University Of Birmingham
School of Physics & Astronomy

Physics Year 2 Project

Second Semester Report

March 2021

Dark photon detection period as a method of true random number generation

By B.D.J.Adlam

(Lab Group Y)

Lab Partner: J.O.Evans

Supervisor: Alan Watson

Abstract:

This experiment examines the use of dark photon detection period as a method of generating random numbers. A detector was placed in a dark room and the period between detections was recorded. The results were passed through the ENT test suite and indicated highly random data. As a method of random number generation, dark photon detection was successful in generating random numbers, but slow and inefficient.

Introduction:

There are many applications for truly random numbers. They are used for simulations and models, gambling and for data encryption. As a result, the quality of the random numbers used is very important. Pseudo-random numbers (numbers that appear random but in fact are not) can cause simulations to provide false results, can cause cheating in gambling and can result in encrypted data being easily hacked.

The fundamental objective of this experiment is to generate random numbers by measuring the detection period of dark photons. These random numbers will then be run through a test suite in order to ascertain the quality of their randomness, and whether random number generation in this way is viable.

Theory:

Random Number Generators (RNG)

When it comes down to generating random numbers, there are two ways of addressing the problem; Pseudo-Random Number Generators (PRNG) and True Random Number Generators (TRNG). PRNGs, as the name suggests, are numbers that appear to be randomly generated but in reality, are not. They are usually generated by predetermined formulae that produce sequences of numbers that appear to be random, but when analysed are found to be periodic and predictable. If you generate more numbers than your PRNG (pseudo-random number generator) produces, then it will loop the numbers meaning they are not truly random[1]. An example of a PRNG is a Linear Congruential Generator (LCG), it is one of the most well-known methods of generating Pseudo-Random Numbers (PRN)[2]. As you can see in Fig.1, over a short period it seems to be completely random, but as we increase the scale, we can see that it becomes periodic and therefore is no longer random. The code for the LGC in Fig.1 can be found in appendix a.

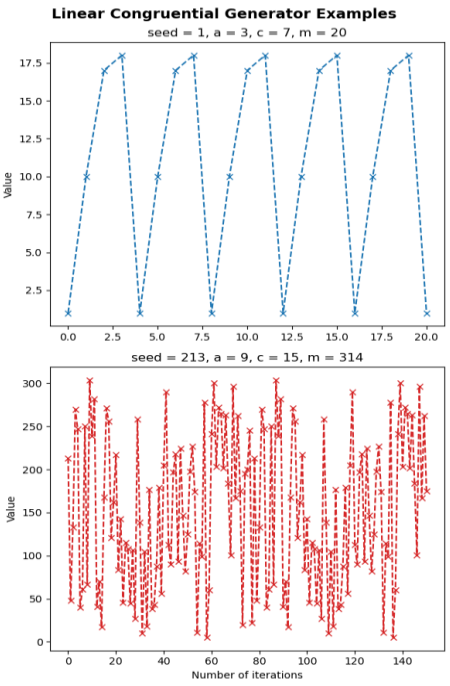

Figure 1: Plots of data generated by Linear Congruential Generator

On the other hand, TRNGs use physical processes to generate random numbers. An example of this would be the decay of a radioactive substance. Although it is possible to calculate the rate of decay of a substance, it is impossible to know which atom will decay next and the interval of time until it decays. As a result, you can detect the decays and convert them to random numbers.

TRNG and PRNG both have their advantages and disadvantages. PRNGs are efficient and deterministic meaning if you wanted to recreate the exact same numbers it would be possible. In comparison, TRNGs are inefficient and non-deterministic. This means for simulations and models where you need a large amount of data, PRNGs would be more appropriate as you could generate the numbers much faster, and be able to recreate exact scenarios. On the other hand, for data encryption or gambling you would use TRNGs as the numbers would be truly random and aperiodic.

Experimental theory

Photon detectors output a signal each time they detect a photon. When placed in a dark room, the time between photon detections becomes random, due to the dark count rate. The dark count rate of a detector is the number of detections of photons without any incident light[3]. The main source of these background detections is due to any remaining visible light photons or photons from the Infra-Red spectrum which are due to the thermal energy of the surroundings.

Further reasons for photon detections to be random, are deadtime and timing jitter.

The time duration spanning from when the detector has detected a photon and is converting it to an electrical output is known as 'deadtime'[1]. During this time the detector is unable to measure any incident photons. This inability to detect incident photons would heavily skew results because in cases when there are too many detections, the time between subsequent detections would become the deadtime. This would result in the data being deterministic and not at all random. Placing the detector in the dark and measuring the dark count rate can help to minimise this effect. This is because when measuring the dark count rate, the average time between detections is much greater, and therefore the likelihood a photon interacts with our detector during its deadtime is greatly reduced. This means that the data will display more randomness.

