universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Optimized Post-Processing Algorithm for Quantum Key Distribution

verfasst von | submitted by

Haxhi Pantina

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 876
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Physics
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Philip Walther

Acknowledgements

I would first like to thank my supervisors, whose guidance and mentorship made it much easier for me to complete this thesis. In particular, I'd like to thank them for always finding time to meet and helping me when I faced difficulties.

A special thank you goes, of course, to my dear parents and siblings for their emotional and financial support throughout my time as a student. Finally, I would like to thank my extended family here in Austria—my dear aunt Hava Pantina, my cousin Zahir Pantina, and his wife Pranvera Nikqi Pantina—without whom I would never have been able to complete this journey.

Abstract

The generation of highly entangled photons employing spontaneous parametric down-conversion is essential to many Quantum Key Distribution schemes and other quantum technologies. SPDC also plays a key role in the implementation of the digital quantum payment scheme (which is our main focus here). Particularly, an important aspect of SPDC-generated photon pairs we need to consider is their temporal cross-correlation. As shown in [4], photon pairs are simultaneously generated and should therefore be detected at the same time (within a very thin time window), but due to different factors, some of which are very hard to control, photon pairs are delayed relative to each other. Thus, an algorithm that is time and resource-efficient for a fine estimation of the time offset between SPDC-generated streams of photons is of great interest.
In this thesis, two different post-processing Python algorithms for estimating the time offset and coincidence counts between two time-tagged datasets are put to the test. To avoid bias and allow for valid conclusions to be drawn, both codes ran the same data collected from the experiments on the quantum digital payment scheme [15] and in the same hardware environment. In general, an algorithm analysis can be very complex, but for all practical purposes of this thesis, only the running time element as a function of the input size n is studied, both experimentally and theoretically. Also, we compare the time it takes for both algorithms to compute coincidence counts. The results of the analysis will show that our proposed Python algorithm is a huge improvement over the previous model, offering a much better time efficiency in finding the time offset and higher resolution (in picoseconds) at the same time. Further on, the analysis will show that both algorithms score high in our time efficiency test for computing the coincidence counts.
Finally, we hope that this work will help in making informed decisions as to what approach to take to find the time delay between huge datasets and coincidence counts.

Kurzfassung

Die Erzeugung stark verschränkter Photonen durch spontane parametrische Abwärtskonversion (SPDC) spielt eine zentrale Rolle bei der Realisierung zahlreicher Verfahren der Quanteninformationstechnologie, insbesondere im Rahmen von Quanten-Schlüsselverteilungsprotokollen. Darüber hinaus bildet SPDC eine wesentliche Grundlage für das digitale, quantenbasierte Zahlungssystem, das im Mittelpunkt dieser Masterarbeit steht. Ein zentraler Aspekt von SPDC im Kontext unserer Untersuchung ist die zeitliche Korrelation der erzeugten Photonenpaare. Theoretisch sowie experimentell ist belegt, dass solche Paare gleichzeitig entstehen und daher idealerweise auch simultan detektiert werden sollten. In der Praxis führen jedoch verschiedene, nur schwer kontrollierbare Einflüsse zu messbaren Zeitunterschieden zwischen den Photonen. Ziel dieser Arbeit ist es daher, zwei verschiedene Python-Algorithmen zur Bestimmung des Zeitversatzes zwischen Photonenströmen, die im Rahmen experimenteller Arbeiten zum Projekt des digitalen Quanten-Zahlungssystems aufgezeichnet wurden, zu analysieren und zu evaluieren. Zwar wäre für einen umfassenden Algorithmusvergleich die Berücksichtigung zahlreicher Faktoren wünschenswert, im Rahmen dieser Arbeit konzentrieren wir uns jedoch ausschließlich auf die zeitliche Effizienz. Die Laufzeit beider Algorithmen wird in Abhängigkeit von der Datenmenge gemessen und analysiert. Diese Analyse soll uns ein tieferes Verständnis über die zeitliche Leistungsfähigkeit beider Ansätze vermitteln und aufzeigen, unter welchen Bedingungen ein Algorithmus dem anderen überlegen ist. Auf diese Weise hoffen wir, fundierte Entscheidungen im Hinblick auf die Auswahl geeigneter Python-Module und -Algorithmen für vergleichbare Anwendungen treffen zu können.

Contents

1. Introduction

A high-resolution estimation of the time delay between streams of photons generated via spontaneous parametric down-conversion is critical for the successful implementation of many quantum protocols, particularly QKD schemes. However, this is typically a very challenging task as one has to deal with huge datasets that often exceed the computational power at disposal. Given that, figuring out a programming algorithm that efficiently handles large data is very important.

1.1. Quantum Key Distribution

Quantum Key Distribution is a key exchange protocol between two trusted parties (traditionally called Bob and Alice) based on the laws of quantum mechanics. Due to its many applications and excellent security features, QKD ranks among the most advanced and widely used quantum technologies. Whereas the security of traditional classical key exchange schemes relies on complex algorithms and hard mathematical problems to solve, QKD exploits the inability of an eavesdropper to intercept the communication without disturbing it because of the well-established laws of quantum mechanics. This enormous advantage to its classical analogs is primarily guaranteed by the no-cloning theorem, according to which there's no way to make an identical copy of some unknown non-orthogonal quantum states.

Proof: To prove the non-cloning theorem, we assume the contrary, that such cloning operation is possible. Suppose we have two non-orthogonal quantum states

$$|\psi >$$

and

$$|\phi >$$

and a cloning machine represented by a unitary operator U. Upon acting on the quantum states with U, we get the following relations

$$U\left(|\psi\rangle \otimes |s\rangle\right) = |\psi\rangle \otimes |\psi\rangle \tag{1.1}$$

$$U\big(|\phi\rangle \otimes |s\rangle\big) = |\phi\rangle \otimes |\phi\rangle \tag{1.2}$$

Where |s> is some auxiliary state.

After taking the inner product of the two, we get

$$\langle\phi|\psi\rangle = (\langle\phi|\psi\rangle)^2 \tag{1.3}$$

Which can stand only if $|\psi\rangle$ and $|\phi\rangle$ are orthogonal or the same state, but since this contradicts the statements we began with in the first place, we can conclude that such a cloning machine can not exist. Note, however, that the theorem does not prevent us from copying orthogonal states. It is, therefore, logical to raise the question of whether there is indeed a way for an untrusted part to intercept the exchange and gain information about non-orthogonal states to be transmitted between Alice and Bob, without disturbing them. It turns out, however, that this is also impossible due to the laws of quantum physics.

Proposition 1.1: (Information gain entails disturbance) Attempting to distinguish non-orthogonal quantum states in the quantum channel will incur disturbance.

Proof: Suppose $|\psi\rangle$ and $|\phi\rangle$ are two non-orthogonal quantum states contained in the signal Bob wants to send to Alice. The untrusted middle party will try to gain information by operating on those states with a unitary operator, so that the following relations stand

$$U\left(|\psi\rangle \otimes |u\rangle\right) = |\psi\rangle \otimes |v\rangle \tag{1.4}$$

$$U\left(|\phi\rangle \otimes |u\rangle\right) = |\phi\rangle \otimes |v'\rangle \tag{1.5}$$

The goal of the eavesdropper is to be able to partially distinguish the non-orthogonal quantum states in the transmission protocol. Taking the inner product between the two equations leads us to the following

$$\langle\phi|\psi\rangle = \langle\phi|\psi\rangle\langle v'|v\rangle \tag{1.6}$$

which can hold only if v=v'. In other words, in an attempt to distinguish these non-orthogonal quantum states, the eavesdropper will inflict a disturbance.

