UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN

INFORMATION AND COMMUNICATION TECHNOLOGY - PHOTONICS

DEVELOPMENT OF A POINTING-ACQUISITION-TRACKING SYSTEM FOR FREE-SPACE QUANTUM COMMUNICATION

Supervisor: Prof. Giuseppe Vallone
Co-Supervisor: Dr. Francesco Vedovato

Author: Berra Federico

Academic Year 2019 – 2020

Date of graduation 07/09/2020

Contents

List of Figures

Chapter 1

Introduction

The aim of this thesis is to present an alignment system for quantum communication. To do this, the following chapters will be developed:

- **Quantum key distribution**: This chapter will explain a method of quantum communication called quantum key distribution;
- **Pointing Acquisition Tracking system**: This chapter will explain why quantum key distribution needs an alignment system and what its main components are;
- **QKD software design**:This chapter will explain the methods and technologies used to implement the software in this thesis;
- **PAT software design**: This chapter will explain how it was decided to create the software that controls the quantum key distribution alignment system using the technologies and methods discussed in the previous chapter;
- **PATLIC experiment**: This chapter will present a practical implementation of the alignment system;
- **Conclusions**: This chapter will summarize the points developed at the time and future extensions of the project.

Chapter 2

Quantum key distribution

Since the discovery of radio transmissions, the security of wireless communications has been continuously improved. Today, the advent of quantum communications is going to change the transmission technique itself, making the cryptographic keys an impregnable frontier for hackers.

This chapter starts with a brief introduction to quantum communication, then touches the key elements of quantum key distribution, and concludes with the need for satellite quantum communication.

Figure 2.1: Micius satellite used for quantum communication.

2.1 Introduction

The security of telecommunication systems is typically ensured by cryptographic protocols that are based on computational assumptions. For example, the complexity of factorizing large prime numbers is at the basis of the RSA public-key cryptosystem. However, this technique will be irremediably compromise with the arrival of quantum computers [1–3], which can provide an exponential speedup [4] in performing complex computational tasks, as factorizing prime numbers. Quantum Key Distribution (**QKD**) [5,6], once paired with the symmetric cryptosystem called One Time Pad (**OTP**), offers a solution to this issue, since the security of the key generation process is guaranteed by physical laws such as monogamy and no cloning theorem [7].

Since the first QKD protocol was introduced in 1984 [8], theoretical developments of security proofs and protocols have led to refined schemes that guarantee security even if devices are not completely safe or trust, as in the device independent framework [7–13].

From the experimental point of view, we can divide QKD into two main branches distinct by the medium in which the quantum communication (that is the exchange of photons) takes place: via fiber, via free-space. This last one can take place via both horizontal and vertical links (via satellite). At the moment free-space QKD is less developed with respect to its fiber-based counterpart, because it has to face additional issues, such as the use of telescopes, the presence of atmospheric turbulence and strong environmental noise. In this regards the Quantum Future research group of DEI department is deeply focused in free-space quantum communication technologies, in particular along satellite links. For instance, they reported on different feasibility studies realized in collaboration with the Italian Space Agency [14–16]. An important milestone in satellite QKD has been reached by China through the launch of the first satellite dedicated to quantum communication in 2016 called Micius [17], and many other research groups around the world are developing their own platform satellites [18–20].

The long-term aim of quantum communication is achieving one day the Quantum Internet [21] on a global scale, which require two conditions to be fulfilled. The first will be an integration between classical and quantum communication in the same physical channel (fiber and/or free-space), and the second one is the hybridization of the network, which must support both free-space and fiber-based channels. For example, it could be built small-scale centralized fiber distribution stations, interconnected with satellite links that presents long-scale characteristics.

2.2 Elements of Quantum Key Distribution

2.2.1 Qubit

Quantum communication is based on the qubit. A classical bit (**bit**) can stays in only two states and it is represented by 0 or 1, in contrast a quantum bit

(**qubit**) can stays in one of the states generated by a basis of the Hilbert space. For example, the two dimensional Hilbert space can be generated by the vectors

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.1}$$

and a generic state can be expressed as a super position of the two

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle , \tag{2.2}$$

where the complex amplitudes α, β must respect the normalization

$$\langle\psi|\psi\rangle = |\alpha|^2 + |\beta|^2 = 1. \tag{2.3}$$

The qubit seems to have infinite possibility of output but when measured can take only two values 0 or 1 as the bit. This because α, β represent the amplitude probability of outcome. For instance, a measure of the qubit $|\psi\rangle$ in the basis $\{|0\rangle , |1\rangle\}$ produce the 0 with probability

$$p(0) = | \langle\psi|0\rangle |^2 = |\alpha|^2 \tag{2.4}$$

and the 1 with probability

$$p(1) = | \langle\psi|1\rangle |^2 = |\beta|^2. \tag{2.5}$$

What is infinite is the possibility to choose the bases of measurement, in fact there are infinite possibility of choosing two orthonormal vectors in the two dimensional Hilbert space.

2.2.2 Photons

In the quantum optics theory, light is the sum of indivisible particles called **photons**. The photons are important for quantum communication because the qubit information can be carried into a single photon. The qubit can be encoded in the single photon by using various encodings or degrees of freedom. For quantum communications in fiber and free-space the two most used encodings are polarization and time-bin.

Polarization encoding takes advantage of the polarization of the single photon. Polarization is a property applying to transverse waves that specifies the geometrical orientation of the oscillations. The electromagnetic waves propagate according the Helmholtz equations and the simpler solutions for the Helmholtz equations are the plane waves, the higher solution can be expressed as a superposition of plane waves [23]. In general, the polarization of a plane wave refers to the orientation of the electric field vector that can be expressed as a super position of two vectors of the basis, in the specific case of a plane propagating in the $\hat{z}$ direction can be written as

$$\vec{E} = (a_x\hat{x} + a_y\hat{y}e^{i\varphi})e^{i\phi}e^{ikz}. \tag{2.6}$$

The polarization can be expressed by a Jones's vector

$$\hat{P} = \begin{bmatrix} a_x \\ a_y e^{i\varphi} \end{bmatrix} \tag{2.7}$$

that is a vector in the $\mathbb{C}^2$ Hilbert space.

Time bin encoding is obtained by sending a photon through a unbalanced Mach-Zehnder interferometer (MZI). In the MZI the photon can take one of the two paths, one path $|1\rangle$ is designed longer than the other one $|0\rangle$ by a length bigger than the coherence length of the photon. The final state can be expressed as a superposition of the two possibilities

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\theta}|1\rangle) \tag{2.8}$$

where $e^{i\theta}$ represent the additional phase accumulated in the longer path. In this way, by varying the phase it is possible to change the state [24].

Another important aspect of the photon is its indivisibility. This was demonstrated for the first time in the 1987 [25]. Indivisibility locks the information in a unique spot so the only way that one has to duplicate its information is to duplicate the entire photon. Here came the second aspect of the photon, because it is a quantum state it cannot be copied because doesn't exist a linear operator that do so. The demonstration is really immediate, consider: a generic input state $|\psi\rangle$, an ancilla for the copy $|0\rangle$, and a machine state $|\Phi\rangle$. They form a unique state

$$|\psi\rangle|0\rangle|\Phi\rangle =: |\psi 0\Phi\rangle, \tag{2.9}$$

where the copy-operator for the two vectors of the basis $\{|0\rangle, |1\rangle\}$ must follow:

$$|00\Phi\rangle \to |00\Phi\rangle, \tag{2.10}$$

$$|10\Phi\rangle \to |11\Phi\rangle. \tag{2.11}$$

This operator does not satisfy linearity:

$$|\psi 0\Phi\rangle = \alpha|00\Phi\rangle + \beta|10\Phi\rangle \to \alpha|00\Phi\rangle + \beta|11\Phi\rangle \neq |\psi\psi\Phi\rangle. \tag{2.12}$$

The single-photon source is difficult to realize and therefore it is preferred to use an *attenuated phase-randomized* laser pulse. A laser generates coherent light, whose quantum state is described by

$$|\alpha\rangle = e^{-|\alpha|^2/2} \sum_{n=0}^{+\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \tag{2.13}$$

that have an average number of photons of $\bar{n} = |\alpha|^2$. The average number $\bar{n}$ can be controlled by attenuation.

2.2.3 BB84

The first quantum distribution protocol **BB84** [11] was devised in 1984 by Charles H. Bennett and Gilles Brassard. It requires the use of a public and authenticated classical channel and a quantum channel. The strength of this protocol lies in the fact that non-orthogonal single-photon symbols are used for data transmission, so that any eavesdropper who starts listening to the channel is forced to consume the photon to read the state. Once consumed, the eavesdropper is not able to reproduce it perfectly and so introducing an error in the system. In other words, the security of the channel can be determined by the number of errors obtained in the communication itself. Because this protocol requires to send a random sequence of symbols, this cannot be used to share data but only for generate secret keys that will be used to encode the data. This process is called quantum key distribution. The safest way to communicate using keys is to divide the data in messages, for each message encrypt it with a random string (key) and then discard that key. In this way, every message became a random message that only who knows the key can decrypt. This procedure is called one time pad.

The functioning details of this protocol are very important to understand the others protocols because they are all variations around the same concept. In this protocol a transmitter called **Alice** and a receiver called **Bob** want to communicate using the following steps:

- Alice and Bob decide to use two non-orthogonal bases, for example

$$B_1 = \{|0\rangle, |1\rangle\} \tag{2.14}$$

$$B_2 = \{|D\rangle, |A\rangle\} \tag{2.15}$$

  with

$$|D\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.16}$$

$$|A\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle); \tag{2.17}$$

- Alice send a sequence of qubits randomly chosen from the four vectors of the bases;
- for each qubit, Bob choose at random a basis and measure it getting a sequence of 0 or 1;
- then Alice and Bob share in the authenticated classical channel the bases choice;
- they keep only the bits of the matching bases and discard the others because they are sure of the outcome. They get two partially correlated sequences of symbols called **sifted key**;
- at this point they do a classical post-processing to be sure no errors are introduced by the channel, so if everything is worked, they share a unique secret key.

The strength of this protocol appears when a third person Eve tries to read the data in the channel. Because the hole information of the qubit is encoded into a single photon that cannot be divided or copied, she must measure it. She can do only one measure per photon because a sequence of measurements does not commutate. When Eve measure the photon, she must guess a basis for the measure and send another photon to Bob with the corresponding guessed state. Doing so Eve introduce an error that Alice and Bob can estimate because it is additional to the typical one introduced by the channel. If the error introduced is too high, they must discard all bits and start again, else they have the *sifted key* that are partially correlated and partially secret. As before, they do a classical post-processing to remove the error and they get a unique key that is still partially secret because Eve has read some qubit correctly. To remove the partial information that Eve has in the secret key, Alice and Bob do an operation called **privacy amplification**. They want to remove the bits that Eve know but they didn't know which are so they discard some bits using a hash function. Now they have unique secret key.

This protocol needs some conditions to communicate:

- Eve cannot intrude into Alice's and Bob's devices because this protocol guarantees a safety on the channel, if Eve reads inside the devices it becomes a problem of safety of the devices;
- Alice and Bob must trust the random number generators, if not Eve can study the statistical distribution of the source and improve its predictions;
- the classical channel is authenticated otherwise Eve can pass itself off as Bob and the communication would no longer be between Alice and Bob but between Alice and Eve;
- devices are trusted and characterized, the further away the devices are from theoretical operation more information Eve manages to recover.

If the conditions have been met the only attacks Eve can do are:

- individual attack: Eve attacks each of the systems going from Alice to Bob independently of all others, using the same strategy. Eve must measure her ancillas before the classical postprocessing;
- collective attack: Eve can keep her ancillas in a quantum memory until the end of the classical postprocessing and perform collective measurements;
- coherent attack: no limits to fantasy. In many cases the bound is the same as for collective attacks.

The best attack Eve can do is *collective attack* where the maximum information rate she can get is

$$I_{AE} = \max_{Eve} \chi(A : E) \tag{2.18}$$

where $\chi(A : E)$ is the *Holevo bound* [5]. The secret information rate shared between Alice and Bob is the hole information rate between them minus the Eve's information rate where in the worst case is

$$r = I_{AB} - I_E. \tag{2.19}$$

The mutual information rate between Alice and Bob can be expressed as total rate minus the error entropy

$$I_{AB} = 1 - h_2(Q) \tag{2.20}$$

where $h_2(Q)$ is the binary entropy of the quantum bit error rate (QBER) [5]. Eve can gain information only at the expense of introducing an error, so the information gained is proportional of the error introduced

$$I_E = f h_2(Q). \tag{2.21}$$

The total rate of secret information is then

$$r = 1 - h_2(Q) - f h_2(Q) \tag{2.22}$$

and is called **secret key rate**. This last parameter is what allows us to make the estimates described in the procedure above.

Having a very low $\bar{n}$ is different from having a single photon because this allow Eve the possibility of *photon number splitting attacks* [5]. Decoy states protocols have been implemented as a useful method for improving the performance of quantum key distribution. A decoy state is a state with a fixed $\bar{n}$. A decoy state protocol takes advantage of the dependence between the errors generated by Eve and the average number of photons per state, to detect the presence of Eve. One example of two-decoy-state protocol (the vacuum $\bar{n} = 0$ and a weak decoy state $\bar{n} \leq 1$) is the BB84 shown in reference [26].

2.2.4 Fiber

The simpler way to communicate with QKD is via fiber because the technology of this sector is very developed, so the costs are not high, and the attenuation of the channel is essentially constant in time. The laser's pulses after being generated are sent into a symbol encoding system (time-bin, polarization) before being sent into the fiber, which is usually the telecom standard fiber.

There are three effects that characterize the fiber: losses, chromatic dispersion, polarization mode dispersion. **Fiber losses** are due to random scattering processes and depend therefore exponentially on the length $t = 10^{-\alpha l/10}$ where α is the attenuation coefficient and it is typically

$$\alpha(\lambda = 1550 \text{ nm}) \approx 0.2 \text{ dB/km}. \tag{2.23}$$

The attenuation coefficient depends from intrinsic or extrinsic properties. Intrinsic properties are electrons transition, molecule agitations, and Rayleigh scattering. Extrinsic properties are contamination of OH^-, H_2, bending, micro-bending,

and leakage due to modes coupling. **Chromatic dispersion** β_2 is a nonlinear effect that is generated by the dependence of the traveling velocity of the singular wavelength

$$\beta(w) \approx \beta_0 + \beta_1 w + \frac{1}{2}\beta_2 w^2 \tag{2.24}$$

where w is the angular frequency. This parameter depends on fiber profile, fiber material and guide mechanism. Polarization mode dispersion **PMD** generate different delays for the polarization's components. The entering pulse is decompose in the two principal states of polarization (PSP) in a *birefringent way*. This components has two different delays. This effect is caused by a symmetry break of the geometry of the fiber, so the two modes LP_{01}^x and LP_{01}^y are no longer degenerative.

In general, all three of these characteristics impose a maximum reachable distance-constraint regarding fiber communication especially for PMD and attenuation. All fiber-based implementations of quantum communication have to face PMD. This is clearly true for polarization encoding but it is equally a concern for the time-bin encoding because the interferometer depends on the polarization [6]. The attenuation is another big problem because unlike the classic communications where the intensity of the impulse sent can be increased, here the single-photon regime forbids it, and largely reduces the maximum communication distance.

When the pulse leaves the fiber, it will be sent to the decoding system which it will converted into a pair "bit and base" used in the communication. From here on the system will depend on the protocol used such as the BB84 protocol described above.

2.2.5 Satellite

In QKD both fiber communication and the free-space ground-to-ground communication have a much shorter maximum link-distance than satellite communication. The first one is due to attenuations, and the second one is due to curvature of the earth and fluctuations of the low atmosphere. The atmosphere behaves as a medium with a variable refractive index, which results in fluctuations of phase and intensity for the light beams that pass through it. Therefore, it would be preferable to use satellite communication for long range links [16].

The composition of the atmosphere offers an excellent transmission window around 770 nm which is very close to the 800 nm of the first telecommunication window used for fiber. This is important because a lot of devices are already developed for this window. In this band it does not present any birefringent behavior and therefore does not have PMD.

Unlike fiber, free-space has no spatial confinement and therefore has beam divergence problems. For reference, a gaussian-beam laser that illuminates a geostationary orbit from earth $r \approx 36000$ km with a beam waist of $w_0 \approx 10$ cm, produces a spot with diameter $d \approx r\theta \approx 88$ m, where

$$\theta = \frac{\lambda}{\pi w_0} \tag{2.25}$$

is the gaussian-beam divergence. Non-confinement makes the channel easily influenced by external noises such as sun and moon emissions. This problem can be mitigated by introducing spatial, temporal and spectral filters to try to confine the channel. New studies are trying to change the wavelength from 800 nm to 1550 nm to mitigate this problem. The 1550 nm wavelength is in the third telecommunication window used for fiber, so it is compatible with the standard fiber-technologies used in our days.

At the moment there are no high-performance free-space receivers at 1550 nm but only for single-mode fiber. Developing an efficient single-mode fiber injection system at 1550 nm would be advantageous because it would provide wavelength compatibility with fiber-based QKD technology which is available at telecom wavelength. Moreover, it would give the possibility of using very good single-mode fiber detectors (SNSPD) in the case of fixed stations, and of medium-good detectors with InGaAs technology [27] to make a portable receiver.

Another disadvantage is that the medium in which the light propagates depends on the atmospheric conditions of the moment. Beam propagation in atmosphere is subject principally to many effects given by temperature, wind, atmospheric conditions and many others. Small variations of these parameters result in local random changes of the density of the atmosphere and hence its refractive index. This is why the atmosphere is often named as a random medium. These small variations on the refractive index, have a similar effects like a series of small lenses positioned through the direction of propagations. This may cause effects of: fluctuations of the spot size; fluctuations of angle-of-arrival; wandering of the instantaneous beam center (**wandering**); deviations of the temporal pulse positions (**jittering**); and irradiance (intensity) fluctuations (**scintillation**). In addition, further signal degradation occurs with weather conditions not optimal to communication such as: fog, rain, clouds [6].

Chapter 3

Pointing Acquisition Tracking system

In physics and automation, feedback is the ability of a dynamic system to take into account the state of the system to compensate for the error by adapting the characteristics of the system itself. For example, a pointing-acquisition-tracking (PAT) system for quantum communications requires to use an optical feedback to keep the two terminals aligned one respect to the other during the photons exchange.

This chapter starts with a brief description of the logical units comprising a free-space QKD system and then it focus on the PAT system describing its modelling and some setups in which it will be used. It concludes with a list of components and an explanation of the fine and coarse versions.

Figure 3.1: Picture of Matera Observatory using a pointing-acquisition-tracking system during a quantum communication experiment.

3.1 Introduction

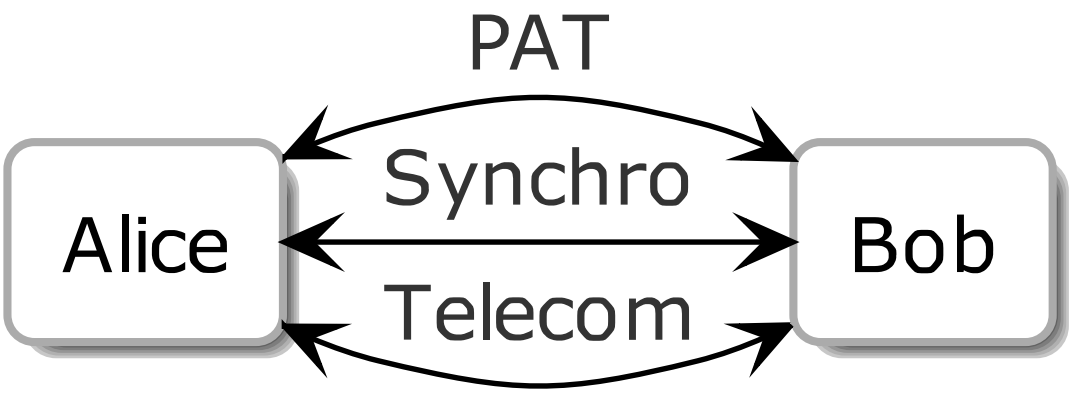


Figure 3.2: The requirements of QKD: PAT, Synchro, and Telecom.