The uncertainty of the timing of individual photon detection is known as timing jitter[1]. In this experiment, the timing jitter of the TBX-04 is 250 pico-seconds. Compared to the uncertainty of the R&S HM8123 Universal Counter, 330 pico-seconds[4], this is much less. As a consequence, the timing jitter will not have any visible effect on the data, as the resolution of this counter is not large enough.

The detector is also affected by quantum efficiency[1]. Quantum efficiency of a detector is the fraction of detections which can be recorded. The interaction between matter and photons is inherently probabilistic, and therefore it is neither guaranteed that an interaction occurs, nor that it is recorded. Nevertheless, this should not matter too much for our experiment because the wavelengths of the visible and infrared photons have an interaction probability of almost 100%.

We expect our data to display an exponential distribution because at any point in time, we have a set probability that the next detection will occur. This means that the more time that passes since the last detection, the more likely it becomes that the next detection will occur. This is similar to a radioactive source.

Test Suite ENT

The test suite ENT from Fourmilab [5] will be used to evaluate randomness of our TRNG. ENT is a test suite that applies various tests to sequences of bytes stored in files and evaluates the results of those test. The test suite consists of six separate tests; entropy, optimum compression, Chi-squared, arithmetic mean, Monte-Carlo value for Pi and serial correlation coefficient[6].

Entropy is a test of the information density of the contents of the file, expressed as a number of bits per character. Ideally for a TRNG, the entropy would be 8 bits per byte.

Optimum compression is a measure of periodicity. It is the process of encoding data using fewer bits than the original file would require. An optimum compression would reduce the file by 0%.

Chi-squared is a very common measure of randomness. It is notoriously sensitive to errors in PRNG, and is expressed as an absolute number as well as a percentage. Ideally a TRNG would have a Chi-squared value of between 10% and 90%, however it is very dependent on sample size.

Arithmetic mean is simply the sum of all bytes divided by file length. Ideally one would have a value of 127.5. If the arithmetic mean was higher, it would indicate that the numbers are consistently high. If it was lower, it would indicate that the numbers are consistently low.

Monte Carlo value for Pi is a method of calculating the value for Pi using random numbers. A circle is placed inside a square and consecutive sequences of 6 bytes are used as 24-bit (X, Y) coordinates. If the separation of the randomly generated point is less than that of the radius of the circle, the 6-byte sequence is a "hit". The percentage of hits is then used to calculate a value for Pi. Of course, an exact value is impossible so one aims for 3.14159.

Serial correlation coefficient measures how much each byte depends on the previous byte in the file. RNG will approach a serial correlation coefficient of 0, whereas predictable data will have a serial correlation coefficient of 1.

Apparatus:

- TBX-04 Picosecond photon detection module
- TBX-PS detector power source
- R&S HM8123 Universal Counter
- Connecting cables
- Cardboard box
- Black cloth to cover detector lens
- Computer

Method:

The apparatus is setup as in Fig.2. A TBX-04 Picosecond photon detection module was connected via connecting cables to both a TBX-PS detector power source and an R&S HM8123 Universal Counter, which was in turn connected to a computer. The TBX-04 Picosecond photon detection lens was covered in black cloth and a cardboard box was placed on top so that only background noise photons were detected. The R&S HM8123 Universal Counter was connected to a computer via USB, where it was found to be configured as a serial port. Using the application Cutecom[7], it was identified as /dev/tty/USB0. With the R&S HM8123 Universal Counter on the remote setting, commands found in the HM8123 user manual[3], could be used to control the counter. The command XMT would request the current reading that the counter was displaying, and would either return a value in milliseconds, or it would return the phrase 'Not Available\r', indicating that no detection had been recorded.

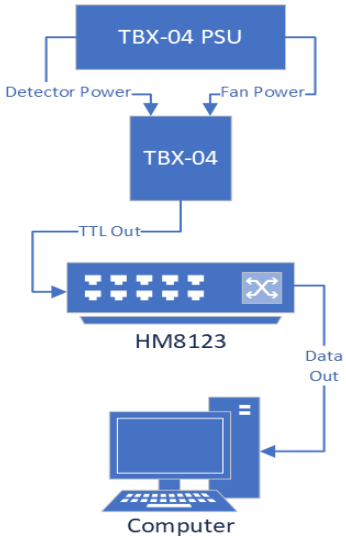

Figure 2: Diagram of experimental setup

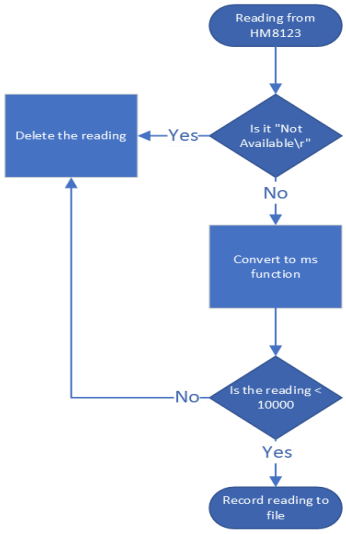

Figure 3: Flowchart describing treatment of data form universal counter

The program (appendix b) sends the command XMT which requests the current reading that the counter is displaying multiple times every second. If the counter returns a reading that is not 'Not Available\r' and is not greater than 10,000 ms, then the program records it and saves it onto the text file 'JoeByronData.txt'. The program was setup to run for three days, in order to collect the necessary amount of data to perform meaningful tests on.

