The Incomplete Technical Document on Pedagogical 'Q'KD Implementation

Experimental Session

November 6, 2018

Dedicated to the people that make this happens.

Contents

Preface

Dear Qcumbers,

This book accompanies the gamified do-it-yourself experiments of QCamp 2018 at the Centre of Quantum Technologies. It gives a cohesive picture on the technical implementation of the experiments, such that the reader may be able to reconstruct parts of the setup elsewhere.

The experiments are designed to convey the ideas and steps of a Quantum Key Distribution implementation and also include a remark about potential security issues. It is imperative to note that this is not a QKD implementation, even though they share a lot of properties and similarities.

In itself, the book will help to clarify the nitty-gritty details of the implementation, and serve as a resource to the students in the Quantum Camp Experimental Session. As this is the first iteration of the event, and probably even the first iteration of writing this, the authors would like to express apologies should there be mistakes in the writings or explanations.

Best of luck to all the participants and have fun!

Qcamp2018 Team Experimental

1

Classical Channel

> *"Probably a common misconception to the general public about QKD is that the whole process runs on the quantum channel. On the contrary, quantum channel is only used to distribute keys securely, and the encrypted message itself needs to run along the classical channel."*
>
> – Some random made-up quote

The first part of this series is about building a classical channel. With the advent of technology and internet, we might already take for granted the pervasiveness of this classical communication. Nowadays, it is very easy to tap into the extensive web of the internet. Every connection, whether to any other parties or to the internet, is by itself a classical channel.

This pedagogical exercise uses a "less common, but probably more visually pleasing" approach to establish a classical channel, which is infrared (IR) signals. Nowadays, IR signals see a lot of application in TV, AC and projector remote controls.

1.1 IR signals

In this section, we discuss how we implement classical communication channel with IR signals. The process itself is very similar to how a TV remote control tells the TV to turn on/off, change channel, or volume. In the IR signal that the remote sends, there will be packets of information, i.e. strings of ones and zeros that are sent to the TV.

1.1.1 Electrical connections

Sender

The sender circuit consists of an IR lED (TSAL7400), a few resistors, and a transistor (BC547). The purpose of the transistor is to provide higher

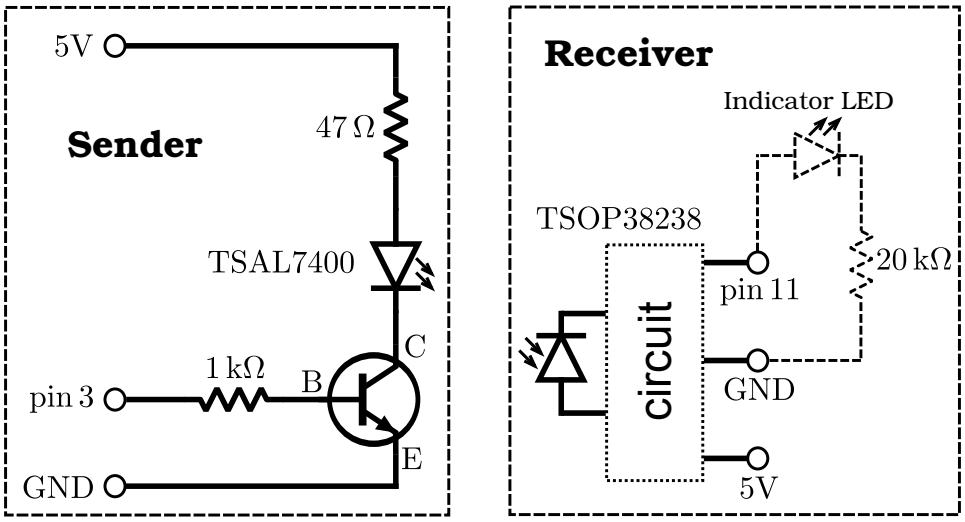

Figure 1.1: Electrical circuit of the IR sender and receiver

current to the IR LED ($\sim 60\,\text{mA}$) than the maximum allowable rating of the Arduino pin (40 mA). To achieve this, we can use $R_B \approx 1\,\text{k}\Omega$ as the base resistor and $R_C \approx 47\,\Omega$ as the collector resistor. We use pin 3 on the Arduino to switch on/off the IR LED. *Warning: The transistor can burn easily if connected in the wrong polarity.*

To test whether the circuit works, you can try sending blinking signals (see Section 1.2.1). You should be able to see the infrared light using your phone camera (except iPhone of course, which has an excellent IR filter).

Receiver

The IR receiver module (TSOP38238) consists of an IR photodiode with a built-in demodulation circuitry. The output of integrated circuit / photodiode module can be directly connected to Arduino pin 11. *Warning: The IR receiver module can burn easily if connected in the wrong polarity.*

You may add in an (optional) indicator LED to monitor the output of the IR receiver module (indicated as dashed components in Figure 1.1). Note that you might need to use quite a high resistance value ($\sim 20\,\text{k}\Omega$) as the demodulation circuitry can not output more than a few mAs. Also, note that the pin logic is opposite, i.e. it turns OFF (0 V pin reading or LED off) when it receives an ON signal from the IR sender, and vice versa.

