

NetXPTO - LinkPlanner

September 11, 2019

Contents

1 Case Studies	5
1.1 BPSK Transmission System	6
1.1.1 Theoretical Analysis	6
1.1.2 Simulation Analysis	7
1.1.3 Comparative Analysis	11
Bibliography	13
1.2 BB84 with Channel Emulator	14
1.2.1 Input parameters	14
1.2.2 Protocol Overview	17
1.2.3 External usage	28
1.2.4 Interface	31
1.2.5 Results	33
1.2.6 Open Issues	35
Bibliography	37
1.3 The Impact of Pulse Shaping in the Security of CV Quantum Communication Systems	38
1.3.1 Classical Frequency and Phase Recovery - State of the art	38
1.3.2 Quantum Frequency and Phase Recovery - State of the art	40
1.3.3 Novelty	40
1.3.4 Implementation issues	40
1.3.5 Error Vector Magnitude	40
1.3.6 Nyquist pulse shaping	41
1.3.7 Simulation Analysis	42
1.3.8 Experimental Setup	51
1.3.9 Open Issues	60
1.3.10 Future work	60
Bibliography	62
1.4 Discrete Variables Quantum Tx/Rx system - Physical Layer	63
1.4.1 Simulation Analysis - Algorithm for polarization compensation	63
1.4.2 Simulation Analysis - TestBench	64

<i>Contents</i>	2
1.4.3 Open Issues	65
Bibliography	66
1.5 Frequency and Phase Recovery in CV-QC Systems	66
1.5.1 Classical Frequency and Phase Recovery - State of the art	66
1.5.2 Quantum Frequency and Phase Recovery - State of the art	69
1.5.3 Open issues	70
1.5.4 Novelty	70
1.5.5 Novel Frequency Mismatch Compensation Technique	70
1.5.6 Implementation issues	72
1.5.7 Error Vector Magnitude	77
1.5.8 Comparative analysis of the frequency ranging techniques	78
1.5.9 Simulation Analysis	80
1.5.10 Simulation Results	81
1.5.11 New study	82
1.5.12 Future work	86
Bibliography	92
2 Library	93
2.1 ADC	94
2.2 Add	97
2.3 Arithmetic Encoder	98
2.4 Arithmetic Decoder	100
2.5 Alice QKD	102
2.6 Alice QKD	104
2.7 AliceQuantumTx	110
2.8 Ascii Source	114
2.9 Ascii To Binary	116
2.10 Balanced Beam Splitter	118
2.11 Bit Error Rate	119
Bibliography	125
2.12 Binary Source	126
2.13 Binary To Ascii	130
2.14 Bob QKD	132
2.15 BobQuantumRx	133
2.16 Bit Decider	136
2.17 Clock	137
2.18 Clock_20171219	139
2.19 Complex To Real	142
2.20 Coupler 2 by 2	144
2.21 Carrier Phase Compensation	145
2.22 Decision Circuit	149
2.23 Decoder	151
2.24 Discrete To Continuous Time	153

2.25	DownSampling	155
2.26	DSP	157
2.27	EDFA	159
2.28	Electrical Signal Generator	162
2.28.1	ContinuousWave	162
2.29	Entropy Estimator	164
2.30	Entropy Estimator	165
2.31	Fork	167
2.32	Gaussian Source	168
2.33	Hamming Decoder	170
2.34	Hamming Encoder	171
2.35	MQAM Receiver	172
2.36	Huffman Decoder	176
2.37	Huffman Encoder	177
2.38	Ideal Amplifier	178
2.39	IQ Modulator	180
	Bibliography	185
2.40	IIR Filter	185
	Bibliography	186
2.41	IP Tunnel MS Windows	186
2.42	Local Oscillator	190
2.43	Local Oscillator	192
2.44	Mutual Information Estimator	195
	Bibliography	198
2.45	MQAM Mapper	199
2.46	M-QAM Receiver	202
2.47	MQAM Transmitter	209
2.48	Netxpto	219
2.48.1	Version 20180118	219
2.48.2	Version 20180418	219
2.49	Alice QKD	220
2.50	Polarizer	222
2.51	Probability Estimator	223
2.52	Quantum Channel Emulator DV	226
2.53	Bob QKD	230
2.54	Eve QKD	231
2.55	Rotator Linear Polarizer	232
2.56	MS Windows IP Tunnel	234
2.57	Mutual Information Estimator	236
	Bibliography	239
2.58	Optical Switch	240
2.59	Optical Hybrid	241

2.60	Photodiode pair	243
2.61	Photoelectron Generator	246
	Bibliography	251
2.62	Power Spectral Density Estimator	252
	Bibliography	258
2.63	Pulse Shaper	258
2.64	Quantizer	260
2.65	Resample	263
2.66	SNR Estimator	266
	Bibliography	271
2.67	Sampler	271
2.68	SNR of the Photoelectron Generator	273
	Bibliography	278
2.69	Sink	279
2.70	SNR Estimator	281
	Bibliography	286
2.71	Single Photon Receiver	286
2.72	SOP Modulator	290
	Bibliography	294
2.73	Source Code Efficiency	295
2.74	White Noise	296
2.75	Ideal Amplifier	299
2.76	Phase Mismatch Compensation	301
	Bibliography	304
2.77	Frequency Mismatch Compensation	305
	Bibliography	308
2.78	Cloner	309
	Bibliography	310
2.79	Error Vector Magnitude	310
	Bibliography	312
2.80	Load Ascii	312
2.81	Load Signal	314
2.82	Matched Filter	316
2.83	Timing Deskew	317
2.84	DC component removal	319
2.85	Orthonormalization	321

Chapter 1

Case Studies

1.1 BPSK Transmission System

Student Name	:	André Mourato (2018/01/28 - 2018/02/27) Daniel Pereira (2017/09/01 - 2017/11/16)
Goal	:	Estimate the BER in a Binary Phase Shift Keying optical transmission system with additive white Gaussian noise. Comparison with theoretical results.
Directory	:	sdf/bpsk_system

Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying (PSK), in which binary information is encoded into a two state constellation with the states being separated by a phase shift of π (see Figure 1.1).

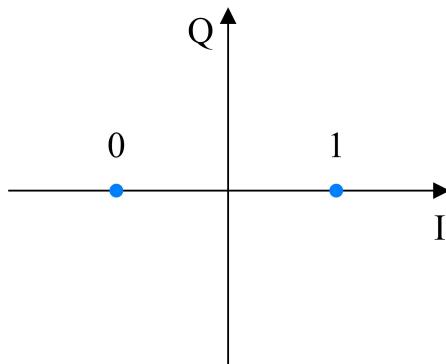


Figure 1.1: BPSK symbol constellation.

White noise is a random signal with equal intensity at all frequencies, having a constant power spectral density. White noise is said to be Gaussian (WGN) if its samples follow a normal distribution with zero mean and a certain variance σ^2 . For WGN its spectral density equals its variance. For the purpose of this work, additive WGN is used to model thermal noise at the receivers.

The purpose of this system is to simulate BPSK transmission in back-to-back configuration with additive WGN at the receiver and to perform an accurate estimation of the BER and validate the estimation using theoretical values.

1.1.1 Theoretical Analysis

The output of the system with added gaussian noise follows a normal distribution, whose first probabilistic moment can be readily obtained by knowledge of the optical power of the received signal and local oscillator,

$$m_i = 2\sqrt{P_L P_S} G_{ele} \cos(\Delta\theta_i), \quad (1.1)$$

where P_L and P_S are the optical powers, in watts, of the local oscillator and signal, respectively, G_{ele} is the gain of the trans-impedance amplifier in the coherent receiver and

$\Delta\theta_i$ is the phase difference between the local oscillator and the signal, for BPSK this takes the values π and 0, in which case (1.1) can be reduced to,

$$m_i = (-1)^{i+1} 2 \sqrt{P_L P_S} G_{ele}, \quad i = 0, 1. \quad (1.2)$$

The second moment is directly chosen by inputting the spectral density of the noise σ^2 , and thus is known *a priori*.

Both probabilist moments being known, the probability distribution of measurement results is given by a simple normal distribution,

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m_0)^2}{2\sigma^2}}. \quad (1.3)$$

The BER is calculated in the following manner,

$$BER = \frac{1}{2} \int_0^{+\infty} f(x|\Delta\theta = \pi) dx + \frac{1}{2} \int_{-\infty}^0 f(x|\Delta\theta = 0) dx, \quad (1.4)$$

given the symmetry of the system, this can be simplified to,

$$BER = \int_0^{+\infty} f(x|\Delta\theta = \pi) dx = \frac{1}{2} \operatorname{erfc} \left(\frac{-m_0}{\sqrt{2}\sigma} \right) \quad (1.5)$$

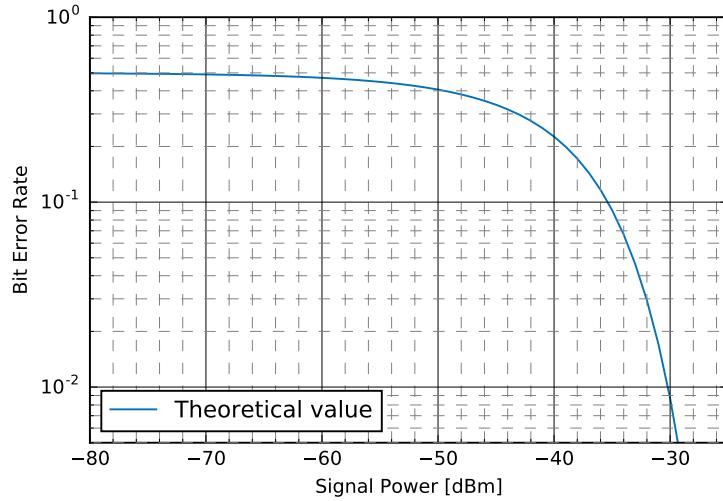


Figure 1.2: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

1.1.2 Simulation Analysis

A diagram of the system being simulated is presented in the Figure 1.3. A random binary sequence is generated and encoded in an optical signal using BPSK modulation. The decoding of the optical signal is accomplished by an homodyne receiver, which combines the

signal with a local oscillator. The received binary signal is compared with the transmitted binary signal in order to estimate the Bit Error Rate (BER). The simulation is repeated for multiple signal power levels. Each corresponding BER is recorded and plotted against the expectation value.

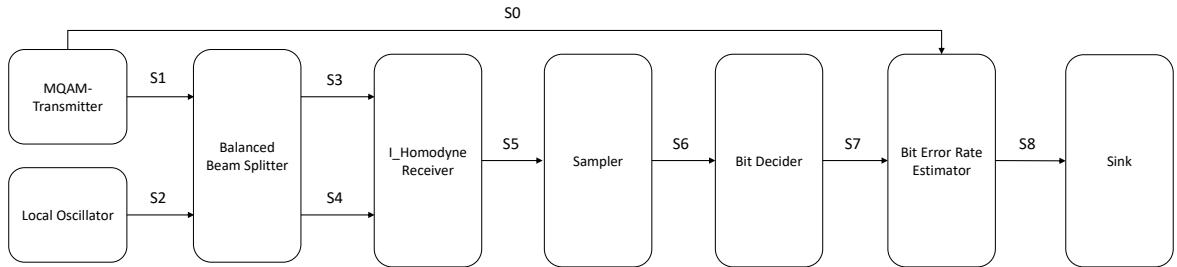


Figure 1.3: Overview of the BPSK system being simulated.

Required files

Header Files		
File	Comments	Status
add_20171116.h		✓
balanced_beam_splitter_20180124.h		✓
binary_source_20180118.h		✓
bit_decider_20170818.h		✓
bit_error_rate_20171810.h		✓
discrete_to_continuous_time_20180118.h		✓
i_homodyne_receiver_20180124.h		✓
ideal_amplifier_20180118.h		✓
iq_modulator_20180130.h		✓
local_oscillator_20180130.h		✓
m_qam_mapper_20180118.h		✓
m_qam_transmitter_20180118.h		✓
netxpto_20180418.h		✓
photodiode_old_20180118.h		✓
pulse_shaper_20180118.h		✓
sampler_20171116.h		✓
sink_20180118.h		✓
super_block_interface_20180118.h		✓
ti_amplifier_20180102.h		✓
white_noise_20180118.h		✓

Source Files		
File	Comments	Status
add_20171116.cpp		✓
balanced_beam_splitter_20180124.cpp		✓
binary_source_20180118.cpp		✓
bit_decider_20170818.cpp		✓
bit_error_rate_20171810.cpp		✓
discrete_to_continuous_time_20180118.cpp		✓
i_homodyne_receiver_20180124.cpp		✓
ideal_amplifier_20180118.cpp		✓
iq_modulator_20180130.cpp		✓
local_oscillator_20180130.cpp		✓
m_qam_mapper_20180118.cpp		✓
m_qam_transmitter_20180118.cpp		✓
netxpto_20180418.cpp		✓
photodiode_old_20180118.cpp		✓
pulse_shaper_20180118.cpp		✓
sampler_20171116.cpp		✓
sink_20180118.cpp		✓
super_block_interface_20180118.cpp		✓
ti_amplifier_20180102.cpp		✓
white_noise_20180118.cpp		✓

System Input Parameters

This system takes into account the following input parameters:

System Input Parameters		
Parameter	Default Value	Comments
numberOfBitsReceived	-1	
numberOfBitsGenerated	1000	
samplesPerSymbol	16	
pLength	5	
bitPeriod	20×10^{-12}	
rollOffFactor	0.3	
signalOutputPower_dBm	-20	
localOscillatorPower_dBm	0	
localOscillatorPhase	0	
iqAmplitudesValues	$\{ \{-1, 0\}, \{1, 0\} \}$	
transferMatrix	$\{ \{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\} \}$	
responsivity	1	
amplification	10^6	
electricalNoiseAmplitude	$5 \times 10^{-4}\sqrt{2}$	
samplesToSkip	$8 \times samplesPerSymbol$	
bufferLength	20	
shotNoise	false	

Outputs

This system outputs the following objects:

System Output Signals	
Signal	Associated File
Initial Binary String (S_0)	S0.sgn
Optical Signal with coded Binary String (S_1)	S1.sgn
Local Oscillator Optical Signal (S_2)	S2.sgn
Beam Splitter Outputs (S_3, S_4)	S3.sgn & S4.sgn
Homodyne Receiver Electrical Output (S_5)	S5.sgn
Sampler Output (S_6)	S6.sgn
Decoded Binary String (S_7)	S7.sgn
BER Result String (S_8)	S8.sgn
Report	Associated File
Bit Error Rate Report	BER.txt

Bit Error Rate - Simulation Results

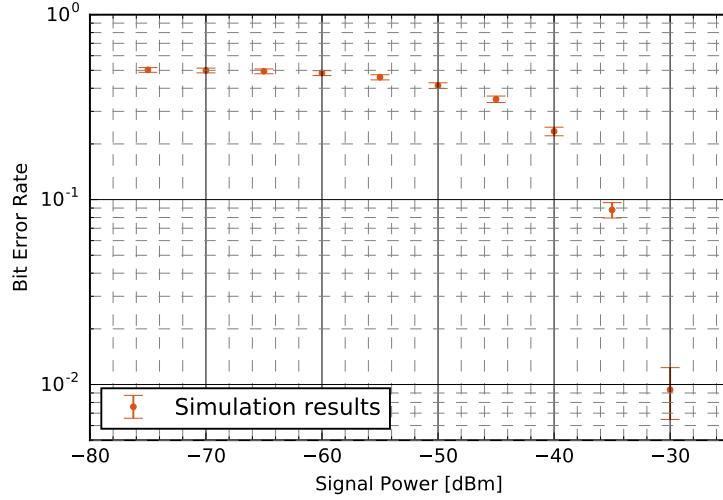


Figure 1.4: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

1.1.3 Comparative Analysis

The following results show the dependence of the error rate with the signal power assuming a constant Local Oscillator power of 0 dBm, the signal power was evaluated at levels between -70 and -25 dBm, in steps of 5 dBm between each. The simulation results are presented in orange with the computed lower and upper bounds, while the expected value, obtained from (1.5), is presented as a full blue line. A close agreement is observed between the simulation results and the expected value. The noise spectral density was set at $5 \times 10^{-4}\sqrt{2} V^2$ [1].

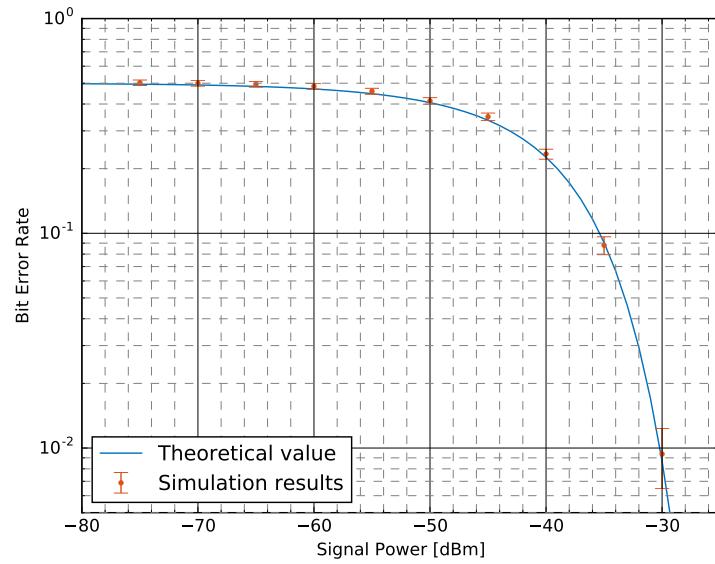


Figure 1.5: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 2.78

References

- [1] Thorlabs. *Thorlabs Balance Amplified Photodetectors: PDB4xx Series Operation Manual.* 2014.

1.2 BB84 with Channel Emulator

Students Name	:	Andoni Santos (10/04/2019 - ...)
Starting Date	:	April 10, 2019
Goal	:	BB84 implementation with discrete variables and simplified transmission.

BB84 is a quantum key distribution protocol which involves three parties: Alice, Bob and Eve [1]. The objective of the protocol is for Alice and Bob to generate private key that only they know, while assuming a third party, Eve, is trying to eavesdrop on their communications. For this, they use both a quantum channel and a classical channel. The protocol allows Alice and Bob to share information between them, creating the shared keys, while minimizing the amount of information leaked to Eve.

The simulation "BB84 with Channel Emulator" implements the top layers of the BB84 protocol, while emulating the observable physical aspects of the communication system in a simplified fashion. Its purpose is to allow implementing, testing, analyzing and improving the software layers of the protocol. The emulation of the quantum physical transmission channel is achieved by using a probabilistic model based on the physical characteristics of a real world system.

1.2.1 Input parameters

Variable	Type	Values	Default	Description
<i>numberOfBits</i>	int	>1	1000000	Total number of bits created on the bit sources
<i>qber</i>	t_real	[0,1]	0.03	Transmission error rate. Also used in error correction and privacy amplification
<i>probabilityOfDoubleClick</i>	t_real	[0,1]	0.1	Rate of double clicks
<i>doubleClickNumber</i>	t_integer	>1	2	Number to signal the double click events
<i>noClickNumber</i>	t_integer	>1	3	Number to signal the events without clicks

<i>numPasses</i>	t_integer	any	4	Number of iterations in the error correction process
<i>qValue</i>	t_real	any	0.73	Parameter for choosing the initial size of error correction segments. Size is equal to <i>qValue</i> divided by the <i>qber</i> .
<i>minimumNumberOfPartitions</i>	t_integer	any	10	Minimum number of segments to use for error correction
<i>maximumSyndromeSize</i>	t_integer	any	8192	Maximum number of samples to be processed at once during error correction
<i>minimumSyndromeSize</i>	t_integer	any	8192	Minimum number of samples required to start error correction.
<i>bypassHash</i>	bool	[true, false]	false	Control whether to hash the key or not, to better analyze error correction results.
<i>bitRate</i>	t_real	any	1000	Bits per second.
<i>systematicSecurityParameter</i>	t_integer	any	1	Number of extra bits to discard after hashing. Discarded after each round.
<i>doublePartitionSize</i>	bool	[true, false]	true	Controls whether the Cascade error correction segments double in each pass after the first.