Typical free-space systems used to implement the BB84 decoy-state share the logical structure sketched in Fig. 3.2. The three logic units necessary to implement the BB84 decoy-state protocol in a free-space channels are a Pointing-Acquisition-Tracking (**PAT**) system, a Synchronization (**Synchro**) system and a **Telecom** interface between Alice and Bob. In this scheme Fig. 3.2 Alice prepares and sends the states to Bob that measure them.

The PAT system is in charge of allowing Alice and Bob to send and receive the quantum signals over the free-space channel. It is usually comprised of a transmitting and a receiving telescope and a close-feedback pointing system to keep the two telescope aligned one with respect to the other.

The Synchro unit is in charge of the real-time timing of the protocol. It controls the lasers and tag the detections, allowing Alice and Bob to compare their raw bit strings a posteriori.

The Telecom unit is in charged of the (authenticated) classical communication between the two terminals. It allows the raw-key exchange to the other post-processing phases, the basis-reconciliation (or sifting), and error-correction and privacy-amplification [11].

In this thesis it will be explained how to implement the PAT system.

3.2 The PAT model

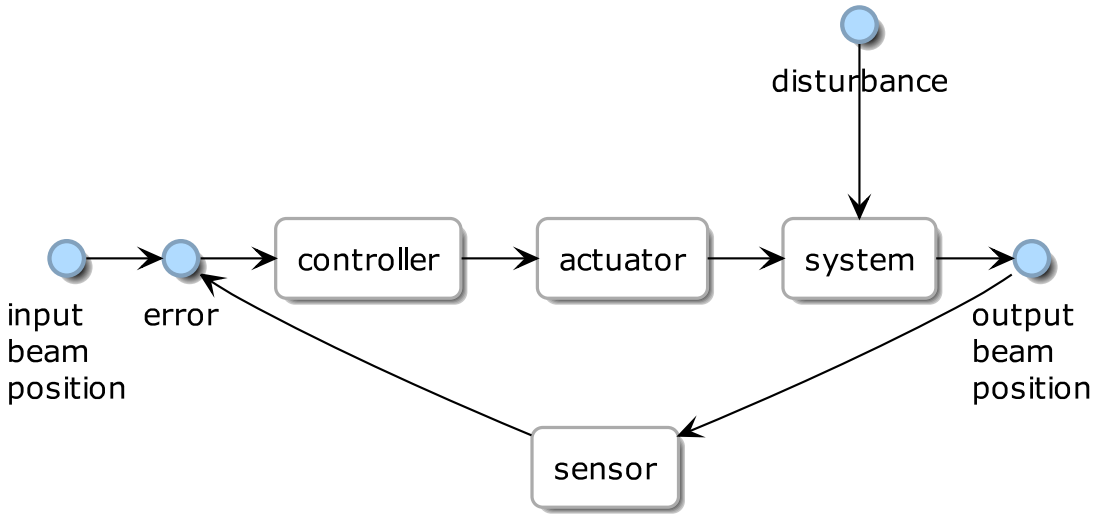


Figure 3.3: Pointing, Acquisition and Tracking system's model.

To establish a communication between two non-isotropic sources (transmitter, receiver) an alignment system is required. The one used in the free-space quantum communication is called *Pointing, Acquisition and Tracking* (**PAT**) system. PAT system operates on two levels: the first level is called *coarse* (**PATC**) while the second level is called *fine* (**PATF**). The PATC aims to align the telescopes, while the PATF aims to compensate the fluctuations of the transmitted signal. The signals typically used in this communication are laser signals. Both systems can be modeled with a feedback-control shown in the Fig. 3.3. Given a nominal position, the controller will estimate the error via a sensor. Once the estimate is completed, it will try to compensate the error by an actuator.

The PAT system can be divide in two parts: the software (**PATSW**), and the hardware (**PATHW**). PATHW can change a lot depending on what experiment you want to do, because different experiments require different components. PATSW must adapt to the PATHW but at the same time must provide an equal interface to users. This is because the goal is always the same: align the system at the desired point.

3.3 PAT implementations

The PATs implemented can be multiple and depend on the specific experiment that is carried out. For example, in this thesis it will be presented in the case of a free-space QKD system with single-mode fiber coupling called PATLIC that will be detailed in chapter 6. However, the same functions can also be used to try to pair single-mode fiber with light from a star, or for a generic quantum communication experiment in free-space on horizontal links.

3.4 Components

In this section will be listed all the components of the PATHW which a software implementation has been made in this thesis. In addition, these components will then be used in the PATLIC experiment.

3.4.1 Sensors

Camera

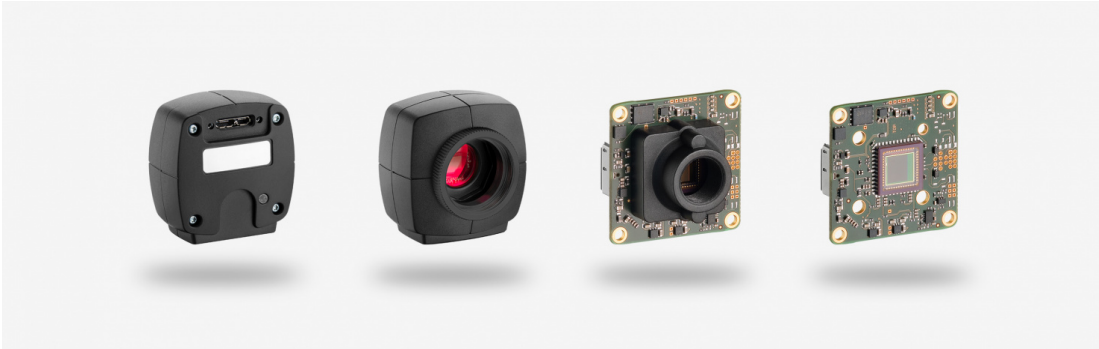


Figure 3.4: IDS camera.

A camera is a common sensor used in a PAT system. Depending on the model it can cover all the useful spectra for lasers: ultraviolet, visible, infrared. Typically, it has a frame rate that can range from 25Hz to 200Hz. The main advantage of the camera over the other sensors is its capability to reduce the frame rate to increase the exposure time to compensate for the typical attenuations of long distances. The camera returns an image typically of order of magnitude of 1000x1000 pixels. This image will then have to be processed by the computer to calculate the location of the laser signal. The technical details for the cameras used in the PAT setup reported in table 3.1.

Name	UI-3240LE
Producer	IDS
Sensor type	CMOS
Frame rate	30 fps
Resolution	1280 x 1024 pixels
Optical area	6.784 mm x 5.427 mm
Pixel size	5.30 µm

Table 3.1: Camera UI-3240LE technical details.

Position Sensing Detectors

Figure 3.5: ThorLabs PSD.

The Position Sensing Detectors (**PSD**) utilizes a silicon photodiode-based pincushion-tetra-lateral sensor to accurately measure the displacement of an incident beam relative to the calibrated center. These devices are ideal for measuring the movement of a beam, the distance traveled, or as feedback for alignment systems. The spectrum covered is from 320nm to 1100nm. This device needs a hardware driver to be used. Depending on the position of the laser in the sensor, it returns three voltages: Δx , Δy, SUM. Δx and Δy are signed voltages proportional to the light intensity captured in the two directions, where the sign signifies if the light intensity captured is before or after the middle point of the sensor. SUM is proportional to the whole intensity capture in the both directions. From these three components it is possible to obtain the position (x, y) of the laser's centroid relative to the center of the sensor (target point) using the following equations:

$$x = \frac{L_x \cdot \Delta x}{2 \cdot SUM}, y = \frac{L_y \cdot \Delta y}{2 \cdot SUM} \tag{3.1}$$

where L_x, L_y are the sensor sizes. The model used in the PAT is the Thorlabs PDP90A, which requires a driver called KPA101. The technical details for the PDP90A used in the PAT setup reported in table 3.2. The PSD is typically better than a camera at short distances, because the integrated controller in the KPA101 is based on a circuit and therefore has a continuous resolution with a bandwidth of 15 kHz, its signal doesn't require a post processing.

Name	PDP90A
Producer	Thorlabs
Wavelength range	320 to 1100 nm
Resolutiona	0.75 µm
Sensor size	9 mm x 9 mm
Bandwidth	15 kHz
Output voltage range	pin-1 (X): -4 V to 4 V pin-2 (Y): -4 V to 4 V pin-3 (SUM): 0 to 4 V

Table 3.2: Camera PDP90A technical details.

3.4.2 Drivers

KPA101

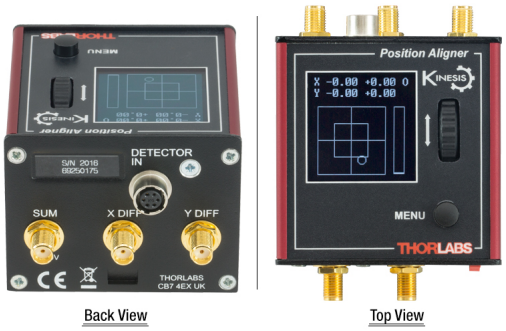


Figure 3.6: ThorLabs KPA101.

The **KPA101** is a device with three main functions: driver for the PSD, driver for the FSM, and natively implement a PID controller. The KPA101 can be controlled by the manual interface on the top of the unit or via a USB connection to a computer running a Kinesis software package or with the PATSW. Both interfaces allow the auto aligner to be operated in either an open-loop or closed-loop mode. In both modes, the KPA101 displays three signals on its screen: SUM that represents the total intensity of the signal read by the sensor, and x and y that are the laser's centroid coordinates obtained from equations 3.1. In the closed-loop mode, a processor inside the KPA101 runs two independent PID-feedback loops that outputs the two signals x and y as voltages on the back of the unit through the two pins X DIFF and Y DIFF. These signals will be used as the inputs to the beam steering elements being used to center the beam on the detector also called actuator.

3.4.3 Actuators

Fast Steering Mirror

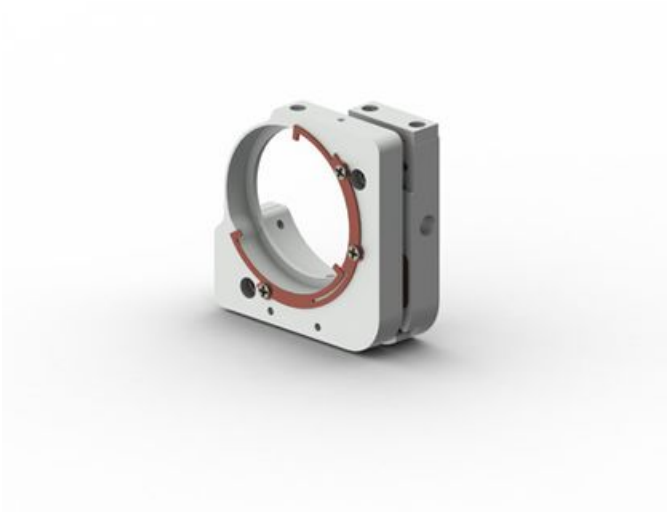


Figure 3.7: SmarAct FSM.

The fast-steering mirror (**FSM**) is a mirror housed on a mount able to do tip and tilt. The FSM used for the PAT is mounted in a STT-25.4 SmarAct frame that takes advantage of the technology stick-slip piezo stages. The piezo actuator combine macroscopic travel with nanometer resolution and high velocities of several millimeters per second. The technical details for the STT-25.4 used in the PAT setup reported in table 3.3. It requires an external driver to be managed because its movement is controlled by two voltages (x,y). The KPA101 is a driver compatible with the technology of this actuator.

Name	STT-25.4
Producer	SmarAct
Tip-tilt angles range	-2.5 to +2.5 degree
Angular velocity	15 degree/s

Table 3.3: Fast-steering mirror STT-25.4 technical details.

Skywatcher

Figure 3.8: Skywatcher telescope mout.

The Skywatcher used in the PAT is the AZ-GTI telescope mount. It is an altitude-azimuth mount controlled by two DC servo motor. It needs a computer to do the targeting. It has 9 speeds of motion called rates, so to perform a movement in one of the two axes a rate-time pair should be provided. The two motors are controlled separately by two encoders that translate the rate-time pair in an angle of motion.

3.4.4 Controllers

Ideally there are countless controllers available for the PAT system, but only two are used/developed in this thesis: One shot, and PID KPA101.

One shot

One shot is a controller developed for computers that reads the error in pixels between the input signal and the desired position, and generates a proportional correction signal for the actuator. Since the devices are not ideal this controller will have to make a calibration for each device, in order to obtain the best voltages to be given to the various actuators.

PID KPA101

The **PID KPA101** controller is Proportional Integral Derivatives (**PID**) and is integrated into the KPA101. A computer is required to specify PID parameters. The integrated PID system has a big limitation, it allows to align the beam only to the center of the system. This position is not always optimal in case of complex setups.

3.4.5 Additional hardware

A problem with the integrated PID KPA101 is that aligns the system only to the center of the sensor. This becomes a problem in very complex systems when the optimal alignment position does not match the center of the sensor. To solve this problem, a circuit was created that adds bias voltages to the signals that are exiting from the sensor and entering in the KPA101, with a range of ±4.096 V and a 16-bit accuracy. This resolution results in approximately 100 nm beam-displacement in the sensor.

Circuit schematic

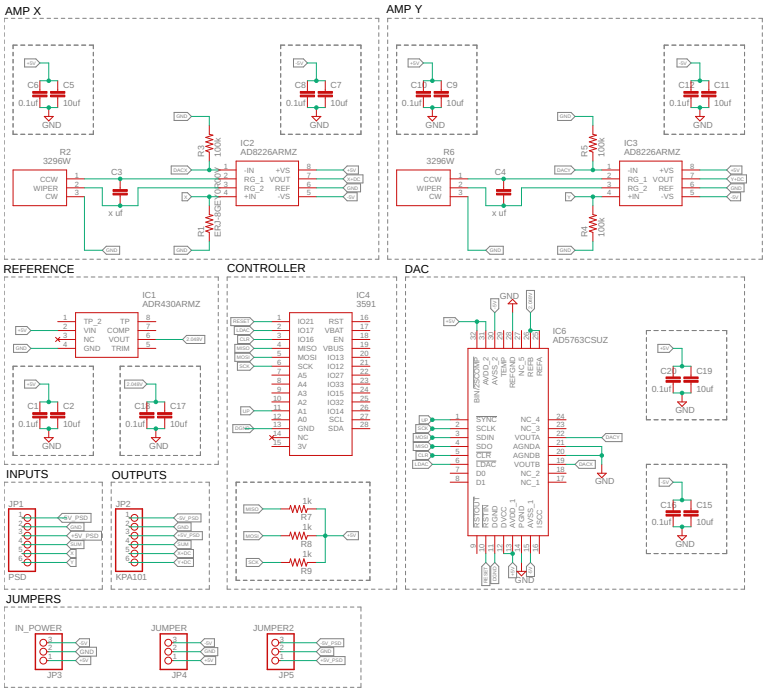


Figure 3.9: Eagle schematic view of the circuit.

The circuit in Fig. 3.9 was made in Eagle from scratch. It is composed of three main components: controller, digital analog converter (DAC), amplifiers (AMP).

The controller (ESP32) is mounted on the feather-board Huzzah32 and plugs into the circuit via a socket. It talks to the DAC (AD5763) through the serial-peripheral-interface (SPI) protocol using the most-significant-bit (MSB) convention with two-complement-data-coding. The output of the DAC that has a range of $\pm 2V_{ref}$ is combined with the two input signals X, Y through two instrumentation amplifiers. The reference voltage $V_{ref} = 2.048V$ is generated by a separate integrated (ADR430ARMZ). The amplifiers (AD8226ARMZ) has been provided with the possibility to add voltage regulators in case of amplification balance necessity. All integrated circuits have two capacitors $0.1\mu F, 10\mu F$ used as power supply filters. The MISO, MOSI, SCK lines have a resistance of $1k\Omega$ used as a pull-up. All amplifiers have on the -IN, IN lines $100k\Omega$ resistors as pull-down. Some jumpers have been added in order to choose whether to use the internal (KPA101) or external power supply for the integrated circuits.

Circuit board

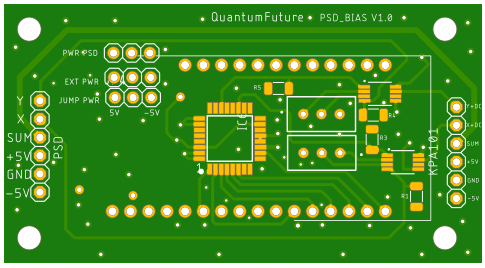


Figure 3.10: Circuit top view.

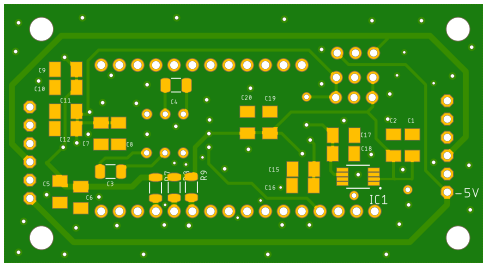


Figure 3.11: Circuit bottom view.

The track design was done using Sparkfun's design rules circuit (DRC). Particular attention has been paid to the isolation between digital and analog circuits, and efforts have been made to maintain serial communication lines of the same length. The final circuit was tinned in the laboratory and soon will be tested.

Software design

The software was created from scratch via PlatformIO. It is divided in two main parts: the first part is a communication interface between the controller and the DAC; the second part is the graphical interface that allows a user to set the bias value. Only the first part has been developed due to temporal restrictions.

Controller-DAC interface

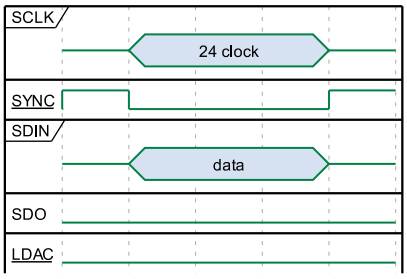


Figure 3.12: Simplify diagram of the sequence to write in a register.

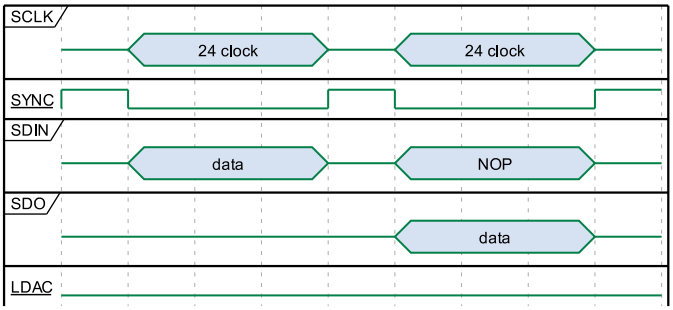


Figure 3.13: Simplify diagram of the sequence to read from a register.