When the program sent commands to the counter to ask for data, the counter would sometimes mess up and give a false reading. This was evident when the program was run for 10s and 3 of the 19 readings were given to be more than 2000 s. This was accounted for by deleting all recordings which were more than 10,000 ms. The resulting data was found to only have one reading in the 8,000 ms range and the rest at less than 6,000 ms. This implies that the act of deleting measurements of 10,000 ms and longer was a reasonable adjustment.

In order to reduce the size of the file, the program was written so that it would delete any 'Not Available\r' readings as well. This is depicted in Fig.3.

Experimental results:

The experiment yielded a sample size of 68,068 data points. The resulting data, when manipulated into histogram bins, displayed an exponential distribution; more accurately a 2nd order exponential distribution. The 1st and 2nd orders of the exponential can be attributed to the remaining natural light in the room and the thermal noise; visible and Infra-Red background noise.

In Fig.4, one can see a clear exponential distribution. However, there are some terms in the first 100 ms which are conflicting with the exponential distribution. In order to calculate the fit more accurately, the first few terms preceding the peak were removed.

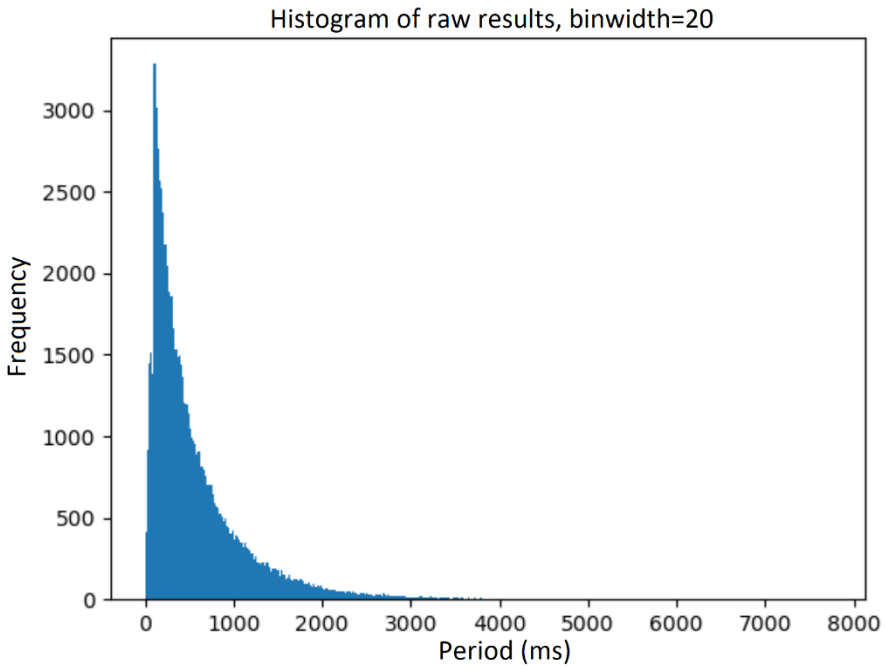

Figure 4: Histogram of our raw data, displaying an exponential distribution

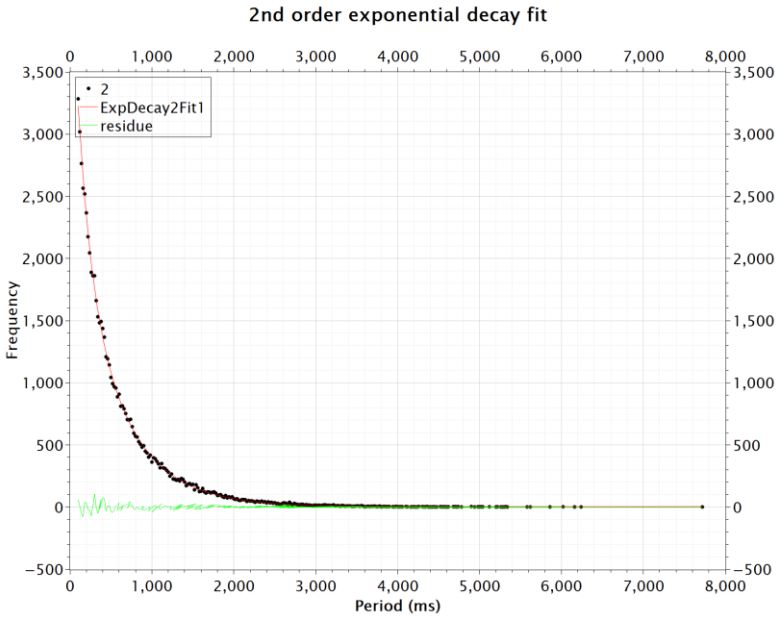

Figure 5: Plot of the 2nd order exponential decay fit, and of the residuals. Coefficients of the fit are: A_1 = 2265.2, A_2 = 2380.5, t_1 = 164.0, t_2 = 558.6 and y_0 = 1.5. The coefficients can be found with higher precision in appendix b.