1.2. BB84 Protocol

Introduced in 1984 by Bennet and Brassard, the BB84 is widely regarded as the first QKD protocol ever. The BB84 protocol consists of the following steps:

Step 1: Alice prepares a string of photons completely at random in any of the four quantum states |0>, |1>, |+>, and |->. Alice and Bob agree to give the 0 bit value to the states |0> and |+> and the bit value 1 to the states |1> and |->, so that any potential

eavesdropper will risk disturbing the state trying to gain information as shown in the previous section (Proposition 1.1).

Step 2: After receiving the string of photons sent by Alice, Bob measures them randomly in Pauli X or Z basis. If Bob measures on the same basis that Alice prepared the photon, he will get the bit encoded in it. If, however, he measures in the complementary basis, he will obtain a random bit.

Step 3: (Key sifting) Upon the completion of measurement by Bob, they switch over to a classical communication channel (which can be susceptible to eavesdropping) to compare the basis that they used. If their measurement bases match, they retain those bits and if their bases don't, they discard them.

Step 4: Alice and Bob share a sample of bits from the key via a classical channel to estimate the error rate and the information gain by the eavesdropper.

Step 5: Finally, they run an error correction algorithm to account for Bob's bits that did not match those of Alice.

QUANTUM TRANSMISSION														
Alice's random bits	1	1	0	1	1	0	0	1	0	1	1	0	0	1
Random sending bases	D	R	D	R	R	R	R	R	D	D	R	D	D	D	R
Photons Alice sends	↗	↕	↘	↔	↕	↕	↔	↔	↘	↗	↕	↘	↗	↗	↕
Random receiving bases	R	D	D	R	R	D	D	R	D	R	D	D	D	D	R
Bits as received by Bob	1		1		1	0	0	0		1	1	1		0	1
PUBLIC DISCUSSION															
Bob reports bases of received bits	R		D		R	D	D	R		R	D	D		D	R
Alice says which bases were correct			OK		OK			OK				OK		OK	OK
Presumably shared information (if no eavesdrop)			1		1			0				1		0	1
Bob reveals some key bits at random					1									0	
Alice confirms them					OK									OK	
OUTCOME															
Remaining shared secret bits			1					0				1			1

Figure 1.1.: The BB84 Protocol [3]

1.3. Spontaneous Parametric Down Conversion

Spontaneous parametric down-conversion (shortly referred to as SPDC) is an optical nonlinear process in which a single photon of higher energy (known as the pump) is converted to two photons of lower energy (the signal and the idler), hence the name down-conversion.

In line with the law of energy conservation, the corresponding wave vectors of the pump, the signal, and the idler satisfy the following relation in the nonlinear medium

$$\vec{k} = \vec{k}_1 + \vec{k}_2 \tag{1.7}$$

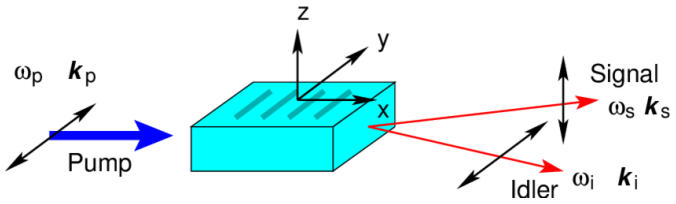


Figure 1.2.: A schematic illustration of SPDC [11]

A widespread use of this nonlinear optical process is the generation of single photons, although real single-photon sources like quantum dots and NV centers are preferable. Other applications of SPDC include:

- Quantum Cryptography
- Generation of entangled photons
- Quantum teleportation
- Quantum metrology, etc.

For a more formal treatment of the SPDC, the reader can refer to the Appendix (A1).

SPDC and the Digital Quantum Payment Scheme

The generation of highly entangled photon pairs via SPDC and the study of their time offset in the context of a digital quantum payment scheme [15] are particularly of great interest to us in this thesis.

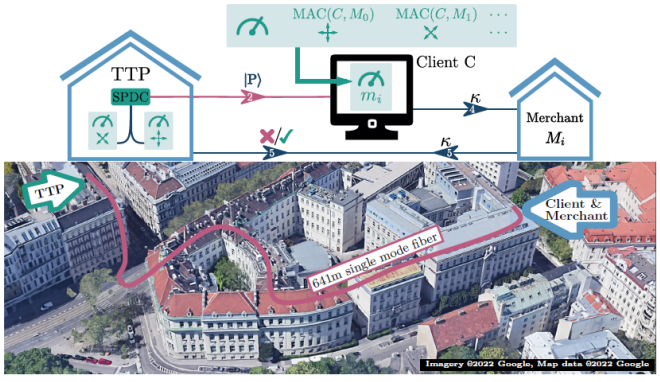


Figure 1.3.: The experimental setup of the digital quantum payment scheme[15]

In such a scheme, a Trusted Token Provider generates a stream of entangled photon pairs, measuring one of each pair in a specific measurement basis so that the second photon for the Client will be remotely prepared in a state P. For one to be able to show that two photons originate from the same pair, a coincidence counting must be carried out. However, due to several inevitable factors, the stream of photons detected by the TTP and Client will be shifted in time relative to each other. Some of these factors include different distances that photons of each pair travel, different drifting rates of the internal clocks of the corresponding TTMs (short for Time-Tagging Modules), and different detectors' response times. Thus, to recover the coincidences, one needs to build a computationally fast and efficient algorithm to account for this delay. However, as already mentioned, this is not always an easy task because in such experiments we deal with huge data that require great computational power. For example, the data we ran our codes presented in this thesis contains millions of time-tagged events spanning only a few milliseconds.

The Distribution of High-Dimensional Entanglement Over Free Space

Another example of where an optimized algorithm for high-resolution estimation of the time offset between SPDC-generated streams of photons would find application is the scheme for the distribution of high-dimensional entanglement over a space-free noisy environment [see 3]. In this experimental work, the authors were able to get 3-dimensional

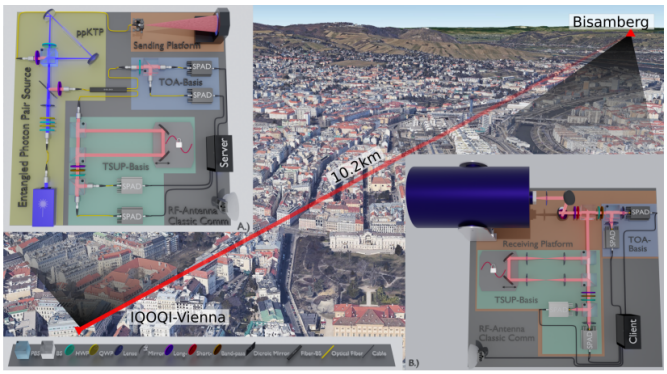


Figure 1.4.: The experimental realization of the distribution of high-dimensional entanglement

entanglement by coarse-graining the arrival time of photons into variable time-bin lengths $\triangle(t)$. They noticed that for relatively larger time-bin lengths, the number of accidental coincidence counts was higher compared to when shorter time bins were used. Moreover, they were able to find a minimal $\triangle(t)$ for which the fidelity of the high-dimensional entangled state was the highest (Fig.1.5). Therefore, in such project the need for a high resolution and time and resource efficient coincidence finding and time offset estimation is of great importance.