An initial analysis led to an implementation of the interface between controller and DAC through three functions: initialization, writing from a register, and reading from a register. This interface has been implemented through the class AD5763, where the initialization is done via the constructor, and the read/write operation is done via the homonymous functions. The read and write procedures were carried out following the sequences described in the DAC documentation, their simplified schemes can be seen in Fig. 3.12 and Fig. 3.13. Some additional structures have been implemented to simplify the use of these three functions. For a quick view of its components, the List. 3.4.5 shows the class header.

```
#ifndef AD75763_H
#define AD75763_H

#include <Arduino.h>
#include <SPI.h>
#include "printf.h"

enum IO {
  R = 1 << 7,
  W = 0 << 7
};

enum Register {
  function  = 0 << 3,
  data      = 2 << 3,
  coarseGain = 3 << 3,
  fineGain  = 4 << 3,
  offset    = 5 << 3
};

enum DAC {
  A  = 0,
  B  = 1,
  AB = 4
```

```
};

struct PINSConfig {
  int syncNegate = 25;
  int sclk       = 5;
  int sdin       = 18;
  int sdo        = 19;
  int clrNegate  = 16;
  int ldacNegate = 17;
  int rstinNegate = 21;
};

class AD5763 {
  unsigned char message[3];
  PINSConfig pins;
  bool debugMode;
  SPISettings spiSettings;
  void printBin(unsigned char* message);

 public:
  AD5763(const PINSConfig& _pins, bool _debugMode = true);
  ~AD5763();
  void write(Register reg, int dac, const unsigned char* message);
  const unsigned char* read(Register reg, int dac);
};

#endif // AD75763_H
```

3.5 PAT-Coarse system

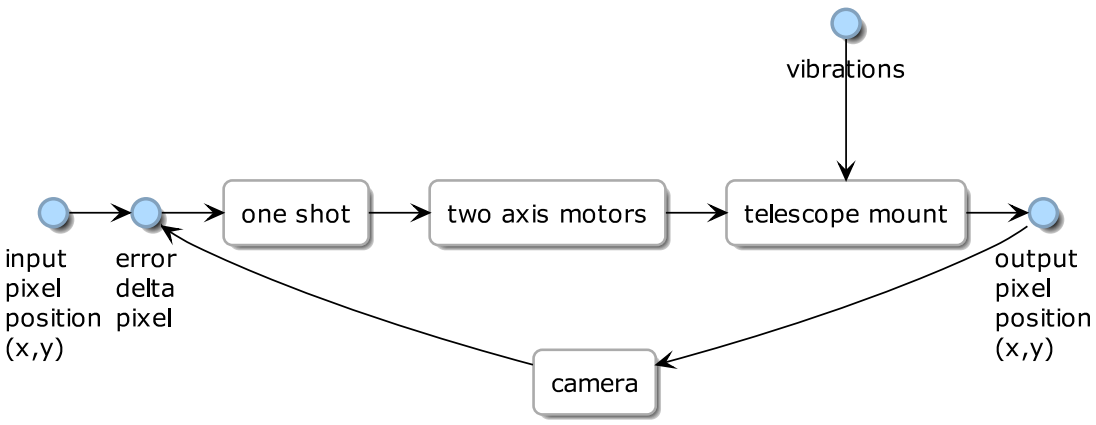


Figure 3.14: PATC model's implementation.

One way to point a telescope in a given region of space is to put in that region a laser pointing in the direction of the telescope, and in the telescope mount a camera to see that laser. If the camera is aligned with the telescope and the laser is centered in the camera, then the telescope will be aligned in the desired direction. To align two telescopes (Alice, Bob) will be needed to mount this system twice, so that Alice has a camera that points towards Bob's laser and vice versa.

In the PATLIC a camera is used as a sensor to express the system error in pixels. This error is handled by an ad-hoc developed controller called One shot. Once the controller has complete the error calculation, it will move the two-axis engine on the mount to align the telescope.

This system will maintain the aiming of both telescopes in both cases: for large movements such as the displacement of the two telescopes, and for smaller ones such as vibrations at low frequencies.

3.6 PAT-Fine system

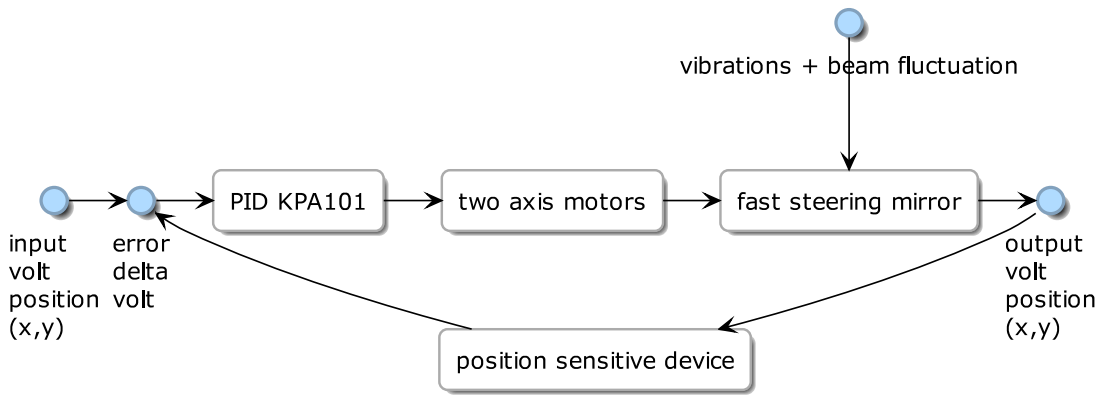


Figure 3.15: PATF model's implementation with position sensitive device.

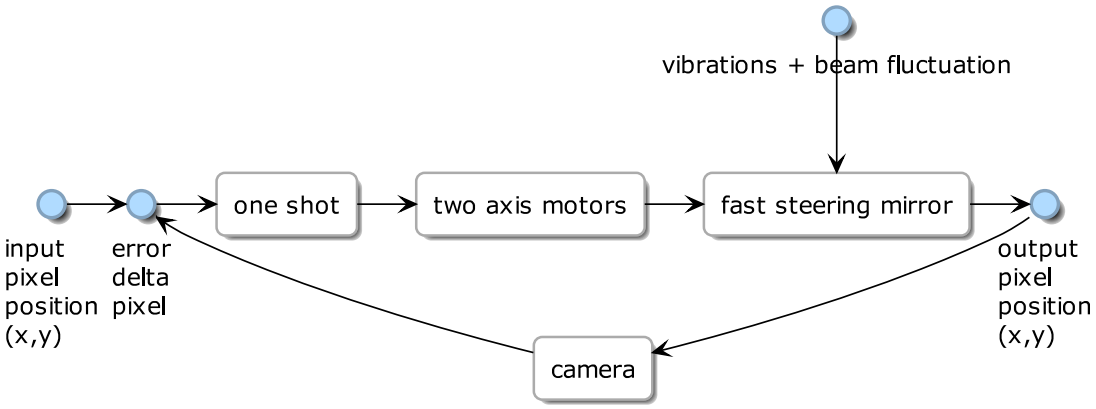


Figure 3.16: PATF model's implementation with camera.

Once the PATC system has aligned the telescopes then the PATF goes in action. To compensate for the fluctuations of the transmitted signal, a second beam is superimposed on the beam of the signal. If the wavelengths of the two beams are sufficiently close together, then both beams will be subject to the same fluctuations and thus compensate the second beam will also compensate the first one. This system is essential if you want to inject the free-space signal into a single-mode fiber. For example in the PATLIC experiment 6, a free-space 4mm-beam should be centered in a lens to be focus on single-mode fiber, but external fluctuations that reach a centimeter range prevent the beam from having a stable injection, PATF system can greatly mitigate these fluctuations. In this case the sensor can be a camera for long distances or a PSD for small distances.

In the PATLIC a PSD is used as a sensor and it exploits another control system PID KPA101. The sensor will read the position of the laser and send it to the controller that will correct it using a FSM.

This system will be able to compensate for the atmosphere fluctuations and high frequency vibrations.

Chapter 4

QKD software design

A software repository, or ”repo” for short, is a storage location for software packages. Often a table of contents is stored in the repo, as well as metadata. Repositories are typically one by one with packages. At client side, a package manager helps installing from and updating the repositories. Repositories are the modern way to develop new software.

One of the task assigned to me was the restructuring of the QKD software in order to develop and manage it in a smart way. Here it is described the solution chosen with an example.

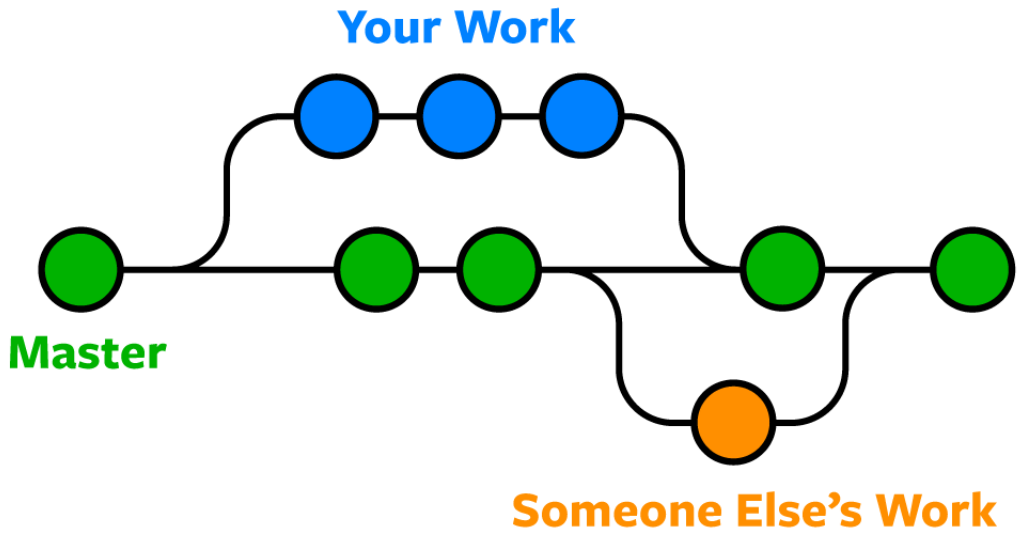


Figure 4.1: Typical scheme of a branch of commits inside a repository.

4.1 Memorization, standardization and expansion of the QKD software

All of the software already developed by the Quantum Future group to control the full QKD system was done temporarily on several computers without a particular organization, and it was therefore necessary to implement an easy system that would allow standardization, memorization, and simplification of code expansion.

To store the code, it was chosen to use GitLab, a web platform that allows the storage and versioning of the code via Git. Git is a free and open-source distributed version control system, it is typically used via shell commands. The code can then be stored in repositories that are organized in groups.

To create a standardization it has been chosen to create a **Boilerplate**, a ready-made code that serves as a template for the realization of various projects. The Boilerplate was built in c++ with the integration of Cmake and Qt. **CMake** is a system that simplifies the compilation of sources, while **Qt** is a set of libraries that simplify the graphical interface for the c++.

To simplify code expansion, it was decided to create a CMake-based packet manager and integrate it directly into the Boilerplate. CMake is an open-source, cross-platform family of tools designed to build and test c++ package software. This new system allows to list in a file "config.json" the dependencies of every project, that are the libraries necessary for its compilation. When the project is compiled, dependencies are automatically downloaded and compiled.

4.2 Program logical division

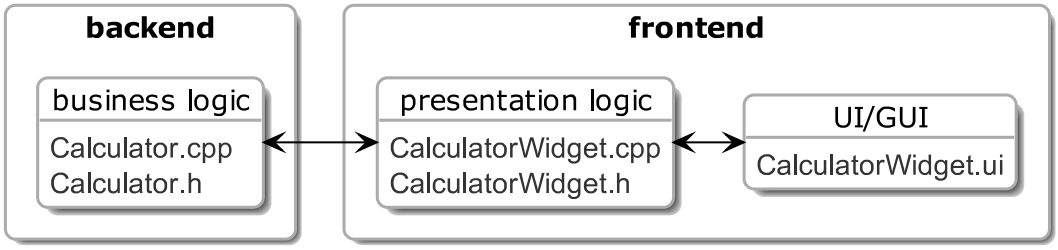


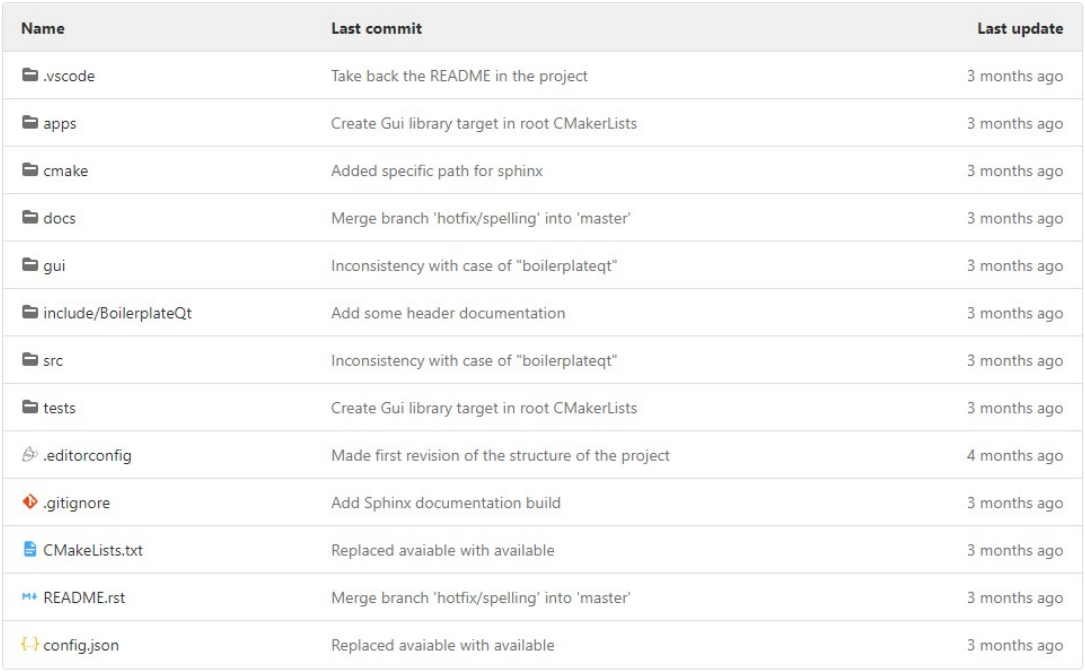
Figure 4.2: Typical structure of a calculator done with Qt.

As shown in Fig. 4.2 simple program like a calculator can be divided in two main logical blocks: backend and frontend. In software design the front end (or frontend) is the part of a software system that manages interaction with the user or with external systems that produce input data (e.g. user interface with a form), the back end (or backend) also called business logic is the part that processes the data generated by the front end. The frontend can be divide in two parts: one that describe the structure, the buttons, the texts of the graphic user interface (GUI); and one that connect each button with the respective business

logic function (presentation logic). Sometimes, the presentation logic is used as synonymous of frontend as was done in this thesis.

For example, a calculator application done with Qt libraries is divided as shown in Fig. 4.2. The business logic doesn't differ from a typical implementation except that the classes used must extend the class QObject. The presentation logic is typically renamed as widget and must extends the class QWidget. The UI is done inside the Qt editor and its information is saved in an xml that has as extension ".ui". In addition, Qt offers a many-to-many callback connection system of the various components called signal-slots.

4.3 Boilerplate structure



Name	Last commit	Last update
.vscode	Take back the README in the project	3 months ago
apps	Create Gui library target in root CMakerLists	3 months ago
cmake	Added specific path for sphinx	3 months ago
docs	Merge branch 'hotfix/spelling' into 'master'	3 months ago
gui	Inconsistency with case of "boilerplateqt"	3 months ago
include/BoilerplateQt	Add some header documentation	3 months ago
src	Inconsistency with case of "boilerplateqt"	3 months ago
tests	Create Gui library target in root CMakerLists	3 months ago
.editorconfig	Made first revision of the structure of the project	4 months ago
.gitignore	Add Sphinx documentation build	3 months ago
CMakeLists.txt	Replaced avaiable with available	3 months ago
README.rst	Merge branch 'hotfix/spelling' into 'master'	3 months ago
config.json	Replaced avaiable with available	3 months ago

Figure 4.3: BoilerplateQt repository's structure.

The Boilerplate contains two types of documentation: README and Sphinx. **README** is a reStructuredText (.rst) file in which a template allows you to quickly describe how to use the library. The documentation in **Sphinx** is more complex, it uses a python program that collects all the inline comments in the code and provides the class structure visible through web pages.

Typically when an application is developed, the developers decide on an interface (.h) that will have to be visible to other developers and all the rest of the code is kept hidden. This is to simplify the work of other developers who have already filtered the content they are interested in. To implement this division it was chosen to create an "include" folder that contains the interface and an "src" folder where all the rest of the code will be put. Moreover, the code is not only

composed by features but also by ready-to-use examples, for these examples it was decided to create the folder ”apps”. Since not all projects have a GUI, it was chosen to create a separate folder for them ”gui” which in case of non-use is easily erasable. In general the gui folder contains all frontend files, while the ”src” and ”include” folders contain all the backend files. For each class made in both backend and frontend, it was chosen to realize unit tests that were put in a separate folder ”tests”.

All other files and folder are typically left to the exclusive use of the packet manager. Also during the dependency assembly pipeline, the packet manager compiles individual folders into objects that will then put together. In particular, it will always compile the folders ”src”, ”include” and eventually ”gui”, but the folders ”apps” and ”tests” will be compiled only in case the repository is the parent and not a dependency. This pipeline is controlled by the ”CMakeList.txt” files placed in each folder. Additionally, the ”Cmakelist.txt” in the parent folder will extract all dependency information contained in the file ”config.json” and then download and compile all dependencies.

4.4 Boilerplate documentation

Boilerplate documentation is stored in a separate repository called *BoilerplateQt.wiki*. It is made in markdown (.md). This section will list its main sections.

4.4.1 Setup your environment

Prerequisites

These libraries need to be installed manually in your system:

- Qt 5.14.2
- Doxygen 1.8.13
- python3

This can be installed through **pip3** using a single command line:

- Sphinx 3.0.3
- Sphinx read the doc theme 0.4.3 (to use the read the doc theme for html documentation)
- Breathe 4.16.0 (to use the xml output of doxygen)
- Sphinx-markdown-builder 0.5.4 (to generate the markdown version for gitlab wiki)

```
pip3 install Sphinx sphinx_rtd_theme breathe sphinx-markdown-builder
```

On **Windows** add the path of the `../Anaconda3/Scripts` folder to the system variable **PATH**.

Setup procedure

- Create a new project on gitlab {YourProject} (the name of the project must be upper camel case).
- Download {YourProject} on the computer.
- Download this repository
  `git clone https://gitlab.dei.unipd.it/Calderaro/BoilerplateQT.git`.
- Copy everything except the folder **.git** from this repository to {YourProject} folder.
- (optional for QtCreator) Configure the project following the wizard procedure.
- Replace every string **BoilerplateQt** with {YourProject} in every file (Recommended to use QtCreator advanced replace).
- Replace every string **boilerplateQt** with {yourProject} in every file (Recommended to use QtCreator advanced replace).
- Replace every string **boilerplateqt** with {yourproject} in every file (Recommended to use QtCreator advanced replace).
- Replace every string **BOILERPLATEQT** with {YOURPROJECT} in every file (Recommended to use QtCreator advanced replace).
- Rename every file that contains **boilerplateqt** with {yourproject}:
  - '/gui/boilerplateqtwidget.cpp' to '/gui/yourprojectwidget.cpp'.
  - '/gui/boilerplateqtwidget.ui' to '/gui/yourprojectwidget.ui'.
  - '/src/boilerplateqt.cpp' to '/src/yourproject.cpp'.
  - '/tests/boilerplateqttest.cpp' to '/tests/yourprojecttest.cpp'.
  - '/include/BoilerplateQt/boilerplateqt.h' to '/include/BoilerplateQt/yourproject.h'.

- Rename the folder `/include/BoilerplateQt` with `/include/{YourProject}`.
- Open `config.json` and substitute with your data:
- **name**: name of the project YourProject.
- **version**: last tag in YourProject master history.
- **description**: short description of YourProject.
- **GUI**: whether YourProject have a GUI. **If you do not have a gui remove** '/gui' **and** '/include/YourProject/gui' **folders**.
- **modules**: external libraries to be fetched from an online repositoty.
- (Optional for QtCreator):
  - Close the project on QtCreator.
  - Remove the file 'CMakeLists.txt.user' and build directory (if any).
  - Configure the project as described in the Installation.

If you correctly executed the above steps the project should build with no problems. From this point on you can start:

- adding new source files and/or change the existing ones.
- adding and removing libraries dependeces.

4.4.2 How to add source files .cpp or .h

Adding new classes to the src folder

In the `src/CMakeLists.txt` add the source and header (only the headers not exported in the library) files in the **add_library** function. If you want the class to be exported in the library, so that an external program can use it, put the header in the `include/ProjName` folder.

Adding new classes to gui folder (should not be necessary but why not)

In the `gui/CMakeLists.txt` add the source and header (only the headers not exported in the library) files in the **add_library** function. If you want the class to be exported in the library, so that an external program can use it, put the header in the `include/ProjName/gui` folder.