To test the circuit, you can try receiving the blinking signals sent by another party (see Section 1.2.1). If you succeed, the connection between the sender and receiver is successfully established.

Modulation

There is still another ingredient to this IR channel: signal modulation. In simple terms, the IR light is switched on and off (modulated) repeatedly with a very high frequency (38 kHz). On the receiver side, there is an array of electronics that demodulates the signal, i.e. reading only the signal that has been repeatedly switched on and off. Thus, the receiver can only detect signal of frequency 38 kHz (±10%, according to the datasheet).

This process will suppress unwanted noise in the IR range that might come from elsewhere, i.e. room fluorescent lamps (∼100 Hz modulation due to power grid).

1.1.2 From NEC to the protocol

We derive our communication protocol from the commonly known NEC (Nippon Electric Company) IR protocol. The logical ones and zeros are defined by the length of the signal in the ON and OFF position (Figure 1.2):

- Logical 1: ON for 560 μs and OFF for 1690 μs
- Logical 0: ON for 560 μs and OFF for 560 μs

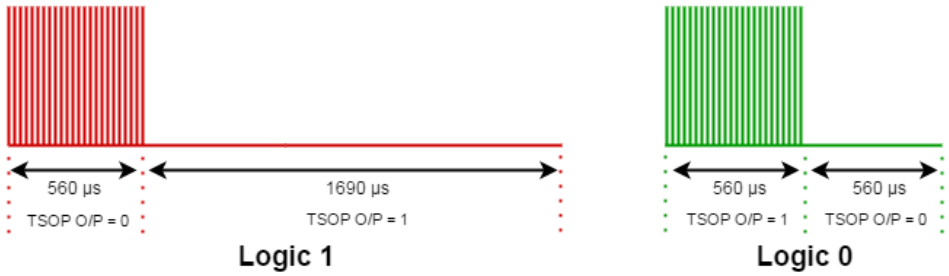

Figure 1.2: The definition of logical 1s and 0s in the NEC protocol. Image source: http://www.snrelectronicsblog.com/8051/nec-protocol-and-interfacing-with-microcontroller/

Every time you press on a button on the remote, it will send a train of pulses, consisting of:

1. Header: ON for 9 ms and OFF for 4.5 ms.
2. Address byte: 8 bits of 1s and 0s containing the information to which device should this signal be delivered to, i.e. to this model of TV.
3. Inverse address byte: The logical opposite of the address byte, for redundancy purpose.
4. Command byte: 8 bits of information of what the device should do, i.e. turn ON/OFF, change channel, etc.

Figure 1.3: Typical example of a pulse train in the NEC protocol. Image source: http://www.snrelectronicsblog.com/8051/nec-protocol-and-interfacing-with-microcontroller/

5. Inverse command byte: The logical opposite of the command byte, for redundancy purpose.

6. Termination pulse: Such that the receiver can determine whether the last bit is 1 or 0.

For our experiment, there is only Alice and Bob, so we don't really need the address byte. Also, as we are not operating in a very noisy environment, we don't need the redundancy bytes either. Thus, we will use all four available bytes (32 bit) in the NEC protocol to send our data. Messages longer than 4 bytes will be cut into 4-byte chunks and send consecutively.

1.2 Programs

The programs to perform Mission 1 are located in the folder "1_Classical".

1.2.1 Arduino program

The Arduino program "ArduinoClassical.ino" consists of several subroutines to perform tasks related to testing and communicating IR signals, which can be accessed via serial communication. It can be used for both the sender and the receiver. There are some set parameters to the program, which can be modified at wish (and if you really know what you are doing). The default values for some of the parameters are given below as: (def: #).

To distinguish between serial sent by computer to Arduino (TX) and the serial sent by Arduino to computer (RX), we use the flags (TX) and (RX) in the discussion below. List of serial (TX) commands:

- HELP
  Prints list of serial commands and their short descriptions (in case you forget). Alternatively, you can keep referring to this notes.

- SBLINK
  Sends a blinking signal with a half-period of "blink_time" (def: 200 ms) for a total of "blink_num" (def: 20) periods.

- RBLINK
  Listens for any blinking signal for "blink_obtime" (def: 10 s). Serial (RX) of "BLINK!\n" will be sent for each blinking period detected.

- SEND X
  Sends a 4-byte character X with the protocol defined in Section 1.1.2.

- RECV
  Listen to any 4-byte character sent by other devices sharing the same protocol, and sends the result to the computer via serial (RX). This process will not continue if no signal is received. To cancel this process, you need to send a termination character "#" via serial (TX).

1.2.2 BinaryComm package

These packages are only used to demonstrate the process of sending and receiving 8 bit binary sequences. There are two programs: "send_binary.ino" and "recv_binary.ino". The former should be used by Alice, and the latter by Bob. The 8 bit binary sequence (for example the letter A: 01000001) should be written on the serial monitor of "send_binary.ino", which will then be sent through the IR channel and appears on the serial monitor of "recv_binary.ino".