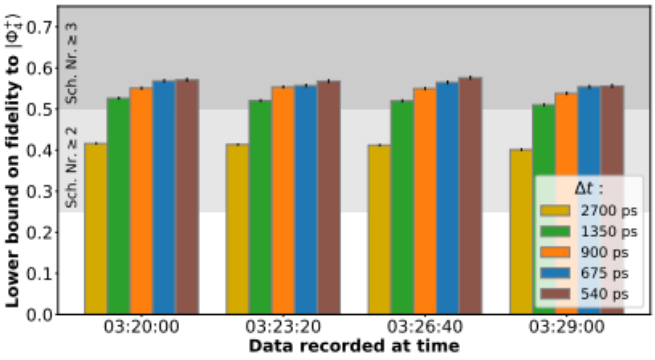


Figure 1.5.: The fidelity of the time-energy part of the SPDC-generated high dimensional quantum state for various time bin lengths [3]

1.4. Goals and Motivation

In a situation where, on one hand, a high-resolution estimation of time offset between time-tagged events and with it the recovery of coincidences is of critical importance for many quantum protocols, and on the other, the data we have to deal with is huge, the need for a time and resource-efficient computational algorithm is inevitable.

Unfortunately, no solution fits all kinds of problems. Therefore, we must have a deep understanding of the data and the ways we can approach a specific problem. In this thesis, we're going to study two different algorithms for finding the time delay between two sets of time-tagged events collected as part of the implementation of the digital quantum payment scheme [15]. In addition, we are going to conduct an algorithm analysis, both experimentally and theoretically, to better evaluate their performance. While an extensive algorithm analysis requires taking into account several different factors, for all practical purposes of this thesis, we are going to focus on the time complexity of each algorithm as a function of the input size n.

In general, the goal of this post-processing analysis is to compare two different Python algorithms for estimating time offsets and coincidence counts to help us make informed decisions as to what approach to take when challenged with tasks of this nature.

2. Python Algorithms for Time Offset Estimation

2.1. Python built-in functions

Before moving any further, let's take a quick look at some Python modules that we'll need here, which are specifically developed for estimating time offsets between two array objects.

scipy.correlate

The most common of all is scipy.signal.correlate from the scipy module.

SYNTAX:

signal.correlate(in1, in2, mode='full', method='auto')

The first two arguments of this method are array-like elements, and in the context of our discussion, they contain the series of timetags whose cross-correlation we want to compute. In short, what the method does is it measures the cross-correlation of two arrays as it scans through the time that both signals span by sliding one array on top of the other. In simple mathematical language, the cross-correlation is a measure of the similarity between two arrays (see Appendix for more). The method returns an array where each element corresponds to the cross-correlation of the two for a certain time offset.

Here is a simple example:

```
import numpy as np
from scipy.signal import correlate

a = np.array([0, 1, 1, 0, 0])
b = np.array([0, 0, 1, 1, 0])

corr = correlate(a, b, mode='full')
print(corr)
```

OUTPUT: [0 0 1 2 1 0 0 0 0]

To extract the index of the element with the highest value in the corr array, we can make use of the argmax() method

```
max_corr=np.argmax(corr)
```

If we were to compare histograms of a and b arrays instead of raw data, we could easily retrieve the time delay for which the correlation is the highest by multiplying the index of the highest value in corr with the width of the time bin defined

```
time_lag=max_corr*bin_width
```

Fortunately, the module does have a method to find the time offset between the arrays, and in our case would look as in the following

```
import numpy as np
from scipy.signal import correlate, correlation_lags

a = np.array([0, 1, 1, 0, 0])
b = np.array([0, 0, 1, 1, 0])

corr = correlate(a, b, mode='full')
lags = correlation_lags(len(a), len(b), mode='full')

delay = lags[np.argmax(corr)]
print("Estimated delay:", delay)
```

Numpy Correlate

Another less common Python method to measure the cross-correlation between two arrays is numpy.correlate

SYNTAX

numpy.correlate(a,b,mode='full')

This method does return the cross-correlation between the a and b arrays. In contrast to scipy.correlate, this method is only limited to 1D arrays. In addition to that, it does not use the FFT to compute the convolution, making it very slow for large arrays. Therefore, in cases when one has to deal with huge data, scipy.correlation is a much better choice.

NOTE: In addition to models that are strongly based on standard Python libraries that are designed to estimate the time offset between series of time-tagged events, there are different modules developed by programmers, physicists and engineers alike, for specific problems or even data formats (an example is listed in the Code section).

2.2. The Shutter Method

Standard Python modules provide us with a solid set of tools to compute the cross-correlation and find the time offset between two signals. In reality, however, the size of the data we have to deal with in experiments with QKD protocols often makes it very impractical to just feed those methods with raw data and expect them to return the output in a short time. It is therefore very important to have a clear overview of the data so we can notice patterns and use them to our advantage, so we can,for example, reduce the size of the data to be handled without compromising the final result.

The data we are working with here is an example of that. To be able to detect such patterns, it is very common to plot a histogram of the data because it offers valuable visual insights. In the graph below, you see the histogram of the first set of data we're working with (named Timetag1)

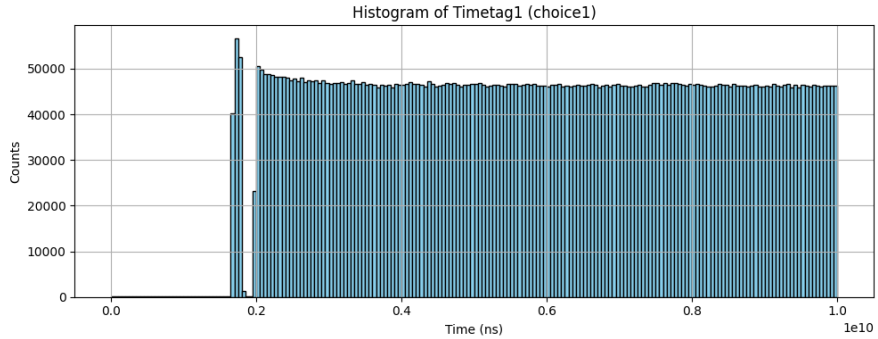


Figure 2.1.: Histogram of raw data from the first set (Timetag1)

One particular thing that does perhaps instantly catch your eye in the figure above is that the histogram is empty up to some point on the time axis. Equipped with this fact, we can write a code so that it leaves out part of the data since that is not relevant.

The following code snippet (see find_start file in the original code) shows how the data can be filtered to remove unwanted noise and reduce the size of the signals to objects of significant relevance.

```
start1 = np.argmax(h1<max(h1)/5)
start2 = np.argmax(h2<max(h2)/5)

h1 = np.delete(h1, slice(start1))
h2 = np.delete(h2, slice(start2))
```

Another important aspect of this approach is that we start big to find a rough estimation, and then we work our way to a finer resolution. The figure above is a visual illustration of what we're referring to as the coarse-to-fine resolution. For instance, say we have two

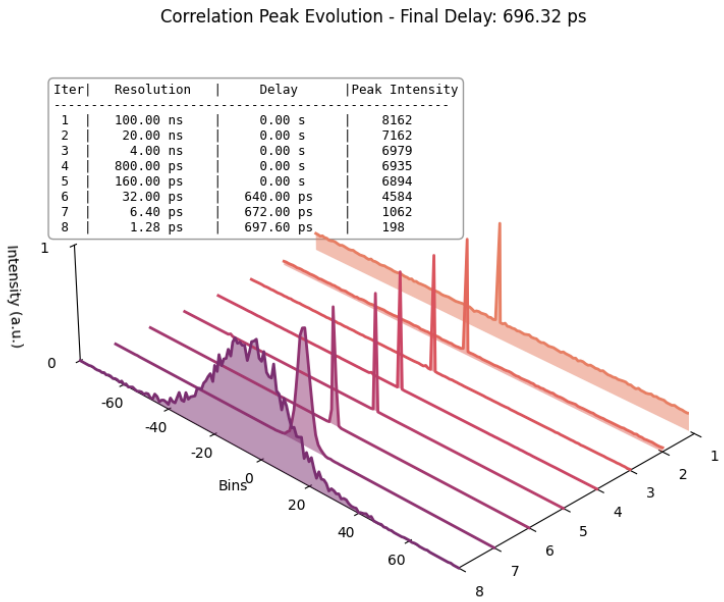


Figure 2.2.: An illustration of the coarse-to-fine approach

sets of time-tagged events spanning 10ms, and we want to find the shift between the two for which the coincidence counts are the highest. In the first step, we could, for example, study the change in the coincidence counts by shifting one signal in ms steps. Given that we have found a time offset in the millisecond range for which the correlation between the two is the highest, we can increase the resolution in the next step by defining a search range around that first rough estimation in higher resolution (say, nanoseconds). By recursively repeating this procedure, we can effectively achieve our goal: a high-resolution estimation of the time offset and coincidence counts.