Adding new executables to the apps folder

In the `apps/CMakeLists.txt` add the **add_executable** function to create a new target executable:

```
add_executable(
"${PROJECT_NAME}NewApp"
newapp.cpp
)
```

Also add the libraries that should be linked to the "NewApp" with target *link* libraries:

```
target_link_libraries(
  "${PROJECT_NAME}NewApp"
  PRIVATE Qt5::Gui Qt5::Widgets "${PROJECT_NAME}Gui" ${PROJECT_NAME}
      ${SUBMODULES_NAME}
)
```

4.4.3 Include other projects inside https://gitlab.dei.unipd.it/

Configuring your root CMakeLists.txt

This project builds two libraries called BoilerplateQt and BoilerplateQtGui. To import other libraries in your personal project you can do the following. Assuming your project structure is based on BoilerplateQt, you can add to the `config.json` the following lines:

```
"modules": [
  {
```

```
    "name": "SimpleLabel",
    "path": "gitlab.dei.unipd.it/Calderaro/SimpleLabel.git",
    "tag": "1.0.0",
    "avaiable": "YES",
    "getGui": "YES"
  }
]
```

Here the additional libraries SimpleLabel is added to BoilerplateQt. SimpleLabel contains a SimpleLabelWidget that is just a label that will be used as example in this tutorial.
Keep in mind that this will force your submodules to use this version (the one described in the ”tag”) of BoilerplateQt. Make sure that your submodule works fine with the version you set.
This will add two library targets: BoilerplateQt and BoilerplateQtGui. Other options:

- If you do not want to import the BoilerplateQtGui library set `"getGui": "NO"`
- You can just force your submodules to use a particular version of the BoilerplateQt libraries by setting `"available": "NO"`. Your project will not import the BoilerplateQt and BoilerplateQtGui targets.

Add the BoilerplateQtWidget to your GUI Widget

Show the BoilerplateQtWidget inside your GUI Widget following these steps:

- create an empty Widget in your GUI Widget and name it as “boilerplateQtWidget”.
- right-click on the boilerplateQtWidget and select “promote to”.
- Use “QWidget” as base class.
- In “Promoted class name” insert “BoilerplateQtWidget”.
- In “Header file” insert <BoilerplateQt/gui/boilerplateqtwidget.h>.

Connect the BoilerplateQt business logic to the BoilerplateQt Widget following these steps:

- add two private pointers of type BoilerplateQt and SimpleLabel to your GUI Widget class.

```
BoilerplateQt *boilerplateQt;
SimpleLabel *simpleLabel;
```

- assign two objects to the pointers in a method of your GUI Widget class.

- call the setBoilerplateQt method of BoilerplateQtWidget, passing the two pointers as argument:

```
ui->boilerplateQtWidget->setBoilerplateQt(boilerplateQt, simpleLabel).
```

4.4.4 Include third part libs .dll or .h

If you need to add some external libs to your project this is the right section. Supposing you want add an external library called "uEye" these are the steps to add it (remember to substitute "uEye" with your lib name):

1. Make the folders:
   - **libs/uEye/libs**
   - **libs/uEye/include**
2. Make the file **cmake/FinduEye.cmake** containing:

```
# Find the uEye libs
# uEye_FOUND - uEye SDK has been found on this system
# uEye_INCLUDE_DIR - where to find the header files
# uEye_LIBRARY_DIR - where to find the library files

# positions where to search the libs
SET(
  uEye_PATH_HINTS
  "${PROJECT_SOURCE_DIR}/libs/uEye/include"
  "${PROJECT_SOURCE_DIR}/libs/uEye/libs"
)

# set uEye_INCLUDE_DIR
find_path(
  uEye_INCLUDE_DIR
  uEye.h
  HINTS
  ${uEye_PATH_HINTS}
)

# set uEye_LIBRARY_DIR
find_library(
  uEye_LIBRARY
  NAMES
  uEye_api_64
  HINTS
  ${uEye_PATH_HINTS}
)
```

```
# if all listed variables are TRUE then set uEye_FOUND = TRUE
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(
  uEye
  DEFAULT_MSG
  uEye_LIBRARY
  uEye_INCLUDE_DIR
)

# now we can use this in the others cmake files
mark_as_advanced(uEye_INCLUDE_DIR uEye_LIBRARY)
set(uEye_LIBRARIES ${uEye_LIBRARY} )
set(uEye_INCLUDE_DIRS ${uEye_INCLUDE_DIR} )

# outputs
message("Adding lib: uEye")
message("-- uEye_FOUND: ${uEye_FOUND}")
message("-- uEye_LIBRARIES: ${uEye_LIBRARIES}")
message("-- uEye_INCLUDE_DIRS: ${uEye_INCLUDE_DIRS}")
```

3. In the file **src/CMakeLists.txt** add the following lines:

```
# Make cmake aware about the presence of Qt source files
# set(CMAKE_AUTOUIC ON)
# set(CMAKE_AUTOMOC ON)
# set(CMAKE_AUTORCC ON)

# file(GLOB HEADER_LIST CONFIGURE_DEPENDS
    "${PROJECT_SOURCE_DIR}/include/${PROJECT_NAME}/*.h")

# Automatically check if Qt5 is installed in the system
# Automatically find Qt5 components. REQUIRED flag: if components
    not found no build
# find_package(
#     Qt5 COMPONENTS
#     Core REQUIRED
# )

find_package(uEye REQUIRED)

# Create a target to build a library for the core sources.
# Specify the sources from which the library will be built
# Make an automatic library - will be static or dynamic based on
    user setting
# add_library(
#     ${PROJECT_NAME} STATIC
```

```
#     Camera.cpp
#     CameraFrame.cpp
#     Cameras/IDS.cpp
#     Cameras/Kiralux.cpp
#     ${HEADER_LIST}
# )

# We need this directory, and users of our library will need it
    too
# target_include_directories(${PROJECT_NAME} PUBLIC "../include")
target_include_directories(${PROJECT_NAME} PRIVATE
    ${uEye_INCLUDE_DIR})

# Specify the libraries needed to build BoilerPlateQt.
# target_link_libraries(${PROJECT_NAME} PRIVATE Qt5::Core
    ${SUBMODULES_NAME})
target_link_libraries(${PROJECT_NAME} PRIVATE ${uEye_LIBRARIES})
```

4. Now because we include the libs using "PRIVATE" you must include the libs using the relative path `#include "../../libs/uEye/include/uEye.h"`

4.4.5 Structure description of the project

CMakeLists.txt is the configure root file used by cmake to set the building process of the project.

- **include** folder contains all the headers that must be shared to external project (the headers, that declare the classes and variables that do **not** need to be accessed through the library, must be placed in src or gui folders).
- **src** folder contains all the sources for the business logic of the project.
- **gui** folder contains all the sources for the definition of the GUI and the interaction between its widgets and the business logic.
- **test** folder contains all the sources for testing the core and gui sources.
- **apps** folder contains all the sources for executing the applications using the src and gui libraries.

4.4.6 Setup git credential in windows

In order to download automatically the repository you need to store your Git credential in your computer. Here is listed the sequence of commands you need to write in your terminal to store the credentials.
(*) this are Enters

```
git credential-wincred fill(*)
protocol=https(*)
host=gitlab.dei.unipd.it(*)
```

```
username=federico(*)
password=secret(*)(*)
```

Chapter 5

PAT software design

User Interface Design (UI-design) in Information Technology refers to the design of the user interface for software, websites or applications. It's about programming the look of things, with a view to facilitating usability and to improve the user experience. Developing a user interface for complex systems simplifies the scientist's work by giving him more time to focus on the experiment. That's why before starting a project in addition to doing the relational study between the various components is also important to do a graphic study of what a user needs.

In this chapter, two analyses were carried out in order to build the PATSW part of this project. An analysis that focuses on the components available, and an analysis that focuses on what a user expects to use. Having these two analyses the construction of the single repositories becomes immediate, since it reduces to connecting the functionalities expected by users to the various components available.

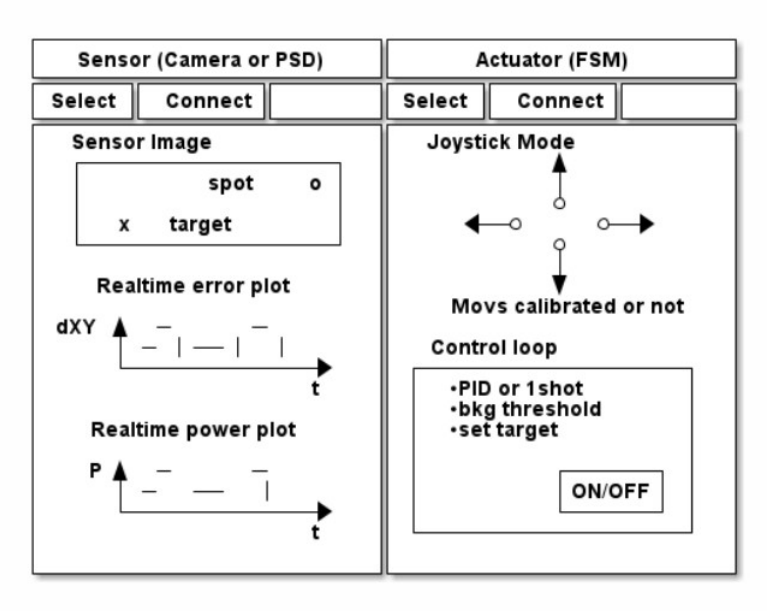


Figure 5.1: Initial draft of the PAT system interface.

5.1 The components analysis

The aim of this project was to develop modular software that could control a PAT system. The PATSW had to be able to align the telescope with different light sources such as the light of a star or the laser of another telescope. In addition, the PATSW had to be able to control all devices in the Sec. 3.4. Starting from the PAT model in Fig. 3.3 a simplified model has been developed as shown in Fig. 5.2 to highlight the various components to be implemented. In this model we see that there are three main families to be implemented: sensors, controllers, and actuators. The arrows in this diagram indicate the information passages, in other words, actuator and sensor will have to communicate with the controller and vice versa. It is therefore necessary to establish a common interface between the various sensors and actuators to connect them with the controllers. The study of these interfaces required multiple iterations to ensure maximum flexibility.

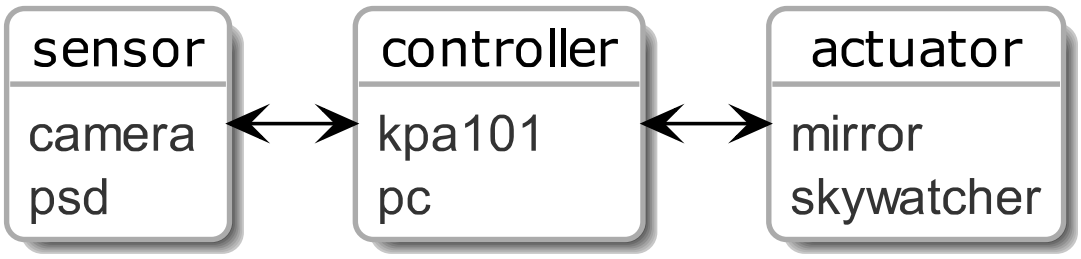


Figure 5.2: Simplified model for PAT system.

5.2 The GUI analysis

To get a complete idea of the goal of the system an other analysis was carry out. This analysis aims to establish the possible setups and what are the features that a user expects from the setup's GUI. In this section will be reported the final interface that an user expects to use and its functionalities. This final interface it was created joining togheter the sub-modules interfaces.

5.2.1 Possible setups

In this system there are three basic setups:

- mirror-camera-pc;
- skywatcher-camera-pc;
- mirror-PSD-KPA101.

Mirror-camera-pc

In this setup the pc is connected with two devices: the first is the KPA101 used to control the mirror; the second is the camera used as sensor. Here two types

of controllers are available: manual, calibrated. Manual give the possibility to move the mirror using a joystick taking in input times and voltages. Calibrated has two functionalities: the first is to use a joystick similar to manual that works taking in input a distance in pixels; the second is to use and automatic controller that align the centroid to the target that is called One shot.

Skywatcher-camera-pc

In this setup the pc is connected with two devices: the first is the skywatcher mout; the second is the camera used as sensor. The controllers available are the same of the mirror-camera-pc setup.

Mirror-PSD-KPA101

In this setup the pc is connected only to the KPA101, the KPA101 is then connected to the mirror and to the PSD. Here three types of controllers are available: manual, calibrated, PIDKPA101. Manual and calibrated are the same as the previous point. PIDKPA101 use the integrated PID-controller available in the KPA101.

5.2.2 Possible GUIs

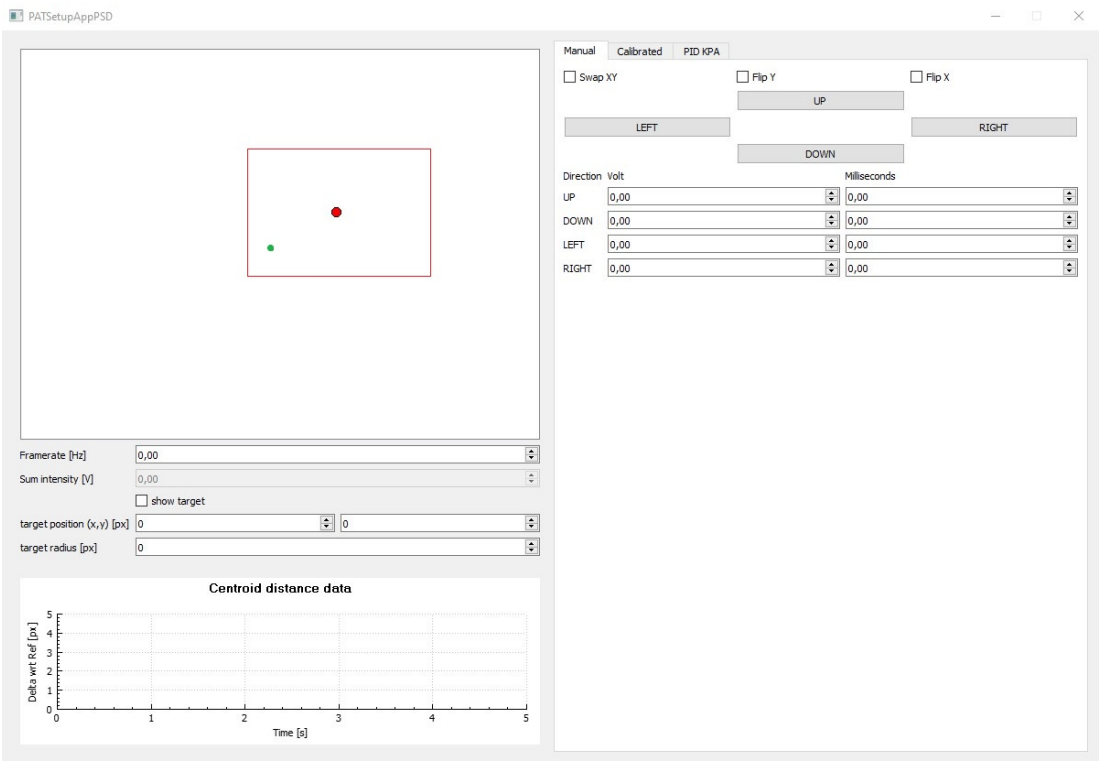


Figure 5.3: Main GUI of the program.

The GUI can be divided in two parts: the left side show the sensor information, the right side show the controllers information.

The sensor's side is composed by (form the top):

- the frame rendering that contains three objects:
  - the red dot represents the centroid of the beam;
  - the red square represents the area where the controller calculates the centroid;
  - the green circle represents the target;
- the set of parameters of the camera/PSD;
- the set of parameters to position the target;
- the plotter that show in real time the distances in pixels between the target and the centroid.

In the controller's side there are three tabs: manual controller, calibrated controller and PIDKPA101 controller (only for PSD). The Manual controller and the PID KPA controller are simpler than calibrated. In the manual controller rate (voltages) and times are the two parameters to be configured to optimize the joystick that will be use for the movement. In the PIDKPA101 after the configuration of the PID parameters a checkbox *enable PID* is avaliable to start the controller. The *auto open closed loop* checkbox is used to disable the controller when the intensity of the beam go outside the threshold range. The calibrated controller is more complex so a separate section is reserved for it.

Calibrated controller

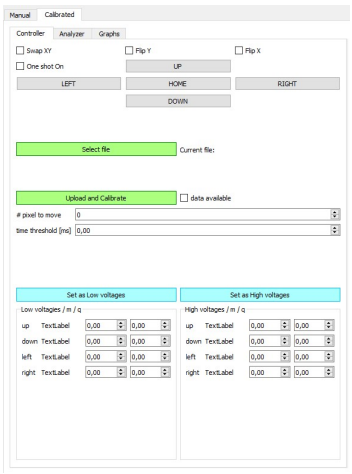


Figure 5.4: Controller interface.

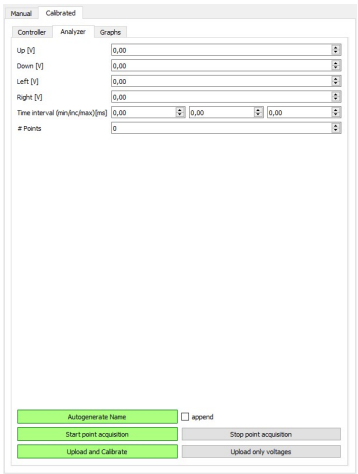


Figure 5.5: Analyzer interface.

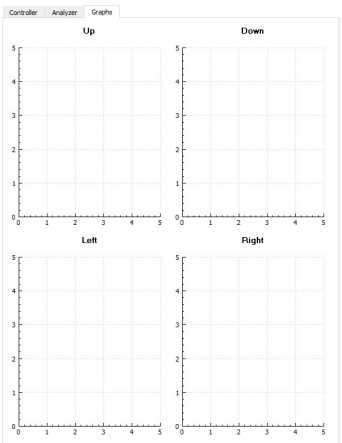


Figure 5.6: Graph interface.

The aim of this controller is to be able to use the *One shot controller*, but before to be able to use it a calibration is required.

Sensor calibration is done mainly for two reasons: the first is to establish the linear relationship between pixel error and actuator motion, and this applies to both PATF and PATC; and the second is to find the best voltage at which the FSM will work, and this applies only to PATF.

The calibration procedure can be decomposed in two parts: data acquisition, and data analysis. The points acquisition consists in moving the actuator in the four directions (up, down, left, right) giving in input time and rate and getting in output a movement in pixels. In this regards, if a FSM is used the rate is a voltage, instead if a Skywatcher is used the rate is a internal value of the instrument. Fixed a rate (voltage) of movement, the rate between pixels and time will result in a perfect line in case of ideality of the system. Unfortunately the system is not ideal and therefore a fit of the points obtained is required as shown in Fig. 5.7.

In the case of the FSM the problem becomes more complicated given the precision required by the instrument, not all voltages stimulate the sensor's movement in the same way. Moreover, the asymmetry of the four axes is highlighted. This result in four different optimal voltages for the directions, in other words by fixing one direction and applying different voltages to it, the generated fits will have different mean-squared-error. Multiple data acquisitions are therefore required to search for optimal values for each direction. The best choice is therefore to choose the combination of voltage-direction that has the minor mean-squared-error, but two voltages are chosen: one high and one low. This is done because typically the best voltage is low and therefore produces movements that are too slow but very precise, so having a high tension for big movements is recommended. Thus, the combination of small precise movements and large rapid movements should be able to ensure good performance to the controller.

This calibration procedure is typically broken into two parts because when you create a system you do not know which are the best rates (voltages) and therefore must be searched. Once the ideal configuration is found, it always remains the same for the system so it is useful to have this configuration saved in a ready-to-use file.

Data acquisition In this phase several calibration points will be taken and stored into a file. The procedure is the following:

- in the *Analyzer* tab set the voltages and the time interval;
- autogenerate the filename;
- start the point acquisition.

Data analysis In this phase the calibration points stored in a file will be analyzed to retrieve the low/high voltages configuration. The procedure is the following:

- in the *Controller* tab select the file and press *Upload and Calibrated*;

- go into *Graphs* tab and check that the analysis is completed correctly;
- if everything is ok then go in the *Controller* tab and set the configuration to the low voltages or high voltages pressing the corresponding button.

