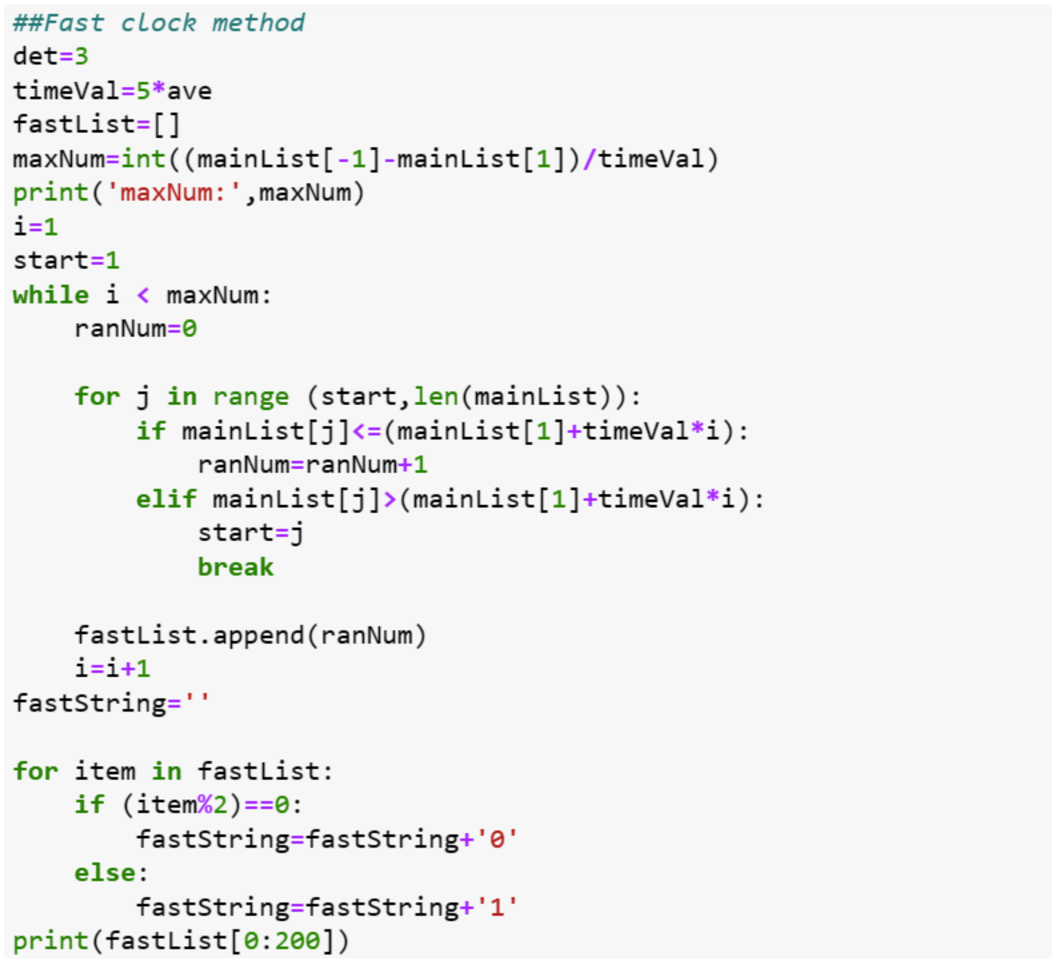
Figure A.3.3: This picture shows the code used for the slow clock method



```
while i < maxNum:
    ranNum=0
    for j in range (start,len(mainList)):
        if mainList[j]<=(mainList[1]+timeVal*i):
            ranNum=ranNum+1
        elif mainList[j]>(mainList[1]+timeVal*i):
            start=j
    ranList.append(ranNum)
    i=i+1
ranString=''

for item in ranList:
    if (item>=1):
        ranString=ranString+'1'
    else:
        ranString=ranString+'0'
print(len(ranString))
print(ranString[0:200])
```