Time Offset Estimation

Now let us check how this algorithm finds out the time offset between the two time-tagged signals and computes the coincidences

Step 1: Histogram the data

```
choice5 = np.around(choice1/binsize)
choice6 = np.around(choice2/binsize)
```

Step 2: Filter out the double counts in the A and B arrays

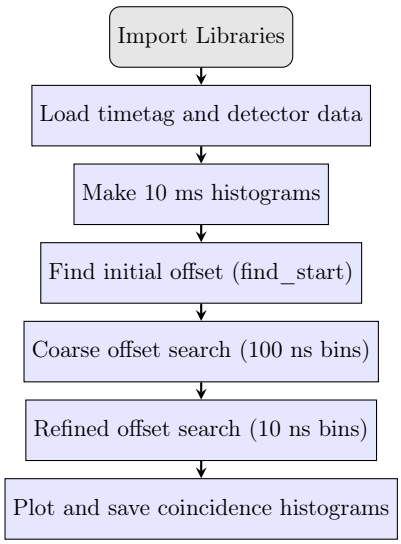
```
        A = choice5
        double = A[:-1][A[1:] == A[:-1]]

```

```
        whereA = np.logical_not(np.isin(choice5,double))
        choice5 = choice5[whereA]
        choice1 = choice1[whereA]
        detector1 = detector1[whereA]

        B = choice6
        double1 = B[:-1][B[1:] == B[:-1]]
        whereB = np.logical_not(np.isin(choice6, double1))
        indexB = np.argwhere(whereB == False)

        #randomly keep one element from every double pair
        r_unif = []
        r = random.randint(2, size=(2))
        for i in range(np.int(len(indexB)/2)):
            r = random.randint(2, size=(2))
            while sum(r) != 1:
                r = random.randint(2, size=(2))
            else:
                r_unif.append(r[0])
                r_unif.append(r[1])

        whereB[indexB] = np.reshape(r_unif, (len(indexB), 1))
        choice6 = choice6[whereB]
        choice2 = choice2[whereB]
        detector2 = detector2[whereB]
```

Step 3: Count the coincidences using the intersect method

```
x = datetime.datetime.now()
result=np.intersect1d(choice5,choice6)
counter_new = len(result)
```

Here's a simple flowchart of this algorithm



2.3. The Cross-Correlation Method

Going through a few iteration steps in our code can give the algorithm an edge in terms of how good the resolution of estimation can get. However, they can also result in a longer processing time, and we may need to explore ways to mitigate them. Think of a situation where your timestamps are uniformly scattered in time, and you cannot apply a filter like in the shutter method to extract a sample that would be a valid representative of the whole. The question is, where would you turn your focus to in such a case?

Fortunately, we don't need to always go through that many iteration steps or even cut to 1ns resolution to figure out the shift between the two signals. Instead, we're going to achieve this in only two steps

Step 1: Compute the cross-correlation between the arrays for a whole range of time scans and find the time delay for which the cross-correlation is the highest.
Step 2: Fine-tune the resolution by finding the distance of the peak of coincidence counts in the cross-correlation plot from the center (zero delay)

Time Offset Estimation

In the following, let us examine the code section responsible for determining the time offset more closely. As it is common, we first histogram the data, as cross-correlating raw data is impractical.

```
common_min = min(min(a), min(b))
common_max = max(max(a), max(b))
time_bin = 10000

bins = np.arange(common_min, common_max + time_bin, time_bin)
hist1, _ = np.histogram(a, bins=bins)
hist2, _ = np.histogram(b, bins=bins)
```

In the next step of our algorithm, we use the scipy.correlate method to find the cross-correlation between hist1 and hist2, and the built-in method signal.correlation_lags to compute the time offset between the two

```
corr = signal.correlate(hist1, hist2, mode='full')
corr_lag = signal.correlation_lags(len(hist1), len(hist2), mode='full')
best_delay = corr_lag[np.argmax(corr)]
best_delay_ps = best_delay * time_bin
print(f"The rough estimate of time delay : {best_delay_ps}")
```

Next, we increase the resolution of the estimation by finding the distance of the peak from the center

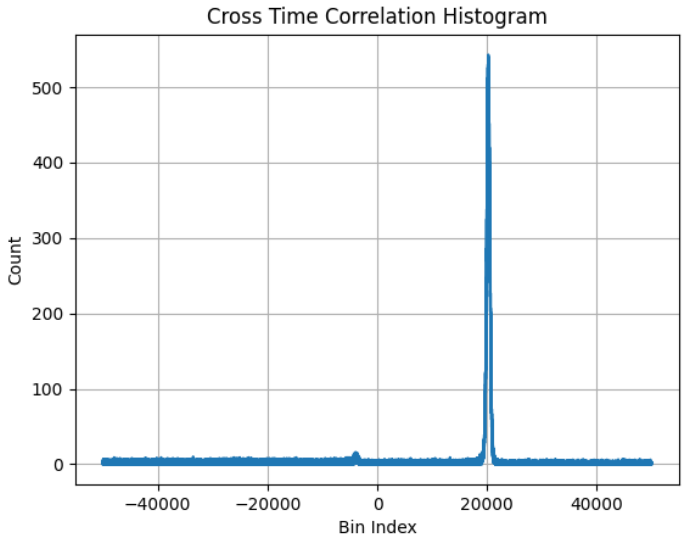


Figure 2.3.: The histogram of coincidence counts for rough estimation of time offset

```
#Find the fine delay
origin_index = len(vec) // 2
peak_index = np.argmax(vec)
x_peak = peak_index - origin_index
distance = abs(x_peak)
binsize = 1 << binshifter if binshifter != 0 else 1
time_distance = distance * binsize
```

This block of code will compute the distance of the peak in Fig.2.3 to the center (zero time delay), which is the fine-tuning of our estimation. Finally, we will update the t_in_Iq variable by this shift to center the peak of the plot around zero delay.

```
#Apply final fine delay
t_in_Iq += time_distance

print(f"Peak index: {peak_index}")

print(f"X-axis value of peak: {x_peak}")

print(f"Distance from origin (bins): {distance}")

print(f"Distance from origin (picoseconds): {time_distance}")
```

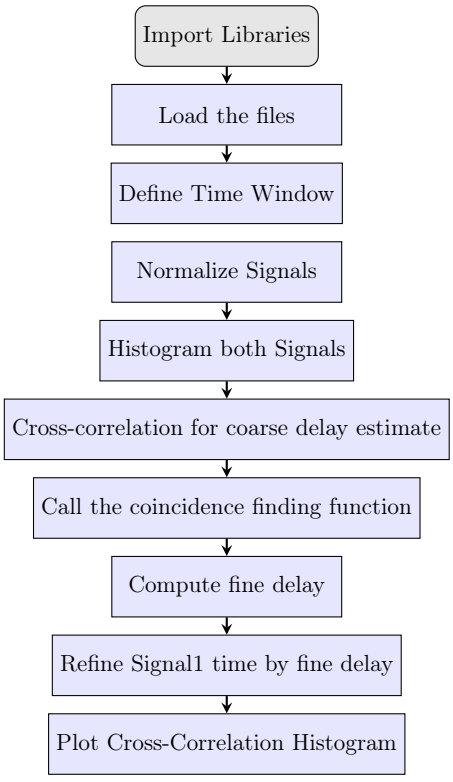
where vec is an array defined within the coin_peak_find function, and it contains the number of coincidences for different time delays of the signals (see Code section).