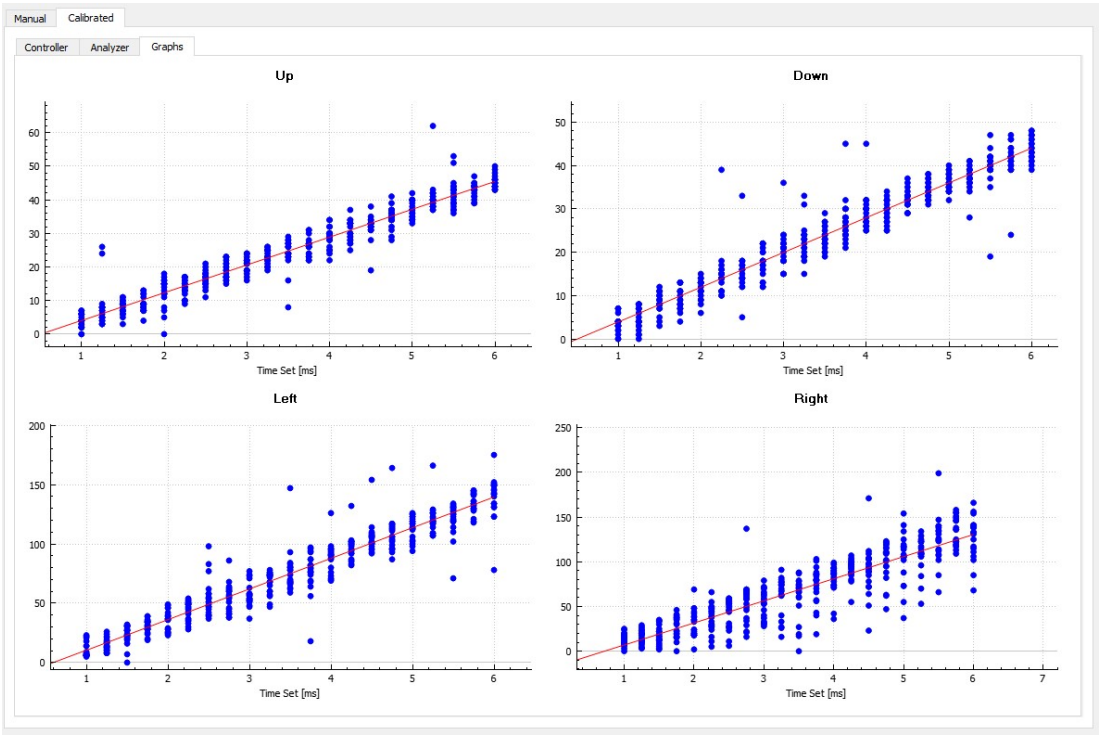


Figure 5.7: Example of data acquisition for the PATF done with mirror-camera-pc configuration. The volta used are: top 2V, bottom 2V, left 3V, right 3.5V.

One shot controller After the selection of both low and high voltages, the One shot checkbox is ready to be pressed in the *Controller* tab. To swap between high and low voltages a time threshold is available in the same tab. If the movement is too long a high voltages will be selected to reduce the time, if not a low voltages will be selected to increase the precision.

5.3 The repositories structure

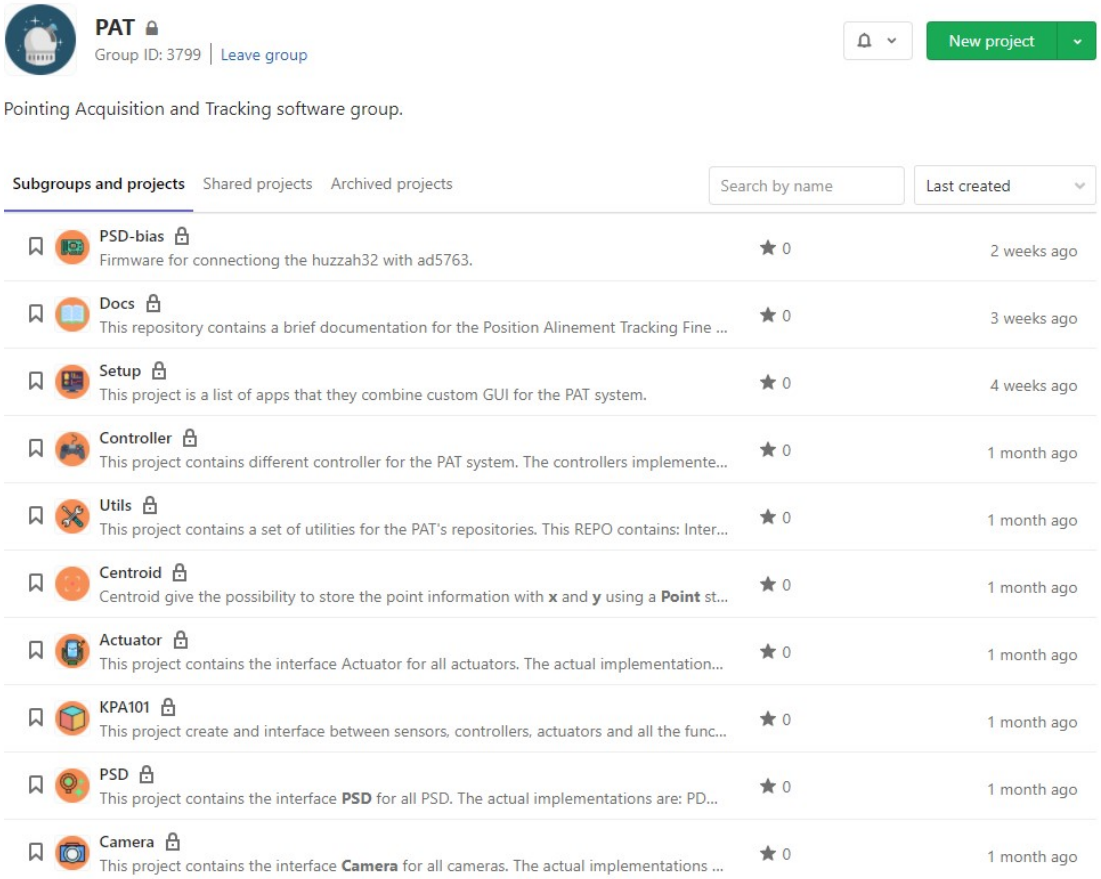


Figure 5.8: This is a screenshot of all the repositories developed in the PAT group.

Starting from the simplified model Fig. 5.2 has been chosen to create: a single repository for all controllers (Controller), a single repository for all actuators (Actuator), and two separate repositories for sensors (Camera, PSD). As a communication interface between Controller and Actuator was made a Actuator class. As a communication interface between the Controller class and Camera/PSD it was chosen to create a Centroid class stored in a separate repository (Centroid). The KPA101 repository was added to contain the KPA101 driver, while the Utils repository was created to contain a list of classes useful to all other repositories. The Setup repository was created with the idea of taking pieces from the various repositories to compose different setups, in other words, it contains a list of possible PAT combinations. The last two repositories that have been created are: Docs that contains quick documentation for programmers who want to contribute to the PAT, and PSD-bias that contains the software created to control a circuit that will be explained in the next chapter.

5.4 The repositories documentation

This section lists all the documentations of the developed repositories.

5.4.1 Utils

This project contains a set of utilities for the PAT's repositories. This REPO contains:

- Interval
- Point
- Range
- BoundedParameter

Requirements

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
  "name"     : "PATUtils",
  "path"     : "gitlab.dei.unipd.it/PAT/Utils.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "NO"
}
```

Prerequisites

The Prerequisites are the same descibed in the section "Setup your enviroment Prerequisites" Sec. 4.4.1.

UML

In figure Fig. 5.9 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

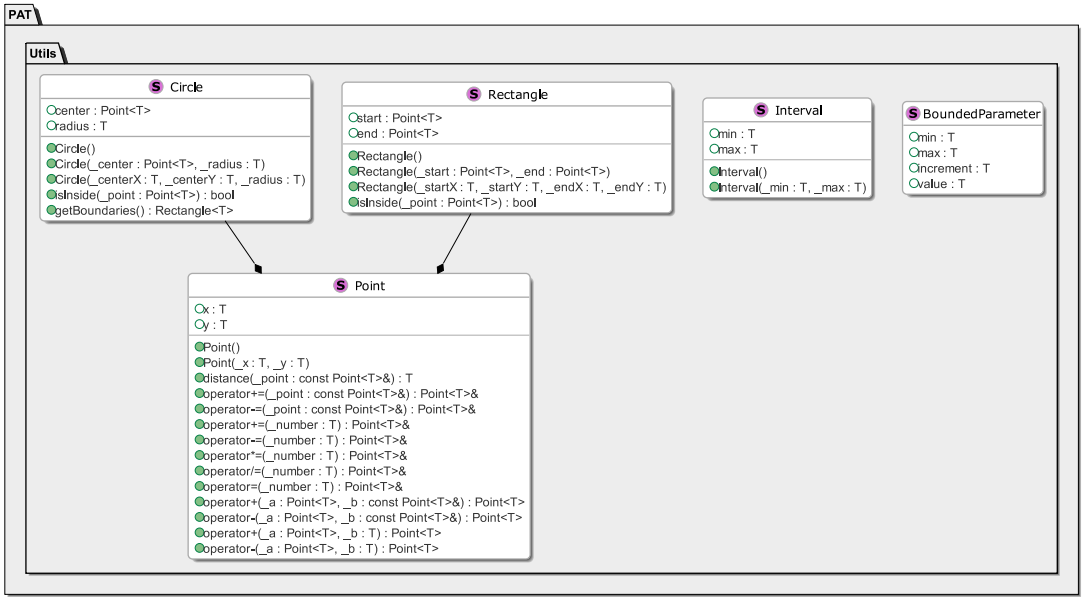


Figure 5.9: Utils UML.

Usage

Interval

Interval give the possibility to generate a base-type structure that has a **min** and **max** value.
First you need to include the header file and use the namespace.

```
#include <PAT/Utils/Interval.h>
using namespace PAT::Utils;
```

Then you can define the variable ready to be used.

```
Interval<double> interval{0, 100};
interval.min = 0;
interval.max = 100;
```

Point

Point give the possibility to generate a 2d-point that has a **x** and **y** value.
First you need to include the header file and use the namespace.

```
#include <PAT/Utils/Point.h>
using namespace PAT::Utils;
```

Then you can define the variable ready to be used.

```
Point<double> point{0, 100};
point.x = 0;
point.y = 100;
```

It has some operations already defined suc as: sum, division, scalar product.

```
Point<double> point2{0, 100};
point += point2;
point -= point2;
point /= 3;
point *= 3;
```

Rectangle

Rectangle give the possibility to generate a rectangular range defined by two points(**Point**) that they are **start** and **end**.
First you need to include the header file and use the namespace.

```
#include <PAT/Utils/Rectangle.h>
using namespace PAT::Utils;
```

Then you can define the variable ready to be used.

```
Rectangle<double> rectangle{{0, 0}, {100, 100}};
rectangle.start.x = 0;
rectangle.start.y = 0;
rectangle.end.x = 100;
rectangle.end.y = 100;
Point<double> point{50, 50};
if(rectangle.isInside(point))
 rectangle.end = point;
```

Circle

Circle give the possibility to generate a circular range defined by a central point(Point) and a radius.
First you need to include the header file and use the namespace.

```
#include <PAT/Utils/Circle.h>
using namespace PAT::Utils;
```

Then you can define the variable ready to be used.

```
Circle<double> circle{{0, 0}, 100};
circle.center.x = 0;
circle.center.y = 0;
circle.radius = 100;
Point<double> point{50, 50};
if(circle.isInside(point))
  circle.radius = point.x;
Rectangle<double> boundaries = circle.getBoundaries();
```

BoundedParameter

BoundedParameter give the possibility to define a parameter that has: **min** value, **max** value, **increment** value, and **value** value.
First you need to include the header file and use the namespace.

```
#include <PAT/Utils/BoundedParameter.h>
using namespace PAT::Utils;
```

Then you can define the variable ready to be used.

```
BoundedParameter<double> parameter{0, 100, 1, 50};
parameter.min = 0;
parameter.max = 100;
parameter.increment = 1;
parameter.value = 50;
```

Warning
The order of these parameters is imporant because is the same of the IDS camera (min, max, increment).

5.4.2 Centroid

This project contains the centroid class. This class contains the centroid information shared between the sensor and the controller in the PAT-system.
Requirements
cpp 11 cmake 3.16 git 2.0 Doxygen 1.8.13 Sphinx 3.0.2 OS Win OS Mac OS Linux

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
```

```
  "name"     : "PATCentroid",
  "path"     : "gitlab.dei.unipd.it/PAT/Centroid.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "NO"
}
```

Prerequisites

The Prerequisites are the same descibed in the section "Setup your enviroment Prerequisites" Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- Utils 1.0.0

UML

In figure Fig. 5.10 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

Figure 5.10: Centroid UML.

Usage

Centroid give the possibility to store the point information with **x** and **y** using a **Point<double>** structure, and also the **instensity** information.

It also contains a signal **newValue** that could be emitted when a new value is ready.
First you need to include the header file and use the namespace.

```
#include <PAT/Centroid.h>
using namespace PAT;
```

Then you can define the variable ready to be used.

```
Centroid centroid(10, 20, 100);
centroid.x = 10;
centroid.y = 20;
centroid.instensity = 100;
```

If you need to emit the signal use this function.

```
emit centroid.newValue();
```

5.4.3 KPA101

This project create and interface between sensors, controllers, actuators and all the functionalities of the KPA101.
Requirements


Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
  "name"     : "PATKPA101",
  "path"     : "gitlab.dei.unipd.it/PAT/KPA101.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "NO"
}
```

Prerequisites

The Prerequisites are the same descibed in the section ”Setup your enviroment Prerequisites” Sec. 4.4.1 with the addition of the following libraries that are auto

fetched from the gitlab.dei.unipd.it:

- Utils 1.0.0

UML

In figure Fig. 5.11 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

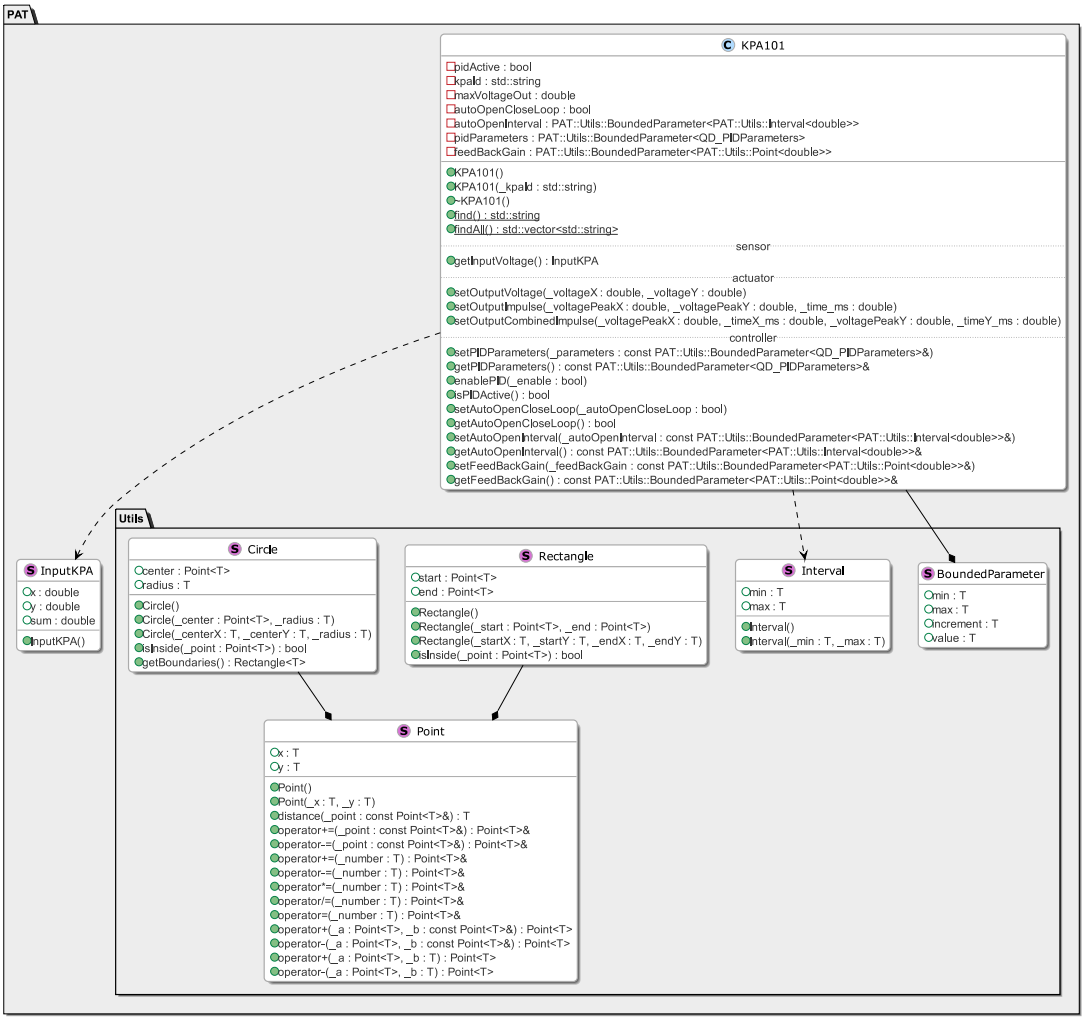


Figure 5.11: KPA101 UML.

Usage

InputKPA

InputKPA is a utility structure that contains the three voltagies that KPA101 takes in input: **x**, **y**, **sum**.
First you need to include the header file and use the namespace.

```
#include <PAT/InputKPA.h>
using namespace PAT;
```

Then you can define the variable ready to be used.

```
InputKPA parameters;
parameters.x = 1;
parameters.y = 2;
parameters.sum = 3;
```

KPA101

KPA101 is a wrapper for the Kinesis libs that enhance their functionalities.
First you need to include the header file and use the namespace.

```
#include <PAT/KPA101.h>
using namespace PAT;
```

In order to define a variable you can choose to pass the **kpa id** or to let it automatically search it. If the search fails the program throw an exception "not found". To search the id you can use **find** and **findAll** functions.

```
KPA101 kpa101;        // auto
KPA101 kpa101("kpaid"); // with id
KPA101 kpa101(KPA101::find());    // with find
KPA101 kpa101(KPA101::findAll()[0]); // with findAll
```

Sensor

Here are listed the functions useful for sensors.
If you need the get the current input state you can use this function.

```
InputKPA getInputVoltage() const;
```

Actuator

Here are listed the functions useful for actuators.
You can set the output state of the KPA using one of these three functions. Output voltages are between 10 and 0.

- The function **setOutputVoltage** set the output voltages at the selected values.
- The function **setOutputImpulse** send a single impulse with length **_time_ms**.
- The function **setOutputCombinedImpulse** send two impulses at the same time with with their respective lengths. I one finish before the other its voltage go to zero.

```
double voltageX = 1;
double voltageY = 1;
kpa101.setOutputVoltage(voltageX, voltageY);

double voltagePeakX = 1;
double voltagePeakY = 1;
double time_ms    = 1;
kpa101.setOutputImpulse(voltagePeakX, voltagePeakY, time_ms);

double voltagePeakX = 1;
double timeX_ms   = 1;
double voltagePeakY = 1;
double timeY_ms   = 1;
kpa101.setOutputCombinedImpulse(voltagePeakX, timeX_ms, voltagePeakY,
    timeY_ms);
```

Controller

Here are listed the functions useful to use the PID controller integrated in the KPA.

You can set the PID parameters using these functions. PID values are between 1 and 0.

```
auto parameters = kpa101.getPIDParameters();
parameters.value.proportionalGain = 1;
parameters.value.integralGain = 0.001;
parameters.value.differentialGain = 0.01;
kpa101.setPIDParameters(parameters);
```

You can enable/disable or check if enabled the PID controller using these functions.

```
if(!kpa101.isPIDActive())
  kpa101.enablePID(true);
else
  kpa101.enablePID(false);
```

You can enable/disable the auto open closed loop. If the intensity of the light is outside the interval **autoOpenInterval** then the PID controller is stopped. The intensity value is bounded between 10 and 0.

```
auto interval = kpa101.getAutoOpenInterval();
interval.value.min = 0.1;
interval.value.max = 1.3;
kpa101.setAutoOpenInterval(interval);
if(!kpa101.getAutoOpenCloseLoop())
  kpa101.setAutoOpenCloseLoop(true);
```

The output of the PID controller is multiplied by a feedback constant that range between -10 to +10 volts. Using this function you can set it.