The fit was generated using the following 2nd order exponential equation (Eqn.1):

$$y = A_1 e^{\left(-\frac{x}{t_1}\right)} + A_2 e^{\left(-\frac{x}{t_2}\right)} + y_0 \,, \qquad (1)$$

where A_1, A_2, t_1, t_2 and y_0 are constants given in the caption of Fig.5.

Clearly, the 2nd order exponential model fit the data points very closely. The residuals can also be seen to be relatively small.

The fit gave an R-squared value of R^2 = 0.999109, indicating a very tight fit. To check where the data deviated most from the fit, a plot of the residuals and of the scaled residuals was generated separately. The scaled residuals were calculated by dividing the residuals by the exponential fit values. These can be seen in Fig.6a & 6b.

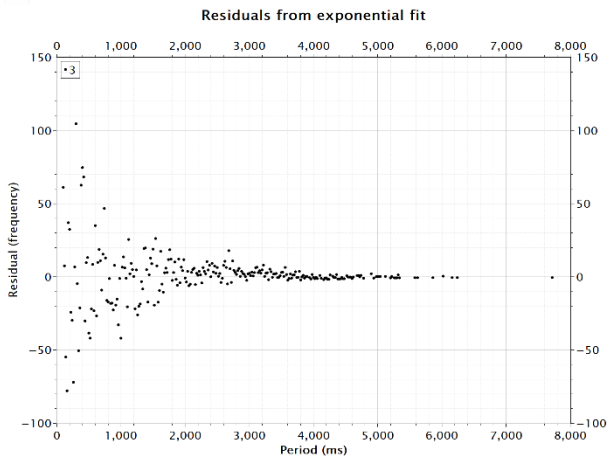

Figure 6a: Plot of residuals of raw data from exponential fit

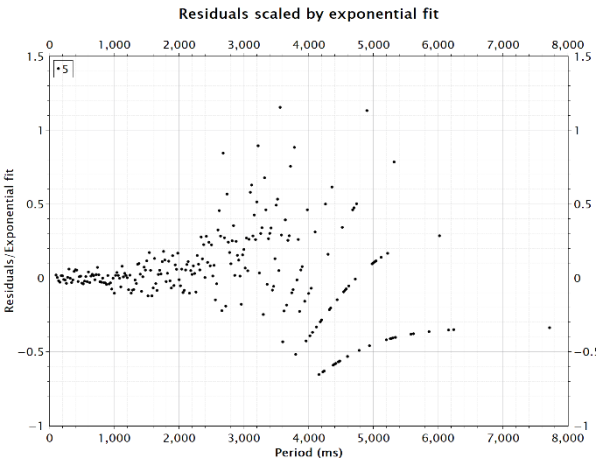

Figure 6b: Plot of residuals of raw data scaled by exponential fit

In Fig.6a, it seems as though the data points up until 2000 ms fit the model much worse than the latter points (3000+ ms). However, after examining Fig.6b, it is clear to see that the data points up until 2500 ms fit the model better than the latter points. This is because the majority of data points fall into the first 1000 ms. As the time period increases, the value of the exponential in Fig.5 falls to zero. From 3000 ms onwards, the exponential drops below 1. However, because we are dealing in integers, it either gets rounded up or down depending on the value of the exponential at that point. This results in errors being massively inflated, hence the poor residuals plot.

Although an exponential distribution was predicted, it was not sufficient enough to indicate the extent of which our data was truly random. Instead, the data was prepared to be run through the ENT test suite.

Analysis:

Method

In order to examine the data using the ENT test suite, the data needed to be manipulated using a probability integral transform (PIT)[8] into a uniform distribution. In order to do this the program found in appendix c was written. To begin with, the cumulative density function (CDF) (Eqn.2) is calculated by taking our 2nd order exponential equation as our probability density function (PDF):

$$CDF = \int_0^x PDF\ dx \qquad (2)$$

$$= \int_0^x A_1 e^{\left(-\frac{x}{t_1}\right)} + A_2 e^{\left(-\frac{x}{t_2}\right)} + y_0\ dx$$

$$= \left[-t_1 A_1 e^{\left(-\frac{x}{t_1}\right)} - t_2 A_2 e^{\left(-\frac{x}{t_2}\right)} + x y_0\right]_0^x$$

$$= \left[-t_1 A_1 e^{\left(-\frac{x}{t_1}\right)} - t_2 A_2 e^{\left(-\frac{x}{t_2}\right)} + x y_0\right] - [-t_1 A_1 - t_2 A_2]$$

$$\therefore CDF = t_1 A_1 \left(1 - e^{\left(-\frac{x}{t_1}\right)}\right) + t_2 A_2 \left(1 - e^{\left(-\frac{x}{t_2}\right)}\right) + x y_0\ , (3)$$