Coincidence Finding Function

In contrast to the shutter method we define a coincidence-finding function as follows

```
def find_Coin(signal_1, signal_2, BinSize):
    Coincidence = 0
    n = 0
    nn = 0
    for _ in range(len(signal_2)):
        diff = signal_1[n] - signal_2[nn]
        if np.abs(diff) < BinSize / 2:                #BinSize is the
    coincidence window
            Coincidence += 1
            n += 1
            nn += 1
        elif diff < 0:
            n += 1
        else:
            nn += 1
        if n >= len(signal_1):
            return Coincidence
    return Coincidence
```

The following is a simple flowchart of the algorithm



3. Algorithm Analysis

To be able to tell the differences and the advantages of one algorithm over the other, we need to run an algorithm analysis. In general, applying a comprehensive algorithm analysis is very complex(although very important) as it requires taking into account many different factors, many of which are hard to control. Since time is a very precious resource, the time it takes for an algorithm to run is a good quality metric. Thus, for all practical purposes of this thesis, we will focus our analysis on one key aspect: Running time as a function of the size of the input. To be more specific, we investigated the time it took both algorithms to estimate the time offset and count the coincidences. Note that to allow proper performance comparison, we have modified our code (see Code section) so that both algorithms have the same time offset resolution (10ns).

3.1. Experimental Algorithm Analysis

A typical way to analyse the time performance of an algorithm as a function of the input size is by running the program for inputs of various sizes and measuring the time each round takes to complete. This is what is known as experimental algorithm analysis.

In our experimental algorithm analysis, we ran the original code and the one proposed by us for inputs of various sizes, ranging from 2 million to 100 thousand. We chose 2 million as the maximum input, as that is the maximum length of one of the arrays loaded in the code. To ensure statistical significance, we ran each code 20 times for each input size n and averaged the running times. To avoid bias and to ensure a consistent hardware and software ecosystem, each run was separated by 15 minutes in time, allowing the CPU to cool down and be at full capacity at every round of measurement. Furthermore, no program was running in the background during the whole test.

At first, we measured the time needed for both algorithms to count the coincidences. For that, we made use of the datetime Python module. It turned out that their performance in coincidence counting is pretty much the same. On average, it took 2.4 seconds for both algorithms to count the coincidences, and this did not change much with input size. Therefore, the results below focus only on the time complexity for finding the time offset of each method.

3.2. Results

In the following section, we present a visualization of the results of our experimental analysis.

The Shutter Method Results

Size input n (in millions)	**Average running time (m:s)**
2.00	6:57
1.75	6:53
1.50	6:49
1.25	6:46
1.00	6:44
0.75	6:39
0.50	6:11
0.25	5:57
0.10	5:32

Table 3.1.: Average running time of the shutter method algorithm for different input sizes

Here's a visualization of this data

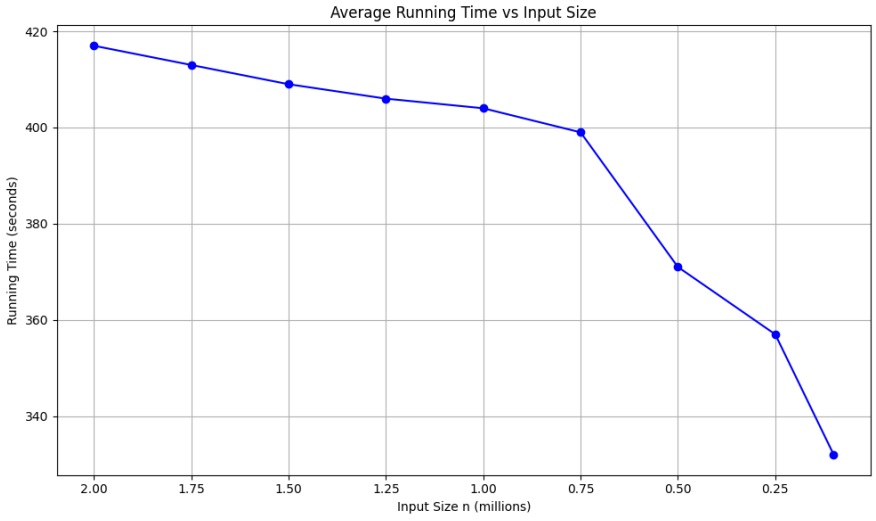


Figure 3.1.: Running time of the Shutter method algorithm as a function of input size n

The Cross-Correlation Method Results

Size input (n, in millions)	**Running time (s)**
2.00	18
1.75	15
1.50	14
1.25	14
1.00	13
0.75	13
0.50	12
0.25	11
0.10	11

Table 3.2.: Average running time of the Cross-Correlation Method algorithm for different input sizes

Here's a visualization of this data

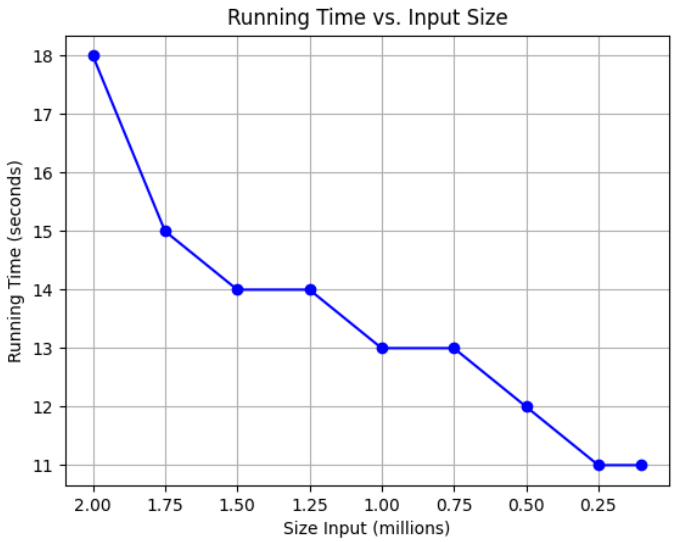


Figure 3.2.: Running time of the Cross-Correlation Method algorithm as a function of input size

Side note: The average running time was rounded up to a second because the difference between the algorithms is in the range of minutes.

An experimental analysis is a solid tool for performance measurement, but with a few obvious limitations:

- The contextual hardware and software environment can affect the running time of our algorithm.
- We can't test the running time for every input size.
- The algorithm must be fully executed to study its running time as a function of the input size. From a practical standpoint, we'd like to avoid this scenario, as we'd have to waste time and computational power implementing an algorithm that we would eventually find out through a deeper algorithm analysis to be not optimal.

To overcome these challenges, in the following, we're going to aim for a better algorithm analysis whose outcome is independent of the hardware and software environment, encompasses test inputs of all sizes, and requires no need for implementation.