```
auto feedback = kpa101.getFeedBackGain();
gain.value.x = 10;
gain.value.y = 10;
kpa101.setFeedBackGain(feedback);
```

5.4.4 Actuator

This project contains the interface **Actuator** for all actuators. The actual implementations are: STT254.

Requirements

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
  "name"     : "PATActuator",
  "path"     : "gitlab.dei.unipd.it/PAT/Actuator.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "NO"
}
```

Prerequisites

The Prerequisites are the same descibed in the section ”Setup your enviroment Prerequisites” Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- KPA101 1.0.0

UML

In figure Fig. 5.12 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

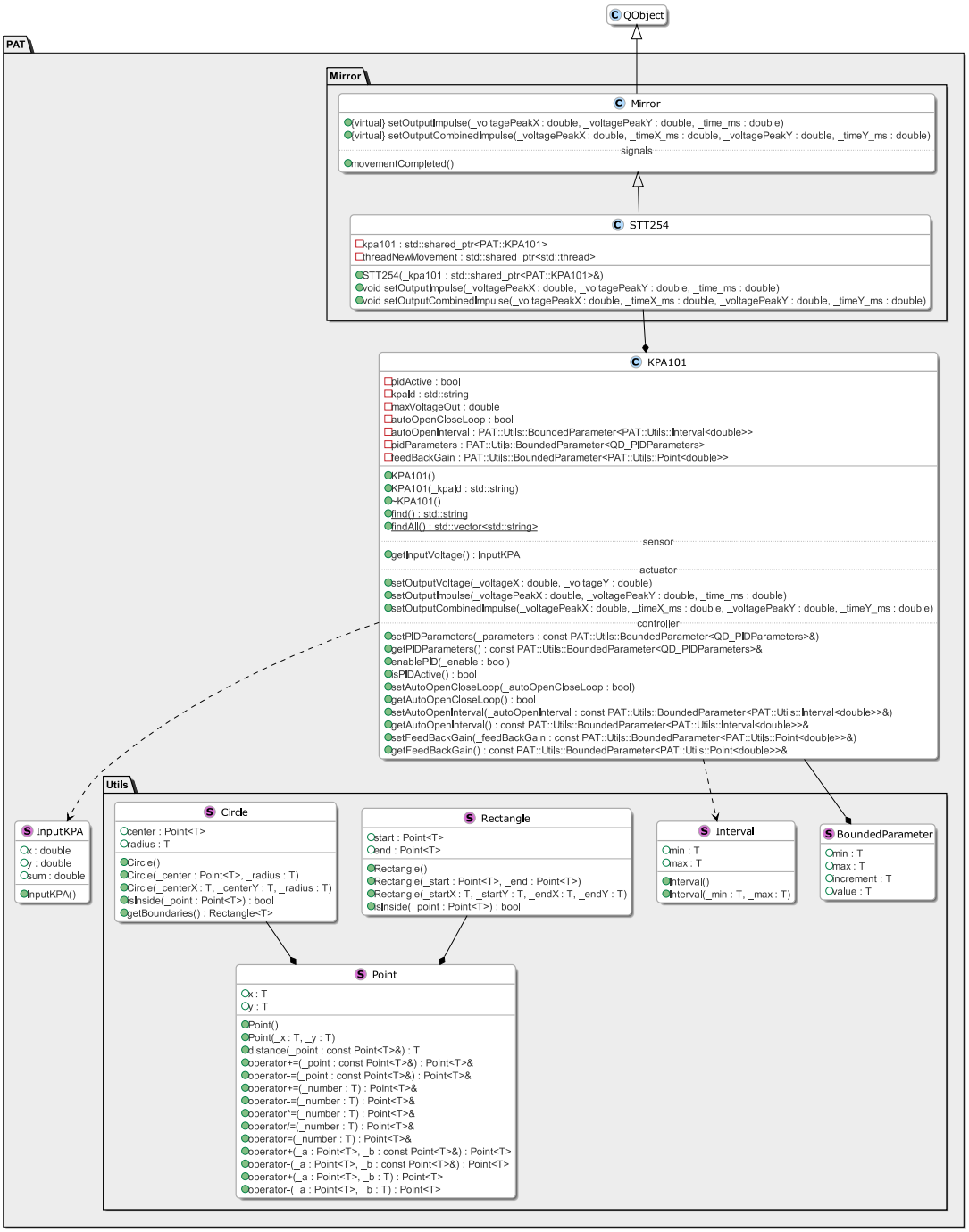


Figure 5.12: Actuator UML.

Usage

Actuator

The **Actuator** class is a interface for all actuators. Here is the include code.

```
#include <PAT/Actuator/Actuator.h>
```

To inherit the class you have to overwrite the two methods: **setOutputImpulse**, **setOutputCombinedImpulse**. When the impulse finish remember to emit a **movementCompleted** signal that is already defined in **Actuator** class.

```
class YourClass : public Actuator {
  public:
    void setOutputImpulse(double _voltagePeakX, double _voltagePeakY,
        double _time_ms) override;
    void setOutputCombinedImpulse(double _voltagePeakX, double
        _timeX_ms, double _voltagePeakY, double _timeY_ms) override;
  };
```

STT254

The **STT254** class is a Actuator implementation that works with the **KPA101**. Here is the include code.

```
#include <PAT/KPA101.h>
#include <PAT/Actuator/STT254.h>
using namespace PAT::Actuator;
```

When you istantiate the **STT254** you must pass a **KPA101** istance. The **STT254** is typically used as polymorphism with **Actuator**.

```
std::shared_ptr<PAT::KPA101> kpa101     =
    std::make_shared<PAT::KPA101>();
std::shared_ptr<PAT::Actuator::Actuator> actuator =
    std::make_shared<PAT::Actuator::STT254>(kpa101);
```

You can set the output state of the **Actuator** using one of these two functions. Output voltages are between 10 and 0. When the functions finish they emit a a **movementCompleted** signal.

- The function **setOutputImpulse** send a single impulse with length **_time_ms**.
- The function **setOutputCombinedImpulse** send two impulses at the same time with with their respective lengths. I one finish before the other its voltage go to zero.

```
double voltagePeakX = 1;
double voltagePeakY = 1;
double time_ms    = 1;
actuator->setOutputImpulse(voltagePeakX, voltagePeakY, time_ms);
```

```
double voltagePeakX = 1;
double timeX_ms   = 1;
double voltagePeakY = 1;
double timeY_ms   = 1;
actuator->setOutputCombinedImpulse(voltagePeakX, timeX_ms,
    voltagePeakY, timeY_ms);
```

5.4.5 PSD

This project contains the interface **PSD** for all PSD. The actual implementations are: PDP90A.

Requirements

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
  "name"     : "PATPSD",
  "path"     : "gitlab.dei.unipd.it/PAT/PSD.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "YES"
}
```

Prerequisites

The Prerequisites are the same descibed in the section ”Setup your enviroment Prerequisites” Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- KPA101 1.0.0
- Centroid 1.0.0

UML

In figure Fig. 5.13 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

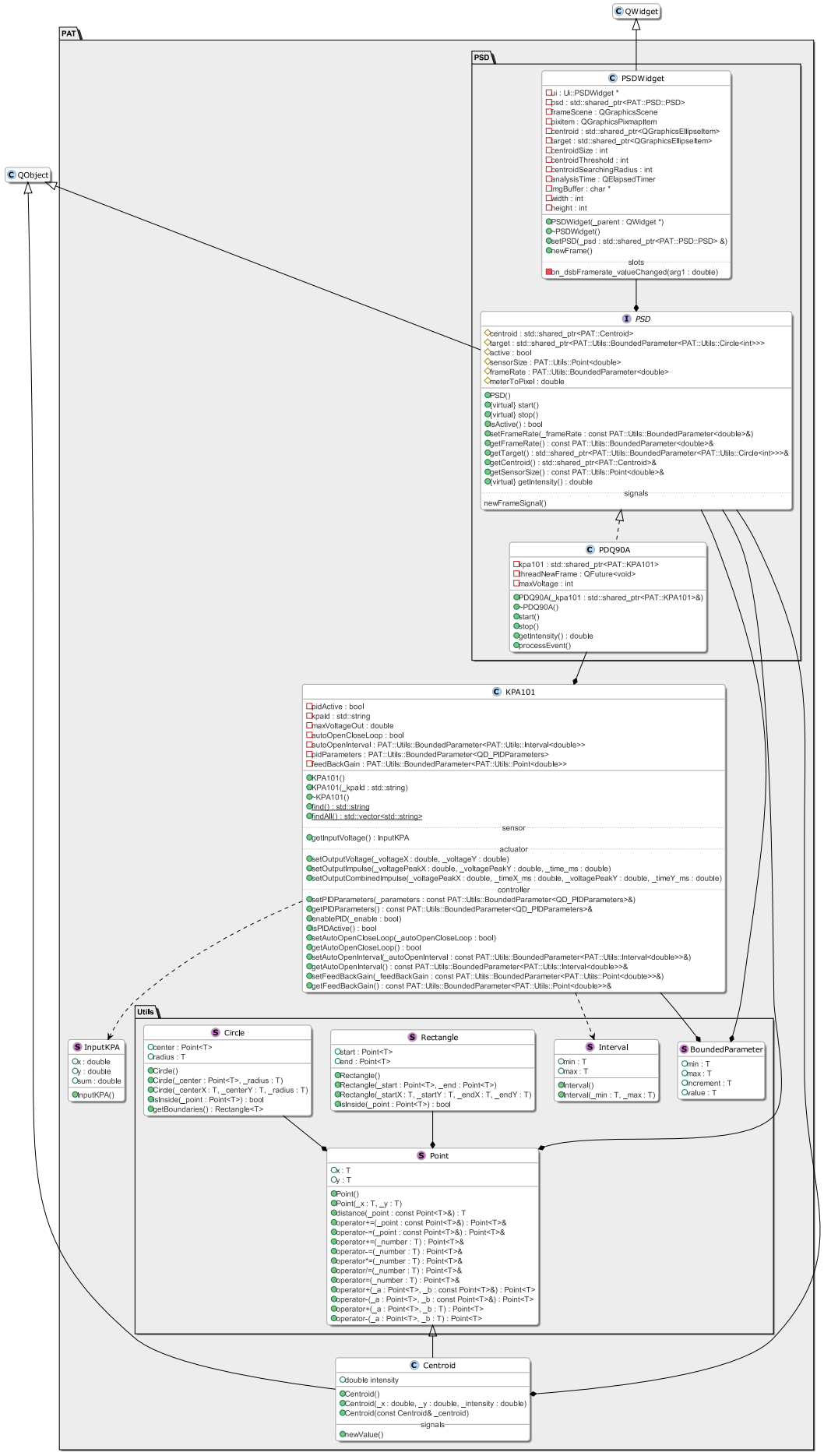


Figure 5.13: PSD UML.

Usage

PSD

The **PSD** class is a interface for all PSD. Here is the include code.

```
#include <PAT/PSD/PSD.h>
```

To inherit the class you have to overwrite the three methods: **start**, **stop**, **getIntensity**.

```
class YourClass : public PSD {
  public:
    void start() override;
    void stop() override;
    double getIntensity() const override;
  };
```

PDP90A

The **PDP90A** class is a PSD implementation that works with the **KPA101**. Here is the include code.

```
#include <PAT/KPA101.h>
#include <PAT/PSD/PDP90A.h>
using namespace PAT::PSD;
```

When you istantiate the **PDP90A** you must pass a **KPA101** istance. The **PDP90A** is typically used as polymorphism with **PSD**.

```
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<PAT::PSD::PSD> psd =
    std::make_shared<PAT::PSD::PDP90A>(kpa101);
```

You can enable/disable or check if enabled the PDP90A's acquisition using these functions.

```
if(!psd->isActive())
  psd->start();
lse
  psd->stop();
```

You can set the frame rate of the acquisition using these functions. At every frame the PSD calculates the centroid and then it emit a signal **newFrameSignal**.

```
auto framerate = psd->getFrameRate();
framerate.value = 30;
psd->setFrameRate(framerate);
```

You can get the other parameters using these functions.

```
PAT::Utils::Point<double> sensorSize = psd->getSensorSize();
std::shared_ptr<PAT::Utils::BoundedParameter<PAT::Utils::Circle<int>>>
    target = psd->getTarget();
std::shared_ptr<PAT::Centroid> centroid = psd->getCentroid();
PAT::Utils::Point<double> sensorSize = psd->getSensorSize();
double intensity = psd->getIntensity();
```

PSDWiget

The **PSDWiget** class is a widget fot the PSD's implementations. Here I present the PDP90A example. The include code is the following.

```
#include <QApplication>
#include <PAT/PSD/gui/PSDWidget.h>
#include <PAT/PSD/PDP90A.h>
using namespace PAT::PSD;
```

This is a simple **main** example.

```
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  PSDWidget cameraWidget;
  std::shared_ptr<PAT::KPA101> kpa101 =
      std::make_shared<PAT::KPA101>();
  std::shared_ptr<PSD> camera      = std::make_shared<PDP90A>(kpa101);
  cameraWidget.setPSD(camera);
  cameraWidget.show();
  camera->start();
  return a.exec();
```

If you want instead integrate the widget into another GUI (layout) you can use this code.

```
QWidget *window                  = new QWidget();
QVBoxLayout *mainLayout          = new QVBoxLayout();
PSDWidget *psdWidget             = new PSDWidget();
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<PSD> camera      = std::make_shared<PDP90A>(kpa101);

psdWidget->setPSD(psd);
```

```
mainLayout->addWidget((QWidget *)psdWidget);
window->setLayout(mainLayout);
window->show();
```

5.4.6 Camera

This project contains the interface **Camera** for all cameras. The actual implementations are: IDS.

Requirements

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

Codice 5.1: PATCamera module inclusion.

```
{
  "name"     : "PATCamera",
  "path"     : "gitlab.dei.unipd.it/PAT/Camera.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "YES"
}
```

Prerequisites

The Prerequisites are the same descibed in the section "Setup your enviroment Prerequisites" Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- Centroid 1.0.0

UML

In figure Fig. 5.14 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

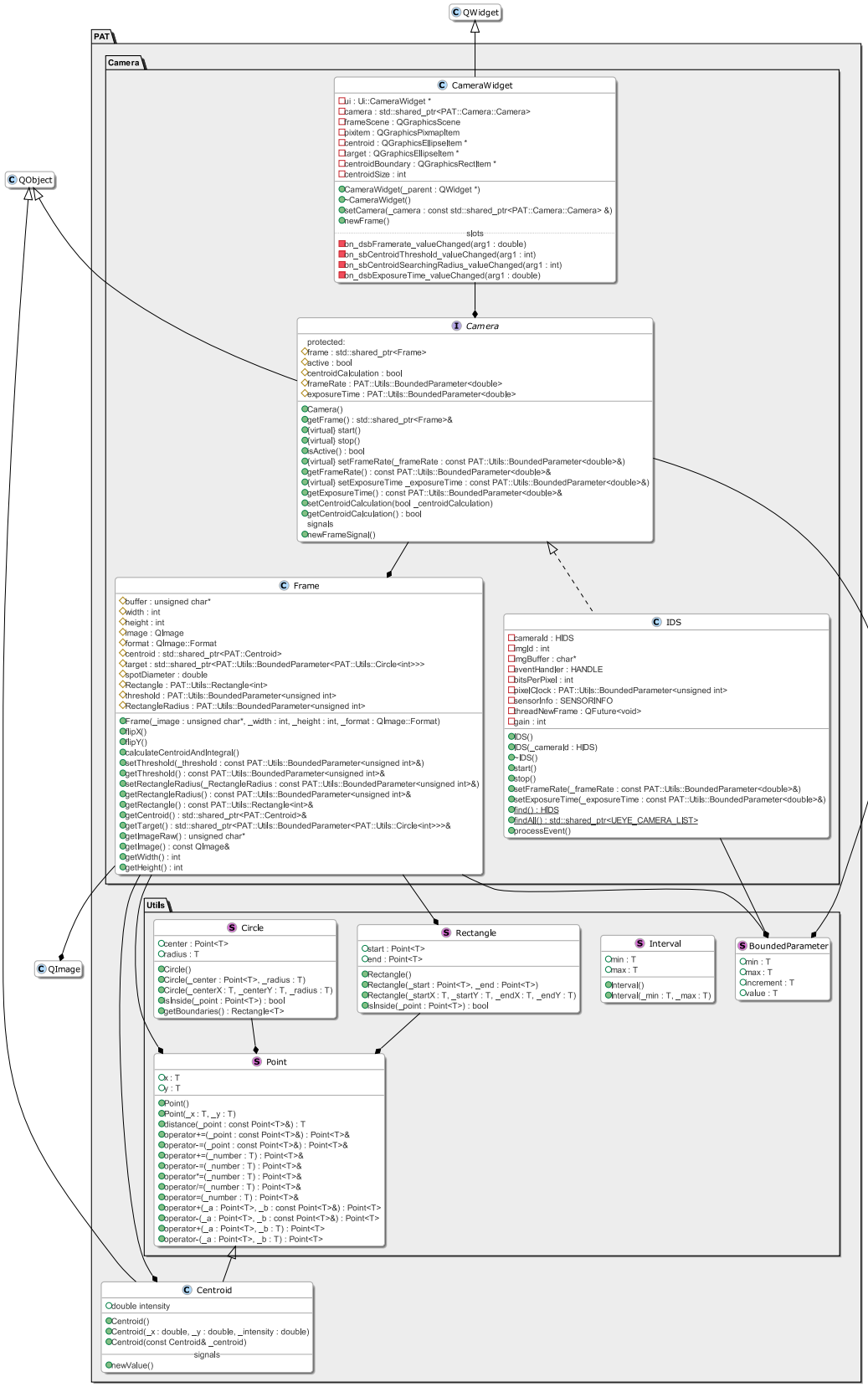


Figure 5.14: Camera UML.

Usage

Frame

The **Frame** class manages all the processing of the images captured by the camera.

Codice 5.2: PAT/Camera/Frame inclusion.

```
#include <PAT/Camera/Frame.h>
```

This is an example of initialization.

Codice 5.3: Frame initialization example.

```
int width = 100;
int height = 100;
unsigned char imgBuffer[width * height] = {0};
std::shared_ptr<Frame> frame = std::make_shared<Frame>(imgBuffer,
    width, height, QImage::Format_Grayscale8);
```

You can flip the image in the two axes usign these functions.

```
frame->flipX();
frame->flipY();
```

You can calculate the centroid, the spotDiameter and the integral (spot intensity) using this function. If it does not find the centroid it returns -1 as invalid value.

```
frame->calculateCentroidAndIntegral();
```

You can also control the centroid calculation setting the parameters **threshold** and **rangeRadius**. - **setThreshold/getThreshold** set/get the minimum value for pixels to be counted in the centroid average. - **setRangeRadius/getRangeRadius** set/get the range radius in pixel where to search the centroid. The value 0 means infinite range. - **getRange** get the four cordinate of the range where to search the centroid.

```
auto threshold = frame->getThreshold();
threshold.value = 30;
frame->setThreshold(threshold);

auto rangeRadius = frame->getRangeRadius();
rangeRadius.value = 200;
frame->setRangeRadius(rangeRadius);

PAT::Utils::Range<int> range = frame->getRange();
```

You can get the other parameters using these functions.

```
std::shared_ptr<PAT::Centroid> centroid = frame->getCentroid();
std::shared_ptr<PAT::Utils::BoundedParameter<PAT::Utils::Circle<int>>>
    target = frame->getTarget();
unsigned char* imgBuffer = frame->getImageRaw();
QImage img = frame->getImage();
int width = frame->getWidth();
int height = frame->getHeight();
```

Camera

The **Camera** class is a interface for all Camera. Here is the include code.

```
#include <PAT/Camera/Camera.h>
```

To inherit the class you have to overwrite the four methods: **start**, **stop**, **setFrameRate**, **setExposureTime**.

```
class YourClass : public Camera {
  public:
    void start() override;
    void stop() override;
    void setFrameRate(const PAT::Utils::BoundedParameter<double>&
        _frameRate) override;
    void setExposureTime(const PAT::Utils::BoundedParameter<double>&
        _exposureTimeValue) override;
  };
```

IDS

The **IDS** class is a Camera implementation. Here is the include code.

```
#include <PAT/Camera/IDS.h>
using namespace PAT::Camera;
```

The **IDS** is typically used as polymorphism with **Camera**. In order to define a variable you can choose to pass the **IDS id** or to let it automatically search it. If the search fails the program throw an exception "not found". To search the id you can use **find** and **findAll** functions.