1.2.3 Python programs

The Arduino program only serves as the backbone for the classical communication. To perform more complicated tasks, we have written several python programs that will aid in performing Mission 1.

devloc_classical.txt

This is a simple text file that contains the information of the Arduino device location (usually "/dev/ttyACM0" or "/dev/ttyUSB0"). The python programs will look into this file prior to running to determine the device location of Arduino, and hence the file needs to be updated should there be any changes in the device or its location.

To check the location of the device after plugging in the USB hub, you might want to run the "dmesg" program in the Linux terminal.

conv_ascii.py

This program converts between different representation of ASCII characters: ASCII string, hex representation, and binary representation.

send_testQ.py

This program sends a test message of "text" (def: "Qc!8") repeatedly every "rep_wait_time" (def: 1 s). This program can be used to ensure that the correct message is being transmitted and received by "recv_testQ.py".

recv_testQ.py

This program listens to a message of "text" (def: "Qc!8"). It should print "Mission successful" if the message is received successfully and the "text" is the same with that in "send_testQ.py". If the "text" is not the same, it will print "Receiving something that does not seem correct".

Both "send_testQ.py" and "recv_testQ.py" can be used to check for the quality of the transmission or for troubleshooting purposes.

send_message.py

This program asks the user to input the message ("tosend_string"), which will be sent through the IR channel afterwards. The message is split into 4-byte chunks (4 characters each time), with a message header of "[STX]×4" and a message footer of "[ETX]×4" to signify the start and end of the message transmission.

Each chunk takes around "rep_wait_time" (def: 0.3 s) to transmit. We have experimented with different waiting time on our computer, and so far 0.3 s is a safe choice (0.2 s is a bit on the edge as some chunks might get lost). This is in part due to serial communication between computer and Arduino ($\sim$ 80 ms), sending and receiving of IR signals ($\sim$ 70 ms), and operating system overhead. For slower computers, you might need to increase "rep_wait_time" to prevent packet loss.

recv_message.py

This program listens to any incoming message sent with our protocol (with "send_message.py"). It first looks for "[STX]×4" message header, and joins 4-byte message chunks. The listening process will terminate when it receives "[ETX]×4" message footer.

chatting.py

This program is a combination of both the programs "send_message.py" and "recv_message.py" in a looping sequential order. Thus, it enables for some form of chatting to happen. The program starts in a sending mode, which can be switched to receiving mode by pressing [Enter].

2

Quantum Channel

> *"If (A1) quantum mechanics is correct, and (A2) authentication is secure, and (A3) our devices are reasonably secure, then with high probability the key established by quantum key distribution is a random secret key independent (up to a negligible difference) of input values."*
>
> – Douglas Stebila, *The Case for Quantum Key Distribution* [1]

In the second part of this series, we describe how to implement a variant [2] of quantum key distribution (QKD) experiment by using lasers, polarisers, and photodetectors. Particularly, we will explore how the secret key can be formed with random keys, random basis choices, and the sifting process.

2.1 Laser signal

The main component of this experiment is the laser signal. Contrary to the previous chapter, which uses the ON/OFF sequence to encode the information, the laser signal described in this chapter uses light polarisation instead.

The section below describes the nitty-gritty details on how the laser signal is produced and detected, along with the protocol that we utilise to ensure that the receiver can receive the signal sent by the sender faithfully.

2.1.1 General schematics

In this section we describe the electrical connections as well as the general schematics of the experiment. The laser light is generated with a laser module (via pin 4), which then passes through a quarter wave plate and a linear polariser (whose angle can be controlled with a stepper module).

[1] https://arxiv.org/abs/0902.2839

[2] Recall that this is not QKD. For more information, read the Preface.

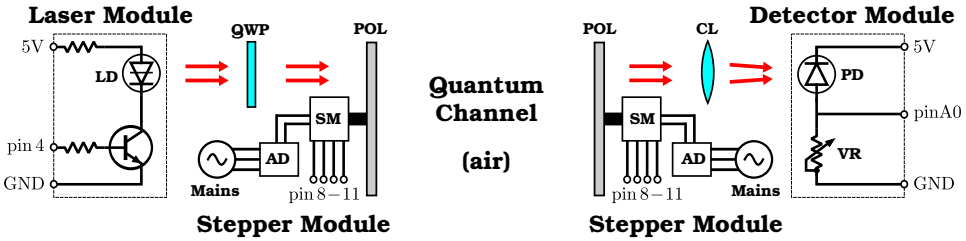

Figure 2.1: The schematics of the QKD experiment. Labels: (LD) Laser Diode, (QWP) Quarter Wave Plate, (POL) Linear Polariser, (SM) Stepper Motor, (AD) Electrical Adapter, (CL) Converging Lens, (PD) Photodiode, and (VR) Variable Resistor.

This linear polariser will produce different polarisations corresponding to different keys and basis choices (or different quantum states).

On the receiver side, another linear polariser will filter the light and project it into a measurement basis. The light will then be focused onto the photodiode with a converging lens. The detector module will output the light power via pin A0.