<i>parameterEstimationNBits</i>	t_integer	> 0	1000	Number of bits to use (and subsequently discard) in parameter estimation.
<i>parameterEstimationNBitsToStart</i>	t_integer	> parameter Estimation NBits	100000	Minimum required number of bits to start parameter estimation.
<i>saveKeys</i>	bool	[true, false]	true	Controls whether the keys are saved to disk or not. Useful for speeding up debugging and testing.
<i>buffSize</i>	t_integer	> 0	4096	Chooses the general buffer size. Bigger buffers may improve simulation speed.
<i>detectorEfficiency</i>	double	[0,1]	0.25	Efficiency of the detectors receiving the photons.
<i>fiberAttenuation_dB</i>	double	≥ 0	0.2	Fiber optical attenuation in dB km.
<i>fiberLength</i>	double	≥ 0	50	Length of the fiber in km.
<i>insertionLosses_dB</i>	double	≥ 0	2	Attenuation by insertion losses.
<i>numberOfPhotonsPerInputPulse</i>	double	≥ 0	0.1	Statistical average of the number of photons per input pulse.
<i>ipAddress</i>	string		"running locally"	IP address where IPTunnel is going to look for connections.
<i>deadtime</i>	double	≥ 0	0.00001	Deadtime of the photon detectors.

<i>printRecv</i>	bool	[true, false]	true	Controls the information shown on the console. If true, it shows the receiver information (Bob). If false, shows the transmitter information (Alice).
<i>qberThreshold</i>	t_real	[0,1]	0.15	Security QBER threshold. If the QBER upper bound is higher than this value, the whole interval is discarded and a warning is shown on the interface.
<i>confidenceInterval</i>	double	[0,1]	0.95	Confidence interval for calculating upper and lower QBER bounds during parameter estimation.

1.2.2 Protocol Overview

The structure of the implemented system is shown in Figure 1.6. Alice and Bob are connected through two different channels, and use them to communicate with each other: a public, authenticated channel represented by the *Messages* signals, and a private, unidirectional quantum channel, represented through the *quantum_channel_emulator* block. The general flow of the system in the simulation is as follows:

The information flowing between Alice and Bob is transmitted through the *Messages* signals and the *Quantum Channel Emulator* block. Data transmitted through the messages signals is publicly available, with its integrity ensured. On the other hand, the information transmitted through the quantum channel is not public, although an eavesdropper could still try to access it. In addition, the integrity of the information in this channel is not guaranteed, as we shall see further ahead.

Two random bitstreams, *AliceData_In* and *AliceBasis_In* are generated and passed to Alice, to serve as the sources for the transmitted key and the basis used to encode the key. Alice sends a copy of those signals to the *quantumChannelEmulator* block, where they will be used to transmit the key to Bob. In addition, Alice saves the information she sent (key and basis) to prepare to do basis reconciliation with Bob. Another random independent bitstream

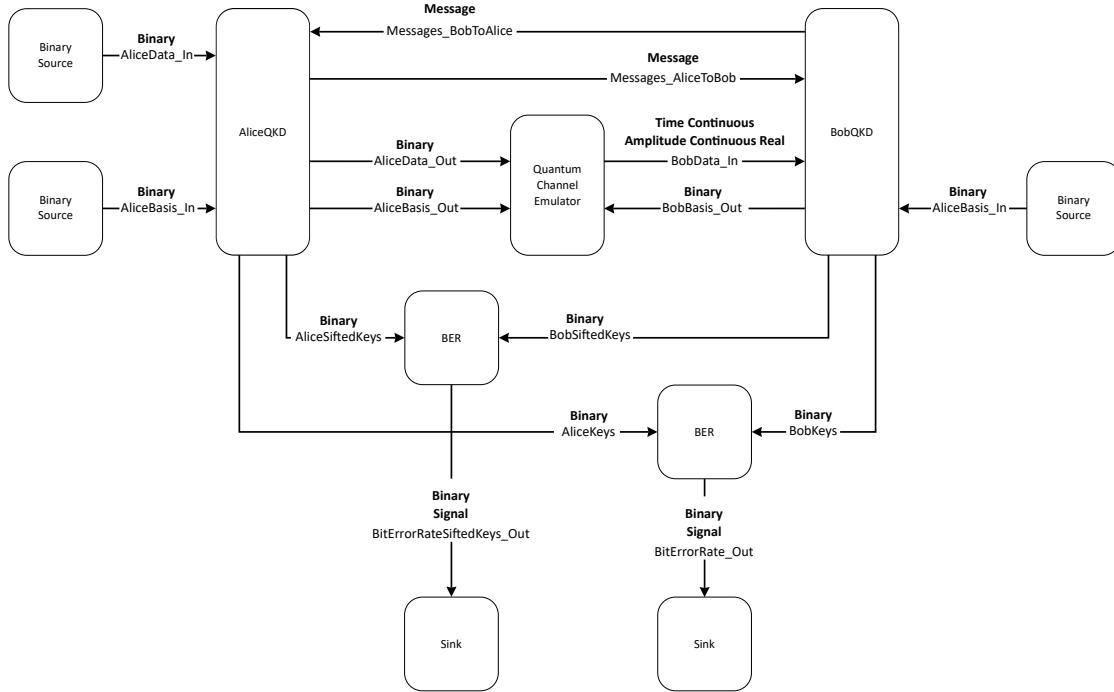


Figure 1.6: Implemented QKD Model. Alice and Bob are connected by a public classical channel, represented by the message signals, and a private quantum channel.

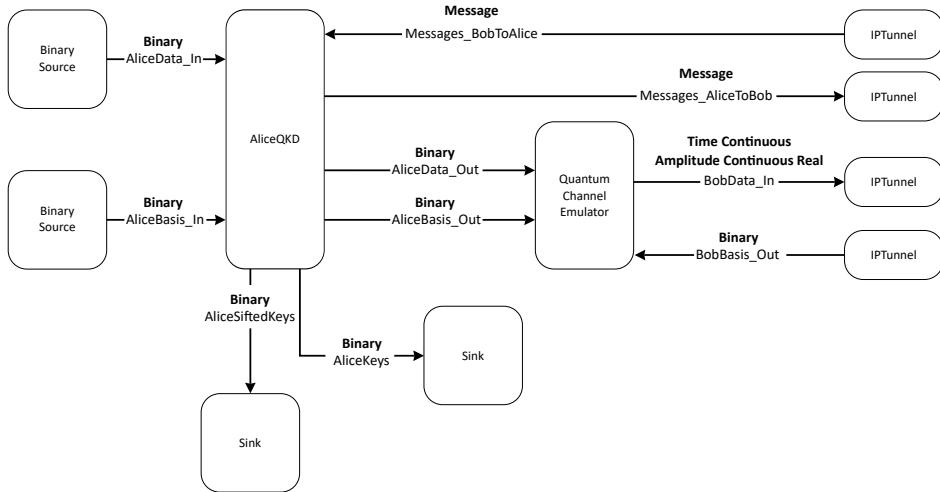


Figure 1.7: Model of the transmitter, using IPTunnel for communicating between multiple computers.

is generated and passed to Bob, to provide the measurement basis. Bob sends this signal to the *quantumChannelEmulator* block and also saves the information to prepare for basis reconciliation. The *quantumChannelEmulator* block now reads all the input and it processes the information before sending it to Bob. We assume a sort of black box model, where we can define the channel's behavior based on certain key parameters. This block is responsible

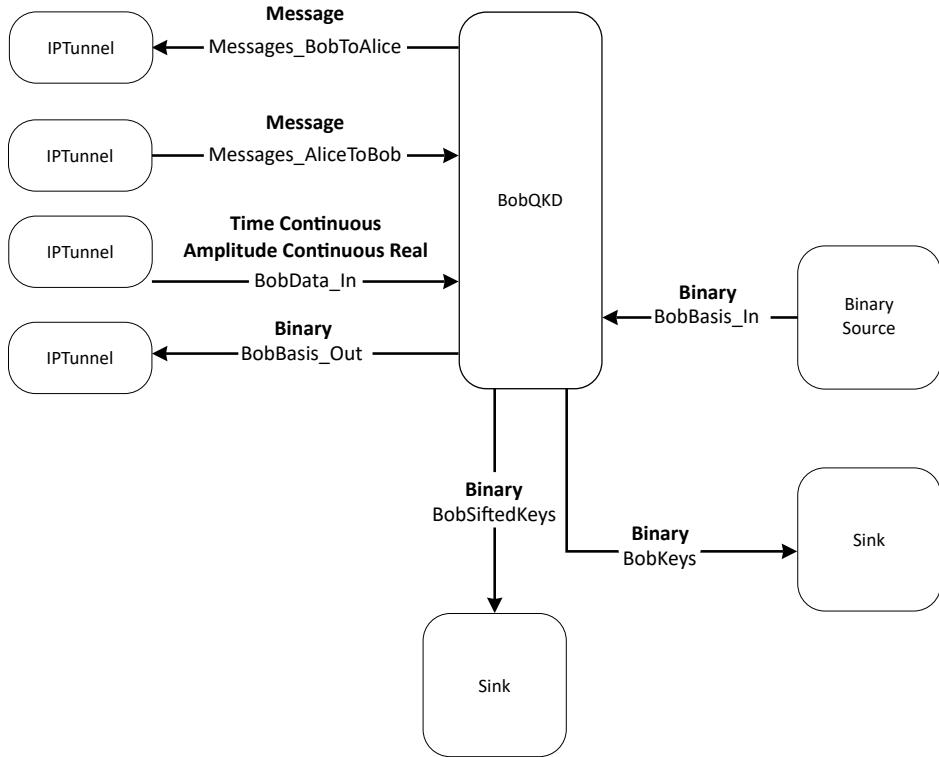


Figure 1.8: Model of the receiver, using IPTunnel for communicating between multiple computers.

for simulating the quantum channel behavior. Currently it assumes a simple probabilistic transmission model, taking into account some of the physical parameters which influence a quantum channel. Specifically, the probability of a qubit being attenuated is calculated based on the fiber attenuation and length, insertion losses, detector efficiency, deadtime and average number of photons per impulse. However, do note that while the model provides a decent approximation, it makes some assumptions which may deviate the results slightly from those obtained in a real system.

Besides the qubit being attenuated, there are several possible outcomes that can be output by the channel emulator. The probability of each of these happening is independent from the probability of attenuation, and is controlled by a set of user defined probabilities.

If a given qubit has suffered attenuation, the *quantumChannelEmulator* sends a *noClickNumber* to Bob. Otherwise, depending on a set of generated random numbers, it sends the following data:

- If a detector has clicked less than *deadtime* seconds ago, a *noClickNumber* is sent;
- A probability for the detectors not clicking is calculated based on the average number of photons arriving at the detectors (using Poisson statistics). We compare the probability to a random number and, if lower than required, a *noClickNumber* is sent;
- If the next random number is lower than *probabilityOfBothClick*, the channel sends a

predefined *doubleClickNumber*, which signals that both the detectors in the physical system clicked;

- Else, if the next random number is lower than *probabilityOfError*, the channel sends the wrong bit;
- If none of these happen, the channel compares Alice and Bob's basis. If they are different, the channel checks another random number, and it has a 50% chance of sending the right bit, and 50% chance of sending the wrong one.

Bob then receives the data from the quantum channel and starts the basis reconciliation process. Alice and Bob then do basis reconciliation, parameter estimation, error correction and privacy amplification, in this order, communicating through the *Message* signals. This processes will be explored in more detail further ahead. For now, in order to better understand what is happening, we should take a look into the *AliceQKD* and *BobQKD* superblocks.

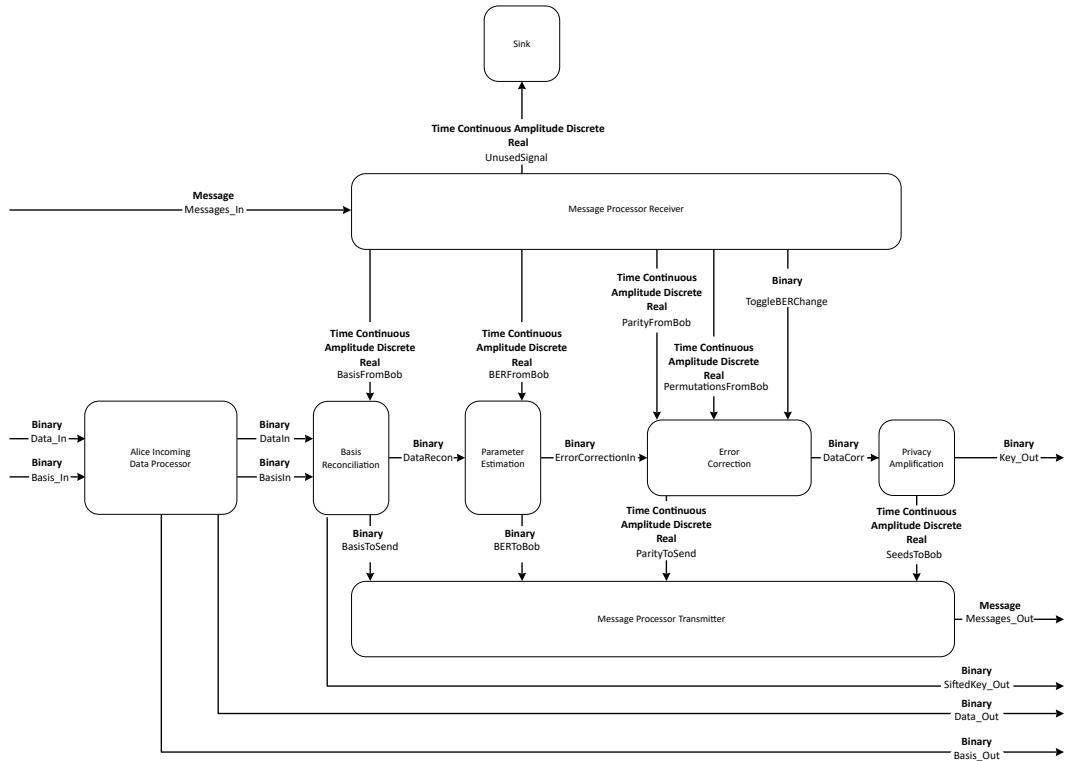


Figure 1.9: AliceQKD superblock.

We can see that they have dedicated blocks to receiving and transmitting messages, and also dedicated blocks for basis reconciliation, parameter estimation, error correction and privacy amplification. Each signal that goes from one of these blocks to one of the message processors carries a certain type of information. The message processors identify the type of information from the signal that they arrive in, and send the data in a message with the appropriate data type. Only one message of each type can be sent in a single run of the block.

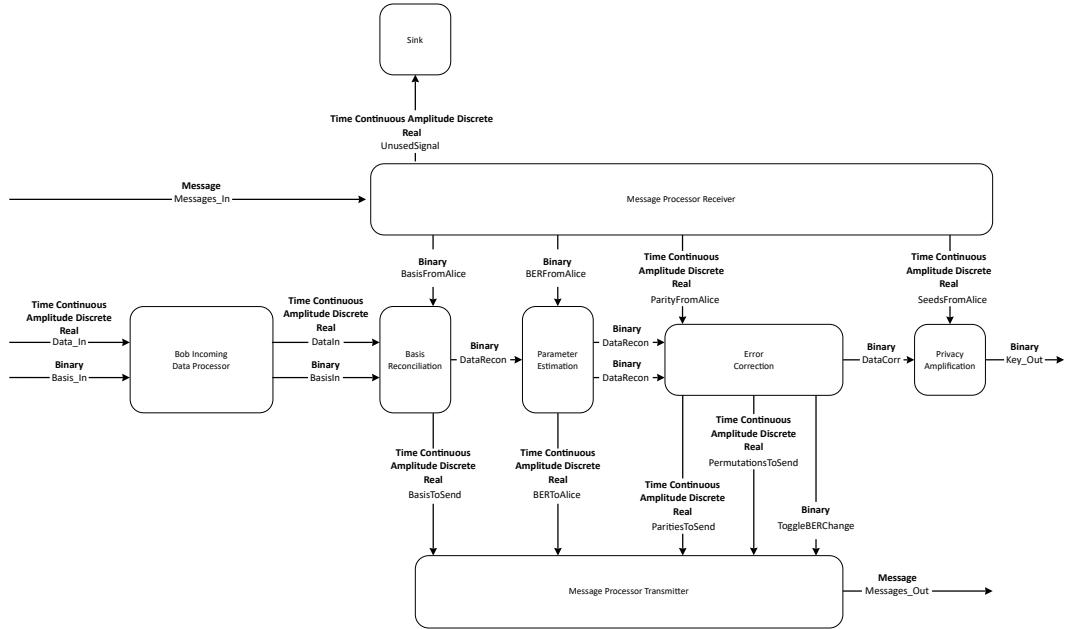


Figure 1.10: BobQKD superblock.

This way, all three processes (basis reconciliation, parameter estimation, error correction and privacy amplification) can be happening simultaneously for different sets of data.

The purpose of each of this processes is as follows:

- **Basis reconciliation:** Removing non-useful information from the obtained keys. The data to be filtered out includes the events when there was a *noClick* or *doubleClick* signal, or when the basis used by Bob was different from the basis used by Alice.
- **Parameter estimation:** Estimate the overall error rate between Alice and Bob's keys by sharing a small amount of randomly selected bits.
- **Error correction:** Reconciling any differences existing between Alice and Bob's keys, making sure that they end up being equal.
- **Privacy amplification:** Transform the keys to minimize the potential information that an attacker may have.

After all these steps, Alice and Bob send the final keys to a *BitErrorRate* block. Here, the BER should be zero (or much smaller than the value measured with the sifted key, if the error correction process failed to remove all errors), and the key size will also be significantly reduced. The final keys may instead be sent to a *Sink*, which is what happens in the versions connected by *IPTunnel*.

Now that we have explained the process by which Alice and Bob communicate using the classical channel, we can delve into the details of basis reconciliation, error correction and privacy amplification.

Basis Reconciliation

During basis reconciliation, Bob sends Alice information about each of the measurements he has made. If the measurement returned a valid value, he sends the basis used to make the measurement. If the measurement returned a *noClick* or *doubleClick*, he sends the corresponding number instead, so that Alice can know about it.

Alice then compares that information with the basis she used to transmit the information. If the basis used is the same, Alice sends a "1" to Bob and sends the current bit to the next stage, Parameter Estimation; otherwise, Alice sends a "0" to Bob and discards the current bit. In addition, for the cases where the measurement returned a double click or no clicks, Alice can readily discard these bits and sends a "0" as confirmation that she discarded them. Bob can then analyze the response from Alice and discard the bits for which he didn't receive a "1", while sending the others to the stage of error correction.

During this stage, the basis used are made public, so Eve may have access to them. This is accounted for and even expected. However, no direct information about the key is shared. The shared information only acts as confirmation for the qubits that Eve has already intercepted, and those are dealt with during the privacy amplification stage.

It's worth noting that this process, unlike the ones that follow it, does not benefit from processing a higher number of bits at once. Processing a higher number of bits reduces the number of messages Alice and Bob require, but does not have any other effect on the key.

Parameter estimation

Even after removing the bits that were measured with the wrong basis, there will still be errors in Bob's key. These may be due either to transmission errors or to interference by an eavesdropper. However, at this point the origin does not matter, and it is safer to assume that all errors are due to Eve.

An estimate of the QBER is required for two purposes: to choose appropriate parameters for error correction, maximizing error correction efficiency for the estimated error rate; and estimating the information that Eve has obtained, which is related to the error rate.

The process for obtaining the error rate is straightforward: Alice and Bob publicly share a given number of bits, and count how many of them are different. After they do this, they discard all the shared bits, so that Eve does not gain any knowledge about the key.

After basis reconciliation, Alice and Bob's parameter estimation blocks store the sifted keys into a vector, while they wait for the number of available bits to be enough to ensure proper performance. There are several reasons to do this, instead of working normally with the values contained in the input buffer:

1. **Better accuracy.** The parameter estimation procedure needs to use a percentage of the bits in order to estimate the BER. Doing this estimation with a larger number of bits usually creates more accurate results, so we need to accumulate larger amounts of data in order to randomly select bits from it.
2. **It keeps the data flowing.** While these processes usually take some time, and require

waiting until a certain amount of data is available, the data continues to be transmitted and received. If the data is not removed from the buffer, the maximum amount of data available at any time for processing is severely limited.

3. **Reducing communication between Alice and Bob.** As previously mentioned, this method requires Alice and Bob to communicate in order to find the errors. If the communication is done after accumulating 100000 bits, with each transmitted message containing 1000 bits, we will be using one percent of the sifted key for estimating the QBER. If we use 500 bits after accumulating 50000, we will still be using one percent of the sifted key, but will have to communicate twice as often. This is undesirable, as the minimizing the number of messages improves performance.