```
std::shared_ptr<PAT::Camera::Camera> camera =
    std::make_shared<PAT::Camera::IDS>();
```

```
std::shared_ptr<PAT::Camera::Camera> camera =
    std::make_shared<PAT::Camera::IDS>(IDS::find());
std::shared_ptr<PAT::Camera::Camera> camera =
    std::make_shared<PAT::Camera::IDS>(IDS::findAll()->uci[0].dwCameraID);
```

You can enable/disable or check if enabled the IDS's acquisition using these functions.

```
if(!camera->isActive())
  camera->start();
lse
  camera->stop();
```

You can set the frame rate of the acquisition using these functions. At every frame the Camera calculates the centroid and then it emit a signal **newFrameSignal**.

```
auto framerate = camera->getFrameRate();
framerate.value = 30;
camera->setFrameRate(framerate);
```

You can set the exposure time of the acquisition using these functions. The exposure time limit depends on the frame rate (1 / framerate).

```
auto exposureTime = camera->getExposureTime();
exposureTime.value = 1 / 30;
camera->setExposureTime(exposureTime);
```

You can enable/disable or check if enabled the centroid calculation using these functions.

```
if(!camera->getCentroidCalculation())
  camera->setCentroidCalculation(true);
lse
  camera->setCentroidCalculation(false);
```

You can alse obtain the frame object that automatically update at every frame.

```
std::shared_ptr<Frame> frame = camera->getFrame();
```

CameraWiget

The **CameraWiget** class is a widget fot the Camera's implementations. Here I present the IDS example. The include code is the following.

```
#include <QApplication>
```

```
#include <PAT/Camera/gui/CameraWidget.h>
#include <PAT/Camera/IDS.h>
using namespace PAT::Camera;
```

This is a simple **main** example.

```
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  CameraWidget cameraWidget;
  std::shared_ptr<Camera> camera = std::make_shared<IDS>();
  cameraWidget.setCamera(camera);
  cameraWidget.show();
  camera->start();
 return a.exec();
}
```

If you want instead integrate the widget into another GUI (layout) you can use this code.

```
QWidget *window             = new QWidget();
QVBoxLayout *mainLayout     = new QVBoxLayout();
CameraWidget *cameraWidget  = new CameraWidget();
std::shared_ptr<Camera> camera = std::make_shared<IDS>();

cameraWidget->setCamera(camera);
mainLayout->addWidget((QWidget *)cameraWidget);
window->setLayout(mainLayout);
window->show();
```

5.4.7 Controller

This project contains different controllers for the PAT system. The controllers implemented at moment are: Manual, Calibrated, PIDKPA101.

Requirements

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
  {
    "name"    : "PATController",
    "path"    : "gitlab.dei.unipd.it/PAT/Controller.git",
```

```
    "tag"      : "HEAD",
    "available": "YES",
    "getGui"   : "YES"
}
```

Prerequisites

The Prerequisites are the same descibed in the section ”Setup your enviroment Prerequisites” Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- Mirror 1.0.0
- PSD 1.0.0
- Camera 1.0.0

UML

In figure Fig. 5.15 is illustrated the UML of the repository that shows the relationships between the classes that will be describe below.

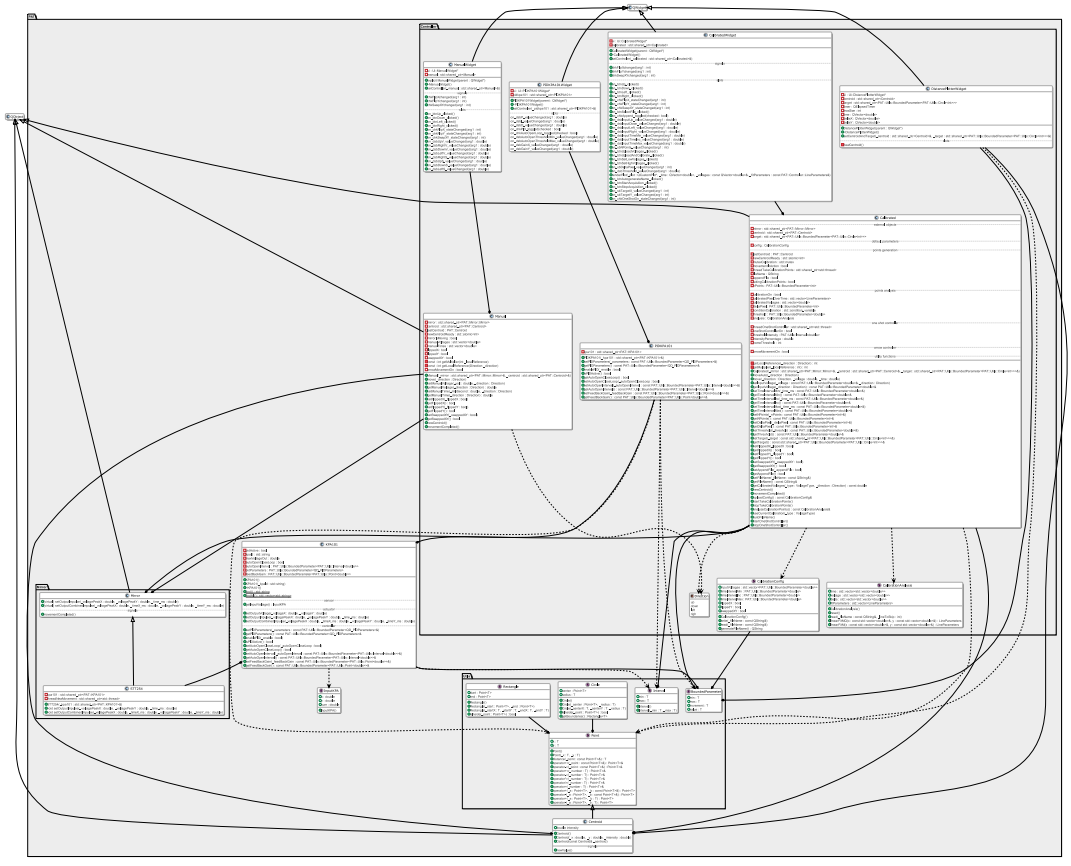


Figure 5.15: Controller UML.

Usage

PIDKPA101

This class offers a set of functions to comunicate with the PID controller integrated in the KPA101.
First you need to include the header file and use the namespace.

```
#include <PAT/Controller/PIDKPA101.h>
using namespace PAT::Controller;
```

When you istantiate the **PIDKPA101** you must pass a **KPA101** istance.

```
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<PAT::Controller::PIDKPA101> pidkpa101 =
    std::make_shared<PAT::Controller::PIDKPA101>(kpa101);
```

You can enable/disable or check if enabled the PIDKPA101 controller using these functions.

```
if(!pidkpa101->isPIDActive())
  pidkpa101->enablePID(true);
lse
  pidkpa101->enablePID(false);
```

You can set the PID parameters using these functions. PID values are between 1 and 0.

```
auto parameters = pidkpa101->getPIDParameters();
parameters.value.proportionalGain = 1;
parameters.value.integralGain = 0.001;
parameters.value.differentialGain = 0.01;
pidkpa101->setPIDParameters(parameters);
```

You can enable/disable the auto open closed loop. If the intensity of the light is outside the interval **autoOpenInterval** then the PID controller is stopped. The intensity value is bounded between 10 and 0.

```
auto interval = pidkpa101->getAutoOpenInterval();
interval.value.min = 0.1;
interval.value.max = 1.3;
pidkpa101->setAutoOpenInterval(interval);
if(!pidkpa101->getAutoOpenCloseLoop())
 pidkpa101->setAutoOpenCloseLoop(true);
```

The output of the PID controller is multiplied by a feedback constant that range

between -10 to +10 volts. Using this function you can set it.

```
auto feedback = pidkpa101->getFeedBackGain();
gain.value.x = 10;
gain.value.y = 10;
pidkpa101->setFeedBackGain(feedback);
```

Manual

Manual is a controller that allow to move a mirror in the four directions. It require a **Mirror** and a **Centroid**. This last one is from a **Frame** inside a **Camera** or from a **PSD**. For semplicity here I provide an example using a **IDS** camera and a **STT254** mirror.
First you need to include the header file and use the namespace.

```
#include <PAT/Controller/Manual.h>
#include <PAT/Mirror/STT254.h>
#include <PAT/Camera/IDS.h>
using namespace PAT::Controller;
using namespace PAT::Camera;
using namespace PAT::Mirror;
```

When you istantiate the **Manual** you must pass a **Mirror** and a **Centroid** istances.

```
std::shared_ptr<Camera> camera   = std::make_shared<IDS>();
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<Mirror> mirror   = std::make_shared<STT254>(kpa101);
std::shared_ptr<Manual> manual   = std::make_shared<Manual>(mirror,
    camera->getFrame()->getCentroid());
```

To move in one of the four directions you can use this code.

```
manual->move(Direction::up);
manual->move(Direction::down);
manual->move(Direction::left);
manual->move(Direction::right);
```

To choose how much move you need to set times (milliseconds) and voltages (volts) in the four directions. You can also control the current values.

```
if(manual->getManualVoltage(Direction::up) > 100)
  manual->setManualVoltage(1, Direction::up);
if(manual->getManualTime(Direction::up) > 100)
  manual->setManualTime(1, Direction::down);
```

You can also invert or check the movement directions in the two axes. You can also swap the two axes.

```
if(!manual->getFlippedX())
  manual->setFlippedX(true);
else
  manual->setFlippedX(false);

if(!manual->getFlippedY())
  manual->setFlippedY(true);
else
  manual->setFlippedY(false);

if(!manual->getSwappedXY())
  manual->setSwappedXY(true);
else
  manual->setSwappedXY(false);
```

ManualWidget

The **ManualWidget** class is a widget for the Manual controller. Here I present the an example consistent with the Manual one. The include code is the following.

```
#include <QApplication>
#include <PAT/Controller/gui/ManualWidget.h>
#include <PAT/Mirror/STT254.h>
#include <memory>
#include <PAT/Camera/IDS.h>
using namespace PAT::Controller;
using namespace PAT::Camera;
using namespace PAT::Mirror;
```

This is a simple **main** example.

```
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  ManualWidget manualWidget;
  std::shared_ptr<Camera> camera  = std::make_shared<IDS>();
  std::shared_ptr<PAT::KPA101> kpa101 =
      std::make_shared<PAT::KPA101>();
  std::shared_ptr<Mirror> mirror  = std::make_shared<STT254>(kpa101);
  std::shared_ptr<Manual> manual  = std::make_shared<Manual>(mirror,
      camera->getFrame()->getCentroid());
  manualWidget.setController(manual);
  manualWidget.show();
  camera->start();
```

```
  return a.exec();
}
```

If you want instead integrate the widget into another GUI (layout) you can use this code.

```
QWidget *window                  = new QWidget();
QVBoxLayout *mainLayout          = new QVBoxLayout();
ManualWidget *manualWidget       = new ManualWidget();

std::shared_ptr<Camera> camera   = std::make_shared<IDS>();
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<Mirror> mirror   = std::make_shared<STT254>(kpa101);
std::shared_ptr<Manual> manual   = std::make_shared<Manual>(mirror,
    camera->getFrame()->getCentroid());

manualWidget->setPSD(manual);
mainLayout->addWidget((QWidget *)manualWidget);
window->setLayout(mainLayout);
window->show();
```

There are also these three signal functions that they are used to propagate the click of their respective buttons in other controllers.

```
void chkFlipXchanged(int arg1);
void chkFlipYchanged(int arg1);
void chkSwapXYchanged(int arg1);
```

CalibrationConfig

This structure contains the main parameters for the calibration. The parameters are: the four voltages, one per direction; the time durations of the movements described by min-max-increment; the inversion or swap of the axes. The include code is the following.

```
#include <PAT/Controller/CalibrationConfig.h>
```

You can also store/unstore from a file these parameters.

```
CalibrationConfig parameters;
QString filename = "test.txt";
parameters.write(filename);
parameters.read(filename);
```

CalibrationAnalysis

This is a support structure to take the data from a file and fit them using a linear fit. The include code is the following.

```
#include <PAT/Controller/CalibrationAnalysis.h>
```

This structure can analyze file data generated by **Calibrated** controller.

```
CalibrationAnalysis analysis;
QString filename = "test.txt";
analysis.read(filename);
```

Calibrated

Calibrated is a controller that allow to move a mirror in the four directions. It is able to calibrate itself using linear fit. It also contain a One shot controller that in one frame it try to align the centroid to the target point. It require a **Mirror**, a **Centroid** and a target point. The **Centroid** and the calibrated point are from a **Frame** inside a **Camera** or from a **PSD**. For semplicity here I provide an example using a **IDS** camera and a **STT254** mirror.
First you need to include the header file and use the namespace.

```
#include <PAT/Controller/Calibrated.h>
#include <PAT/Mirror/STT254.h>
#include <PAT/Camera/IDS.h>
using namespace PAT::Controller;
using namespace PAT::Camera;
using namespace PAT::Mirror;
```

When you istantiate the **Calibrated** you must pass a **Mirror** and a **Centroid** istances.

```
std::shared_ptr<Camera> camera     = std::make_shared<IDS>();
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<Mirror> mirror     = std::make_shared<STT254>(kpa101);
std::shared_ptr<Calibrated> calibrated =
    std::make_shared<Calibrated>(mirror,
    camera->getFrame()->getCentroid(), camera->getFrame()->getTarget());
```

To calibrate the controller you need first to collect the data. To do so you need to choose a configuration and a file name. You can use this code as example of command you require.

```
auto upVoltage   = calibrated->getInputVoltage(Direction::up);
```

```
auto downVoltage = calibrated->getInputVoltage(Direction::down);
auto leftVoltage = calibrated->getInputVoltage(Direction::left);
auto rightVoltage = calibrated->getInputVoltage(Direction::right);
upVoltage.value  = 2;
downVoltage.value = 2;
leftVoltage.value = 2;
rightVoltage.value = 2;
calibrated->setInputVoltage(upVoltage, Direction::up);
calibrated->setInputVoltage(downVoltage, Direction::down);
calibrated->setInputVoltage(leftVoltage, Direction::left);
calibrated->setInputVoltage(rightVoltage, Direction::right);

auto timeMin = calibrated->getTimeIntervalMin();
auto timeInc = calibrated->getTimeIntervalInc();
auto timeMax = calibrated->getTimeIntervalMax();
timeMin.value = 1;
timeInc.value = 0.5;
timeMax.value = 5;
calibrated->setTimeIntervalMin(timeMin);
calibrated->setTimeIntervalInc(timeInc);
calibrated->setTimeIntervalMax(timeMax);

auto nPoints = calibrated->getNPoints();
nPoints.value = 10;
calibrated->setNPoints(nPoints);

calibrated->setFlippedX(true);
calibrated->setFlippedY(false);
calibrated->setSwappedXY(false);

calibrated->autoFileName();
```

After selected the config you can start/stop to collect the data.

```
calibrated->startTakeCalibrationPoints();
calibrated->stopTakeCalibrationPoints();
```

After collected the data you can analyze it and store the result in cache.

```
calibrated->analyzeCalibrationPoints();
```

To conclude the process, when you are happy with the cached data you can use it as configuration to the calibrated movement. Remember to select the time threshold in milliseconds. If the calibrated movement time is higher than threshold the controller use an high voltage else a low voltage.

```
calibrated->useCurrentCalibration(VoltageType::low);
```

```
calibrated->useCurrentCalibration(VoltageType::high);
auto threshold = calibrated->getThreshold();
threshold.value = 2;
calibrated->setThreshold(threshold);
```

Completed this process you can use the directional movement or the One Shot controller.

```
auto deltaPx = calibrated->getDeltaPixel();
deltaPx.value = 50;
calibrated->setDeltaPixel(deltaPx);
calibrated->moveAuto(Direction::up);

calibrated->startOneShotController();
calibrated->stopOneShotController();
```

CalibratedWidget

The **CalibratedWidget** class is a widget for the Calibrated controller. Here I present the an example consistent with the Calibrated one. The include code is the following.

```
#include <QApplication>
#include <PAT/Controller/gui/CalibratedWidget.h>
#include <PAT/Mirror/STT254.h>
#include <memory>
#include <PAT/Camera/IDS.h>
using namespace PAT::Controller;
using namespace PAT::Camera;
using namespace PAT::Mirror;
```

This is a simple **main** example.

```
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  CalibratedWidget calibratedWidget;
  std::shared_ptr<Camera> camera   = std::make_shared<IDS>();
  std::shared_ptr<PAT::KPA101> kpa101 =
      std::make_shared<PAT::KPA101>();
  std::shared_ptr<Mirror> mirror   = std::make_shared<STT254>(kpa101);
  std::shared_ptr<Calibrated> calibrated =
      std::make_shared<Calibrated>(mirror,
      camera->getFrame()->getCentroid(),
      camera->getFrame()->getTarget());
  calibratedWidget.setController(calibrated);
  calibratedWidget.show();
```

```
  camera->start();
  return a.exec();
}
```

If you want instead integrate the widget into another GUI (layout) you can use this code.

```
QWidget *window                  = new QWidget();
QVBoxLayout *mainLayout          = new QVBoxLayout();
CalibratedWidget *calibratedWidget = new CalibratedWidget();

std::shared_ptr<Camera> camera     = std::make_shared<IDS>();
std::shared_ptr<PAT::KPA101> kpa101 = std::make_shared<PAT::KPA101>();
std::shared_ptr<Mirror> mirror     = std::make_shared<STT254>(kpa101);
std::shared_ptr<Calibrated> calibrated =
    std::make_shared<Calibrated>(mirror,
    camera->getFrame()->getCentroid());

calibratedWidget->setPSD(calibrated);
mainLayout->addWidget((QWidget *)calibratedWidget);
window->setLayout(mainLayout);
window->show();
```

There are also these three signal functions that they are used to propagate the click of their respective buttons in other controllers.

```
void chkFlipXchanged(int arg1);
void chkFlipYchanged(int arg1);
void chkSwapXYchanged(int arg1);
```

DistancePlotterWidget

The **DistancePlotterWidget** class is a widget for plot the distance between a centroid and a point. This widget is used in combination with **CalibratedWiget** and **ManualWidget**. The include code is the following.

```
#include <QApplication>
#include <PAT/Controller.h>
#include <PAT/Centroid.h>
#include <PAT/Utils/Point.h>
#include <memory>
#include <thread>
#include <stdlib.h>
#include <QDebug>
using namespace PAT::Controller;
```

This is a simple **main** example.

```
int main(int argc, char *argv[]) {
  QApplication a(argc, argv);
  DistancePlotterWidget distancePlotterWidget;
  std::shared_ptr<PAT::Centroid> centroid(new PAT::Centroid());
  std::shared_ptr<PAT::Utils::BoundedParameter<PAT::Utils::Circle<int>>>
      target(new
      PAT::Utils::BoundedParameter<PAT::Utils::Circle<int>>({0, 1000,
      1, 500}, {0, 1000, 1, 500}));
  double frameRate = 30;
  distancePlotterWidget.setCentroidAndTarget(centroid, target);
  std::thread t1([&]() {
    for (size_t i = 0; i < 50000; i++) {
      centroid->x = (double)rand() / RAND_MAX * 1000;
      centroid->y = (double)rand() / RAND_MAX * 1000;
      emit centroid->newValue();
      std::this_thread::sleep_for(std::chrono::milliseconds((int)round(1000
          / frameRate)));
    }
  });
  distancePlotterWidget.show();
  t1.detach();
  return a.exec();
}
```

5.4.8 Setup

This project is a list of apps that they combine custom GUI for the PAT system.
Requirements

cpp 11 | cmake 3.16 | git 2.0 | Doxygen 1.8.13 | Sphinx 3.0.2 | OS Win

Generality

Import

Import as an external library into your project by copy-paste the following lines in your `config.json`.

```
{
  "name"     : "PATSetup",
  "path"     : "gitlab.dei.unipd.it/PAT/Setup.git",
  "tag"      : "HEAD",
  "available": "YES",
  "getGui"   : "NO"
}
```

Prerequisites

The Prerequisites are the same descibed in the section ”Setup your enviroment Prerequisites” Sec. 4.4.1 with the addition of the following libraries that are auto fetched from the gitlab.dei.unipd.it:

- Controller HEAD

Usage

In this repo you can create your own GUI for the PAT system in the `apps` folder. As reference you can use one of the already defined apps. There is no need to explain additional things because in this repository you want to compose ready-made pieces and not create new ones. To simplify your orientation, you can find a global image of all the repositories in the picture Fig. 5.16.