Next the transformed data is normalised by dividing each data point by the largest data point. As is evident in Fig.6b, the data does not fit our model as closely as the rest of our data. Because we were using the equation of our fit for the transformation, it made sense to remove the beginning few values that were skewing our model. In order to perform this, the transformed data was trimmed and a multiplier was used to shift the data back so that it covers the spread. To find the point at which the data fit the curve much better, the transformed data was run through the ENT test software and the value corresponding to the point at which the data flattens out was chosen. In our case this was the number 58, which meant we wanted to select the numbers corresponding to the range of 58-255 and spread them across the full range (0-255).

Finally, the program addressed the storage of data for testing in the ENT test suite. The storage of the transformed data values presented a problem as the ENT test suite reads data in the form of bytes (decimal value 0-255); however, our data was encoded in ASCII. The flowchart in Fig.7 describes this process in more detail.

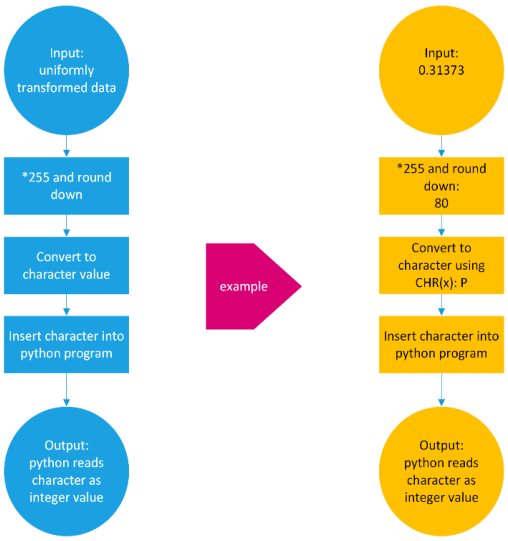

Figure 7: Flowchart describing how data is manipulated for ENT test suite.

Results

The result of the code (appendix c), was the histogram plot seen in Fig.8. Our Histogram displays a clearly uniform distribution. Having seen the 2nd order exponential fit in Fig.5, this is as expected. Assuming our code, and method of transformation is correct, this further indicates randomness.

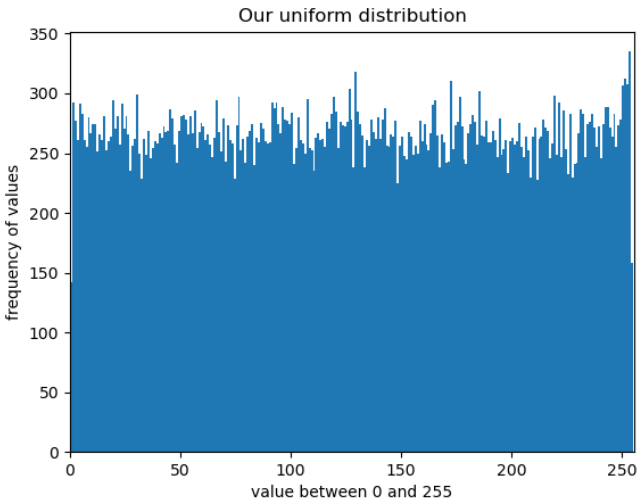

Figure 8: Histogram of our transformed data, showing a clear uniform distribution

Finally, it was run through the ENT test suite to obtain an accurate analysis of the randomness.

As can be seen below in Fig.9, everything except for the Chi-square distribution implies highly random numbers. The entropy is 7.988105 bits per byte, implying a file density of 99.85%. The optimum compression is 0% meaning the file cannot be compressed, and hence shows no periodicity, the arithmetic mean value, at 126.6265, is 0.8735 away from 127.5. The Monte-Carlo value for Pi is very close at

```
Total:          68068   1.000000

Entropy = 7.988105 bits per byte.

Optimum compression would reduce the size
of this 68068 byte file by 0 percent.

Chi square distribution for 68068 samples is 932.12, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 126.6265 (127.5 = random).
Monte Carlo value for Pi is 3.154795487 (error 0.42 percent).
Serial correlation coefficient is 0.022124 (totally uncorrelated = 0.0).
```

Figure 9: Results after running transformed data through ENT test suite.

3.154795487, an error of 0.42%. Finally, the serial correlation coefficient is 0.0022124, which is very close to 0. However, the Chi-square distribution is 0.01% which would imply a complete lack of randomness. However, we believe that this is due to the ENT test suite being inadequate. To prove this, 20,000 data points were taken from *random.org*[9] and run through the ENT test suite. They were open-source random numbers generated by atmospheric noise.

```
Total:         20086    1.000000

Entropy = 7.987550 bits per byte.

Optimum compression would reduce the size
of this 20086 byte file by 0 percent.

Chi square distribution for 20086 samples is 363.54, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 127.5265 (127.5 = random).
Monte Carlo value for Pi is 3.198087840 (error 1.80 percent).
Serial correlation coefficient is 0.011425 (totally uncorrelated = 0.0).
```

Figure 10: Results after running random data from random.org through the ENT test suite.