Sender

The electrical circuit for the sender consists of two main parts: laser module and stepper module. The laser module consists of a 780 nm laser that can be turned on/off with pin 4. *Be careful, as the laser is actually quite bright, even though it appears to be pretty dim, as the wavelength is in near-infrared regime. The laser can actually damage your eyes (similar power to a fancy laser pointer, Class 3A, below 5 mW).*

The stepper module consists of a stepper motor (with its electronics) which rotates a linear polariser. As the stepper motor requires relatively high current ($\sim$ 220-280 mA) and relatively fast switching ($\sim$ 2 ms), we used a Darlington transistor array chip (ULN2002A) powered by an external 5V power adapter. The stepper motor is driven with pin 8 - 11 of Arduino.

As the laser output is linearly polarised, we introduce a 780 nm quarter wave plate to 'depolarise' the laser output, i.e. 'depolarise' in this case means making it circularly polarised, which has very similar effect with depolarisation. Then, after going through the linear polariser, the light will be linearly polarised with similar intensity at any polarisation axis angle.

Receiver

The receiver also consists of two main parts: stepper module (similar to the sender) and detector module. The detector module consists of a photodiode (any 'generic' near-infrared photodiode can do, i.e. SFH213) operating in

reverse biased mode. Before saturation, the light irradiance on the photodiode is proportional to the voltage at A0, and the sensitivity can be tuned with the variable resistor. To produce optimal results and to reduce misalignment sensitivity, we use a converging lens to focus the laser light on the photodiode.

2.1.2 In the sequence

For each polarisation configuration (each bit of 'data'), the stepper motor needs to move to the correct angle, and the laser (detector) needs to send (receive) the light. The main bottleneck of this process is the movement of the stepper motor, which requires around 1.07 s to move 90°. We limit the turning angle [3] between each bit to be 90°, and set that each bits uses 1.5 s, consisting of 1.1 s to rotate the polarisers, 0.1 s empty space [4], 0.2 s to turn on the lasers, and end with 0.1 s empty space. The detector is set to read at the 1.3 s mark of each bit, right in the middle when the laser is on.

2.1.3 Synchronisation

To start each sequence of bits, Alice sends a synchronisation pulse to Bob (500 ms on, 500 ms off, and set to D polarisation). When Bob detects this pulse, he starts his own sequence. From our observation, there is a small offset (less than 5 ms) between Alice and Bob's sequences. Also, Alice and Bob's clock tick in a slightly different rate, so there is a maximum length that Alice can send before the sequences go out of sync. We set the length of our sequence to be 16 bits, but it can well be extended to 128 or 256 bits.

2.2 Key generation

In this section, we will describe each step that needs to be undertaken to generate the secure key. In this experiment, we employ a variant of BB84 quantum key distribution (QKD) scheme, which relies on some basis comparisons and sifting process. To generate the key that is 'independent on the input' [5], we need to find a method to generate 'true' random numbers. We also need to find a way to represent the keys and random numbers in light polarisation.

[3] Because polariser with angles 0° and 180° are basically the same polarisation, you can reduce the range of motion of the polariser to be max 90°, which reduces time of motion.

[4] The empty space is just an added waiting period where Arduino does nothing and only waits. The purpose of this is to time-separate the different steps (easier for debugging).

[5] Contrast this with pseudo random number generator, in which case after you know the initial seed, you can obtain the whole sequence.

2.2.1 Random numbers

For the random numbers, we use the Entropy library that was developed by Walter Anderson [6]. This library harvests random numbers from the timing jitter between two different clocks (the watchdog RC timer v.s. the system clock). The author does not claim that this is the true random source [7], but some quick tests show that the numbers are quite random, unpredictable and pretty superior to other libraries [8], albeit with a relatively low bit generation rate ($\sim 2 \times 32$ bits per second).

2.2.2 Representation

There are 4 polarisation axes in this experiment: (H) horizontal, (D) diagonal, (V) vertical, and (A) antidiagonal, each of them separated by 45°. We represent each polarisation direction with a basis bit and a value bit according to Table 2.1.

Basis	Value	Polarisation
0	0	H
1	0	D
0	1	V
1	1	A

Table 2.1: Representation of polarisation on the basis bit and value bit

To make sense of the table, one can think of the basis bit as either Horizontal-Vertical or Diagonal-Antidiagonal, which we then assign to be 0_B and 1_B respectively [9]. In each basis, there can be two values: 0_V (assigned to H or D) and 1_V (assigned to V or A).

2.2.3 BB84 QKD scheme

In the experiment, we implement a variant of BB84 QKD scheme which follows the following steps:

1. Alice generates 2 sequences of random numbers (basis and value), while Bob generates a sequence of random numbers (basis). The length of the sequence is 16 bits.

2. For each bit, Alice sends the polarised light depending on the basis and value bit sequences, i.e. if the basis bit is 1_B and the value bit is 0_V, then she sends diagonally polarised (D) light.

[6] https://sites.google.com/site/astudyofentropy/home

[7] Probably not when you compare it with randomness from quantum processes

[8] There are some other libraries that make use of the 'random' fluctuation in the analog pin input, for example.