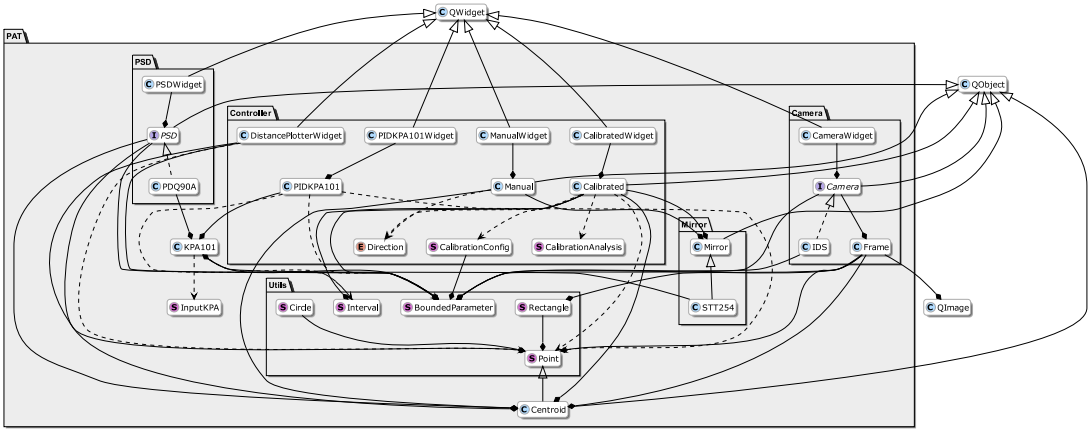


Figure 5.16: Setup UML without attributes.

Chapter 6

PATLIC experiment

In the previous chapters, quantum communication and QKD were introduced, then the importance of the PAT for the QKD and its main components PATC and PATF were explained. After that, it was shown how the code was created to control the PAT. Now it will be shown how to use what was previously realized in a real setup that has been implemented and tested in the laboratory.

This chapter contains the description of the experiment used to test the PAT system. It starts with a description of the setup and then describe how to assemble it.

Figure 6.1: Pictures of the Alice and Bob telescopes used in the PATLIC experiment.

6.1 Introduction

In this experiment, a PAT system was developed between two telescopes in order to realize free-space QKD at 1550 nm with single-mode fiber injection. This experiment is called PATLIC (PAT system for Light Injection on C-band fiber). The objective of this preliminary test in the laboratory was to check that the functionality of the developed software works in a controlled environment.

6.2 Optical setup

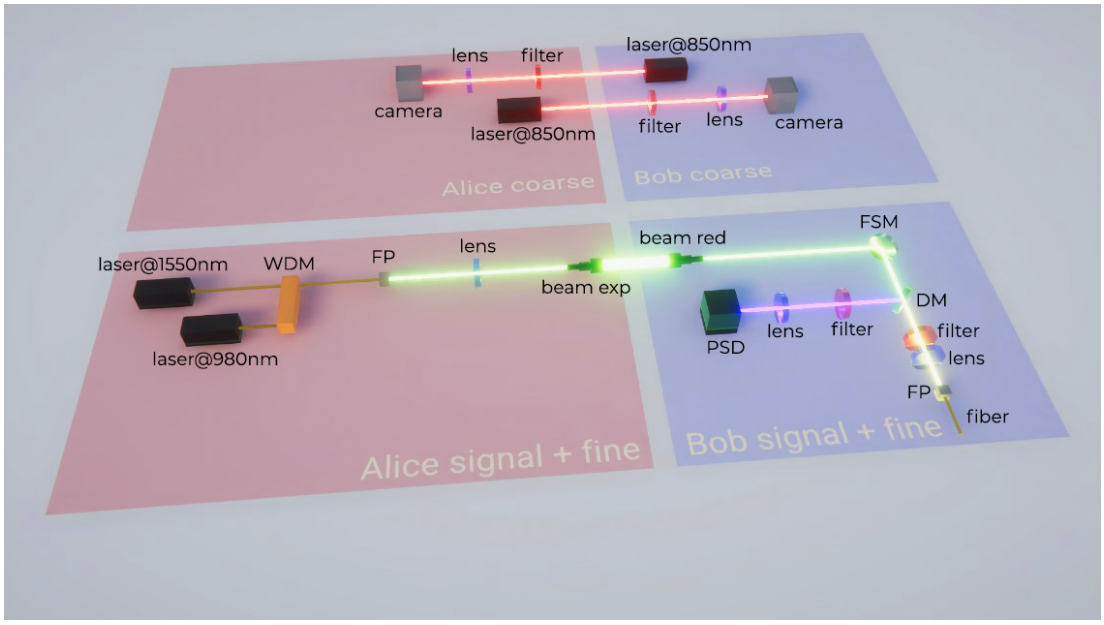


Figure 6.2: PATLIC optical setup. The acronyms are: wavelength division multiplexing (WDM), dichroic mirror (DM), beam expander (beam exp), and beam reducer (beam red).

In this experiment there is a transmitter (Alice) and a receiver (Bob). As shown in Fig. 6.2 they use a PATF's beam (blue) at 980 nm and two PATC's beams (red) at 850 nm, to inject a 1550 nm signal's beam (yellow) named SIG into a single-mode fiber.

From Alice two lasers at 1550 nm and at 980 nm are guided by single-mode fibers into a wavelength division multiplexing that combine them. The fiber's signal is released in free-space by a fiber port. It is collimated by a lens before entering a beam expander that prepare it for the communications link. The signal is then received by Bob where a beam reducer convert it into a smaller collimated beam. The beam reflected by a FSM impacts on a dichroic mirror. The dichroic mirror separates the SIG's beam at 1550 nm from the PATF's beam at 980 nm. The PATF's beam cross a band-pass filter at 980 nm and then cross a lens that focuses it into a PSD. Instead, the SIG's beam cross a band-pass filter at 1550 nm and then cross a lens that focuses it into a single-mode fiber.

From Alice there is a third laser at 850 nm that shines a divergent beam into the free-space. This beam is then focused at Bob's side on a camera. To increase the signal to noise ratio, a filter is put in front of the camera. This setup was also duplicated in the Bob-to-Alice direction.

6.3 Physical setup

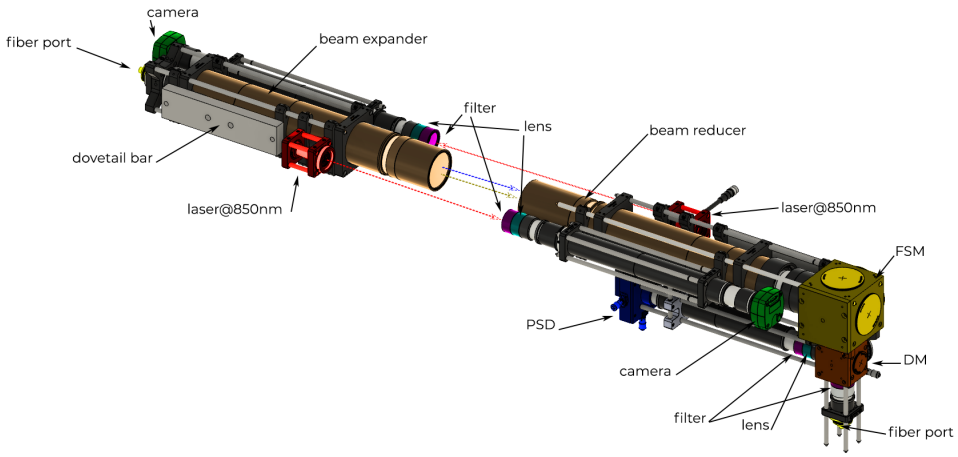


Figure 6.3: PATLIC physical setup with Alice on the left and Bob on the right. In this picture we can see how PATF and PATC are mechanically combined ina single platform.

As shown in the Fig. 6.3 PATF and PATC are united to create a single PAT system. These two structures represent Alice on the left, and Bob on the right. Alice and Bob are then attached to their respective mounts through the dovetail bars. The input point for Alice is the fiber port, where the WDM will be attached through a fiber shown in Fig. 6.2. The fiber port (yellow) contains a lens plane-convex with focal 15.3 mm that collimate the beam exiting from the fiber and traveling for 60 mm. Then the beam enter in the expander (ochre-yellow) composed by three lenses used to increase the beam waist. The beam finds: a achromatic-doublet convex-plane lens with focal 50 mm and then travels for 85 mm; a plane-concave lens with focal -150 mm and then travels for 197 mm; and a achromatic-doublet plane-convex lens with focal 250 mm. The beam then exits from Alice and propagates toward Bob.

When reach Bob the beam is a little bit expanded due to the propagation and so it finds a beam reducer (ochre-yellow) with lenses similar to the beam expander, where the differences are to compensate this expansion. The beam finds: a achromatic-doublet convex-plane lens with focal 300 mm and then travels

for 228 mm; a concave-plane lens with focal -150 mm and then travels for 130 mm, and achromatic-doublet plane-convex with focal 50 mm. The resulting 4mm-diameter beam is then reflected to the DM (orange) by a FSM (yellow) that is placed in the pupil plane of the receiving system. Then, the beam is divided in to paths: PATF, SIG. In the PATF line, the beam finds a plane-convex lens (light-blue) with a focal 300 mm, followed by a band-pass filter (magenta) for 980 nm with a FWHM of 10 nm, and than it is focused on a PSD (dark-blue) after 300 mm of distance. In the SIG line, a band-pass filter (magenta) for 1550 nm with a FWHM of 10 nm, and than it is focused on a fiber port (yellow) with a convex-plane lens with focal 18 mm.

Both PATC are attached and aligned to the main structures. For each, a laser beam (red) is received into a achromatic-doublet convex-concave lens (light-blue) with focal 300 mm through a filter (magenta) for 850 nm with a FWHM of 10 nm, and then it is focused after 300 mm into a 25Hz-camera (green).

6.4 Alignment procedure

A raw manual alignment of the optical components is required before starting the PATSW. The complete alignment procedure for PATHW will be divided into parts. At the end of each part it will be needed an activation of the corresponding PATSW in order to move into the next one.

6.4.1 Coarse alignment

- connect two visible lasers to the two fiber ports. This simplify the initial alignment procedure;
- diverge the two lasers that should be used in the PATC;
- mount both PATC's lasers and ensure that their beams are "parallel" to the beams of the visible lasers mounted in the fiber ports;
- ensure that PATC's lasers are immaged by the PATC's cameras;
- aim Alice on Bob and vice versa. You should try to get the beam-expander exiting beam into the beam-reducer. Make sure the PATC's lasers are "quite" centered in the cameras' pictures. Otherwise correct with laser's tip-tilt. Perfect center is impossible;
- take note of the camera's alignment coordinates in pixels given by PATSW. This is important if something went wrong and you need to start again the procedure. In the PATSW set the alignment-position targets using the alignment coordinates;
- make a calibration with PATSW on both mounts. This is required to use the One shot controller;
- enable the One shot controller.

6.4.2 Fine alignment

- Collimate the Alice's laser @1550 nm used for communication, and the Alice's laser @980 nm used for PAT fine. Connect them on the fiber port removing the visible lasers. Make sure their beams are coaxial;
- check the FSM on Bob. Make sure it is centered in x/y and has 45 degrees angle respect to the optical path;
- check the DM on Bob. Make sure it is centered in x/y and has 45 degrees angle respect to the optical path;
- mount the PATF lens and PSD on Bob. Be sure the PSD is positioned on the lens focus;
- turn on the Alice's lasers @1550 nm, @980 nm and be sure they arrive centered in Bob's fiber port and in the Bob's PSD. Otherwise correct with the placement of FSM, DM;
- make a calibration with PATSW on PSD. This is required to use the Home command;
- make a Home command with PATSW to put the FSM in its tip-tilt neutral position, then correct its placement so the laser is still visible in the PSD. For the correction you can use or the DM placement or the xy-axis controller on the PSD;
- recalibrate the PATSW. This is required to be sure the system is stable;
- activate the PID KPA101 control, paying attention to any inversion of axes due to the positioning of the PSD.

6.4.3 Multi-mode fiber alignment

- Check that the SIG's collimated beam after the DM has a diameter that allows the optimal matching with fiber. Measure the laser power where the fiber port will be placed. This power will be used as reference to understand if the amount of losses in the channel;
- disassemble the Bob fiber-port collimator and put it on a table inside a cage, and attach it on a multi-mode fiber;
- shoot a visible laser through the multi-mode fiber attached to Bob's fiber-port collimator and ensure that the outgoing beam is vertically and horizontally aligned and is collimated;
- reassemble fiber port in the mount with visible laser attached;

- check that the SIG's beam and the fiber-port exiting beam are aligned. Otherwise correct with the precision movements of the DM frame. Be sure that all PAT systems are switched on or the procedure doesn't work;
- mount a power meter at the end of the Bob's fiber swapping the visible laser.
- make sure to have a good fiber coupling by checking that the losses using the power before and after the fiber.

6.4.4 Single-mode fiber alignment

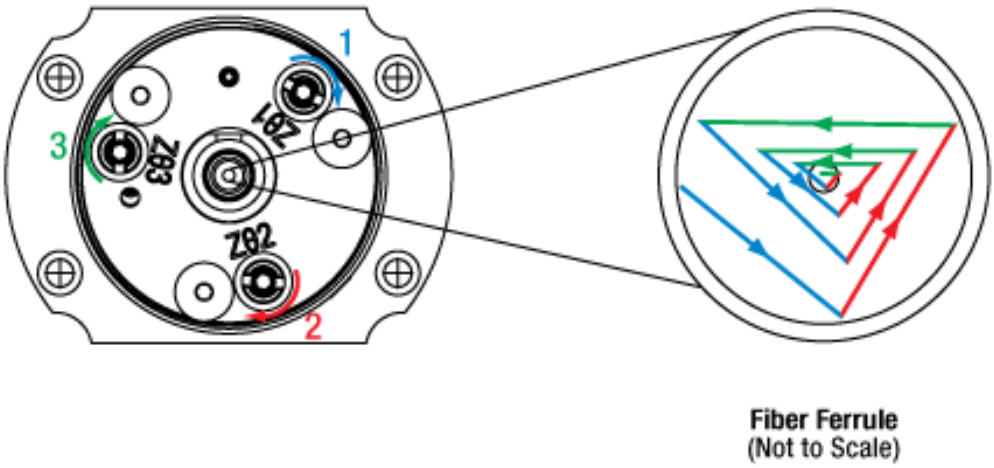


Figure 6.4: This image shows the adjustments sequence for the fiber-port. The image is taken in the Thorlabs website.

- Disassemble the multi-mode fiber and mount the single-mode fiber attached to the Bob's fiber port;
- choose an sequence to make your adjustments, and keep to that sequence as shown in Fig. 6.4;
- turn each adjuster clockwise to maximize the output, then continue to turn slightly beyond local maxima (to 95% of your local maximum). If turning an adjuster clockwise decreases output, skip that adjuster for that round of adjustments and repeat. Once the local maxima values begin to decrease, reverse the direction, and turn each adjuster to maximize the output, and not beyond;
- make sure to have a good fiber coupling by checking that the losses using the power before and after the fiber.

Chapter 7

Conclusions

This work described the realization of a PAT system for QKD experiments. Using the new QKD software structure along with the PAT repository system, the new version of PATSW was designed with the aim of providing a clear and easy-to-use system. The initial objectives were well achieved. Now the system has:

- a new reliable software with a modular structure;
- the possibility to set many different parameters for improved exibility to different scenarios;
- a new board to optimize the center of the KPA101.

Both PATSW and PATHW new features have been developed, implemented and successfully tested on the laboratory. Tests have shown that the system allows excellent performance thanks to its independence from the physical hardware. It easily adapts to 25 Hz frame rate cameras to 200 Hz frame rate cameras. The system has also shown great adaptability with actuators where movements vary from a few milliseconds to seconds. Another example of the excellent hardware independence can be seen in the fiber injection experiment where three instances of the same program were used for coarse alignment of Alice and Bob and for fine alignment of Bob.

For example, the same software will be implemented to try coupling into a single-mode fiber the light coming from a star and collected by a 400 mm telescope, as the one recently placed on the roof of our department (DEI). Furthermore, the system offers a great potential for the implementation of further features and facilities. Future developments could expand the system functionality easily thanks to the help of ready-to-use documentation. The software will be integrated with the algorithms and the adapters necessary for the raw pointing that will be used in links with range distances of the order 1-10 km. These will be nothing more than an extension of the core features that have been developed in this thesis, since they are not requiring additional hardware.

Index

Bibliography

[1] Terhal, B. M. Quantum supremacy, here we come. Nature Physics 14, 530–531 (2018).

[2] Lutchyn, R. M. et al. Majorana zero modes in superconductor-semiconductor heterostructures. Nature Reviews Materials 3, 52–68 (2018).

[3] Johnson, M. W. et al. Quantum annealing with manufactured spins. Nature 473, 194–198 (2011).

[4] Rønnow, T. F. et al. Defining and detecting quantum speedup. Science 345, 420–424 (2014).

[5] Valerio Scarani et al. The security of practical quantum key distribution, Rev. Mod. Phys. 81, 1301, (2009).

[6] Nicolas Gisin et al. Quantum cryptography, (2002).

[7] Ekert, A. K. Quantum Cryptography Based on Bell's Theorem. 413–418 (1992) doi:10.1007/978-1-4615-3386-3_34.

[8] Bennett, C. H. & Brassard, G. Quantum cryptography: Public key distribution and coin tossing. Theoretical Computer Science 560, 7–11 (2014).

[9] Lo, H. K., Curty, M. & Qi, B. Measurement device independent quantum key distribution. Physical Review Letters 108, 1–7 (2012).

[10] Pawłowski, M. & Brunner, N. Semi-device-independent security of one-way quantum key distribution. Physical Review A - Atomic, Molecular, and Optical Physics 84, 1–5 (2011).

[11] Bennett, C. H. Quantum cryptography using any two nonorthogonal states. Physical Review Letters 68, 3121–3124 (1992).

[12] Kato, G. & Tamaki, K. Security of six-state quantum key distribution protocol with threshold detectors. Scientific Reports 6, 1–5 (2016).

[13] Acín, A., Gisin, N. & Masanes, L. From Bell's theorem to secure quantum key distribution. Physical Review Letters 97, 1–4 (2006).

[14] Calderaro, L. et al. Towards quantum communication from global navigation satellite system. Quantum Science and Technology 4, (2019).

[15] Vallone, G. et al. Interference at the Single Photon Level Along Satellite-Ground Channels. Physical Review Letters 116, 1–6 (2016).

[16] Vallone, G. et al. Experimental Satellite Quantum Communications. Physical Review Letters 115, 1–5 (2015).

[17] Ji, W. Strategic Priority Program on Space Science. 34, 505–515 (2015).

[18] Bedington, R. et al. Nanosatellite experiments to enable future space-based QKD missions. EPJ Quantum Technology 3, (2016).

[19] Naughton, D. et al. Design considerations for an optical link supporting intersatellite quantum key distribution. Optical Engineering 58, 1 (2019).

[20] Neumann, S. P. et al. Q 3 Sat: Quantum communications uplink to a 3U CubeSat—feasibility & design. EPJ Quantum Technology 5, 1–24 (2018).

[21] Kimble, H. J. The quantum internet. Nature 453, 1023–1030 (2008).

[22] Benenti Giuliano, Principles Of Quantum Computation, (2004).

[23] David M. Pozar, Microwave Engineering, (1990).

[24] Peter C. et al. Linear Optical Quantum Computing in a Single Spatial Mode, (2014).

[25] P. Grangier et al. Experimental evidence for a photon anticorrelation effect on a beam splitter: A new light on single-photon interferences, (1987).

[26] Xiongfeng Ma et al. Practical decoy state for quantum key distribution, Rev. Mod. Phys. 72, 012326, (2005).

[27] http://www.micro-photon-devices.com/Products/SPAD-by-Wavelength/900nm-1700nm/PDM-IR