From Fig.10 it is clear to see that the Chi-square distribution is indicating that the data is not at all random, although it was generated by a valid TRNG. This clearly indicates that the test is faulty. However, it most likely is a result of the small sample size of 68,068, as according to the ENT website[5] the Chi-square distribution is very inaccurate for small data sets. Although this indicates that the Chi-square is highly inaccurate, it does give some perspective on the results in Fig.9. In all tests other than the Chi-square, it implies our numbers are truly random.

Uncertainty and sources of error:

In the experiment we came across multiple small sources of error that could compound into a larger general uncertainty. When the experiment was running for three days, it was kept unattended in a locked room. When we came back to collect our results, the universal counter displayed the error 'INTERNAL REFERENCE CHECK FAILED'. According to the user manual, it was an error related to the frequency range and the internal clock. We tried to run it on the external setting instead, but it would not record any data.

To obtain an error on the fit was a difficult task. Originally, we were going to take an average of the absolute scaled residuals and use that as a percentage uncertainty on our data. However, as is clear in Fig.6b, the scaled residuals plot is small at low periods, and very large at high periods. This meant our error value would be too inaccurate. Instead, we opted to calculate the error propagation on the values of the model. The errors on the equipment used were ignored, as they were incredibly small and especially after rounding (in the program (appendix c)) they would be negligible. The uncertainty was calculated using the general equation (Eqn.4):

$$\sigma_F = \sqrt{\left(\frac{\partial F}{\partial x}\right)^2 \sigma_x^2 + \left(\frac{\partial F}{\partial y}\right)^2 \sigma_y^2 + \left(\frac{\partial F}{\partial z}\right)^2 \sigma_z^2}\,. \qquad (4)$$

The full method can be found in appendix d. The results of this are a maximum uncertainty value of 4.7372%, a minimum uncertainty value of 0.1473% and a mean uncertainty value of 2.5561%. The minimum and maximum uncertainty values give a reasonable understanding of the general uncertainty on the fit values. Generally, the error on the transformed data is very small, which indicates a high level of precision throughout the experiment. What this corresponds to as an uncertainty on the final ENT test results is difficult to ascertain. However, as the error is small, it would be a fair assumption to make that the final uncertainty would be minute.

Conclusion:

The generation of random numbers from the detection of dark photons was shown to be successful. The test suite showed that the data passed all tests (Fig.9) except for the Chi-squared, however, this is a result of the data set being too small. From the uncertainty propagation we have clear indication that our data is accurate and that our modifications, such as removing the first few values when modelling the data to improve the fit, have not reduced the integrity of our experiment.

As a form of TRNG it produced truly random data, however, it also displayed the limitations often found in TRNG. It was slow and only generated a small sample size. In comparison to the more common technique of measuring the radioactive decay of a sample, it is definitely an improvement. Measuring the decay count of a radioactive source involves a potentially harmful substance, that over time diminishes. In order to generate large amounts of random numbers, the radioactive substance must be allowed to decay for a substantial amount of time. Over time this decay destroys the substance. If one wanted to increase the count, and hence decrease the amount of time taken, a more unstable substance would be required. This would be more dangerous, as the substance would be even more harmful. In comparison, the generation of random numbers using dark photon count is infinite (the source does not decay away) and safe.

Acknowledgements:

Many thanks to all who helped us with our experiment and helped us whilst struggling with code. A special thanks to Alan Watson for being such a great demonstrator and a particular thanks to John Goldwin for giving invaluable advice and lending us equipment. Thanks to Mark Colclough as well for help and advice with our code. Finally, a massive thanks to Joe Evans for being a wonderful lab partner.

References:

[1] Dr Rüdiger Paschotta, "Photon Counting," *RP Photonics Encyclopedia*. https://www.rp-photonics.com/photon_counting.html (accessed Feb. 18, 2022).

[2] Ursa Pantle, "Linear Congruential Generators," Jul. 20, 2006. https://www.mathematik.uni-ulm.de/stochastik/lehre/ss06/markov/skript_engl/node26.html (accessed Apr. 04, 2022).

[3] "HM8123 HM8123-X 3 GHz Programmable Counter Benutzerhandbuch User Manual." Accessed: Mar. 01, 2022. [Online]. Available: https: //scdn.rohde-schwarz.com/ur/pws/ dl_downloads/dl_common_library/ dl_manuals/gb_1/h/hm8123_x/ HM8123_UserManual_de_en_06.pdf

[4] Horiba Jobim Yvon, "TBX picosecond photon detection module User Guide." https://www.horiba.com/fileadmin/uploads/Scientific/Downloads/UserArea/Fluorescence/Manuals/TBX_User_Guide.pdf (accessed Feb. 04, 2022).