[9] We use the subscript notation X_y, where X is either 0 or 1, and y is its representation.

3. Bob measures the light sent by Alice according to his basis sequence [10]. He then takes note of the measurement results. His measurement results should be binary, i.e. above or below some threshold value. We denote this as 0_R and 1_R respectively [11].

4. Bob sends his measurement basis via classical public channel to Alice. Alice then compares her basis and Bob's basis. She takes note of the basis that matches, which she represents with 0_M if the basis does not match, and 1_M if the basis match. She sends this information to Bob via classical public channel.

5. Both Alice and Bob go through a process called sifting, i.e. removing the value bits (for Alice) and measurement result bits (for Bob) whose bases do not match (0_M).

6. Alice and Bob then compile the rest of bits whose bases match (1_M) by appending them next to each other. If the measurement is done properly, for matched bases, $0_V = 0_R$ and $1_V = 1_R$. Thus, in the end they would have obtained the same symmetric key.

2.3 Implementation

This section gives a brief summary of the implementation of this QKD scheme. In principle, one needs to perform these three steps:

1. Polarisation basis alignment. This step is to make sure that Alice's polarisation is aligned to Bob's polarisation.

2. Light generation and measurement. This step deals with the process of sending the polarised light and measuring them. This process occupies the quantum channel and takes the bulk of the time.

3. Secure key generation. This step comes after the transmission through the quantum channel, and is performed through the classical channel. This process is essential to ensure that both Alice and Bob obtain the same key.

2.3.1 Polarisation basis alignment

The first thing to do in any QKD system is to make sure that the polarisation axis are aligned with each other. In our experiment, we perform this in two steps:

[10] We assume that the measurement is done in the 0_V bases, i.e. H and D polarisation

[11] There is no typo: above the threshold means the polarisation is correct, which would be 0_V. If the polarisation is incorrect, it would be 1_V.

		Bob			
		H	D	V	A
Alice	H	1	0.5	0	0.5
	D	0.5	1	0.5	0
	V	0	0.5	1	0.5
	A	0.5	0	0.5	1

Table 2.2: Expected value of intensity of different polarisation settings between Alice and Bob, normalised to the maximum transmission.

1. Aligning the angle of the H polarisation axis between Alice and Bob. There are two ways to do this:

   - Alice fixes her H polarisation, and Bob finds and sets an angle offset which maximises the light transmission. This angle offset for maximum transmission makes Bob's H polarisation align with that of Alice.
   - Similar to the above, but now with Alice finding and setting her angle offset instead. This way, Alice's H polarisation is aligned to that of Bob.

2. After the alignment procedure, we construct the intensity matrix with various polarisation settings set by Alice and Bob. The theoretical values for the intensity matrix is given in Table 2.2. Due to imperfections of the devices (lasers, quarter wave plates, polarisers), the measured intensity matrix deviates from the expected value. We can define this deviation ("signal degradation") with the formula:

$$\eta = \frac{\sum_{ij} |d_{ij} - d_{ij}^{\exp}|}{\sum_{ij} d_{ij}} \tag{2.1}$$

   where d_{ij} is the measured intensity on i-th Alice's polarisation setting and j-th Bob's polarisation setting, and $d_{ij}^{\exp}$ is the expected intensity with the aforementioned settings, which is obtained from Table 2.2 and normalised to the measured values with the following relation:

$$\sum_{ij} d_{ij}^{\exp} = \sum_{ij} d_{ij} \tag{2.2}$$

   To interpret the value of η, we can consider the following cases:

   (a) When the measured intensity matrix follows the expected matrix very closely. In this case, $|d_{ij} - d_{ij}^{\exp}| \approx 0$ for all i and j. Thus, the signal degradation is very small: $\eta \approx 0$.

(b) When the measured intensity matrix is constant for all cases, i.e. the polariser does nothing to the light ('unpolarised'). In this case, $d_{ij} \approx 0.5$ for all i and j, and thus $\eta \approx 0.5$.

(c) When the measured intensity matrix 'anti-correlates' with the expected matrix, i.e. when there is an offset of 45° or 90° between Alice and Bob's H polarisation. In this case, $\eta \approx 1$.

Thus, η represents, to some extent, the degree of misalignment and the 'visibility' of the signal. In our experiment, ideally we would require η to be as small as possible, but based on our experience, it is very hard to get $\eta \leq 0.1$ [12], and the experiment would still work fine if $\eta \leq 0.2$.

2.3.2 Light generation and measurement

After the polarisation axes are aligned and the intensity matrix is characterised, we run a predefined sequence consisting of synchronisation pulse (see Section 2.1.3) and 16-bit sending and receiving cycle (see Section 2.1.2).

The light is generated with a 780 nm laser (see Section 2.1.1) which passes through a quarter wave plate to 'depolarise' the light, and then sent through the first polariser (controlled by Alice) to generate a polarised light in either H, D, V, or A polarisation. On the Bob side, he will project the light (from Alice) to either H or D basis with another polariser, and then measure the intensity of the light. This light generation is done at the $1.2 - 1.4$ s mark of each cycle, while the measurement is performed at the 1.3 s mark.