There is an obvious trade-off that needs to be taken into account: increasing the number of shared bits directly decreases the amount of bits; however, by using a higher amount of bits, a better estimate of the error rate can be achieved, which is translated into better privacy (as the amount of errors is indicative of Eve's activities) and possibly into a better error correction efficiency, particularly if we use the upper bound of this estimate to choose the partition sizes. Therefore, increasing the number of bits wasted at this stage may diminish the number of wasted bits on the long run. We should choose the number of bits to be shared taking into account the total number of bits they are extracted from, how often we will do this estimate and even the total size of the key that is processed during error correction.

Error Correction

Alice and Bob need to make sure they have the same key, and they obviously cannot compare them directly. In these circumstances error correcting codes might not be practical to use, so other methods need to be used.

Cascade is one of the most used algorithms for correcting the errors in these cases. The current implementation will now be described in detail:

After parameter estimation, Alice and Bob's error correction blocks store the sifted keys into a vector, while they wait for the number of available bits to be enough to ensure proper performance. There are three reasons to do this, instead of working normally with the values contained in the input buffer:

1. **It keeps the data flowing.** One of the problems with using Cascade for error correction is the amount of classical messages transmitted between Bob and Alice. If the error correction block only emptied its input buffer when the process ended, it is likely that some of the other blocks would have to pause due to filled buffers, as they wait for Cascade to stop;
2. **Reducing communication between Alice and Bob.** As previously mentioned, this method requires Alice and Bob to communicate in order to find the errors. For a single run of the algorithm, the same average amount of channel uses is required whether the key to correct contains 500 or 20000 bits. However, if the key being corrected

contains 500 bits, it would take forty runs of the algorithm to correct a key with 20000 bits, which implies forty times more channel uses.

3. **Effectiveness of error correction.** This error correction process greatly reduces the probability of an error between Alice and Bob's bit strings. It is not guaranteed to reduce this probability to zero, but usually it becomes a value low enough so that there will not be an error in a bit string of a given size. The original proposed algorithm used strings of 10000 bits and claimed that these would be kept error free within the given parameters [2]. It has been shown that increasing the length of the bit strings has a positive impact on the probability of failing to remove all errors [3]. The achieved error rate becomes lower when more bits are processed at once. For instance, in our simulations, using the same set of parameters, the measured error rate after the cascade dropped from 1.0×10^{-5} when using sets of 512 bits to 6.2×10^{-7} when using sets of 2048 bits. A possibly more relevant metric than the bit error rate is the failure error rate [3], which describes the probability of at least one error not being corrected. As we intend to have completely equal keys in both sides, this is indeed a better statistic to study the probability of not achieving the key. It also presents a different aspect, as the increased key length improves the error correction but makes the probability of at least one error more likely. Nevertheless, this value is a directly related to the bit error rate, so the BER is still an accurate measurement of the error correction effectiveness.

For this reasons, the values are stored so that the input buffer has space available for incoming sifted keys, and once enough values have been saved, the cascade protocol starts. The minimum number of values is defined in a variable of the *ErrorCorrection* block, *minimumSyndromeSize*. In addition, a maximum possible value is established, defined by *maximumSyndromeSize*. The actual number of used values is the maximum multiple of the chosen partitioning size which is below the *maximumSyndromeSize*. Previously, *maximumSyndromeSize* was necessary due to constraints in the way that messages worked. Right now it is not strictly required, and can be set to nearly any value above *minimumSyndromeSize*, if more freedom is desired to allow the algorithm to process as many values as possible at any given moment.

Once the process starts, it works as follows. For the extent of this explanation, assume that any communication between Alice and Bob is done by Messages:

1. Bob retrieves a given amount bits from the stored data, where the number is largest multiple of the *partitionSize* which does not exceed the amount of available data, or *maximumSyndromeSize*. Having that string, Bob will partition it according to *partitionSize* and calculate the parity of each partition. He then sends these parities to Alice.
2. Alice retrieves the parities sent by Bob. As she also knows the *partitionSize*, she can also retrieve the same amount bits from the data she stored. She then partitions it, calculates the parities of her partitions and compares her results to what Bob sent. For each of the parities, if they are equal, she returns a "2", to indicate that the parities

are equal and no more comparison is required. Otherwise, she returns a "0". She also stores her previous parities for the next step.

3. When Bob receives the response, he processes it in the following fashion: if the returned value for a given partition is "2", he leaves it alone and also sends back a "2". Otherwise, Bob "updates" the partition he uses to calculate the parities, calculates the new parity and sends the parity to Alice. By "updating", we mean that Bob changes the number of bits he is currently considering. We shall explain this in detail further ahead.
4. When Alice receives the new response from Bob she does two things: first, she looks over her previous list of parities to also "update" her partitions, so that they match Bob's; secondly, she calculates her parities and sends her response to Bob: the "2" values remain intact, she sends "1" if the parities are equal and "0" if they are different;
5. Bob receives the response, "updates" his partitions accordingly and sends his new parities. This process is repeated until an updated partition has only one bit. When this happens, that bit is flipped and Bob sends a "5" instead of a parity (for now the "5" works as sending a "2", but the difference will be explained further ahead). When all values Bob sends are either "5" or "2", he has finished this iteration of the cascade, and can proceed to the next.
6. Alice knows the process is over when all parities she receives are either "5" or "2". She then proceeds to the next iteration.

Before proceeding we need to clear up two issues.

First, to keep Eve from gaining additional information, we need to discard a bit for every parity check. However, the Cascade process relies on not discarding any bits until the process is complete. Therefore, to achieve both these objectives, the following is done: each time a parity check is performed, one of the bits is marked. If the bit is "1", it is changed to a "7"; if it is "0", it is changed to an "8". As we are treating "0" and "1" as simple integers anyway, it does not have an impact on performance. This way, when the process finally concludes and the key is written to the output buffer, all values which are not "0" or "1" are ignored, and in the meantime, they are still even or odd.

Secondly, we shall now explain the partition "updating" process. In order to find the errors, we start with partitions of a given size, and if the parities don't match, we have to compare shorter partitions, which are continuous subsets of the original partition. These shorter partitions are calculated based on the responses from Alice. The process of calculating these shorter subsets based on Alice's responses is what we referred to as "updating" the partitions. The process is as follows: if the partition has its original size, it is split in half, and the first half becomes the new partition; if the partition has not the original size, and Alice response was "0", the new partition is the first half of the current one; if the partition has not the original size, and Alice response was "1", the new partition is the first half of the second half of the previous one. We can see a representation of these three cases in Figure 1.11.

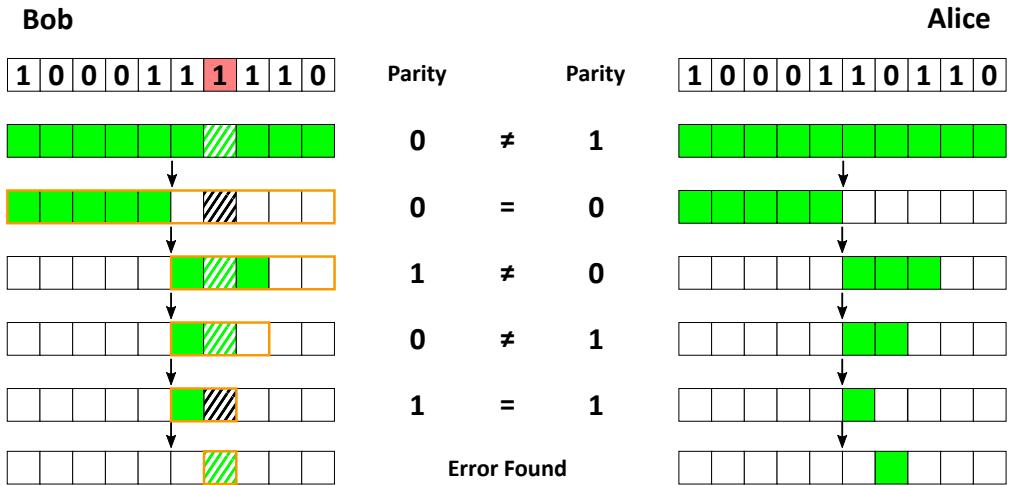


Figure 1.11: Process of updating partitions, demonstrated with a 10 bit partition. In order to find the error, the partition is sequentially divided in smaller subsets, with the particular choice of subset depending on whether Alice and Bob’s parities agree. Bob’s key is on the left, Alice’s is one the right and the error in Bob’s key is shown in red. The green squares represent the bits chosen to calculate the parity to compare, and the orange line delineates all the possible error positions. In the last line, no parity is required, as there is only a single bit left, which must be the error. That bit is then flipped. In addition, each time a parity is compared, one of the bits in the green squares is marked to be discarded. In this case, 5 in 10 bits would be discarded from this bit string.

Now we can continue explaining the other iterations. All iterations after the first are similar, only the first one is different.

1. Bob takes the key obtained at the end of the previous iteration and creates a scrambled copy of it using random permutations. He then sends the permutations to Alice, so she can also create a scrambled copy of her key. This is required, because at the end of the previous iteration, all partitions either contained no errors, or an even number of errors. Scrambling the key allows to spread the errors, so that the likelihood of a partition containing an even number of errors is smaller. The scrambling process and the corresponding rationale is demonstrated in Figure 1.12. This scrambled key is then treated as the initial data: partitioned, parities calculated, etc. It works mostly like the first iteration. One relevant aspect is that the size of the partition does not necessarily need to remain the same. As the number of errors diminishes, the partition size can be increased. The original article recommends doubling the partition size with each subsequent iteration. That is an option on the simulation, however it currently does not perform as we would hope. Therefore, for now, the partition size remains constant, which limits the efficiency of key generation, but ensures a very low error rate.
2. Alice also creates a scrambled copy her key, so it has the same one as Bob. Again the process remains mostly the same.

3. When an error is finally found, Bob sends a "5" in the parities. He corrects the error in the original version of the key. In addition, he also stores the position of the corrected bit in the original key (the one obtained at the end of the first pass). He also activates a Boolean flag to indicate that the lookback process is to be done when this iteration ends;
4. When Alice receives a "5", she also stores the original position of that bit, corrects the original version of the key and activates a similar flag.
5. After all errors have been corrected, Alice and Bob start the lookback process. The rationale for this is that for each error corrected in this pass, in the original key that error was paired with at least one other error. As we now corrected some errors, their old pairs should now be easily found, if they haven't already been corrected. Therefore, we look into the smallest previous partition which contained the bit we corrected, and look for the error with the process described above (they are all the same size, but the first one would be the smallest otherwise).
6. After the lookback process is completed, we can proceed to the next iteration. The error correction finishes after a predetermined number of iterations.

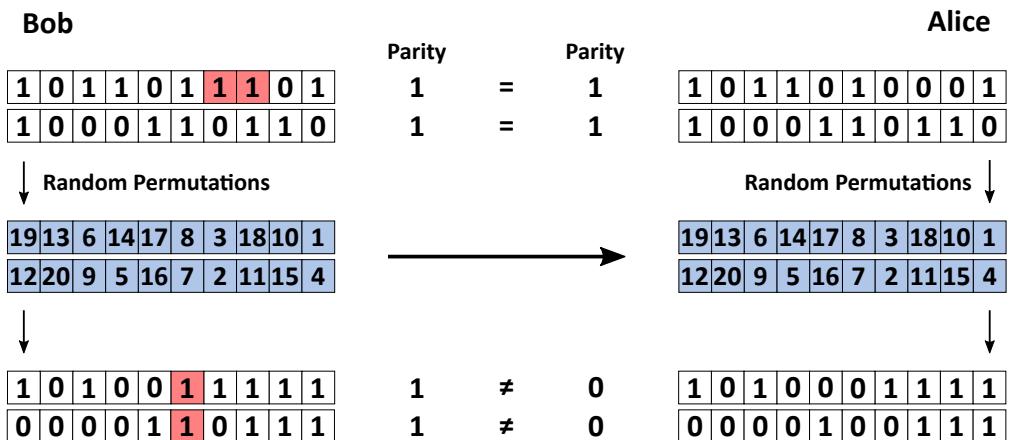


Figure 1.12: Process of creating a scrambled copy of the key. Demonstrated here using only two partitions of 10 bits each. Initially, no error is detected. Alice and Bob have the same parity for both partitions. However, after generating a random list of indexes and permutating the bits according to it, the errors become spread out between both partitions. Using only two partitions, the likelihood of the errors staying in the same partition is still high. Processing longer keys improves this aspect.

Privacy Amplification

The purpose of privacy amplification is to reduce as much as possible the information that Eve has on the transmitted key, while keeping it as long as possible.

There are two sources where Eve is able to acquire information about the transmitted key: she can directly intercept the communications in the quantum channel; and she also can also learn information during the error correction stage, when Alice and Bob share the parities of bit strings in order to find and correct errors.

The error correction process already discards a bit for each parity check that Alice and Bob conduct, so that Eve does not learn any new information. However, Eve likely still retains some information about the key, at least from intercepting qubits in the quantum channel.

For instance, let us assume that Bob knows that the QBER is equal to a certain percentage z , and that every error in the received key is due to a qubit which Alice wrongly measured and subsequently re-sent to Bob.

From Eve perspective, she will measure with the correct basis half of the time. When she uses the wrong basis, she has a 50% probability of introducing an error. Therefore, the total number of bits she might know is given by the number of times she measured correctly, which is twice the number of times she introduced an error, $2z$.

In order to reduce her information, we can transform the key so that the information Eve retains from it is almost certainly none (or, at most, a single bit). Again, this necessarily implies a reduction of the key size. If we assume that Eve probably knows K bits, and our key is N bits long, the final key would be $R = N - K - S$. Here, S is a security parameter that can be chosen as a function of the desired security.

We can use a hash function to transform the key that Alice and Bob have generated into a key of size R . To do this, the hash function should be randomly chosen from a class of strongly universal hash functions, by randomly selecting a key and having Alice and Bob use it. The amount of information that Eve is expected to have on the final key in this case is less than $2^{-S} / \ln(2)$ bits. If the hash function is instead chosen from an almost strongly universal class this still works, and the expected knowledge is no higher than $1 + \log(1 + 2^{-(S+1)})$ [4].

Another option is to generate a new key by checking the parities of random subsets from the initial key. This reduces the knowledge that Eve has on the key, as she cannot know the parity of a subset containing bits that she does not know. The key still needs to be reduced by the same amount as in the previous case.

1.2.3 External usage

The work described in this case study was done with the intention of developing and testing the *AliceQKD* and *BobQKD* blocks. Their function and signal interface was designed so that it would be compatible with other simulations and with the experimental setup currently in development. We shall now detail these blocks, their signal interfaces and what their expected inputs, outputs and functions are expected to be.

General overview

If we look at Figure 1.6, we can immediately take notice that some components were included with the intention of being easily replaceable.

First we have the binary sources. These are easily replaceable with any other source of binary data, such as a quantum random number generator.

Secondly, we have the quantum channel emulator, which tries to emulate the expected behavior of a quantum channel while trying to avoid overcomplicating the simulation. Currently, it simply directly passes the data, introducing errors, attenuation and double clicks according to the defined probabilities and channel characteristics. We can replace it by another channel that follows its guidelines. Thus, it would allow connecting the *AliceQKD* and *BobQKD* blocks, either for using a more complex simulation, testing different configuration or slightly different transmission protocol, or even for implementing the Alice and Bob software parts to use on an experimental setup.

Messages

A classical channel between Alice and Bob is required for most of the software processing, namely basis reconciliation, error correction, and even for sharing seeds to use with hash functions during privacy amplification. This is done using the two message signals which directly connect *AliceQKD* to *BobQKD*. In a simulation setting, they should always remain connected to each other. In a future implementation where Alice and Bob run in different computers, these should be the only signals which need to communicate directly in a classical channel between both computers.

The message signals implementation is a bit more liberal than the other signals, but all messages share some common properties. In all of them the data is transmitted in strings. They are identified by their type. In the blocks responsible for sending and receiving messages, *MessageProcessorReceiver* and *MessageProcessorTransmitter*, each message type is associated with a corresponding input or output signal, respectively.

Each message usually contains a complete set of data, corresponding to a single iteration step of the corresponding type. It is assumed that the blocks won't send a message if the previous one hasn't been received. This is ensured by the way the data flows between the different processing layers.

- During basis reconciliation, Alice never sends a message by herself, she only responds to Bob's messages. In addition, Bob sends a message to Alice, and does not send a second one until he has received an answer for the first.
- The process for parameter estimation is essentially the same, only one element starts it and it does not proceed until an answer is received.
- During error correction, the process is a little more complex, but the principle remains. Bob starts the process, and from there he requires Alice's answer to proceed until the current processing stage is done. Then he starts it again, following the same *modus operandi*.
- During privacy amplification or changing the cascade blocks size, the requirement to only have one message is not as important, as the amount of data sent is much smaller,

and accumulation of a small number of messages has little impact. However, the ratio of the number of bits processed at different stages should ensure that there are never enough accumulated messages to fill the buffers (which typically won't happen easily).

The way it currently works is that when data is sent to a *MessageProcessorTransmitter* or from a *MessageProcessorReceiver*, the first value contains the total number of samples to retrieve (excluding this value from the count). This way, the transmission of data is not constrained by the buffer sizes, and a full set of data can be sent over as many iterations as necessary. Of course, reducing the number of iterations required to transmit the data to and from the message processors greatly improves performance. The preferable way to send and receive data from the *MessageProcessor* blocks is resorting to two specific functions provided by the *message_processor_common* header. To send data we use the *MessageProcessors::sendToMsgProc* function:

```
bool MessageProcessors::sendToMsgProc(vector <t_integer>& data,
Signal& signalToMsg, bool& started)
```

The vector *data* contains the data to transmit, and should not be altered until all its contents have been transmitted, nor should it be constant; *signalToMsg* is the signal which connects to the *MessageProcessorTransmitter*; and finally *started* is a boolean variable, which should be initialized as *false*, and which the function will update according to the transmission status. This variable should not be manually altered after initialization, and cannot be constant.

Similarly, to receive data we use the *MessageProcessors::recvFromMsgProc* function:

```
bool MessageProcessors::recvFromMsgProc(vector <t_integer>&
data, Signal& signalToMsg, bool& started, int&
numberOfValuesToRecv)
```

The vector *data* is where the received data will be saved, and should not be altered until all its contents have been transmitted, nor should it be constant; *signalToMsg* is the signal which connects to the *MessageProcessorReceiver*; *started* is a boolean variable, which should be initialized as *false*, and which the function will update according to the transmission status; and *numberOfValuesToRecv* is an integer variable where the the function keeps track of how many values are still missing. Similarly to the *sendToMsgProc*, these two variables should not be manually changed.

Currently there is no enforced size limit on a single message. While the message buffers have the default space for 512 values, there is no specified maximum size for a message. This means that each message can, in principle, have an arbitrary size. In practice, they are limited by external constraints, such as the sizes of some buffers, the maximum string that can be transmitted through an IPTunnel, and so on.

One could think that reducing the key size in order to process less bits at a time, therefore reducing message size, would be practical. However, this would both reduce the error correction effectiveness and increase the total number of channel uses, delaying the error correction process. Both of these effects are highly undesirable. In fact, in an ideal scenario, the key would be processed all at once, after the transmission ended. This would maximize the effectiveness of the error correction, and the number of classical channel uses would be

minimal (the partition size would be the same, and the number of classical channel uses done by the Cascade algorithm is independent of key length, assuming message size is arbitrary). However, this is not an option when operating in continuous transmission.

Quantum channel emulator

The quantum channel emulator is currently a black box which communicates the bits from Alice to Bob, probabilistically introducing errors, double clicks or no clicks. These are the signals expected from a laboratorial BB84 implementation, and as long as the interface is correct, it should be possible to easily replace the *QuantumChannelEmulator* block with a different, more complex channel model, or even a real quantum channel.

Lets start on Alice's side. As mentioned before, *AliceQKD* block expects input from two binary sources. These can be replaced by other sources, but they should contain only the bits to be transmitted or the bits indicating which base to use. Any synchronization bits, or any other kind of deterministic data, is not expected, and will be treated as if they were either data or basis. That would be a problem, as the randomness of the data is crucial to security. The other input are the messages from Bob, but those were already described.

In addition to the messages to Bob, Alice outputs three different signals. The first and second are binary signals, copies of the two binary input signals, intended to be used as data and basis by the quantum channel. Again, they contain only bits to be used as data and basis, they don't contain any headers or other type of additional information.

These are all the signals relevant to the quantum channel on Alice's side. Let us now observe Bob's side.

Bob also has the messages signals to communicate with Alice in the classical channel, as previously discussed. He also makes use of a binary source to generate the bases used to measure the incoming signal in the quantum channel. Again, it could easily be replaced by any other binary source. A direct copy of this signal is then output into the *QuantumChannelEmulator*.