[5] John Walker, "ENT A Pseudorandom Number Sequence Test Program," Jan. 28, 2008. https://www.fourmilab.ch/random/ (accessed Feb. 04, 2022).

[6] "ent - pseudorandom number sequence test," Mar. 18, 2018. http://manpages.ubuntu.com/manpages/bionic/man1/ent.1.html (accessed Mar. 15, 2022).

[7] Alexander Neundorf, "Welcome to CuteCom," May 2009. http://cutecom.sourceforge.net/ (accessed Feb. 26, 2022).

[8] J. E. Angus, "The Probability Integral Transform and Related Results," 1994.

[9] “Random Integer Generator.” https://www.random.org/integers/ (accessed Mar. 24, 2022).

Appendix:

a) Code for Linear Congruential Generator;

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

values1 = [1]
values2 = [213]   # define seed values
a = [3, 9]
c = [7, 15]
modulus = [20, 314]   # define function parameters
for y in range(0, 20):   # generate sequences
  values1.append((a[0] * values1[y] + c[0]) % modulus[0])
for y in range(0, 150):
    values2.append((a[1] * values2[y] + c[1]) % modulus[1])
x1 = np.arange(0, len(values1))   # set x values for the graph
x2 = np.arange(0, len(values2))
fig, ax = plt.subplots(2)   # plot the generated numbers
fig.set_figheight(10)
fig.set_figwidth(6)
fig.suptitle('Linear Congruential Generator Examples', fontsize=14,
             fontweight='bold')
ax[0].plot(x1, values1, linestyle='--', marker='x', color='tab:blue')
ax[0].set_title("seed = "+str(values1[0])+", a = "+str(a[0])+", c = "
                +str(c[0])+", m = "+str(modulus[0]))
ax[1].plot(x2, values2, linestyle='--', marker='x', color='tab:red')
ax[1].set_title("seed = "+str(values2[0])+", a = "+str(a[1])+", c = "
                +str(c[1])+", m = "+str(modulus[1]))
ax[0].set(ylabel='Value')
ax[1].set(xlabel='Number of iterations', ylabel='Value')
fig.tight_layout()
plt.savefig('LCGexample')
plt.show()
```

b) Code for running photon detector;

```
import serial
import time

def ConvertToms(data_in):
    """
    Returns the period of detections in ms rather than s or ns and removes
    the unit
    """
    if 'ms' in data_in:
        data_in = float(data_in.translate({ord(c): None for c in " ms"}))
    elif 'ns' in data_in:
        data_in = float(data_in.translate({ord(c): None for c in " ns"}))
        data_in = data_in / 1000
    elif '$s' in data_in:
        data_in = float(data_in.translate({ord(c): None for c in " $s"}))
        data_in = data_in / 1000
    elif 's' in data_in:
        data_in = float(data_in.translate({ord(c): None for c in " s"}))
        data_in = data_in * 1000
    return data_in
#configuring the serial port to the parameters of the programmable counter
data = []
ser = serial.Serial(
    port='/dev/ttyUSB1',
    baudrate=115200,
    parity=serial.PARITY_NONE,
```

```
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout = 5
)

with open('JoeByronDatatest.txt', 'w') as file:
    start_time = time.time()
    seconds = 259200 #this is the control for three days of operation
    while True:
        try:
            current_time = time.time()
            elapsed_time = current_time - start_time
        except:
            pass
#try and except is in the off chance any errors occur
        if elapsed_time > seconds:
            break

        try:
            ser.flush()
            ser.write(b'XMT\r')      #request the most recent period
            data_in = ser.read_until(b'\r').decode('ascii')

            if data_in != 'Not Available\r':
                data_in = ConvertToms(data_in)
                if data_in < 10000:
                    file.write(str(data_in) + '\n') #this removes results that
#are too high, and 'Not Available' results. Records all other results
        except:
            pass
    file.close()
print('Data collection complete')
```

c) Code for manipulating data from exponential distribution to uniform distribution;

```
import numpy as np
from matplotlib import pyplot as plt

data = []  # read the data in ms from the file and convert it to a numpy array for processing
with open('JoeByronDatatest.txt', 'r+') as file:
    for line in file:
        line = float(line.translate({ord(c): None for c in '\n'}))
        data.append(line)
    file.close()

data = np.array(data)