2.3.3 Secure key generation

From the value bits, basis choices, and the measurement results, we can perform a key sifting procedure according to Section 2.2.3. Assuming that the detection efficiency is 100%, which is probably always the case in our experiment, we do not need to perform any further error correction and privacy amplification.

The key resulted from the sifting process can then be used to encrypt secure materials. Chapter 3 will be dedicated to it, where we actually use the generated key to perform encryption (and decryption) of some high-security materials.

2.4 Programs

The programs necessary to perform Mission 2 are located in the folder "2_QuantumKey".

[12] This problem has something to do with our polarisers, particularly when the difference between the polarisation axes is 45°: the intensity is ~ 0.6 units instead of 0.5 units.

2.4.1 Arduino programs

Similar to the previous chapter, the Arduino program "ArduinoQuantum.ino" consists of several subroutines to perform tasks related to rotating the stepper motors, turning the laser on/off, and generating and running the random sequence, which can be accessed via serial communication. The program is compatible with both Alice and Bob. There are some set parameters to the program, which can be modified at wish (and if you really know what you are doing). The default values for some of the parameters are given as below: (def: #).

To distinguish between serial sent by computer to Arduino (TX) and the serial sent by Arduino to computer (RX), we use the flags (TX) and (RX) in the discussion below. List of serial (TX) commands:

- HELP
  Prints list of serial commands and their short descriptions (in case you forget). Alternatively, you can keep referring to this notes.

- SETANG X
  Rotates the stepper motor to a specified angle X in degrees. The value of the latest set angle will be stored in the Arduino EEPROM, so that one does not lose the value after a shutdown cycle or any accidental misconnection. This program assumes that the stepper motor has "stepsPerRevolution" (def: 2048) steps in each revolution, and that the stepper motor is connected through pin 8, 9, 10, and 11.
  The serial (RX) prints "OK" after the task is finished.

- ANG?
  Asks for the current angle. The serial (RX) prints the value in degrees.

- SETPOL X
  Rotates the stepper motor to a specified polarisation X: 0 (H), 1 (D), 2 (V), and 3 (A). It does not understand any values besides 0, 1, 2, and 3, and will assume it is 0 (H). The angle set with a specific polarisation is calculated with the following formula:

  $$\mathrm{ANG} = 45^\circ \times \mathrm{POL} + \mathrm{HOF} \tag{2.3}$$

  where ANG is the set angle in degrees, POL is the set polarisation, and HOF is angle offset of the H polarisation (see below).
  The serial (RX) prints "OK" after the task is finished.

- POL?
  Asks for the current polarisation. The serial (RX) prints the value in floating point number based on equation 2.3.

- SETHOF X
  Sets the angle offset of the H polarisation (HOF) in degrees (see equation 2.3). This value will also be stored in EEPROM.
  The serial (RX) prints "OK" after the task is finished.

- POLSEQ X
  Sets the polariation sequence according to X, a sequence of characters (consisting of 0, 1, 2, and 3) with length of "seqLength" (def: 16 bits). This function will not run the sequence.
  The serial (RX) prints "OK" after the task is finished.

- RNDSEQ X
  Generates the polarisation sequence (POLSEQ) with random numbers (0 to 3) with the Entropy library. This is due to the following relation:

$$\mathrm{POL} = 2 \times X_V + X_B \tag{2.4}$$

  where X_B is the basis bit of the sequence and X_v is the value bit of the sequence (see Section 2.2.2).
  The serial (RX) prints "OK" after the task is finished.

- RNDBAS X
  Similar to RNDSEQ, but with only random numbers 0 and 1 (X_B). This will set Bob's measurement basis to be on either H or D.
  The serial (RX) prints "OK" after the task is finished.

- SEQ?
  Asks for the set sequence. The serial (RX) prints the sequence in string with values of 0 to 3.

- LASON
  Turns on the laser connected to "pinLsr" (def: 4).
  The serial (RX) prints "OK" after the task is finished.

- LASOFF
  Turns off the laser connected to "pinLsr" (def: 4).
  The serial (RX) prints "OK" after the task is finished.

- VOLT?
  Asks for the voltage at "sensorLoc" (def: A0). The serial (RX) prints the result in floating point number (in units of V).

- CATCH
  Waits until the voltage at "sensorLoc" (def: A0) gets higher than "catchTh" (def: 200 units, which is around 1 V). This command blocks until the condition is fulfilled. The serial (RX) prints the time when it happens (in milliseconds).

- RUNSEQ
  Runs the sequence set by either POLSEQ, RNDSEQ, or RNDBAS. The sequence is defined in Section 2.1.2, but the 'laser' in this case is "pinDeb" (def: 13). This command is useful for debugging purpose. The serial (RX) prints "OK" after the task is finished.

- TXSEQ
  Similar to RUNSEQ, but is customised for Alice's sequence (TX). It switches on/off the laser at "pinLsr" (def: 4).
  The serial (RX) prints "OK" after the task is finished.