3.3. Theoretical Algorithm Analysis

In this section, we're going to compare the performance of our code by digging a bit deeper into a more complex algorithm analysis. In contrast to the experimental analysis, this approach does not require running the code for every input size and seeing how the running time changes with that, for reasons already listed above. Instead, we will try to identify a set of basic operations (also referred to as primitive operations) whose running time with input size can be easily recognized. Here are a few primitive operations to give you an idea:

- Accessing a list element
- Arithmetic calculations (adding two numbers, for example)
- Calling a function
- A for loop, etc.

By definition, basic operations are instructions with a constant execution time. Assuming that different basic operations take almost the same time to execute, we can then take the running time of the whole algorithm to be approximately equal to the number of primitive operations contained in the code.

To study the time performance of an algorithm with input size, we define a function f(n) that characterizes the number of primitive operations to be performed. Here are seven such functions that are often encountered in algorithm analysis

- The constant function
- The logarithm function
- The linear function
- The NLogN function
- The quadratic function
- The cubic function and other polynomials
- The exponential function

The Big Oh notation

As already mentioned, conducting a deeper algorithm analysis is ideal, but for all practical purposes of this project, it is enough to consider the time factor in the asymptotic case. For instance, it is often enough to just show that the running time of an algorithm grows proportionally with input size n, like this simple code given below

```
def find_max(data):
      biggest=data[0]
      for val in data:
            if val >biggest:
                  biggest=val
      return biggest
```

A very common theoretical formalism to study the running time of an algorithm as a function of the input size n is the Big Oh notation. Given two functions f(n) and g(n), we say that f(n) is Big Oh of g(n) if the following condition is met

$$f(n) < c \cdot g(n) \quad \text{for all } n > n_0 \tag{3.1}$$

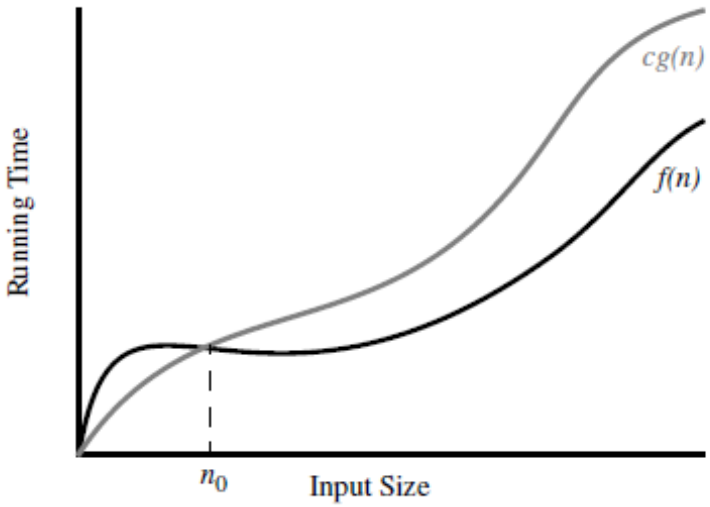


Figure 3.3.: A visualization of the Big Oh notation

Big Oh notation: The Shutter Method

Now, let us see how the running time of our code for finding the time offset in the shutter method changes with input size by using the Big Oh notation. To do so, we're going to define an O(n) function for each stage of the algorithm and, in the end, pick out only the dominant term.

Stage 1: Loading files

```
choice1 = np.load('Timetag1.npy')
choice2 = np.load('Timetag2.npy')
detector1 = np.load('Detectors1.npy')
detector2 = np.load('Detectors2.npy')
```

The time it takes for the program to load the data is linear with size n (with n referring to the total size of the loaded files), O(n).

Stage 2: Histogramming

```
hist1=np.delete(choice1, np.argwhere(choice1>hmx1))
hist2=np.delete(choice2,np.argwhere(choice2>hmx2))

h1,ax1=np.histogram(hist1,bins=bins1,range=(mn1,hmx1)
h2,ax2=np.histogram(hist2,bins=bins2,range=(mn2,hmx2))
```

This also changes linearly with input size; therefore, O(n)

Stage 4: Slicing arrays The slicing operation is implemented a few times in the code, and it is straightforward to argue that its running time changes linearly with n, so its corresponding Big O notation is O(n)

Stage 5: For loop This is the part that contributes the most to the running time of the algorithm.

```
for offset in range(-off_range,off_range):
            result=np.intersect1d(choice7,choice8-offset)
            counter_new=len(result)
            result_ all.append(counter_new)
            off_all.append(offset)
            if counter_new>counter_store:
            counter_store=counter_new
            best_offset=offset
            counter_new= 0
```

The time complexity for the intersection method is O(min(len(a),len(b))), however, if both arrays are not sorted, then the O(nlogn) function best describes the running time of this operation.

Final Stage: Plotting
Obviously O(n)

Now putting everything together, the Big O notation for the whole code is the sum of these parts

$$f(n) = O(n) + O(n) + O(n) + O(n) + O(n) + O(r \cdot n \log n) + O(n) \tag{3.2}$$

Note that we're assuming that both arrays are of the same size and we're denoting the total size of the input as n, since we're only interested in the order of growth and not certain constants.

As is customary for Big O analysis, we only consider the dominant term, which is O(r $\cdot n \log n$).

Big O Notation: The Cross-Correlation Method

Now, let us figure out the Big O notation for the algorithm we propose for finding the time offset.

Stage 1: Loading data Similarly, the Big O notation for this is O(n)

Stage 2: find_coin function The worst-case scenario in this function is O(n) as the for loop goes at most once over an element of the array (signal2).

Stage 3: coin_peak_find function Depending on the binshifter, there are two ways the algorithm can proceed
Option a: binshifter !=0
There is a while loop which, in the worst case scenario, can have a time complexity of O(n) where n, in this case, is not the length of the input, but the sample number. The for loop just beneath has a similar linear dependence on the limit parameter.
Option b: binshifter==0
Again O(sample+limit)

To conclude, the Big O function of this coin_peak_find is O(sample+limit), but since these are constants and we're not including them

Stage 4: Cross-correlation
The scipy.correlate method applies FFT to compute the cross-correlation, and the well-known Big O function for this method is O(nlogn)

Stage 5: High-resolution estimation and plotting

The argmax method runs once over n elements, thus its running time is linearly dependent on the size of n. Finally, plotting's time complexity also changes linearly with input size n.

To conclude, the dominant term in the Big O function of our code is

$$O(nlogn) \tag{3.3}$$

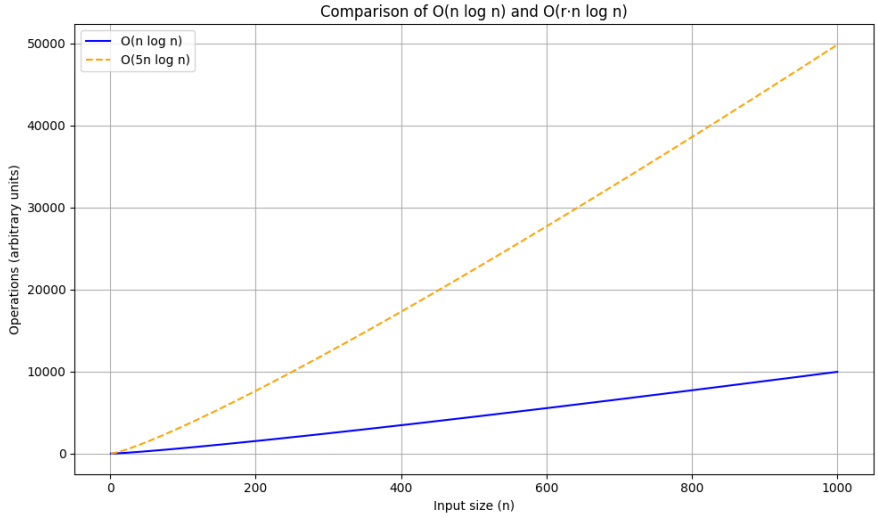


Figure 3.4.: The Big Oh notation of the Shutter Method(blue) and the Cross-Correlation Method(orange)

Discussion

In this thesis, we introduced an improved Python algorithm designed to find the time offset and recover the coincidence counts of SPDC-generated streams of photons. By conducting a sound experimental and theoretical algorithm analysis, we were able to prove the efficiency of our model compared to the one developed for the paper on the Digital Quantum Payment Scheme. Our analysis focused mainly on the time performance as the most important quality metric of the algorithm. In addition, we ran the codes for different input sizes n in the same hardware environment and were able to visualize the dependence of the time performance on the data size.