Other than that, Bob has two additional inputs. One is the *BobData_In* signal. Unlike all the other signals discussed here, this is not binary. Although the data transmitted in the quantum channel should be binary, the channel also outputs certain numbers when it was unable to make a proper measurement, either due to no detector clicking, or both the detectors clicking at the same time. Therefore, this signal is *TimeContinuousAmplitudeContinuousReal*. The numbers used to identify the *doubleClick* or *noClick* events can be defined in the blocks, both in *BobQKD* and in the *QuantumChannelEmulator*. By default, these numbers are set to 2 and 3, respectively.

1.2.4 Interface

Some information is displayed on the dashboard console, in order to pass information to the user about the performance of the protocol. Two different versions exist, one for Alice and one for Bob. The information shown is mostly the same, but some of it is slightly different, due to the difference in perspectives.

```
Alice
#####
Transmitted qubits rate: 1.0e+03 Hz
Photons per pulse: 1.0e-01 Hz
Average QBER: 1.50000 %
Sifted key rate: 7.9e+00 Hz
Key rate after QBER estimation: 4.2e+00 Hz
Key rate after error correction: 3.3e+00 Hz
Key rate after privacy amplification: 3.2e+00 Hz
Overall efficiency: 0.32397 %
#####
Local IP Address: 192.168.92.131
Remote IP Address: 127.0.0.1
```

Figure 1.13: Console interface - Alice.

```
Bob
#####
Received qubits rate: 1.6e+01 Hz
Double click rate: 1.4e-03 Hz
Average QBER: 1.50000 %
Sifted key rate: 7.9e+00 Hz
Key rate after QBER estimation: 4.2e+00 Hz
Key rate after error correction: 3.3e+00 Hz
Key rate after privacy amplification: 3.2e+00 Hz
Overall efficiency: 0.32397 %
#####
Local IP Address: 192.168.92.131
Remote IP Address: 127.0.0.1
```

Figure 1.14: Console interface - Bob.

There are three different section, separated by rows of "#".

The first section contains only the first line, which identifies whether the data shown pertains to Alice or Bob. This is controlled with the *printRecv* input parameter: when set to true, it tries to show Bob, and if false, it tries to show Alice. If the attempted person does not exist, it shows nothing.

The next section describes key statistics about the process:

- **Transmitted qubits rate** - Number of qubits sent by Alice per second. Is equal to the bitrate defined in the input parameters.
- **Received qubits rate** - Number of qubits received per second by Bob. Includes only single clicks and double clicks.
- **Photons per pulse** - Number of average photons per input pulse at the start of the quantum channel.
- **Double click rate** - Number of double clicks per second on the receivers.

- **Latest estimated QBER Upper Bound** - Upper bound obtained for the QBER estimation, for a given confidence interval. This line also shows a warning when the QBER Upper Bound is higher than the QBER Security Threshold.
- **Sifted key rate** - Rate of generation for the sifted key in bits per second. Contains only values correctly measured, after discarding all the attenuated measurements, double clicks and measurements where the wrong basis was used.
- **Overall efficiency** - Ratio of final key size to initially transmitted qubits.

Lastly, the third section shows the IP of the computer running the interface and the IP of the remote computer, to which the current one is connected.

1.2.5 Results

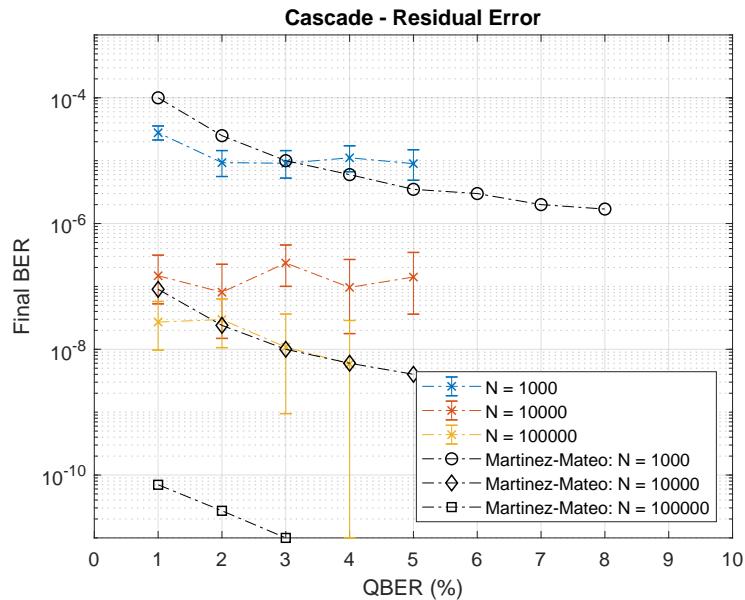


Figure 1.15: Residual BER after error correction as a function of the processed key size, for several QBER values.

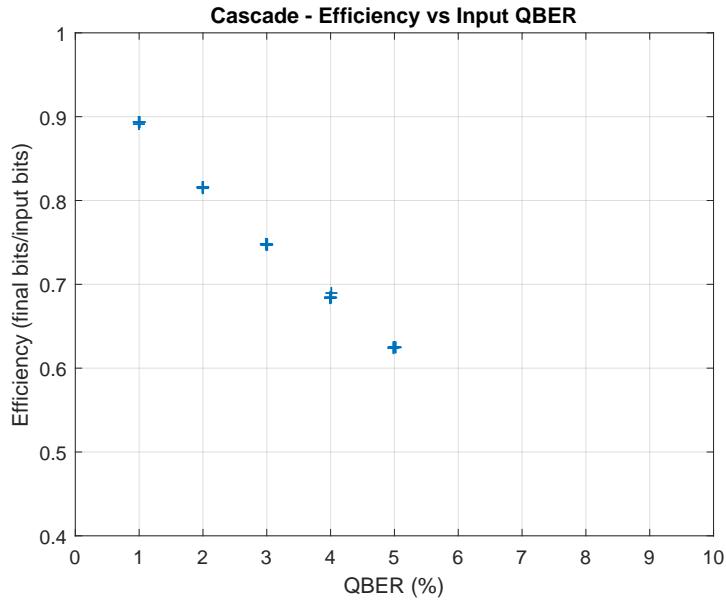


Figure 1.16: Cascade efficiency: ratio between number of bits at the beginning of cascade and number of bits after discarding one bit for each new bit revealed during parity check. These results assume that the initial block sizes were chosen based on the upper bound for the BER estimated during parameter estimation. 1000 bits were used for each BER estimate.

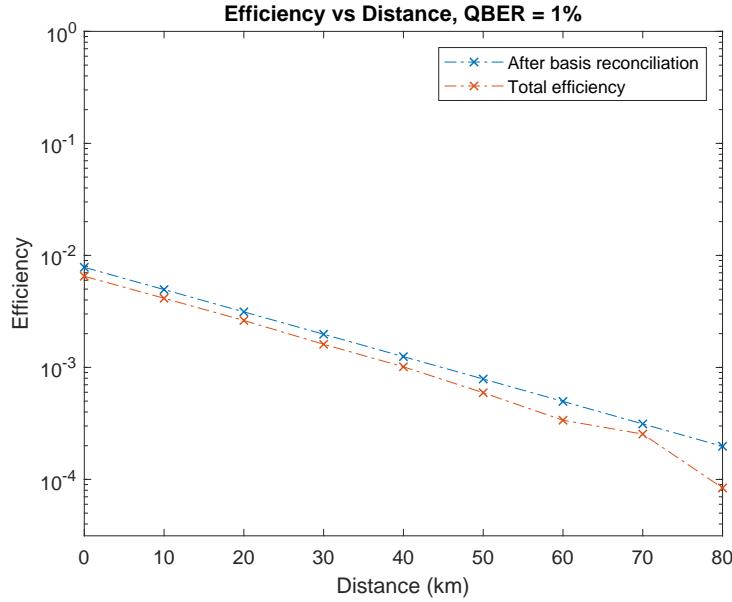


Figure 1.17: Protocol efficiency vs transmission distance for a QBER of 1%. The blue curve shows the ratio between the size of the sifted key and the number of transmitted qubits; the orange curve shows the ratio between the final key and the transmitted qubits. Probability of double click was set at 0.0001, the parameter estimation used 1000 bits out of every 100000, the detector efficiency was set at 0.25, the fiber attenuation at 0.2 dB/Km, insertion losses were 2 dB, deadtime was $10 \mu s$ and the average number of photons per pulse sent by Alice was 0.1.

1.2.6 Open Issues

1. Cascade is not working as well as described in some sources. It is able to correct errors with a reasonably high efficiency and achieving low BERs, but the BER does not go down to values as low as it should (view [3]).
2. Hash function currently used in privacy amplification is probably not adequate. The function currently used is *hash* from the standard library, and is not seeded. A 2-universal hash function should be used, and should use random seeds shared between Bob and Alice (so that a random strongly universal hash function is used, instead of a deterministic one).
3. The number of bits to ignore in privacy amplification should be verified. Different sources claim different values (number of parities exchanged, estimated number of measurements from Alice, etc.). Currently, privacy amplification discards a number of bits k per each processed block, where $k = 2n\text{BER} + s$. Here, n is the number of bits in the processed block and s is a constant number similar to the security parameter. The security parameter is usually only needed for the whole key, and implementing it this way may discard much more bits than necessary, particularly for small keys.

- **2019-05-28 Update:** Updating the system so it works in discrete segments of data. Process starts once N data points are available for doing parameter estimation using K random bits. Error correction proceeds once this is done, using N-K bits. M bits are discarded by this process. Privacy amplification should start once it has N-K-M bits available, providing a secure key with N-K-N-S bits.

References

- [1] Charles H Bennet. "Quantum cryptography: Public key distribution and coin tossing". In: *Proc. of IEEE Int. Conf. on Comp., Syst. and Signal Proc., Bangalore, India, Dec. 10-12, 1984.* 1984.
- [2] Gilles Brassard and Louis Salvail. "Secret-key reconciliation by public discussion". In: *Workshop on the Theory and Application of Cryptographic Techniques*. Springer. 1993, pp. 410–423.
- [3] Jesus Martinez-Mateo et al. "Demystifying the information reconciliation protocol Cascade". In: *arXiv preprint arXiv:1407.3257* (2014).
- [4] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. "Privacy Amplification by Public Discussion". en. In: *SIAM Journal on Computing* 17.2 (Apr. 1988), pp. 210–229. ISSN: 0097-5397, 1095-7111. DOI: [10.1137/0217014](https://doi.org/10.1137/0217014). URL: <http://pubs.siam.org/doi/10.1137/0217014> (visited on 05/07/2019).

1.3 The Impact of Pulse Shaping in the Security of CV Quantum Communication Systems

Student Name	:	Daniel Pereira (21/05/2019 -)
Goal	:	Theoretical, simulated and experimental study of the use of Nyquist shaping for Continuous Variables Quantum Communications.
Directory	:	sdf/cv_m_qam_system
Related Links	:	https://www.overleaf.com/project/5cdbdd58d0b38c095359e0b7

Continuous Variable Quantum Communications (CV-QC) can encode information in the phase of weak coherent states. The weak nature of these states makes frequency and phase noise a substantial impairment, as such it is important to implement optimal techniques for their compensation.

1.3.1 Classical Frequency and Phase Recovery - State of the art

The traditional method for compensating noise in classical coherent communications is to use some form of phase-locked loop (PLL), synchronizing the frequency and the phase of the local oscillator (LO) with that of the transmitter laser [1]. Alternatively, frequency offset compensation and phase recovery can be performed in the digital domain by applying digital signal processing (DSP) [2]. It has been shown that DSP is more tolerant to laser phase noise than PLL based techniques [1]. This work will be focused on the study of DSP techniques for Continuous Variables Quantum Communications (CV-QCs) systems.

Phase Mismatch Compensation Techniques

Deterministic Phase Mismatch

Phase mismatch occurs when the two employed oscillators have differing optical phases, as illustrated in Figure 1.57. This causes the phase modulation applied to one of the signals to be misread in the detection scheme.

Phase Noise

Phase noise arises from the non-zero linewidth of the transmission and reception local oscillators, as illustrated in Figure 1.58. Phase noise consists of a time evolving phase mismatch, which can be modelled as a Weiner process described as

$$\phi(t_k) = \phi(t_{k-1}) + \Delta\phi(k), \quad (1.6)$$

where $\Delta\phi(k)$ are phase increments, which are independent and identically distributed random Gaussian variables with zero mean and variance

$$\sigma^2 = 2\pi\Delta\nu|t_k - t_{k-1}|, \quad (1.7)$$

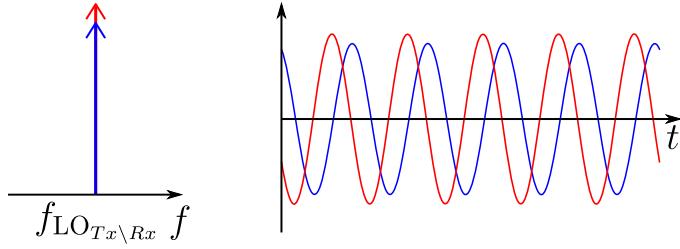


Figure 1.18: Frequency (left) and time (right) domain representation of a phase mismatch between two cosines of equal frequency.

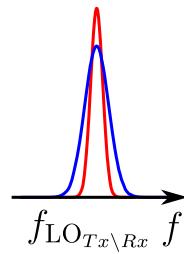


Figure 1.19: Phase noise arises from the non-zero linewidth of the transmission and reception oscillators, being present even if their central frequencies match.

where $\Delta\nu$ is the laser linewidth and $t_{k/k-1}$ are the sampling instants.

In order to show the action of the various phase recovery techniques we assume that a perfect frequency mismatch compensation has been accomplished for the signal (1.16), having resulted in

$$x(n) = x_{\text{sym}}(n)e^{i\Delta\theta(n)}, \quad (1.8)$$

where, again, $x_{\text{sym}}(n)$ is the complex, phase-modulated signal and $\Delta\theta(n)$ is the phase noise contribution.

The phase mismatch can be compensated either through blind estimation [3] or through pilot-aided estimation [4]. In the blind estimation method a m th-order non-linearity is used to remove the phase modulation, yielding an estimate of the phase mismatch [2]. The phase estimation is given by [3]

$$\Delta\theta_{\text{est}}(n) = \frac{1}{m} \arg \left\{ \frac{1}{2L+1} \sum_{l=-L}^L f(l) x^m(n+l) \right\}, \quad (1.9)$$

where $f(l)$ is a weighting function. The m th power removes the phase modulation, thus the only remaining phase is due to the phase noise contribution. The sum $-\frac{1}{2L+1} \sum_{l=-L}^L f(l) e^{i\Delta\theta(n+l)}$ functions then as a weighted average of the phase noise contributions up to L symbols before and L symbols after the current one.

In the pilot aided scheme, pre-agreed on symbols are inserted, time multiplexed, with the data payload at a pilot rate of $1/P$ (meaning one pilot symbol is inserted after $P - 1$ data

symbols). The phase mismatch is determined from the pilot symbols and this estimation is used to compensate the phase mismatch on the data carrying symbols [4].

A alternative blind estimation scheme is proposed in [5], taking advantage of the unmodulated x^m signal to perform both frequency and phase mismatch compensation. The phase added by both the frequency and the phase mismatch is estimated as

$$\Delta\theta_{\text{est}}(n) = \Delta\theta_{\text{est}}(n-1) + \mu \arg \left\{ \frac{1}{L} \sum_{l=0}^{L-1} x^m(n-l)[x^m(n-l-1)]^* \right\}, \quad (1.10)$$

for every symbol except for the first/last L symbols, where μ corresponds to the integral gain. The frequency and phase mismatch compensation is accomplished by taking the uncompensated signal (1.16) and performing

$$x_{\text{rec}}(n) = x(n)e^{-i\sum_{k=L}^n \Delta\theta_{\text{est}}(k)T_s} \quad (1.11)$$

1.3.2 Quantum Frequency and Phase Recovery - State of the art

Due to the low amplitude of the quantum coherent states employed in CV-QC, any auxiliary processes, such as frequency and phase mismatch compensation, need to be employed in a non-intrusive manner. Another consequence of the low amplitude quantum coherent states is the necessity for high sensitivity, low thermal noise coherent receivers [6]. This requirement limits the bandwidth of the receivers [7, 8], which in turn limits the maximum operational baud rate of CV-QC systems. To avoid the added noise from disparities between the transmission and reception LO lasers, the first CV-QC protocols used a high intensity signal sent polarization multiplexed with the quantum signal, to be used as the LO in the detection scheme [9, 10]. This co-propagating technique presents a security risk, with attacks proposed that tamper with the LO's wavelength [11] or the shape of its pulses [12, 13, 14]. Locally generated LO (LLO) CV-QC protocols were proposed simultaneously in [15] and [16]. The usage of two different laser sources for the signal and the LO necessitates the application of frequency and phase recovery techniques.

Phase Mismatch Compensation Techniques

1.3.3 Novelty

1.3.4 Implementation issues

Optimal amplitude of Pilot Signal

1.3.5 Error Vector Magnitude

The figure of merit employed for the comparisons presented in this document will be the Error Vector Magnitude (EVM), where the error is evaluated as the relation between the magnitude of the error vector e_V and the magnitude of the vector of the ideal symbol position ref_V

$$EVM(\%) = 100 \sqrt{\frac{|e_V|}{|\text{ref}_V|}} = 100 \sqrt{\frac{|\mathbf{m}_V - \mathbf{ref}_V|}{|\mathbf{ref}_V|}} \quad (1.12)$$

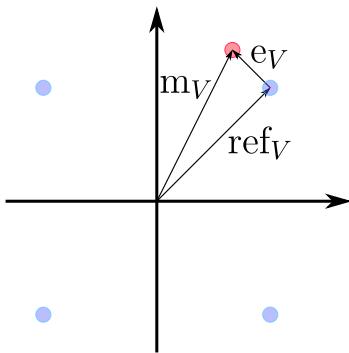


Figure 1.20: Visual representation of the EVM method in a QPSK constellation. The ideal constellation points are presented in blue, while an example for an actual measured symbol is presented in red.

1.3.6 Nyquist pulse shaping

Inter-Symbol-Interference with Root-Raised-Cosine modulation

root-raised-cosine (RRC)

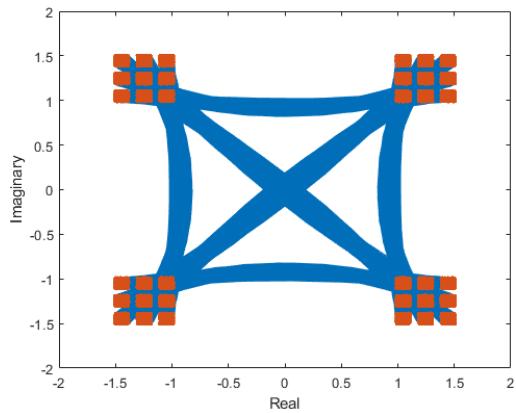


Figure 1.21: Constellation of driving signal. Steady states of the signal highlighted in orange.

1.3.7 Simulation Analysis

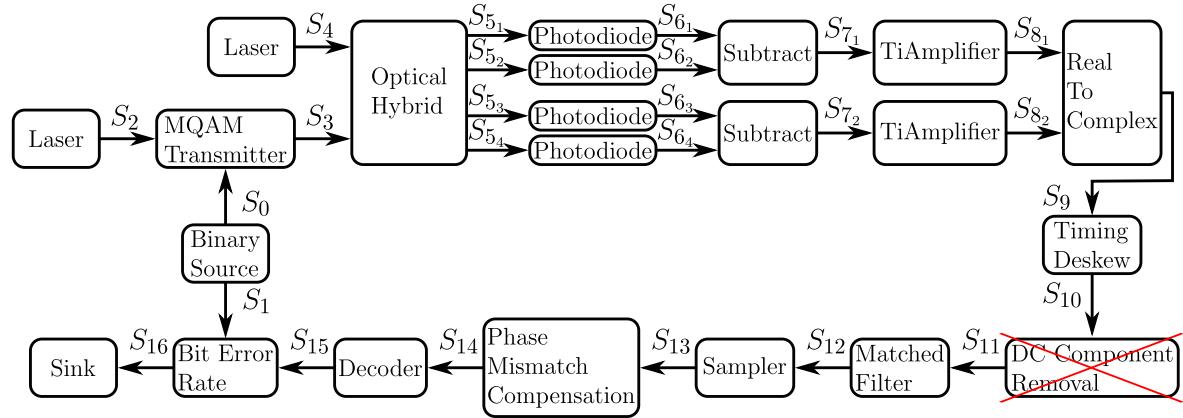


Figure 1.22: Simulation setup for the LLO-sequential CV-QC transmission system employed. DC Component Removal block currently disabled.