- TXSEQ
  Similar to RUNSEQ, but is customised for Alice's sequence (TX). It measures the voltage "sensorLoc" (def: A0) at the "seqReadTime" (def: 1300 ms) mark.
  The serial (RX) prints the measurement result in sequences of integers (0 - 1023 for 0 - 5 V), separated with spaces, after the task is finished.

2.4.2 Python programs

The Arduino program only serves as the backbone for the classical communication. To perform more complicated tasks, we have written several python programs that will aid in performing Mission 2. For alignment purposes, we have also developed a GUI program which is described in Section 2.4.3.

devloc_quantum.txt

This is a simple text file that contains the information of the Arduino device location (usually "/dev/ttyACM0" or "/dev/ttyUSB0"). The python programs will look into this file prior to running to determine the device location of Arduino, and hence the file needs to be updated should there be any changes in the device or its location.

To check the location of the device after plugging in the USB hub, you might want to run the "dmesg" program in the Linux terminal.

threshold.txt

This is a simple text file that contains the threshold value for Bob's polarisation measurement.

send_calibrate.py

This is the program to perform the polarisation calibration (Section 2.3.1) between Alice and Bob. This program is supposed to be run by Alice, and it uploads a sequence of polarisation and obtains intensity matrix (displayed on the Bob's side). From the intensity matrix, we can estimate the signal

degradation parameter η. Note that Alice needs to run this program after Bob starts the complementary program "recv_calibrate.py".

recv_calibrate.py

This is the complementary program of Alice's "send_calibrate.py". This program is supposed to be run by Bob, and it uploads a sequence of polarisation and measures the intensity matrix. Note that Bob needs to run this program first before Alice runs "send_calibrate.py".

send_key.py

This is the program to perform a key sharing procedure (Section 2.2.3) in our variant of BB84 QKD scheme. It will send 16-bit (unsifted) randomised keys in randomised basis to Bob. The key sifting procedure needs to be performed manually, or with the help of "keysift_hint.py". Note that Alice can only start this program after Bob runs his complementary program "recv_key.py".

recv_key.py

This is the complementary program of Alice's "send_key.py". It measures the 16-bit polarisation sent by Alice in randomised basis. Alice and Bob need to perform the key sifting procedure manually afterwards. Note that Bob needs to start his program first before Alice runs "send_key.py".

Remember to set the threshold value in the text file "threshold.txt", which determines whether a measurement is classified as 0_R or 1_R. A good value to set as threshold is the mean of the intensity matrix from "recv_calibrate.py".

keysift_hint.py

This is a helper program to perform the (manual) key sifting between Alice and Bob. It takes in Alice's unsifted key and basis choices, and compares it with Bob's measurement result and basis choice, to produce the final sifted key. It needs to be run by Alice and Bob (concurrently), and they need to use a separate public channel (i.e. a piece of paper that they pass back and forth) to communicate Bob's basis choice and the matched basis.

2.4.3 Python GUI program

The Python graphical user interface (GUI) program is a very useful tool for the alignment procedure. We use the PyQt library [13] to implement the two GUI (for sender and receiver) with functions as following.

[13] https://riverbankcomputing.com/software/pyqt/intro

Sender GUI

1. First, select the device & press start. To stop the device, press stop.
2. Toggle laser: to turn laser on/off.
3. Abs angle: to set absolute angle of stepper motor. Press "Set Angle" to perform the operation (polariser should rotate).
4. Offset: to set the absolute angle of stepper motor for H polarisation. Press "Set Offset" to perform the operation (polariser does not rotate).
5. Polarisation: set the polarisation as follows: 0 (H), 1 (D), 2 (V), and 3 (A). Press "Set Polarisation" to perform the operation. Polariser will rotate according to Equation 2.3.
6. Reset all: to reset absolute angle, offset, and polarisation to 0 (polariser should rotate to 0° absolute angle).

Receiver GUI

1. First, select the device & press start. To stop the device, press stop.
2. Start measure (stop measure): to start (stop) measuring the voltage readout on the photodetector.
3. Abs angle: to set absolute angle of stepper motor. Press "Set Angle" to perform the operation (polariser should rotate).
4. Offset: to set the absolute angle of stepper motor for H polarisation. Press "Set Offset" to perform the operation (polariser does not rotate).
5. Polarisation: set the polarisation as follows: 0 (H), 1 (D), 2 (V), and 3 (A). Press "Set Polarisation" to perform the operation. Polariser will rotate according to Equation 2.3.
6. Reset all: to reset absolute angle, offset, and polarisation to 0 (polariser should rotate to 0° absolute angle).
7. Scan res: to set the steps (in degrees) that you would like to scan. The scan ranges from (-180°+ offset) to (180°+ offset) with the offset you have set in step 4. When done, press "scan" and wait.

3

Putting It Altogether

In Chapter 2, we have explored the technicalities in building up classical communication channel with infrared (IR) sender and receiver. In Chapter 3, we have used laser light and polarisation to represent "quantum bits", which can be used to construct secure keys. This chapter will combine elements from both chapters to produce a working setup where we send "QKD-encrypted" messages from Alice to Bob.