Although our model performed better in this post-processing algorithm analysis, the results of this work must be treated with caution and by no means considered a universal solution to every problem of this nature. Nevertheless, we believe that the overall methodologies and the principles applied here are worth noting and, with proper modification, can find applications in many problems related to the time-offset estimation and coincidence counting. In particular, what we could call the "coarse to fine" approach can prove very useful in dealing with huge data, such that we start at the top and work our way to higher resolution.

In conclusion, we developed a highly efficient post-processing Python algorithm for QKD that manages to determine the time offset between two signals in a matter of seconds, compared to the previous model, which took a few minutes. In the future, it would be interesting to test the upper limits of the algorithm for larger data inputs and similarly its resilience to noise.

Code

This is the code proposed by us to estimate the time offset between two sets of time-tagged events and compute the coincidences.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

# function for finding coincidence
def find_Coin(signal_1, signal_2, BinSize):
    Coincidence = 0
    n = 0
    nn = 0
    for _ in range(len(signal_2)):
        diff = signal_1[n] - signal_2[nn]
        if np.abs(diff) < BinSize / 2:
            Coincidence += 1
            n += 1
            nn += 1
        elif diff < 0:
            n += 1
        else:
            nn += 1
        if n >= len(signal_1):
            return Coincidence
    return Coincidence

# function for finding the peak and histogramming the coincidences
def coin_peak_find(chA, chB, limit, sample, binshifter):
    mean = 0.0
    vec = []

    if binshifter != 0:
        binsize = 1 << binshifter
        vec = [0] * (limit >> binshifter)

        a, b = 0, 0
        HSpos = binsize * len(vec) // 2
        HSneg = -HSpos

        while a < len(chA) and b < len(chB) and (a + b) < sample:
            deltaT = chA[a] - chB[b]
            if deltaT <= HSneg:
                a += 1
            elif deltaT >= HSpos:
                b += 1
```

```
            else:
                vec[(HSpos + deltaT) >> binshifter] += 1
                if deltaT < 0:
                    a += 1
                else:
                    b += 1

        sum_vals = 0.0
        for i in range(len(vec)):
            mean += vec[i] * (i + 1)
            sum_vals += vec[i]

        if sum_vals == 0 or (mean / sum_vals - HSpos) != (mean / sum_vals - HSpos):
            mean = 0
        else:
            mean = binsize * mean / sum_vals - HSpos + binsize / 2
    else:
        vec = [0] * limit

        a, b = 0, 0
        HSpos = len(vec) // 2
        HSneg = -HSpos

        while a < len(chA) and b < len(chB) and a < sample:
            deltaT = chA[a] - chB[b]
            if deltaT <= HSneg:
                a += 1
            elif deltaT >= HSpos:
                b += 1
            else:
                vec[HSpos + deltaT] += 1
                if deltaT < 0:
                    a += 1
                else:
                    b += 1

        sum_vals = 0.0
        for i in range(len(vec)):
            mean += vec[i] * (i + 1)
            sum_vals += vec[i]

        if sum_vals == 0 or (mean / sum_vals - HSpos) != (mean / sum_vals - HSpos):
            mean = 0
        else:
            mean = mean / sum_vals - HSpos

    return mean, vec


#Load data
Signal_1_Iqoqi2 = np.load('Signal_1_Iqoqi2.npy')
```

```
Signal_1_Bis = np.load('Signal_1_Bis.npy')


Time_Duration = 0.5 * 1e12  # in picoseconds
BinSize = 1000  # picoseconds
X = 495

t_in_Iq = 383475907881742
# t_in_Iq = 383475907881742 + 0
t_end_Iq = t_in_Iq + Time_Duration
Signal1 = Signal_1_Iqoqi2[(Signal_1_Iqoqi2 >= t_in_Iq) & (Signal_1_Iqoqi2
    < t_end_Iq)]
print(len(Signal1))

t_in_Bis = 374428836466669
t_end_Bis = t_in_Bis + Time_Duration
Signal2 = Signal_1_Bis[(Signal_1_Bis >= t_in_Bis) & (Signal_1_Bis <
   t_end_Bis)]

#Normalize the signals for histogram
Signal1 = Signal1 - t_in_Iq
Signal2 = Signal2 - t_in_Bis

#Trim the signals
a = Signal1[:500000]
b = Signal2[:500000]

#histogram the data and find the correlation
common_min = min(min(a), min(b))
common_max = max(max(a), max(b))
time_bin = 10000

bins = np.arange(common_min, common_max + time_bin, time_bin)
hist1, _ = np.histogram(a, bins=bins)
hist2, _ = np.histogram(b, bins=bins)

corr = signal.correlate(hist1, hist2, mode='full')
corr_lag = signal.correlation_lags(len(hist1), len(hist2), mode='full')
best_delay = corr_lag[np.argmax(corr)]
best_delay_ps = best_delay * time_bin
print(f"The rough estimate of time delay : {best_delay_ps}")

#Shift Signal1 by the coarse time delay
t_in_Iq += best_delay_ps
t_end_Iq = t_in_Iq + Time_Duration

# Re-align Signal1 and Signal2
Signal1 = Signal_1_Iqoqi2[(Signal_1_Iqoqi2 >= t_in_Iq) & (Signal_1_Iqoqi2
    < t_end_Iq)]
Signal1 = Signal1 - t_in_Iq
Signal2 = Signal2

#call and print the coincidence function
```

Code

```
A_0 = Signal1 + X
B_0 = Signal2
output = find_Coin(A_0, B_0, BinSize)
print(f"Initial coincidence count: {output}")

#Parameters
limit = 100000
sample = 10000000
binshifter = 0

#call the coin_peak_find function
mean, vec = coin_peak_find(A_0, B_0, limit, sample, binshifter)

#Find the fine delay
origin_index = len(vec) // 2
peak_index = np.argmax(vec)
x_peak = peak_index - origin_index
distance = abs(x_peak)
binsize = 1 << binshifter if binshifter != 0 else 1
time_distance = distance * binsize

#Apply final fine delay
t_in_Iq += time_distance
print(f"Peak index: {peak_index}")
print(f"X-axis value of peak: {x_peak}")
print(f"Distance from origin (bins): {distance}")
print(f"Distance from origin (picoseconds): {time_distance}")