Simulation Blocks	
Necessary	Available
binary_source_20180815	✓
bit_error_rate_20180815	✓
dc_component_removal_20190225	✓
decoder_20190611	✓
discrete_to_continuous_time_20180815	✓
electrical_filter_20180815	✓
fft_20180208	✓
ideal_amplifier_20180815	✓
laser_20180815	✓
load_signal_20190503	✓
m_qam_mapper_20190815	✓
m_qam_transmitter_20180815	✓
matched_filter_20190225	✓
netxpto_20180815	✓
optical_hybrid_20180815	✓
orthonormalization_20190225	✓
phase_mismatch_compensation_20190506	✓
photodiode_20190515	✓
pulse_shaper_20180815	✓
sampler_20190218	✓
subtract_20190515	✓
ti_amplifier_20180815	✓
timing_deskew_20190225	✓
white_noise_20180815	✓

No electrical noise

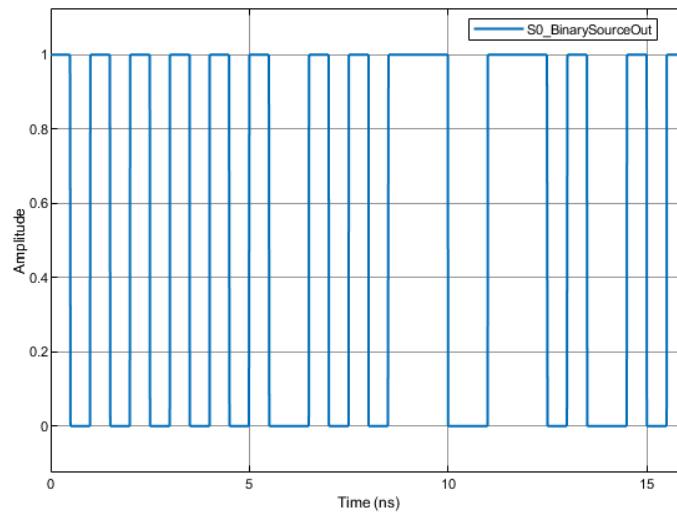


Figure 1.23: Input Binary Signal.

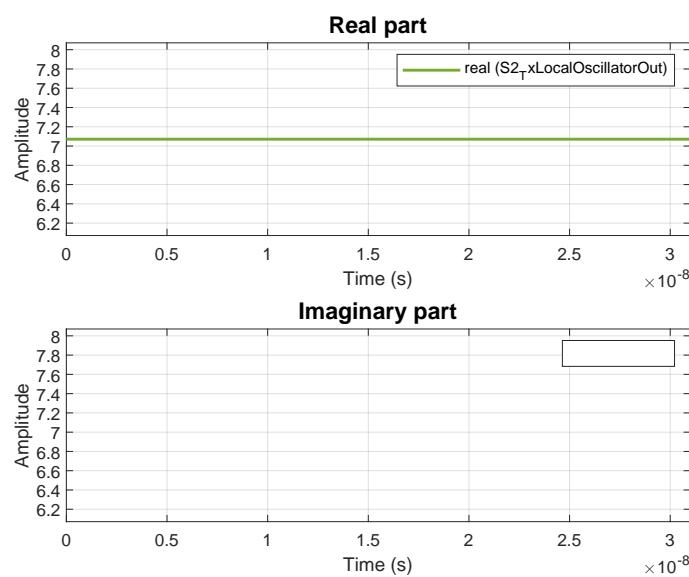
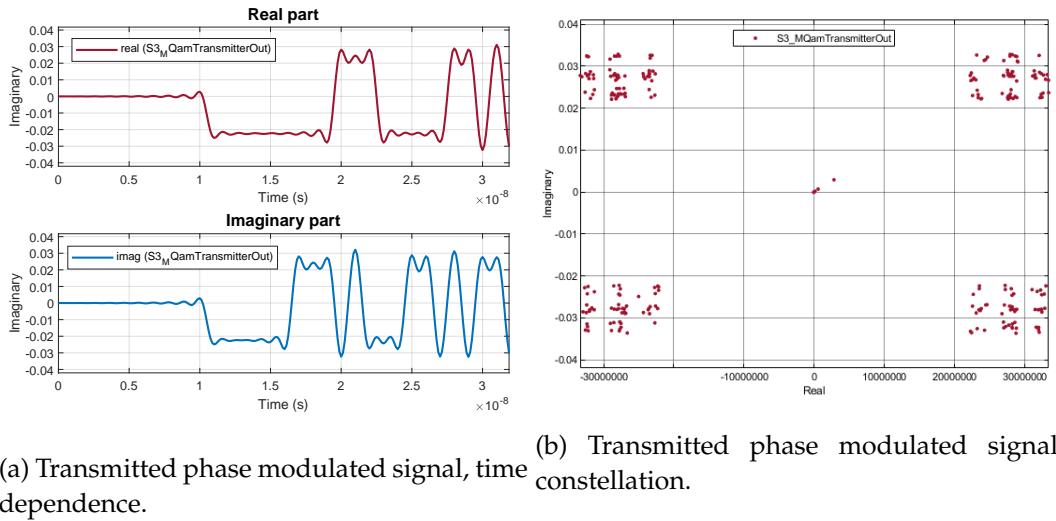


Figure 1.24: Transmission Laser.



(a) Transmitted phase modulated signal, time dependence.

(b) Transmitted phase modulated signal, constellation.

Figure 1.25: Transmitted phase modulated signal.

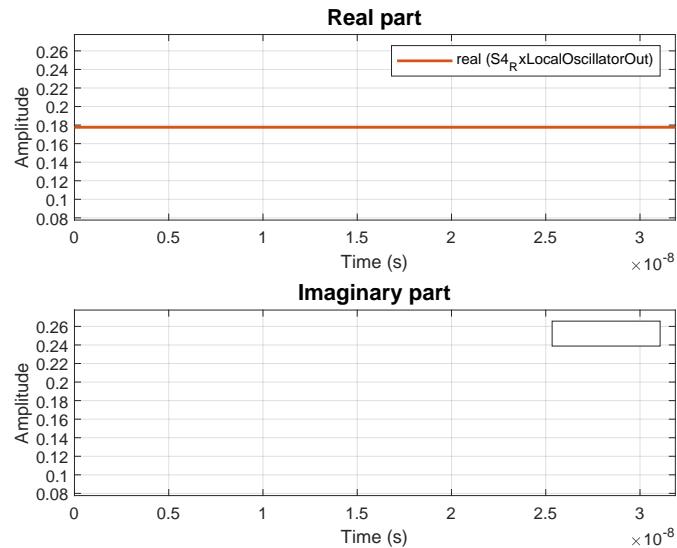
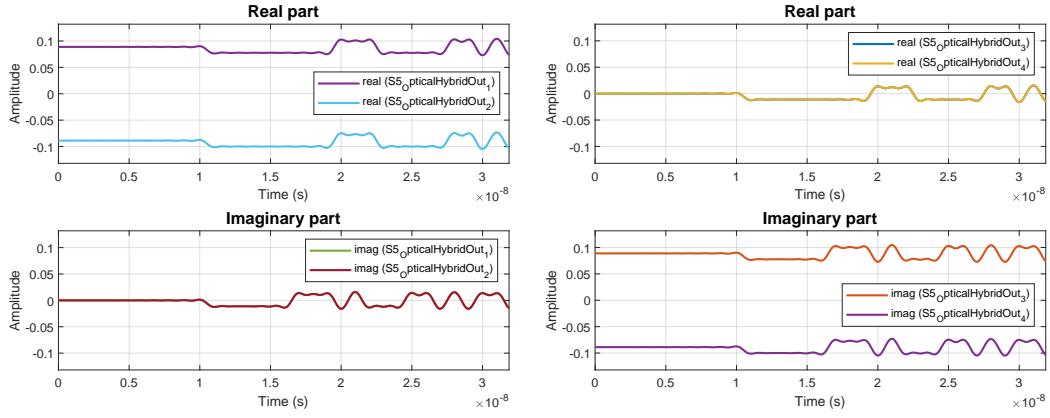
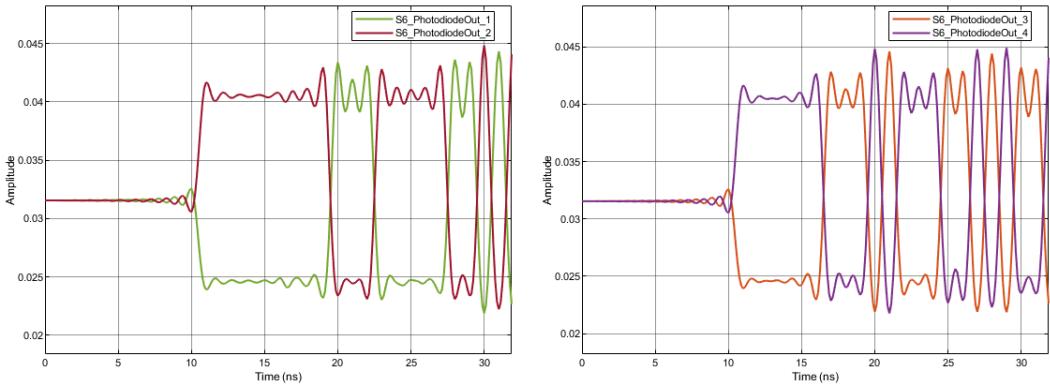


Figure 1.26: Reception Laser.



(a) Output of Optical Hybrid ports 1 and 2. (b) Output of Optical Hybrid ports 3 and 4.

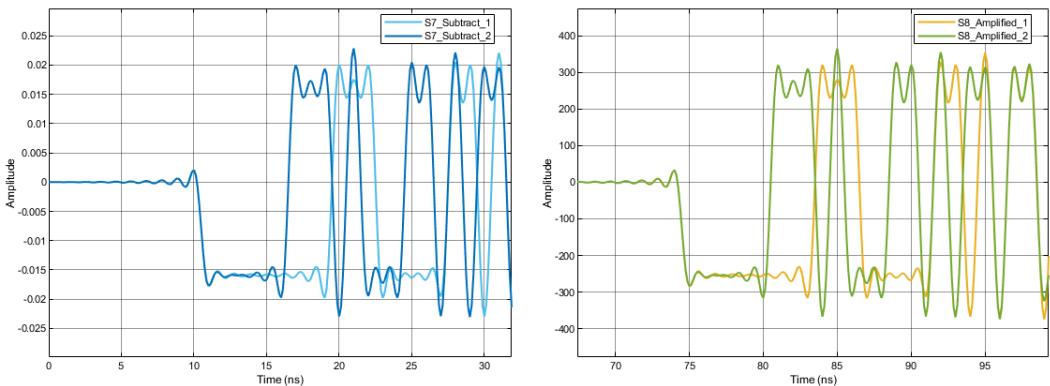
Figure 1.27: Optical-Hybrid outputs.



(a) Output of Photodiodes 1 and 2.

(b) Output of Photodiodes 3 and 4.

Figure 1.28: Photodiodes outputs.



(a) Output of Subtractor.

(b) Output of Trans-Impedance Amplifier.

Figure 1.29: Optical detector outputs.

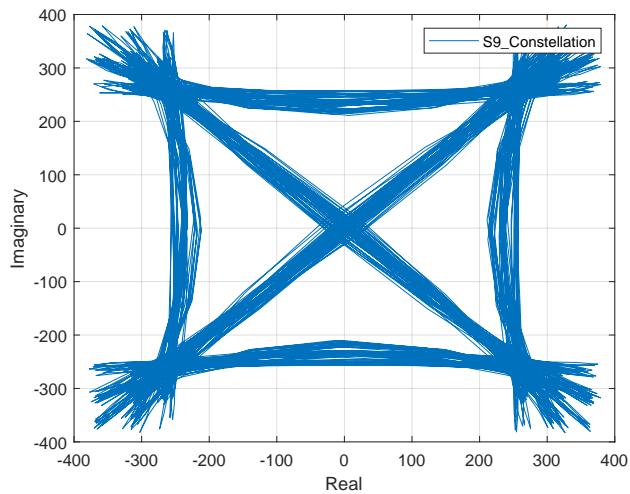


Figure 1.30: IQ diagram of the signal at the output of the optical receivers.

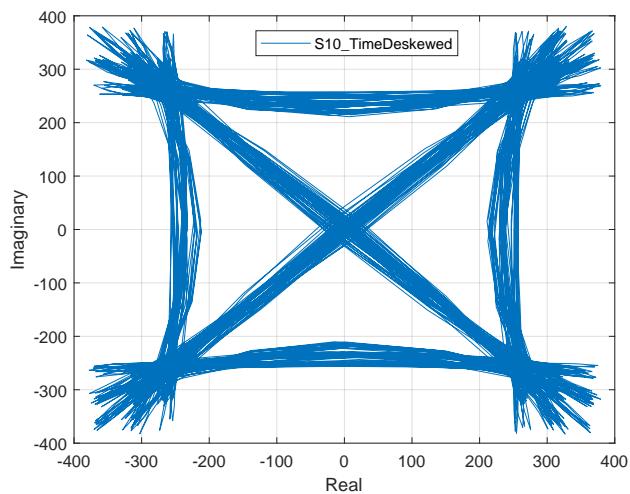


Figure 1.31: IQ diagram of the signal at the output of the timing-deskew DSP step.

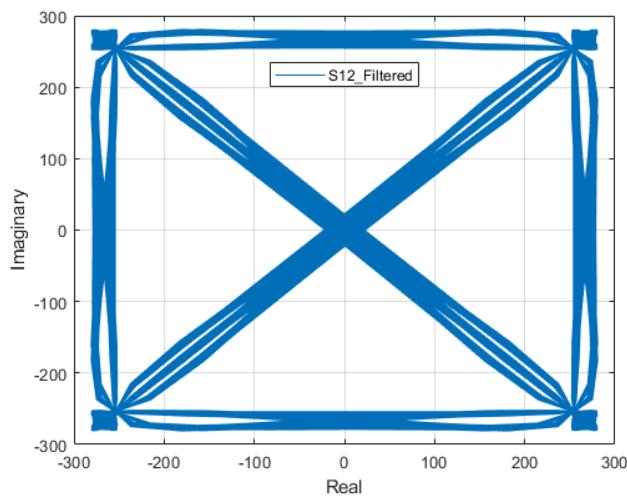


Figure 1.32: IQ diagram of the signal at the output of the matched filter DSP step.

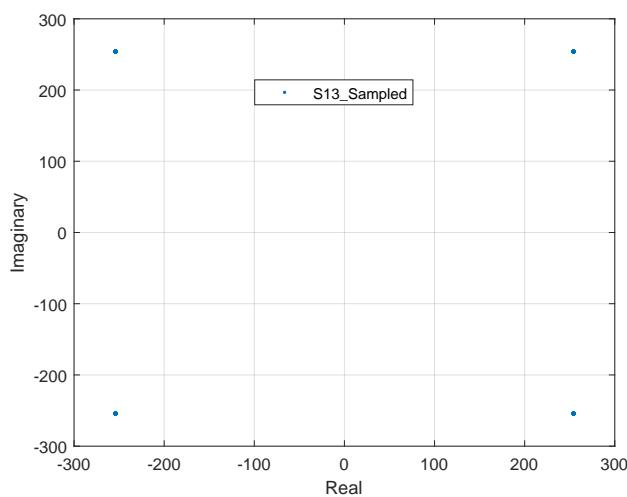


Figure 1.33: Sampled constellation.

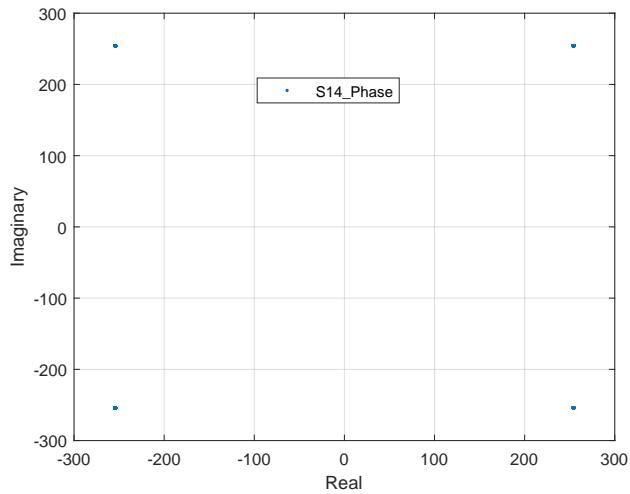


Figure 1.34: Constellation at the output of the phase mismatch compensation DSP step.

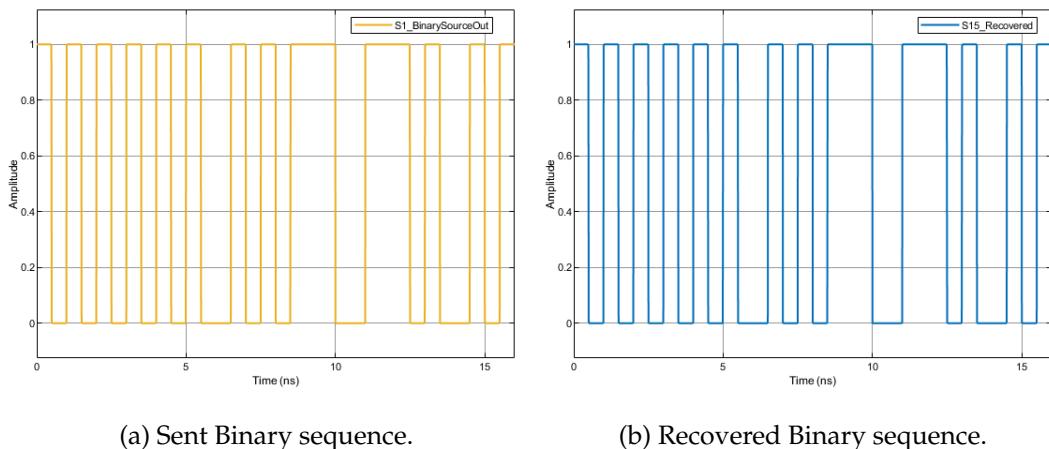


Figure 1.35: Comparison between the sent and recovered binary sequences.

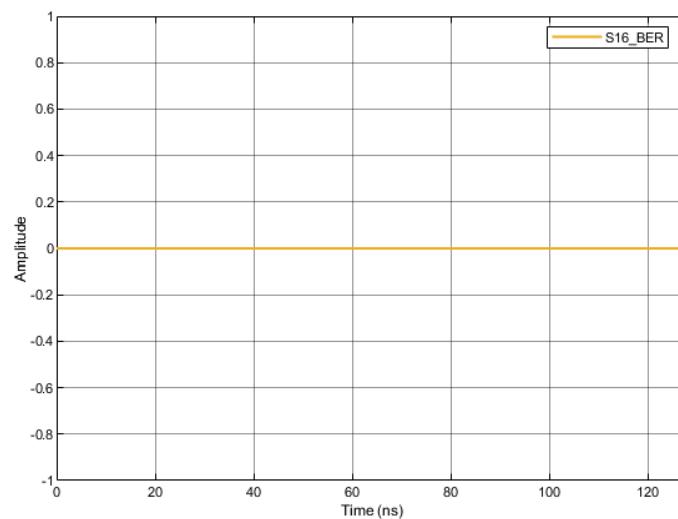


Figure 1.36: Output of the BER block.

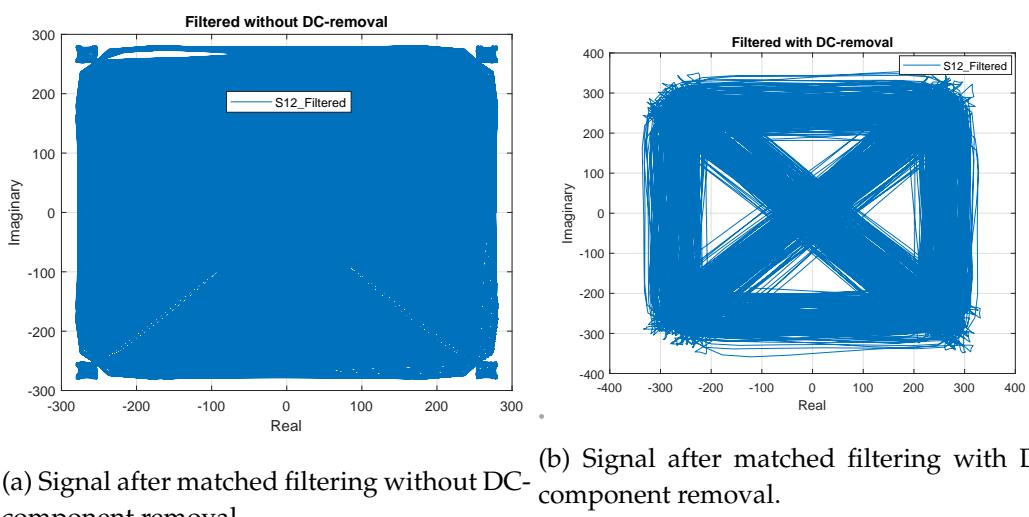


Figure 1.37: Applying DC-component removal to a RRC shaped signal yields weird results.