A1 = 2265.22216  # define values from exponential fit
A2 = 2380.501905
t1 = 163.9803523
t2 = 558.5804155
y0 = 1.50687457

s_A1 = 47.9153343
s_A2 = 64.171205
s_t1 = 6.15408236
s_t2 = 7.85687244
s_y0 = 0.92571907

s_data = 0.000001

transformed_data = A1 * t1*(1 - np.exp(-data / t1)) + A2 * t2*(1 - np.exp(-data / t2))  + data *
y0
s_transormed_data = ((s_A1*t1*(1-np.exp(-data/t1)))**2 + (s_A2*t2*(1-np.exp(-data/t2)))**2 +
                     (A1*s_t1*(1-np.exp(-data/t1))-s_t1*((A1*data*np.exp(-data/t1))/t1))**2 +
                     (A2*s_t2*(1-np.exp(-data/t2))-s_t2*((A2*data*np.exp(-data/t2))/t2))**2 +
                     (data*s_y0)**2 + ((A1*(np.exp(-data/t1)) + A2*(np.exp(-data/t2)) +
y0)*s_data)**2)**0.5


max_data = max(transformed_data)
```

```
s_max_data = max(s_transormed_data)


transformed_normalised_data = transformed_data / max(transformed_data)
transformed_normalised_data_error = ((1/max_data*s_transormed_data)**2 +
                                     (-(transformed_data/max_data**2)*s_max_data)**2)**0.5
num = 58
multiplier = 1 / (1 - (num/255))    # 0.33333... is the value where the data starts to fit well,
therefore we can expand
                                # the upper part of the distribution to fill the range 0 - 1
expanded_transformed_normalised_data = transformed_normalised_data * multiplier - (num/255) *
multiplier

s_multiplier = (255/(255-num)**2)

s_expanded_data = (((transformed_normalised_data-(num/255))*s_multiplier)**2 +
                   (multiplier*transformed_normalised_data_error)**2)**0.5

print(max(s_expanded_data))
print(min(s_expanded_data))
print(np.mean(s_expanded_data))

trimmed_normalised_data = []
for datapoint in expanded_transformed_normalised_data:  # remove any values that are below 0 due
to the previous trim
    if datapoint > 0:  # and transform
        trimmed_normalised_data.append(datapoint)
trimmed_normalised_data = np.array(trimmed_normalised_data)

with open('uniform_normalised_data.txt', 'w') as file:  # save the normalised data to a file
    for item in trimmed_normalised_data:
        file.write(str(item) + '\n')
    file.close()

ENT_data = trimmed_normalised_data * 255 + 0.5  # ENT takes data in bytes (decimal value 0-255)
and the integer rounding
# later on means you to add 0.5 for normal mathematical rounding

x_values = np.arange(0, 255)  # plotting the uniform normalised data as a histogram once it has
been distributed
# between 0 and 255 to visualise what ENT will receive
binwidth = 1
(x, bins, z) = plt.hist(ENT_data, bins=range(int(min(ENT_data)), int(max(ENT_data) + binwidth),
binwidth))
plt.xlim(0, 255.5)
plt.xlabel('value between 0 and 255')
plt.ylabel('frequency of values')
plt.title('Our uniform distribution')
plt.savefig('Uniform distribution')
plt.show()

character_code_string = ''

for x in ENT_data:  # for each data point convert to int which rounds down and convert value to
character code append
    x = int(x)  # these to the end of a string, this stores the data in bytes for ENT to read
    y = chr(x)
    character_code_string = character_code_string + str(y)

with open('ENT_encoded_data.txt', 'w', encoding="latin-1") as file:
    file.write(str(character_code_string))
    file.close()
```

d) Error propagation calculations;

i. Transformed Data Error:

$$\sigma_{TransData} = \sqrt{\begin{array}{l}\left(t_1\left(1-e^{\left(-\frac{data}{t_1}\right)}\right)\sigma_{A_1}\right)^2+\left(t_2\left(1-e^{\left(-\frac{data}{t_2}\right)}\right)\sigma_{A_2}\right)^2+\\ \left(\left(A_1\left(1-e^{\left(-\frac{data}{t_1}\right)}\right)-\frac{(A_1\cdot data)e^{\left(-\frac{data}{t_1}\right)}}{t_1}\right)\sigma_{t_1}\right)^2+\\ \left(\left(A_2\left(1-e^{\left(-\frac{data}{t_2}\right)}\right)-\frac{(A_2\cdot data)e^{\left(-\frac{data}{t_2}\right)}}{t_2}\right)\sigma_{t_2}\right)^2+\\ \left(data\cdot\sigma_{y_0}\right)^2+\left(\left(A_1e^{\left(-\frac{data}{t_1}\right)}+A_2e^{\left(-\frac{data}{t_2}\right)}+y_0\right)\sigma_{data}\right)^2\end{array}}$$

ii. Error after normalisation:

$$\sigma_{NormData} = \sqrt{\left(\frac{\sigma_{TransData}}{\max(data)}\right)^2+\left(-\frac{TransData}{\max(data)^2}\sigma_{\max(data)}\right)^2}$$

iii. Error after multiplication and shift:

$$\sigma_{58} = 8 \pm 1\,,\ m = \frac{255}{197}\,,\ \sigma_m = \frac{255}{(255-58)^2}$$

$$\therefore \sigma_{ExpandData} = \sqrt{\left(\left(NormData-\frac{58}{255}\right)\sigma_m\right)^2+(m\cdot\sigma_{NormData})^2}$$