#Plot
plt.plot(range(-len(vec) // 2, len(vec) // 2), vec)
plt.xlabel('Bin Index')
plt.ylabel('Count')
plt.title('Cross Time Correlation Histogram')
plt.grid(True)
plt.show()
```

Sidenote: Notice that the code used in the test was modified to 10ns resolution to allow for a correct comparison, since the Cross-Correlation Method has a higher resolution estimation order (in picoseconds)

The original code for the demonstration of digital quantum payment is accessible via this link: https://zenodo.org/records/8020667 whereas the corresponding data is found

here: https://zenodo.org/records/7979319

Additional resource: https://github.com/Peter-Barrow/tangy/tree/main/.github

A. Appendix

A1 SPONTANEOUS PARAMETRIC DOWN-CONVERSION

Spontaneous Parametric Down Conversion (shortly referred to as SPDC) is a second-order nonlinear optical process during which a single photon interacts with a nonlinear medium, leading to the creation of two photons of lower energy. By convention, these two photons are referred to as the signal and the idler photon.
Depending on the relationship of the polarization states of the pump, the signal and the idler, we differentiate two types of SPDC:

Type I - The signal and the idler have the same polarization state, but orthogonal to that of the pump
Type II - The signal and the idler have orthogonal polarization states

Upon making the parametric approximation (also known as the non-depleted pump approximation), the Hamiltonian governing the interaction of the photon with the nonlinear media in the Type I SPDC is reduced to:

$$\hat{H}_I = \hbar\eta \hat{a}_s^\dagger \hat{a}_i^\dagger + \text{H.c.} \tag{A.1}$$

where $a_s^\dagger$ and $a_i^\dagger$ are the creation operators of the signal and idler photons and *eta* is approximately given by $\chi^2 E$, where χ^2 denotes the second-order susceptibility of the medium and E is the classical electric field of the pump.

Let us assume that before the interaction, the joint quantum state of both the signal and the idler is given by

$$|\psi\rangle_0 = |0\rangle_s |0\rangle_i \tag{A.2}$$

In time, this initial quantum state will change according to

$$|\psi(t)\rangle = e^{-it\hat{H}_I/\hbar}|0\rangle \tag{A.3}$$

Since $\hat{H}_I$ it has no explicit time dependence, we can Taylor expand this object to get the following

$$|\psi(t)\rangle \approx \left[1 - it\frac{\hat{H}_I}{\hbar} + \frac{1}{2}\left(-it\frac{\hat{H}_I}{\hbar}\right)^2\right]|\psi(0)\rangle \tag{A.4}$$

where only terms up to the second order are considered. Plugging in the expression for $\hat{H}_I$ given above and doing the math, we finally get

$$|\psi(t)\rangle = \left(1 - \frac{\mu^2}{2}\right)|0\rangle_s|0\rangle_i - i\mu|1\rangle_s|1\rangle_i \tag{A.5}$$

where $\mu = \chi t$ and the term of second order is left out.

A2 Fast Fourier Transform(FFT)

The Fast Fourier Transform is a highly efficient algorithm for computing the Discrete Fourier Transform of a function. Cooley, J.W and Turkey, J.W are credited with the development of this technique in their paper in 1965 ''An Algorithm for the Machine Computation of the Complex Fourier Series". However, the famous German mathematician, Karl Gauss, had used FFT in his work over a century ago. The Discrete Fourier Transform of a sequence consisting of n points is given by

$$X(j) = \sum_{k=0}^{N-1} A(k)W^{jk} \tag{A.6}$$

where

$$W = e^{-\frac{2\pi i}{N}} \tag{A.7}$$

One can check that it takes $(N-1)^2$ complex multiplications and $N(N-1)$ complex additions to calculate W. For large N, the complexity of each can be approximated to N^2. As such, a straightforward calculation of this mathematical object is long and tedious.
Suppose N is a composite number, meaning $N = r_1 r_2$, and let us denote the indices in (A2.1) as

$$j = j_1 r_1 + j_0, \quad \text{with } j_0 = 0, 1, \ldots, r_1 - 1, \quad j_1 = 0, 1, \ldots, r_2 - 1. \tag{A.8}$$

Therefore, we can write

$$X(j_1, j_0) = \sum_{k_0} \sum_{k_1} A(k_1, k_0)\, W^{jk_1 r_2}\, W^{jk_0} \tag{A.9}$$

which can be written in the following way

$$X(j_1, j_0) = \sum_{k_0} A_1(j_0, k_0)\, W^{(j_1 r_1 + j_0)k_0} \tag{A.10}$$

where we defined the new array

$$A_1(j_0, k_0) = \sum_{k_1} A(k_1, k_0)\, W^{(j_1 r_1 + j_0) k_0} \tag{A.11}$$

At this point, it is easy to tell that the total number of operations needed to obtain A_1 and X is

$$T = N(r_1 + r_2) \tag{A2.4}$$

By repeating this procedure recursively, we find that the number of operations is

$$T(r) = rN \log_r N \tag{A2.5}$$

The mathematically inclined reader is encouraged to read the paper by James W. Cooley and John W. Tukey [10].

A3 Cross-correlation

Cross-correlation is a measure of the similarity between two signals, and it is mathematically defined as follows

$$f(t) = \int_{-\infty}^{+\infty} x(\tau)\, h(t + \tau)\, d\tau \tag{A3.1}$$

$$f[n] = \sum_{k=-\infty}^{+\infty} x[k]\, h[k - n] \tag{A.12}$$

for discrete time sequences x[t] and h[t].

A4 The documentation of scipy.correlate

Syntax:

signal.correlate(in1, in2, mode='full', method='auto)

Parameters:
in1 and in2 are array-like objects whose correlation is to be computed

IMPORTANT NOTE: the second object must have as many dimensions as the first.

Optional parameters:

- **mode** - a string input indicating the size of the output (full, valid, same)

- **method** - a string input indicating the method to compute the cross-correlation (auto, direct,fft). Note that if not specified, the function will choose 'auto'.

Bibliography

[1] Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *IEEE International Conference on Computers, Systems and Signal Processing*, page 175–179, 1984.

[2] Robert W. Boyd. *Nonlinear Optics*. Academic Press, 2008.

[3] Lukas Bulla, Kristian Hjorth, Oskar Kohout, Jan Lang, Sebastian Ecker, Sebastian P. Neumann, Julius Bittermann, Robert Kindler, Marcus Huber, Martin Bohmann, Rupert Ursin, and Matej Pivoluska. Distribution of genuine high-dimensional entanglement over 10.2 km of noisy metropolitan atmosphere. *Phys. Rev. A*, 107:L050402, May 2023.

[4] David C. Burnham and Donald L. Weinberg. Observation of simultaneity in parametric production of optical photon pairs. *Phys. Rev. Lett.*, 25:84–87, Jul 1970.

[5] J.W. Cooley and J.W. Tukey. An algorithm for the machine computation of the complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[6] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006. Online edition, Oxford Academic, 31 Oct. 2023.

[7] S. Friberg, C.K. Hong, and L. Mandel. Measurements of time delays in the parametric production of photon pairs. *Phys. Rev. Lett.*, 54:2011, 1985.

[8] Christopher C. Gerry and Peter L. Knight. *Introductory Quantum Optics*. Cambridge University Press, 2005.

[9] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley, 2013.

[10] Federico Grasselli. *Introduction*, pages 1–5. Springer International Publishing, Cham, 2021.

[11] Taehyun Kim. *Applications of Single-Photon Two-Qubit Quantum Logic to the Quantum Information Science*. PhD thesis, Your University Name, Year.

[12] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary edition, 2010.

[13] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals and Systems*. Prentice Hall, 1996.

[14] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications*. Prentice Hall, 2006.

[15] P. Schiansky, J. Kalb, E. Sztatecsny, et al. Demonstration of quantum-digital payments. *Nature Communications*, 14:3849, 2023.