1.3.8 Experimental Setup

The experimental setup currently assembled in the lab is presented in Figure 1.38 in the form of a block diagram. Two signals containing a repeating pseudo-random sequence of

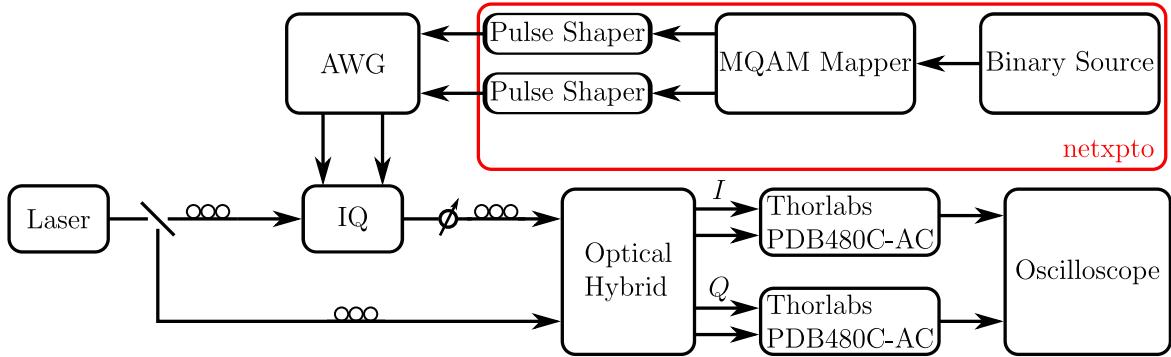


Figure 1.38: Block diagram of the currently implemented experimental setup. DC Component Removal block currently disabled.

2^{15} symbols with RRC pulse shaping. After coherent detection, the signals are converted to the digital domain using an oscilloscope and saved under a .csv format. The .csv files are then loaded into the netxpto environment using the `load_ascii` block and DSP is applied to them.

The DSP applied to the signals obtained from the experimental setup is presented in Figure 1.39 Complex plane representations of selected signals are presented in Figures 1.40

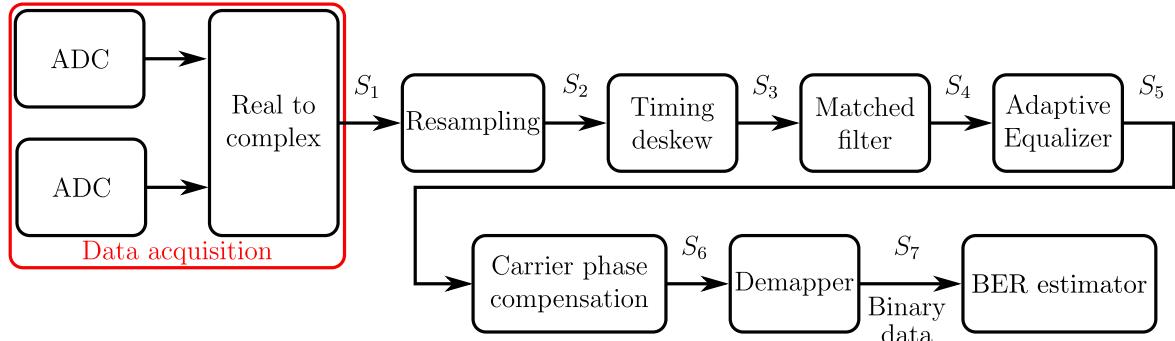


Figure 1.39: Block diagram of the DSP applied to the following results.

and 1.41. The constellation shown in Figure 1.41b is obtained after matched filtering and phase noise compensation. We would expect to see no ISI, but an unfolding of the symbols is clearly visible, this result led to the study presented in 1.3.8.

ISI study

From the result shown at the end of the previous section, it was observed that after the matched filtering stage there still remained considerable inter symbol interference. This was

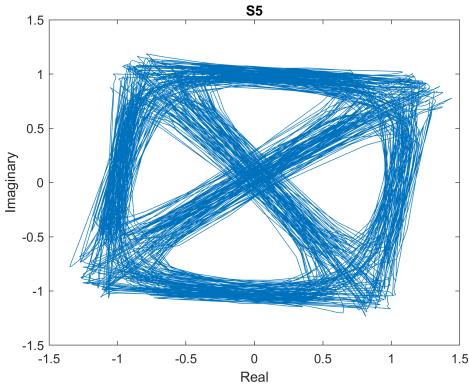
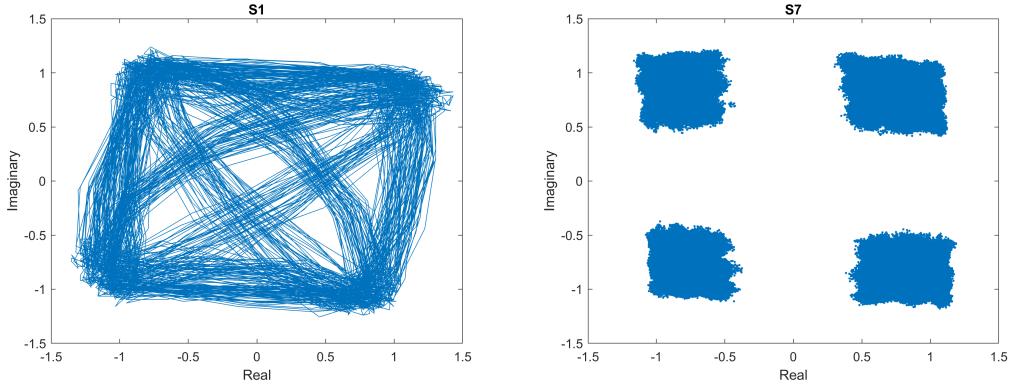


Figure 1.40: QPSK constellation after matched filter.



(a) QPSK constellation at the start of the DSP (b) QPSK constellation obtained at the end of the DSP stage.

Figure 1.41: Comparison of the constellations at the start and at the end of the DSP.

attributed to a filtering effect from the spectral response of the AWG, which can be seen in Figure 1.42, where the spectrum of the input RRC signal is plotted against the spectrum of the output of the AWG. The output spectrum of the AWG was obtained by taking 160000 consecutive symbols directly at the output of the AWG, dividing them into 4000 blocks of 40 symbols each and computing the Fourier transform for each block, the average of these Fourier transforms is then computed to yield the spectrum presented in Figure 1.42. From these spectra we clearly see that the AWG is transforming the signal, a process that can be roughly described by the diagram in Figure 1.43. In the frequency domain, this effect can be described by

$$\hat{Y} = H \hat{X}, \quad (1.13)$$

where \hat{X} is the Fourier transform of the driving signal X , H is the frequency domain representation of the AWG's filter and \hat{Y} is the Fourier transform of the output of the AWG. To recover the original signal X from the output Y we require H^{-1} , which can be obtained

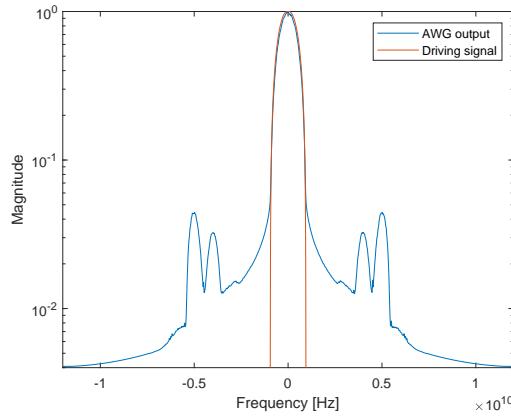


Figure 1.42: Spectral response of the AWG compared to the spectrum of the driving signal.

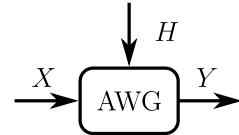


Figure 1.43: Filtering effect of the AWG.

by simply performing

$$\hat{X} = H^{-1}\hat{Y}^{-1} \iff H^{-1} = \hat{X}\hat{Y}^{-1}. \quad (1.14)$$

To obtain the filter we then need only to divide the input by the output spectrum. The inverse filter obtained from the spectra in Figure 1.42 is presented in Figure 1.44.

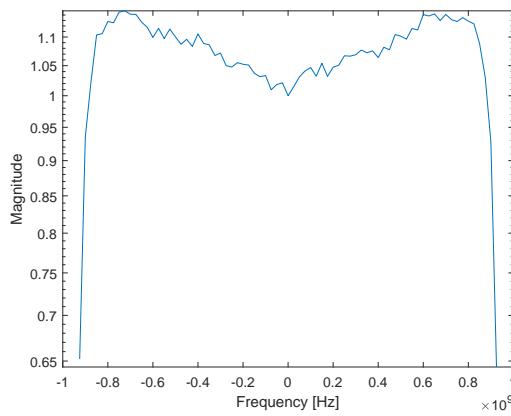


Figure 1.44: Filter extracted from the spectra presented in Figure 1.42.

To apply the extracted filter an overlap save method was employed, following the

method described in Section ???. The spectra before and after the application of the equalization filter are presented in Figure 1.45.

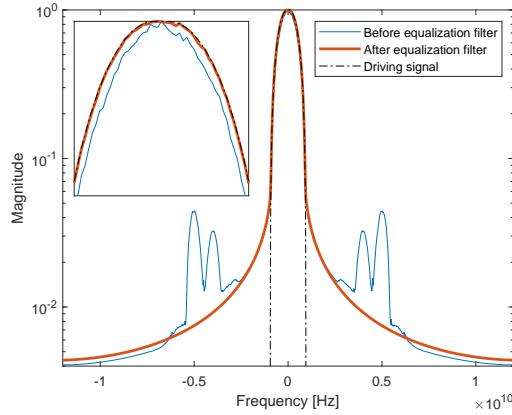


Figure 1.45: Spectra before and after the application of the extracted filter presented in Figure 1.44. The inset shows how closely the spectrum of the signal after the application of the filter follows the spectrum of the driving signal.

To better show the ISI in question and the effect of the extracted filter, the constellations of the signals before and after passing through the AWG, before and after being passed through the matched filter and before and after the application of the extracted filter are presented in Figures 1.46, 1.47 and 1.48. Figure 1.46 shows the results expected theoretically, where the

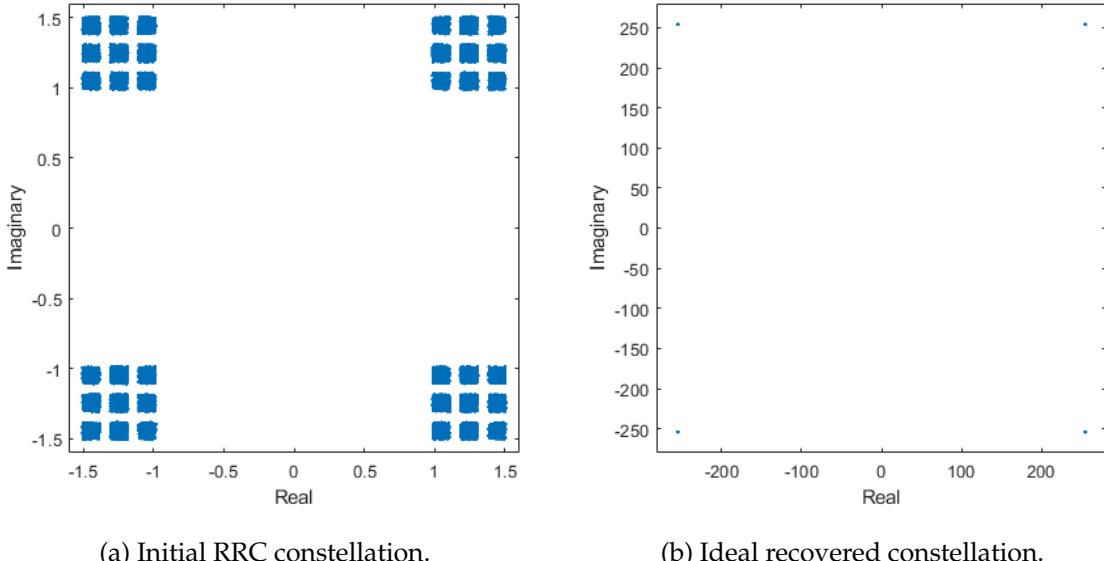


Figure 1.46: Constellations of the RRC modulated signal and before and after the application of a RRC matched filtering step.

ISI of the RRC constellation is completely removed after application of the matched filter. Figure 1.47 shows that, in fact, the signal extracted from the AWG cannot be passed directly

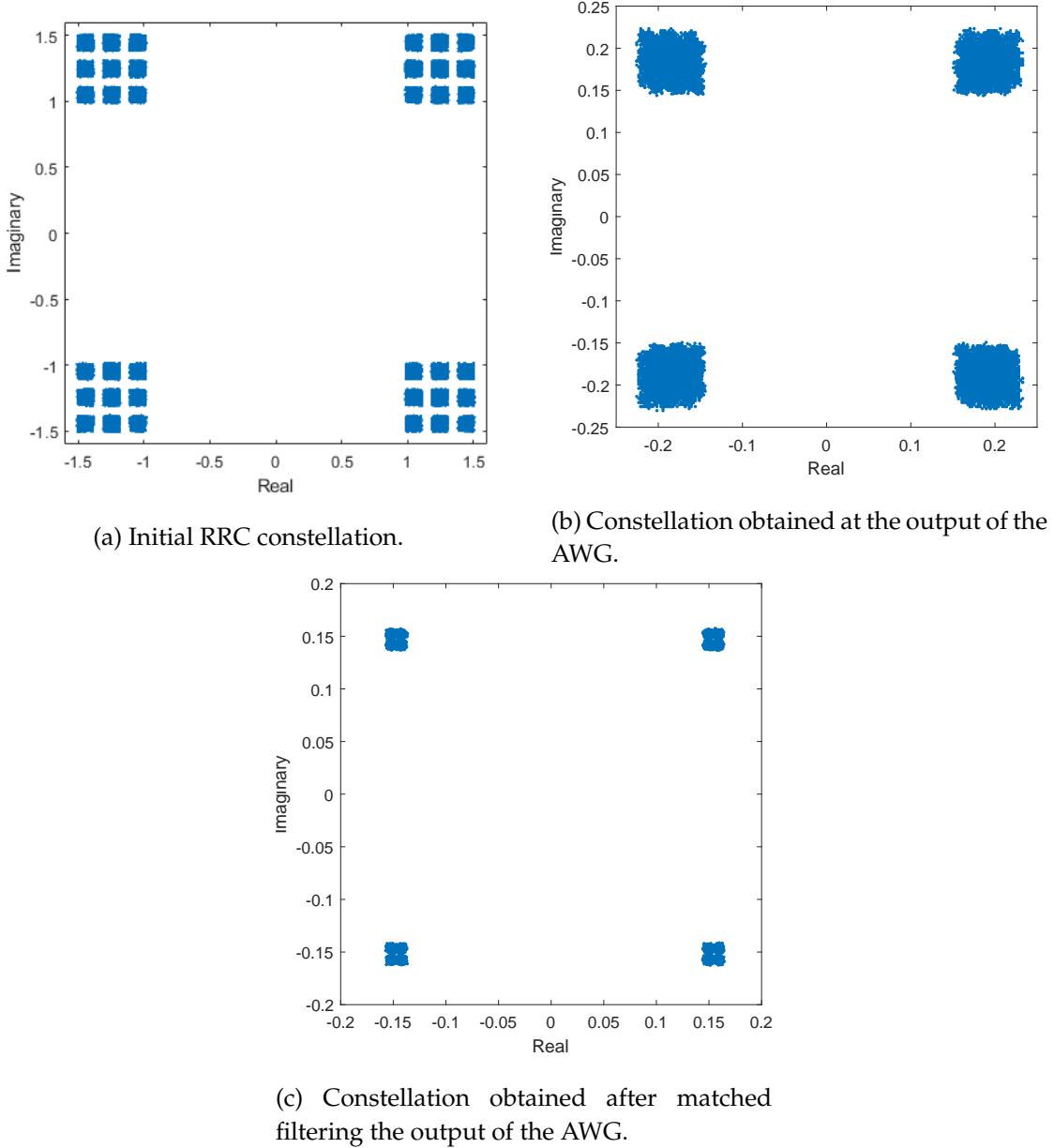


Figure 1.47: Constellations of the input RRC signal, of the signal outputted by the AWG and of the AWG output signal after application of a RRC matched filter.

through the matched filter, as some residual ISI still remains. Figure 1.48 shows the results with the application of the expected filter. From the top-right to the bottom-left signal one can see that the square states observed in the original constellation are much better defined, which in turn leads to the drastic reduction (if not outright removal) of ISI in the recovered constellation.

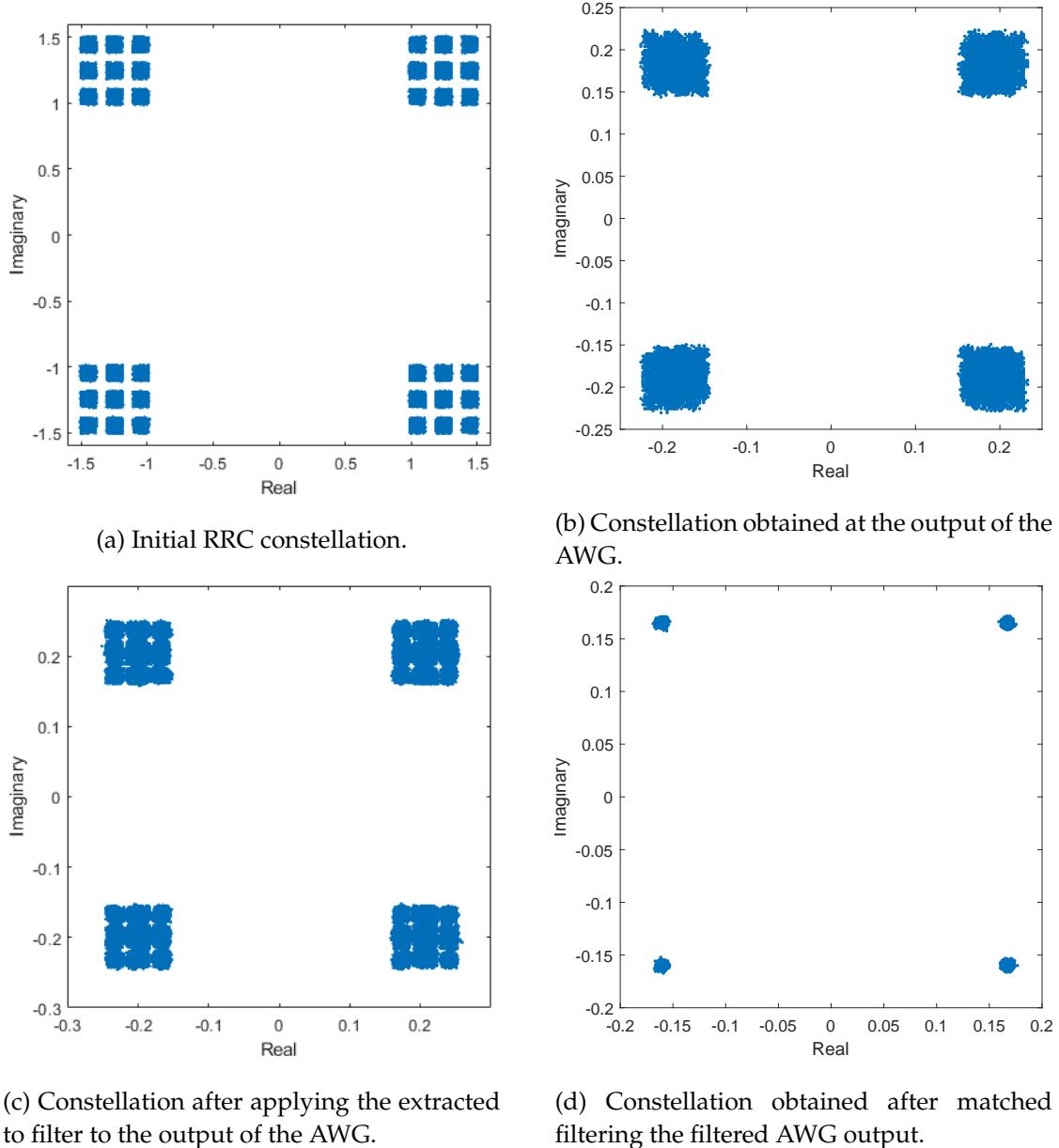


Figure 1.48: Constellations of the input RRC signal, of the signal outputted by the AWG before and after the application of the extracted equalization filter and of the equalized signal after application of a RRC matched filter.