3.1 Overview

The process to send a QKD-encrypted message is summarised as follows:

1. Preparation steps:
   (Classical channel) Alice and Bob ensure that the classical channel works properly, i.e. "chatting.py" works.
   (Quantum channel) Alice and Bob align their polarisation axes with respect to each other, perform the calibration procedure, and set the "threshold" parameter properly.

2. Alice and Bob construct a 32-bit secure key, which will be explained in depth in Section 3.1.1

3. Using the 32-bit secure key, Alice encrypts the message. The encryption procedure and key expansion are explained in Section 3.1.2.

4. Alice then sends the encrypted text via the public channel to Bob, i.e. by using "chatting.py".

5. After receiving the encrypted text, Bob decrypts the text with the procedures outlined in Section 3.1.2. If the messages and keys are received with high fidelity, Bob will obtain the correct message.

3.1.1 Secure key generation

The key generation follows the steps outlined in Section 2.3. However, we implement programs "send_32bitQKD.py" and "recv_32bitQKD.py" which basically automate the entire process. In each cycle/attempt, 16 polarisation keys and bases are generated by Alice and Bob, which after the sifting procedure produces on average 8 bit of keys. Thus, the program repeats the cycle until 32 bit of keys are obtained.

3.1.2 Message encryption and decryption

Ideally, for optimum security, the secure key length should be as long as the message [1]. However, the key generation rate is very low [2]. As a perspective, in our experiment, 32-bit key generation takes approximately 2 to 3 minutes, whereas a normal "Twitter" message has a character limit of 280 (2240 bits). It will take them approximately 3 hours to generate enough key length to encode 280 character messages in this ideal case.

In this experiment, we perform expansion on the key length as follows. We use the 32-bit key as a seed to a pseudo-random number generator (PRNG) [3], which will then generate a pseudo-random sequence of bits enough to encode the entire message. We call this sequence of bits the "expanded" key.

Encryption

To encrypt the message, we perform a bit-wise XOR operation [4] between the "expanded key" and the message string (ASCII) in binary representation. This produces a binary sequence, and for some technical reasons [5], we represent the encrypted text in sequence of hexadecimal (HEX) numbers.

Decryption

To decrypt the message, we perform a bit-wise XOR between the "expanded key" and the encrypted text (represented in binaries). This process is basically identical to the encryption process (in the binary representation). This is because performing an XOR operation twice on a message gives back the message.

[1] https://arxiv.org/pdf/0902.2839.pdf

[2] This is also the case in real world QKD applications. Thus, many propose to expand the key by using ciphers such as advanced encryption system (AES).

[3] We used the Mersenne Twister pseudo random number generator implemented in the random library in python: https://docs.python.org/3/library/random.html

[4] https://en.wikipedia.org/wiki/Exclusive_or

[5] If we represent the encrypted text in ASCII, it might print out a lot of non-characters, which is not very nice for the purpose of copy-pasting. Another alternative yet compact way is to use base64 (6 bit length) representation.

3.2 Programs

The programs necessary to perform Mission 3 are located in the folder "3_QKDComm".

3.2.1 Arduino programs

For the Arduino program, we upload the same Arduino programs as in Mission 1 ("ArduinoClassical.ino") and Mission 2 ("ArduinoQuantum.ino") to two different Arduinos. Please take note of the location of the devices.

3.2.2 Python programs

The Arduino program only serves as the backbone for the classical communication. To perform more complicated tasks, we have written several python programs that will aid in performing Mission 3. Please note that you might need to connect both Arduinos together (one for classical communications and another one for quantum communication).

devloc_classical.txt

This is a simple text file that contains the information of the Arduino device location (usually "/dev/ttyACM0" or "/dev/ttyUSB0") – for the classical communication Arduino device. The python programs will look into this file prior to running to determine the device location of Arduino, and hence the file needs to be updated should there be any changes in the device or its location.

To check the location of the device after plugging in the USB hub, you might want to run the "dmesg" program in the Linux terminal.

devloc_quantum.txt

Similar to "devloc_classical.txt", but for quantum communication device.

threshold.txt

This is a simple text file that contains the threshold value for Bob's polarisation measurement.

send_32bitQKD.py

This is the program to perform a 32-bit secure key generation (for Alice). This program performs and repeats procedures in "send_key.py" and "keysift_hint.py" until 32-bit key is obtained. Note that Alice can only start this program after Bob runs his complementary program "recv_key.py".

recv_32bitQKD.py

This is the program to perform a 32-bit secure key generation (for Bob). This program performs and repeats procedures in "recv_key.py" and "keysift_hint.py" until 32-bit key is obtained. Note that Bob needs to start his program first before Alice runs "send_key.py".

Remember to set the threshold value in the text file "threshold.txt", which determines whether a measurement is classified as 0_R or 1_R. A good value to set as threshold is the mean of the intensity matrix from "recv_calibrate.py".

encrypt.py

This program encrypts a string of message with 32-bit key, and outputs the encrypted text represented in HEX numbers.

decrypt.py

This program decrypts an encrypted text (represented in HEX) with 32-bit key, and outputs the original unencrypted message.