Figure A.3.4: This picture shows the code used for the slow clock method



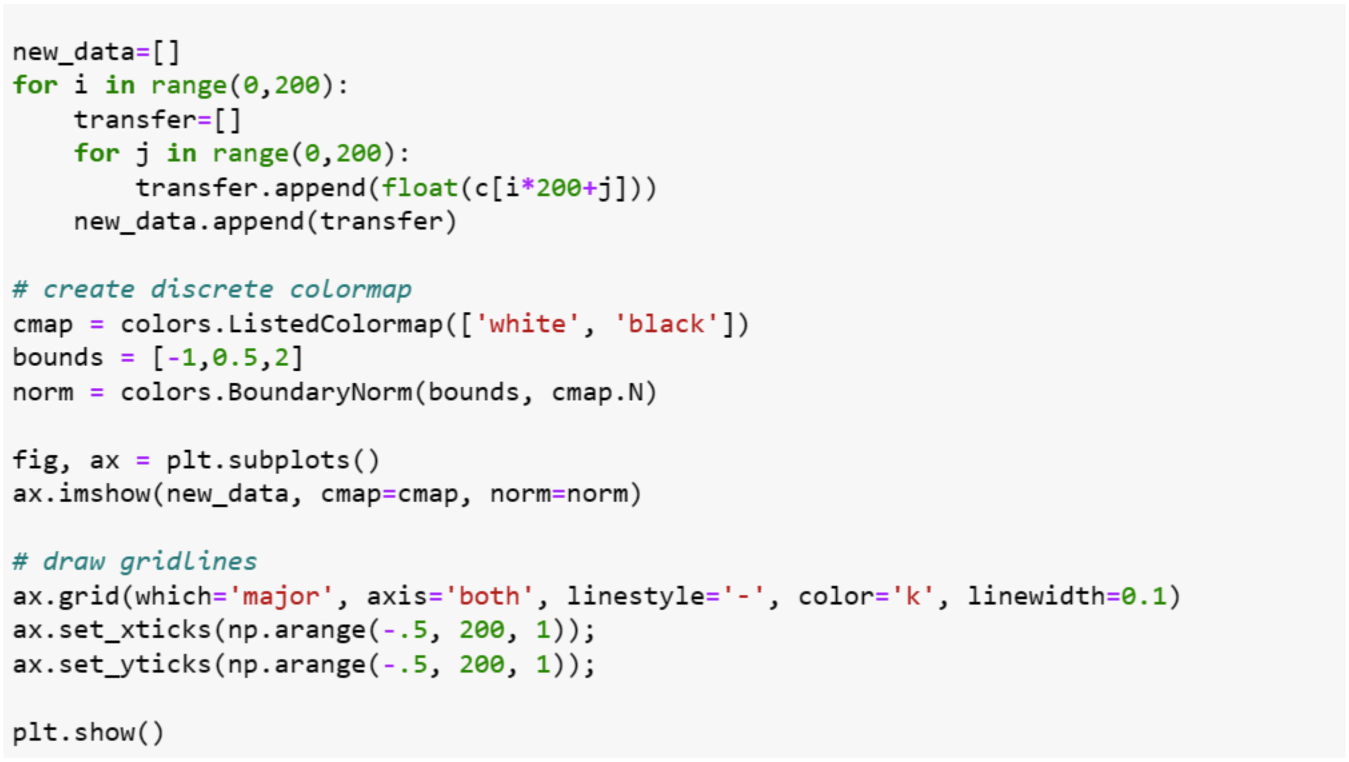
```
##Fast clock method
det=3
timeVal=5*ave
fastList=[]
maxNum=int((mainList[-1]-mainList[1])/timeVal)
print('maxNum:',maxNum)
i=1
start=1
while i < maxNum:
    ranNum=0

    for j in range (start,len(mainList)):
        if mainList[j]<=(mainList[1]+timeVal*i):
            ranNum=ranNum+1
        elif mainList[j]>(mainList[1]+timeVal*i):
            start=j
            break

    fastList.append(ranNum)
    i=i+1
fastString=''

for item in fastList:
    if (item%2)==0:
        fastString=fastString+'0'
    else:
        fastString=fastString+'1'
print(fastList[0:200])
```

Figure A.3.5: This picture shows the code used for the fast clock method



```
new_data=[]
for i in range(0,200):
    transfer=[]
    for j in range(0,200):
        transfer.append(float(c[i*200+j]))
    new_data.append(transfer)

# create discrete colormap
cmap = colors.ListedColormap(['white', 'black'])
bounds = [-1,0.5,2]
norm = colors.BoundaryNorm(bounds, cmap.N)

fig, ax = plt.subplots()
ax.imshow(new_data, cmap=cmap, norm=norm)

# draw gridlines
ax.grid(which='major', axis='both', linestyle='-', color='k', linewidth=0.1)
ax.set_xticks(np.arange(-.5, 200, 1));
ax.set_yticks(np.arange(-.5, 200, 1));

plt.show()
```

Figure A.3.6: This picture shows the code used to create the bitmap

Bibliography

[1] *Statistical analysis*. `https://www.random.org/analysis/`.

[2] Carl W. Akerlof, *Measurement of the muon lifetime*, University of Michigan, Department of Physics.

[3] AndrewRukhin, JuanSoto, JamesNechvatal, Miles Smid, Stefan Leigh ElaineBarker, MarkLevenson, Mark Vangel, DavidBanks, AlanHeckert, JamesDray, and SanVo, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, National Institute of Standards and Technology Special Publication **800** (2010), no. 22.

[4] Steven Kho Ang, *randomness$_t$estsuite*. `https://github.com/stevenang/randomness_testsuite.git`.

[5] Thomas K. Gaisser, Ralph Engel, and Elisa Resconi, *Cosmic rays and particle physics*, 2nd ed., Cambridge University Press, 2016.

[6] Enrique J. Galvez, *Experiments with correlated photons* (2019), 1–13.

[7] M. Herrero-Collantes and J. C. Garcia-Escartin, *Quantum random number generators*, REVIEWS OF MODERN PHYSICS **89**.

[8] Glenn F. Knoll, *Radiation detection and measurement*, 3rd ed., John Wiley Sons, Inc., 1999.

[9] M.P. Silverman, W. Strange C., R. Silverman, and T.C. Lipscombe, *Tests of alpha-, beta-, and electron capture decays for randomness*, Physics Letters A.