A similar process was followed to extract the filter for the AWG in conjunction with the coherent optical receivers, which is plotted in Figure 1.49, where the filter obtained from the AWG output is included for comparison. We see that the two filters are considerably different, which hints at a non-flat response of the optical receivers.

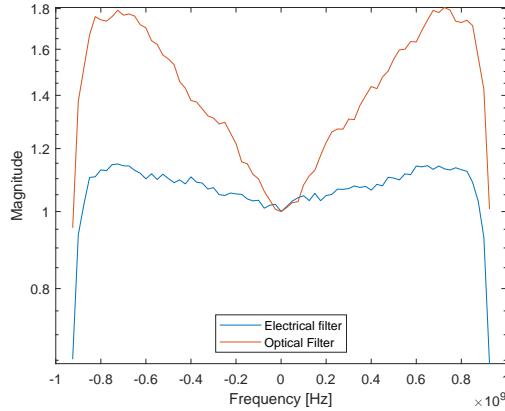


Figure 1.49: Comparison of the filters extracted directly from the AWG with the filter extracted from the output of the optical receivers.

Frequency response of the THORLABS PDB480C-AC

After the previous study on the distortion imposed by the AWG and the receivers, an in depth study on the frequency response of the latter was performed. Following the description of the setup employed by the manufacturer in their characterization of the receiver, present in in [17], the experimental setup represented in Figure 1.50 was assembled. A direct modulation laser was driven using a combination a sine wave of progressively

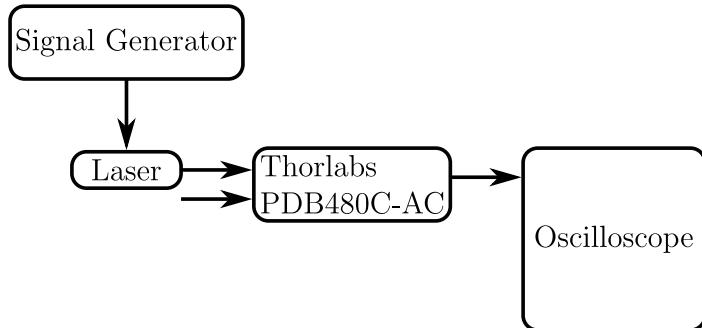


Figure 1.50: placeholder.

larger frequency. Due to bandwidth limits, two signal generators were used to generate the sine wave:

- Agilent Technologies 33250A, to drive the laser from 1 kHz to 80 MHz;
- Rohde & Schwarz SMR40, to drive the laser from 80 MHz to 3 GHz.

The output of the receiver was recorded using an oscilloscope. Example signals for 2 different frequencies are presented in Figure 1.51, where the quenching effect is clearly

visible by the considerable drop in amplitude of the signal from the lower frequency to the higher one. The amplitude of the signal was evaluated by taking the average value of

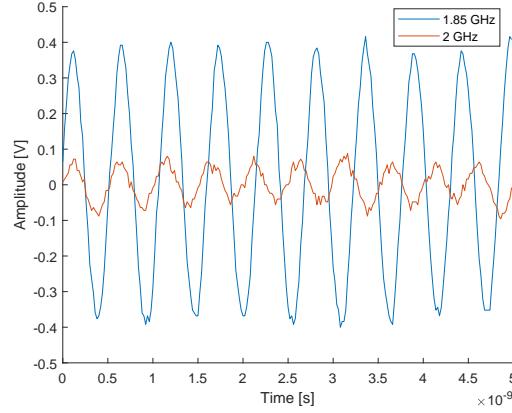


Figure 1.51: placeholder.

the peak-to-peak voltage. The response in function of frequency is presented in Figure 1.52, where an inset is included comprising of the response from 10 to 80 MHz. For this inset signals were recorded with a 1 MHz spacing. This region was chosen due to it appearing to have a consistently flat frequency response.

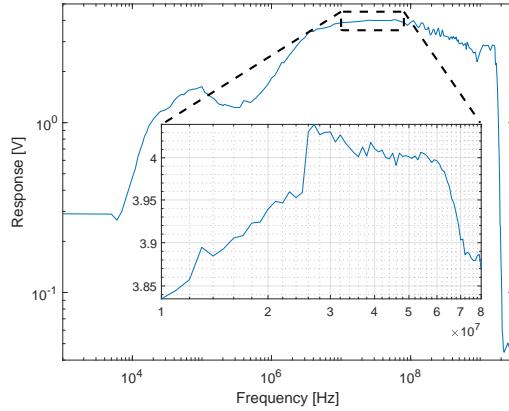


Figure 1.52: placeholder.

Upconversion of the signal

In order to place the signal in the flat region of the receiver, it was chosen to use a RC modulated 12.5 Mbaud signal and to upconvert it to 25 MHz. This is accomplished by simply multiplying the real part of the driving voltage by a cosine and the imaginary part with a sine, both with frequency of $f_c = 25$ MHz. The signals before and after this upconversion step are presented in Figure 1.53.

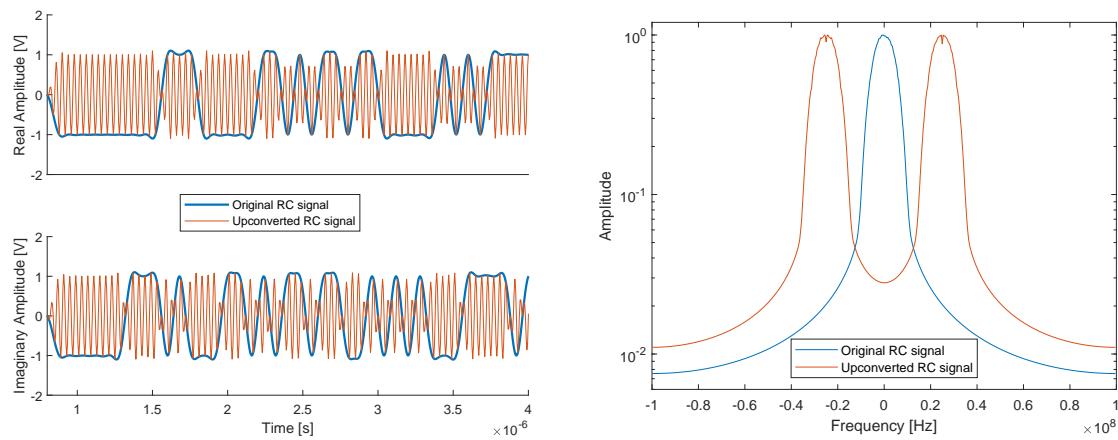


Figure 1.53: Time (a) and frequency (b) domain comparison between the original and upconverted RC signals.

1.3.9 Open Issues

1. Nothing so far

1.3.10 Future work

References

- [1] Ezra Ip and Joseph M Kahn. "Feedforward carrier recovery for coherent optical communications". In: *Journal of Lightwave Technology* 25.9 (2007), pp. 2675–2692.
- [2] Md Saifuddin Faruk and Seb J Savory. "Digital signal processing for coherent transceivers employing multilevel formats". In: *Journal of Lightwave Technology* 35.5 (2017), pp. 1125–1141.
- [3] Michael G Taylor. "Phase estimation methods for optical coherent detection using digital signal processing". In: *Journal of Lightwave Technology* 27.7 (2009), pp. 901–914.
- [4] Maurizio Magarini et al. "Pilot-symbols-aided carrier-phase recovery for 100-G PM-QPSK digital coherent receivers". In: *IEEE Photonics Technology Letters* 24.9 (2012), p. 739.
- [5] Ricardo M Ferreira et al. "Optimized Carrier Frequency and Phase Recovery Based on Blind M th Power Schemes". In: *IEEE Photonics Technology Letters* 28.21 (2016), pp. 2439–2442.
- [6] MB Costa Silva et al. "Homodyne detection for quantum key distribution: an alternative to photon counting in BB84 protocol". In: *Photonics North 2006*. Vol. 6343. International Society for Optics and Photonics. 2006, 63431R.
- [7] John Bertrand Johnson. "Thermal agitation of electricity in conductors". In: *Physical review* 32.1 (1928), p. 97.
- [8] Harry Nyquist. "Thermal agitation of electric charge in conductors". In: *Physical review* 32.1 (1928), p. 110.
- [9] T. C. Ralph. "Continuous variable quantum cryptography". In: *Phys. Rev. A* 61 (1 Dec. 1999), p. 010303. DOI: [10.1103/PhysRevA.61.010303](https://doi.org/10.1103/PhysRevA.61.010303). URL: <https://link.aps.org/doi/10.1103/PhysRevA.61.010303>.
- [10] Mark Hillery. "Quantum cryptography with squeezed states". In: *Physical Review A* 61.2 (2000), p. 022309.
- [11] Xiang-Chun Ma et al. "Wavelength attack on practical continuous-variable quantum-key-distribution system with a heterodyne protocol". In: *Physical Review A* 87.5 (2013), p. 052309.
- [12] Jing-Zheng Huang et al. "Quantum hacking on quantum key distribution using homodyne detection". In: *Physical Review A* 89.3 (2014), p. 032304.
- [13] Paul Jouguet, Sébastien Kunz-Jacques, and Eleni Diamanti. "Preventing calibration attacks on the local oscillator in continuous-variable quantum key distribution". In: *Physical Review A* 87.6 (2013), p. 062313.
- [14] Jing-Zheng Huang et al. "Quantum hacking of a continuous-variable quantum-key-distribution system using a wavelength attack". In: *Physical Review A* 87.6 (2013), p. 062329.

- [15] Daniel BS Soh et al. “Self-referenced continuous-variable quantum key distribution protocol”. In: *Physical Review X* 5.4 (2015), p. 041010.
- [16] Bing Qi et al. “Generating the local oscillator “locally” in continuous-variable quantum key distribution based on coherent detection”. In: *Physical Review X* 5.4 (2015), p. 041009.
- [17] *PDB48xC-AC Operation Manual*. Thorlabs. 2018.

1.4 Discrete Variables Quantum Tx/Rx system - Physical Layer

Student Name	:	Mariana Ramos
Starting Date	:	May 05, 2019
Goal	:	Algorithm for polarization drift compensation with discrete variables.
Directory	:	sdf/dv_polarization_encoding_system.

1.4.1 Simulation Analysis - Algorithm for polarization compensation

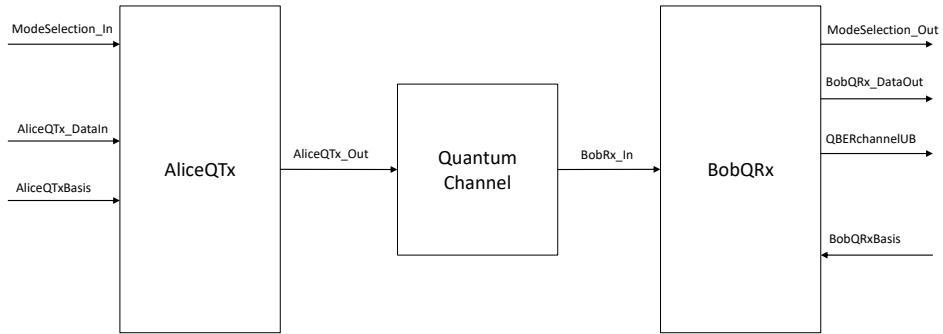


Figure 1.54: Block diagram of a DV quantum Tx/Rx system to emulate the physical layer.

This system emulates the physical layer of a quantum communication system. In this way, it comprises three super blocks. The first is the AliceQTx, which emulates the quantum transmitter of single-photons using polarization encoding. This block has three inputs: the ModeSelection, when it has value '0' selects the monitoring mode in which Alice transmits 1 control qubit in each N transmitted qubits. The remaining 99 qubits are data qubits which are encoded taking into account the other two inputs that are binary signals with basis, AliceQTxBasis, and other with the bit values, AliceQTxBits. Combining these two binary signals Alice is capable of encode 4 different states of polarization in two non-orthogonal basis in the photonStreamXY signal AliceQTxOut. When ModeSelection has the value '1' AliceQTx only transmits control qubits continuously according with a pre-defined sequence. The super block quantum channel accepts and outputs a photonStreamXY signal, and intends to emulate the polarization random drift throughout the optical fiber and the attenuation induced by it. The super block BobQRx receives the photonStreamXY signal and will measure and decode the single photon. This block outputs a binary signal ModeSelection that later inputs Alice super block. It also outputs BobQRxDataOut, which is a signal with Bob's measurements. This is a real signal has five possible value: 5 if it is a control qubit, and the upper layer should simply discard it, 3 if no-click happened, 2 if both

detectors clicked, 0 if bit zero was measured and 1 if bit one was measured. It also outputs a real signal with the quantum channel QBER, QBER_Qchannel, and the ModeSelection. BobQRx accepts a binary signal, BobQRxBasis, which comprises the basis for data qubits measurements.

1.4.2 Simulation Analysis - TestBench

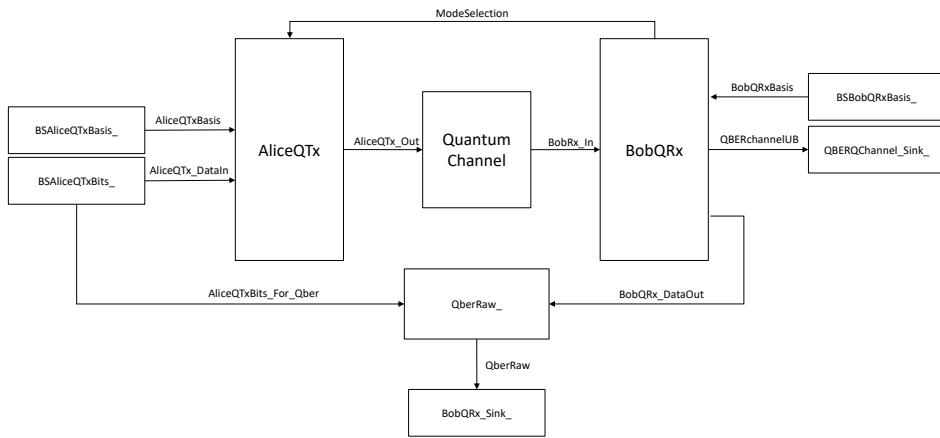


Figure 1.55: TestBench to test the DV quantum Tx/Rx system the physical layer emulator.

This testbench was developed for testing the quantum physical layer presented in figure 1.54. In this way, the `modeSelection` signal was directly connected from `BobQRx` to `AliceQTx`.

This system must be set with the following input parameters:

- `double bitRate{ 1e3 },`
- `double channelAttenuationDb{ 0.2 },`
- `sop_rotation_type channelModel{ sop_rotation_type::Stochastic },`
- `double PolarizationLinewidth{ 900e-9 },`
- `double fiberLength_km{ 50 },`
- `double nPhotonsData{ 1 },`
- `double nPhotonsControl{ 80 },`
- `int nIncrementControl{ 100 }.`

Furthermore, six more blocks were added with the following input parameters:

- `BSAliceQTxBasis_:`

- setMode(BinarySourceMode::DeterministicCyclic)
- setBitPeriod(1 / bitRate)
- setBitStream("1")
- setNumberOfBits(nBits);
- BSAliceQTxBits_:
 - setMode(BinarySourceMode::DeterministicCyclic)
 - setBitPeriod(1 / bitRate)
 - setBitStream("0");
- BSBobQRxBasis_:
 - setMode(BinarySourceMode::DeterministicCyclic)
 - setBitStream("1")
 - setBitPeriod(1 / bitRate);
- QberRaw_;
- BobQRx_Sink_;
- QBERQChannel_Sink_.

The three main blocks were defined with the following parameters:

- AliceQTx_:
 - setControlBasisSequence("0")
 - setControlBitsSequence("0")
 - setNumberOfPhotonsData(nPhotonsData)
 - setNumberOfPhotonsControl(nPhotonsControl)
 - setIncrementControl(nIncrementControl);
- QuantumChannel_ :
 - setAttenuationDbmPerKm(channelAttenuationDb)
 - setFiberLength(fiberLength_km)
 - setFiberModel(channelModel)
 - setPolarizationLinewidth(PolarizationLinewidth);
- BobQRx_:
 - setControlSequenceBits("0")
 - setControlBasisMeasurement("0")
 - setIncrementControl(nIncrementControl).

1.4.3 Open Issues

1.5 Frequency and Phase Recovery in CV-QC Systems

Student Name	: Daniel Pereira (01/05/2017 -)
Goal	: Theoretical and simulation study of techniques for frequency and phase recovery in CV-QC systems.
Directory	: sdf/cv_system
Related Links	: https://www.overleaf.com/14348245sjvzbkzpxwxf

Continuous Variable Quantum Communications (CV-QC) can encode information in the phase of weak coherent states. The weak nature of these states makes frequency and phase noise a substantial impairment, as such it is important to implement optimal techniques for their compensation.

1.5.1 Classical Frequency and Phase Recovery - State of the art

The traditional method for compensating noise in classical coherent communications is to use some form of phase-locked loop (PLL), synchronizing the frequency and the phase of the local oscillator (LO) with that of the transmitter laser [1]. Alternatively, frequency offset compensation and phase recovery can be performed in the digital domain by applying digital signal processing (DSP) [2]. It has been shown that DSP is more tolerant to laser phase noise than PLL based techniques [1]. This work will be focused on the study of DSP techniques for Continuous Variables Quantum Communications (CV-QCs) systems.

Frequency Mismatch Compensation Techniques

Frequency mismatch occurs when the central frequencies of the transmission and reception local oscillators aren't equal, as illustrated in Figure 1.56. A frequency offset of $\Delta f = f_{\text{LO}_{Tx}} - f_{\text{LO}_{Rx}}$ introduces a phase of $\omega_I Tn$ to the n th symbol relative to the 0th symbol, where

$$\omega_I = 2\pi \frac{f_{\text{LO}_{Tx}} - f_{\text{LO}_{Rx}}}{2} \quad (1.15)$$

is the intermediate frequency.

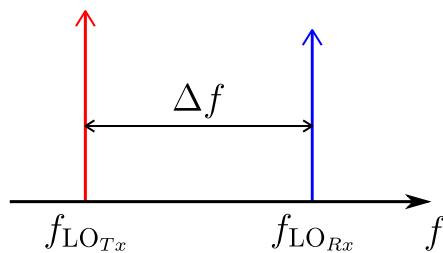


Figure 1.56: Visual representation of an optical frequency offset.

In order to show the action of the various frequency mismatch compensation techniques we will assume that an optical communication system has resulted in a signal given by

$$x(n) = x_{\text{sym}}(n)e^{i(\omega_I T n + \Delta\theta(n))}, \quad (1.16)$$

where $x_{\text{sym}}(n)$ is the complex modulated signal, $\Delta\theta(n)$ is the phase noise contribution and T is the symbol period. Compensation is accomplished by first estimating ω_I , yielding ω_{est} , and then removing the added phase for each symbol via [2]

$$\tilde{x}(n) = x(n)e^{-i\omega_{\text{est}}Tn}, \quad (1.17)$$

Frequency estimation can be accomplished through a blind frequency search method [3], in which the frequency is scanned over a predetermined range, symbol decisions made and the minimum square error used as the frequency-selection criteria. Alternatively the frequency can be estimated by evaluating the spectrum of the m th power of the input signal [4], which exhibits a peak at the frequency $m\omega_I$. A third option involves the usage of training symbols inserted periodically with the data payload [5]. These training symbols are used to remove the applied phase modulation. The resulting signal $s(n)$ is used to obtain an estimate of the offset frequency as

$$\omega_{\text{est}} = \frac{1}{T} \arg \left\{ \sum_{n=1}^L s(n)s^*(n-1) \right\} \quad (1.18)$$

Phase Mismatch Compensation Techniques

Deterministic Phase Mismatch

Phase mismatch occurs when the two employed oscillators have differing optical phases, as illustrated in Figure 1.57. This causes the phase modulation applied to one of the signals to be misread in the detection scheme.

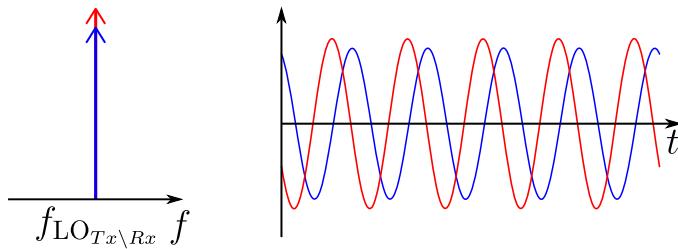


Figure 1.57: Frequency (left) and time (right) domain representation of a phase mismatch between two cosines of equal frequency.

Phase Noise

Phase noise arises from the non-zero linewidth of the transmission and reception local

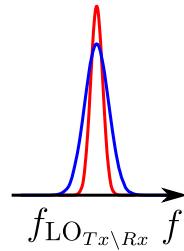


Figure 1.58: Phase noise arises from the non-zero linewidth of the transmission and reception oscillators, being present even if their central frequencies match.

oscillators, as illustrated in Figure 1.58. Phase noise consists of a time evolving phase mismatch, which can be modelled as a Weiner process described as

$$\phi(t_k) = \phi(t_{k-1}) + \Delta\phi(k), \quad (1.19)$$

where $\Delta\phi(k)$ are phase increments, which are independent and identically distributed random Gaussian variables with zero mean and variance

$$\sigma^2 = 2\pi\Delta\nu|t_k - t_{k-1}|, \quad (1.20)$$

where $\Delta\nu$ is the laser linewidth and $t_{k/k-1}$ are the sampling instants.

In order to show the action of the various phase recovery techniques we assume that a perfect frequency mismatch compensation has been accomplished for the signal (1.16), having resulted in

$$x(n) = x_{\text{sym}}(n)e^{i\Delta\theta(n)}, \quad (1.21)$$

where, again, $x_{\text{sym}}(n)$ is the complex, phase-modulated signal and $\Delta\theta(n)$ is the phase noise contribution.

The phase mismatch can be compensated either through blind estimation [6] or through pilot-aided estimation [7]. In the blind estimation method a m th-order non-linearity is used to remove the phase modulation, yielding an estimate of the phase mismatch [2]. The phase estimation is given by [6]

$$\Delta\theta_{\text{est}}(n) = \frac{1}{m} \arg \left\{ \frac{1}{2L+1} \sum_{l=-L}^L f(l) x^m(n+l) \right\}, \quad (1.22)$$

where $f(l)$ is a weighting function. The m th power removes the phase modulation, thus the only remaining phase is due to the phase noise contribution. The sum $-\frac{1}{2L+1} \sum_{l=-L}^L f(l) e^{i\Delta\theta(n+l)}$ functions then as a weighted average of the phase noise contributions up to L symbols before and L symbols after the current one.

In the pilot aided scheme, pre-agreed on symbols are inserted, time multiplexed, with the data payload at a pilot rate of $1/P$ (meaning one pilot symbol is inserted after $P - 1$ data symbols). The phase mismatch is determined from the pilot symbols and this estimation is used to compensate the phase mismatch on the data carrying symbols [7].

A alternative blind estimation scheme is proposed in [8], taking advantage of the unmodulated x^m signal to perform both frequency and phase mismatch compensation. The phase added by both the frequency and the phase mismatch is estimated as

$$\Delta\theta_{\text{est}}(n) = \Delta\theta_{\text{est}}(n-1) + \mu \arg \left\{ \frac{1}{L} \sum_{l=0}^{L-1} x^m(n-l)[x^m(n-l-1)]^* \right\}, \quad (1.23)$$

for every symbol except for the first/last L symbols, where μ corresponds to the integral gain. The frequency and phase mismatch compensation is accomplished by taking the uncompensated signal (1.16) and performing

$$x_{\text{rec}}(n) = x(n)e^{-i\sum_{k=L}^n \Delta\theta_{\text{est}}(k)T_s} \quad (1.24)$$

1.5.2 Quantum Frequency and Phase Recovery - State of the art

Due to the low amplitude of the quantum coherent states employed in CV-QC, any auxiliary processes, such as frequency and phase mismatch compensation, need to be employed in a non-intrusive manner. Another consequence of the low amplitude quantum coherent states is the necessity for high sensitivity, low thermal noise coherent receivers [9]. This requirement limits the bandwidth of the receivers [10, 11], which in turn limits the maximum operational baud rate of CV-QC systems. To avoid the added noise from disparities between the transmission and reception LO lasers, the first CV-QC protocols used a high intensity signal sent polarization multiplexed with the quantum signal, to be used as the LO in the detection scheme [12, 13]. This co-propagating technique presents a security risk, with attacks proposed that tamper with the LO's wavelength [14] or the shape of its pulses [15, 16, 17]. Locally generated LO (LLO) CV-QC protocols were proposed simultaneously in [18] and [19]. The usage of two different laser sources for the signal and the LO necessitates the application of frequency and phase recovery techniques.

Frequency Mismatch Compensation Techniques

In [18, 19, 20] the frequency mismatch was assumed to be minimal, accomplished by employing high quality tunable laser sources. The resulting noise introduced in this situation is treated as extra phase mismatch.

Given the low amplitude of the quantum pulses, frequency offset compensation methods cannot be applied directly to the quantum pulses [19]. To overcome this, a high intensity reference can be shared, with the application of both blind frequency search and frequency domain methods to this high intensity reference having been proposed [21].

Phase Mismatch Compensation Techniques

Given their low intensity, phase recovery methods cannot be applied directly to the quantum pulses [19]. In [18, 19] phase recovery is based on a pilot aided estimation scheme, with the pilot signal consisting of high intensity reference pulses and low intensity quantum

pulses carved sequentially from the continuously running transmission local oscillator. This scheme is dubbed LLO-sequential.

In [20] two LLO CV-QC schemes were proposed. The first one is similar to the pilot assisted phase recovery proposed in [18, 19], with the difference being in that the reference pulses are extracted from the same wavefront as the quantum pulses and time multiplexed with the latter by using a delayline, for this reason being dubbed LLO-delayline. In this setup the reception local oscillator is also pulsed and subjected to the delayline scheme. The second scheme proposed in [20] consisted of displacing the modulated constellation in phase space by a constant amplitude and a phase determined by the instantaneous phase of the transmission laser. This last scheme is dubbed LLO-displacement.

In [22] an alternative solution is presented in which the pilot signal is prepared from a carrier-suppressed optical single-sideband and sent polarization and frequency multiplexed with the quantum signal.

1.5.3 Open issues

- Available frequency and phase recovery DSP techniques for classical systems do not function well at low baud rates. Techniques that work at baud rates of the order of a few MBd are required.
- Available techniques do not function well for high frequency mismatches and low baud-rates. Robust techniques capable of compensating for large offset frequencies are necessary.
- Available CV-QC DSP techniques do not function well for large linewidths and low baud-rates. Techniques more capable of working at high phase noise levels are required.

1.5.4 Novelty

A novel frequency mismatch compensation technique is proposed which can operate at low baud rates, is resistant to large frequency deviations and large laser linewidths. In theoretical tests, our technique performs well in the presence of strong phase noise, yielding EVM results close to perfect frequency mismatch compensation (that is, compensation using the actual frequency mismatch value) for symbol rates as low as 1 kHz and for frequency mismatches as high as 10^9 Hz. The other studied techniques fail for symbol rates below 1 GHz and for frequency mismatches above 10^7 Hz.

1.5.5 Novel Frequency Mismatch Compensation Technique

Our novel frequency mismatch compensation technique uses a pilot signal similar to the ones employed in pilot assisted phase mismatch compensation techniques. In an pilot aided technique a reference signal (the pilot), composed of pre-agreed on symbols, is inserted, time multiplexed, with the data payload at a pre-agreed on rate. A visual representation of

the output of the modulation stage of a pilot-assisted LLO CV-QC scheme is presented in Figure 1.59. At the receiver stage, after coherent detection, the signal is described by

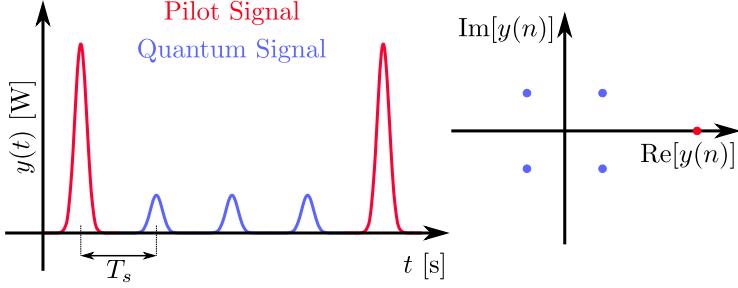


Figure 1.59: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted LLO CV-QC. Pilot and signal pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

$$x(t) = \begin{cases} x_q(t) = |x_q| \Pi(t) e^{i(\omega_I t + \phi(t) + \phi_0 - \varphi(t) - \varphi_0 + \theta(t))}, & n \neq jP \\ x_p(t) = |x_p| \Pi(t) e^{i(\omega_I t + \phi(t) + \phi_0 - \varphi(t) - \varphi_0)}, & n = jP \end{cases}, \quad n, j \in \mathbb{N}, \quad (1.25)$$

where $x_q(t)/x_p(t)$ is the time continuous quantum/pilot signal, P is the pilot rate, ω_I is the offset frequency, $\phi(t)/\varphi(t)$ is the instantaneous phase of the transmitter/receiver laser and $\theta(t)$ is the phase modulation at the sampling instant t and ϕ_0/φ_0 is the initial phase of the transmitter/receiver laser. $\Pi(t) = \sum M(t + nT_s)$ describes the time continuous amplitude modulation applied to both the quantum and pilot signals, while $M(t)$ describes the shape of each individual pulse. Sampling signal (1.25), assuming $\Pi(nT_s) = 1$ for any n , results in

$$x(n) = x(t = nT_s) = \begin{cases} x_q(n) = |x_q| e^{i(\omega_I nT_s + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq jP \\ x_p(n) = |x_p| e^{i(\omega_I nT_s + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = jP \end{cases}, \quad n, j \in \mathbb{N}, \quad (1.26)$$

where $x_q(n)/x_p(n)$ corresponds to the quantum/pilot constellation. The relative phase between the pilot and quantum signal constellations is known, so the pilot constellation can be used as a reference point from which to recover the phase encoded in the quantum signal without having to recover the absolute phase difference between the transmission and reception LOs.

Our novel technique functions by first multiplying an incremented pilot constellation by the complex conjugate of the unincremented pilot constellation yielding

$$x_{\text{freq}}(n = x_p(n + P)x_p^*(n)) = |x_p|^2 e^{i(\omega_I PT_s + \phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s)))}, \quad (1.27)$$

with the frequency estimation obtained by taking the expected value of the complex

argument of $x_{\text{freq}}(n)$ and dividing it by PT_s . Explicitly

$$\begin{aligned}\omega_{\text{est}} &= E \left[\frac{1}{PT_s} \arg(x_{\text{freq}}(n)) \right] \\ &= E \left[\omega_I + \frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s)) + 2k\pi}{PT_s} \right] \\ &= \omega_I + k\pi(PT_s)^{-1}, \quad n = jP, \quad n, j \in \mathbb{N},\end{aligned}\quad (1.28)$$

where the $k\pi(PT_s)^{-1}$ term is due to the non-injectiveness of the \arg function, forcing the value of $\omega_{\text{est}}PT_s$ to remain within the $[0, 2\pi]$ interval. The value of k is bounded by

$$0 \leq \omega_{\text{est}}PT_s < 2\pi \iff -\frac{\omega_I PT_s}{2\pi} \leq k < -\frac{\omega_I PT_s}{2\pi} + 1, \quad k \in \mathbb{Z}. \quad (1.29)$$

$\frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s))}{PT_s}$ is a normally distributed random variable with null mean and variance given by

$$\text{Var} \left[\frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s))}{PT_s} \right] = \frac{4\pi\Delta\nu PT_s}{(PT_s)^2} = 4\pi\Delta\nu(PT_s)^{-1}. \quad (1.30)$$

The frequency mismatch compensated signals are obtained by multiplying the constellations in (1.26) by $e^{-i\omega_{\text{est}}t_n}$, resulting in

$$\begin{aligned}\bar{x}(n) &= \begin{cases} \bar{x}_q(n) = |x_q| e^{i(-nk\pi P^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq jP \\ \bar{x}_p(n) = |x_p| e^{i(-nk\pi P^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = jP \end{cases}, \\ &\quad n, j \in \mathbb{N} \quad -\frac{\omega_I PT_s}{2\pi} \leq k \leq -\frac{\omega_I PT_s}{2\pi} + 1.\end{aligned}\quad (1.31)$$

The phase modulated signal can then be recovered by estimating the instantaneous phase difference using the pilot signal, this step depends on the applied LLO scheme.

The exactness of ω_{est} can be evaluated through a confidence interval. Assuming that the variance is not well known, being estimated from the experimental results, the confidence interval is given by

$$P \left(\omega_I - t \frac{s}{\sqrt{n}} < \omega_{\text{est}} < \omega_I + t \frac{s}{\sqrt{n}} \right) = 1 - \alpha, \quad (1.32)$$

where t is the $1 - \frac{\alpha}{2}$ 'th percentile of the Student's t-distribution and s^2 is the estimated value of the variance. For an offset frequency of $\omega_I = 10$ MHz, assuming a pilot rate $P = 2$ and that $n = 2 \times 10^5$ samples were used, using a variance estimate given by $s^2 = 2\pi\Delta\nu T_s^{-1}$, the 99 % confidence interval is less than 1 % of ω_I .

1.5.6 Implementation issues

The proposed compensation technique is compatible with all pilot assisted phase mismatch compensation techniques, with few differences observed for each case. In the interest of demonstrating both this compatibility and the distinctions between the different pilot assisted LLO techniques, we now proceed with a detailed study of multiple pilot assisted CV-QC methods.

Optimal amplitude of Pilot Signal

Given the sensitivity of CV-QC protocols to phase noise, the Pilot Signal's amplitude should be optimized in order to minimize the uncertainty of the measured instantaneous phase difference between the transmission and reception LOs. Assuming that the quadrature noise does not change meaningfully for an increasing amplitude, the phase uncertainty will drop as the Pilot signal's amplitude increases. This effect is explained graphically in Figure 1.60.

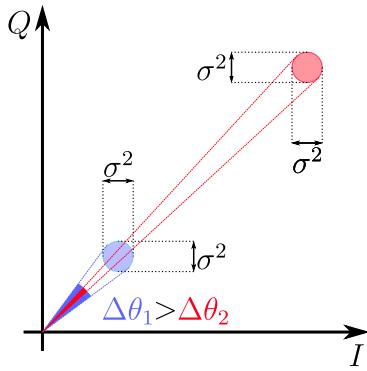


Figure 1.60: The phase uncertainty for Gaussian distributions with the same variance drops with an increase in amplitude.

Given the low intensity of the quantum signal, a very powerful LO is necessary in order to extract any phase information from it. Given this, the pilot signal, having an amplitude much larger than the quantum signal, can very easily saturate the output of the coherent receiver. The results of this study are presented in Figure 1.61. The values for the gain, thermal noise and saturation voltage were extracted from [23].

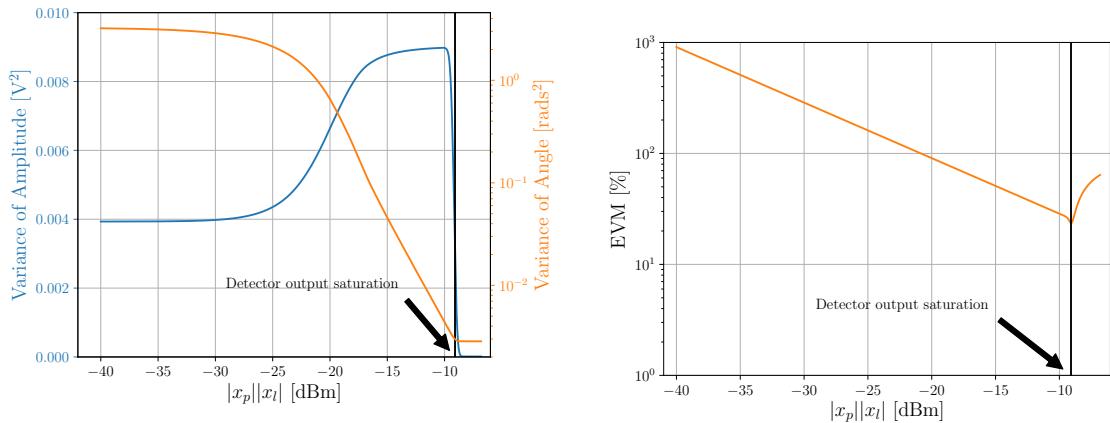


Figure 1.61: Evolution of the phase variance (left) and EVM (right) with the product of the amplitudes of the pilot signal $|x_p|$ and LO $|x_l|$.

LLO-sequential

The initially proposed LLO schemes, a simplified diagram of which is presented in Figure 1.62, carved the pilot and signal pulses sequentially from the free running transmission local oscillator, thus being dubbed LLO-sequential. Coherent detection is performed with a similarly free running reception local oscillator. The phase difference between the transmission and reception laser for each signal symbol is estimated from an average of the instantaneous phase of the previous and ensuing pilot pulses.

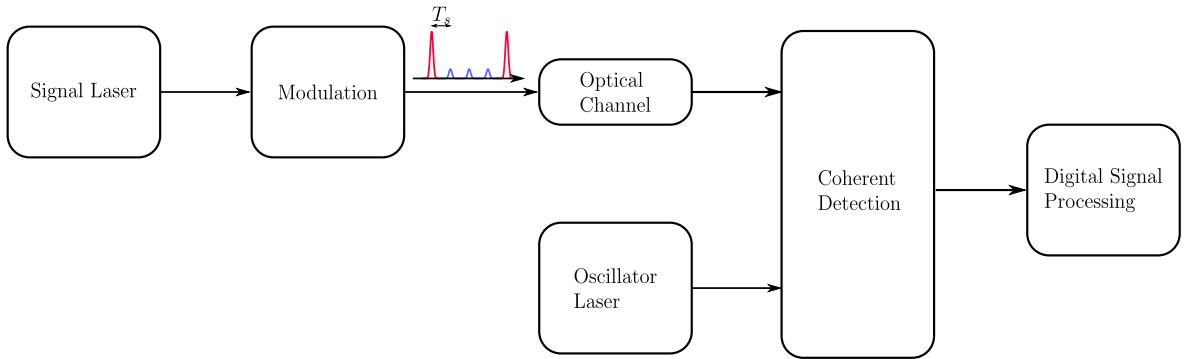


Figure 1.62: Simplified diagram of a LLO-sequential CV-QC scheme. First proposed in [18, 19]

Recalling the frequency mismatch compensated constellations (1.31), the phase mismatch of the n -th quantum signal constellation point is estimated from an average of the instantaneous phase of the previous and the following pilot signal pulses, yielding

$$\Theta_{\text{est}} = -n\pi k P^{-1} + \phi_0 - \varphi_0 + \frac{d[\phi((n+P-d)T_s) - \varphi((n+P-d)T_s)] + (P-d)[\phi((n-d)T_s) - \varphi((n-d)T_s)]}{P}, \quad (1.33)$$

and then removing this phase from the quantum signal constellation. The final recovered constellation is given by

$$\begin{aligned} \tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} = \\ &= |x_q|e^{i\left(\frac{d[\phi(nT_s) - \varphi(nT_s) - \phi((n+P-d)T_s) + \varphi((n+P-d)T_s)] + (P-d)[\phi(nT_s) - \varphi(nT_s) - \phi((n-d)T_s) + \varphi((n-d)T_s)]}{P} + \theta(nT_s)\right)} \\ &= |x_q|e^{i(\Delta\Theta(nT_s) + \theta(nT_s))}, \quad n \neq jP, \quad n, j \in \mathbb{N}, \end{aligned} \quad (1.34)$$

where d is defined as the time distance between the quantum signal pulse and the previous pilot signal pulse, measured in number of symbol periods, and $\Delta\Theta$ is a randomly distributed Gaussian variable with null mean and variance given by

$$\text{Var}[\Delta\Theta(nT_s)] = 2\pi T_s (\nu_T + \nu_R) \frac{d(P-d)}{P}, \quad (1.35)$$

where $\Delta\nu_T$ and $\Delta\nu_R$ are the linewidth of the transmission and reception local oscillators, respectively. From this result we see that the remaining variance will depend on the relation

between the distance to the previous pilot pulse d and the pilot rate P and that different values of d will have different remaining variances. In the LLO-sequential CV-QC scheme the uncertainty introduced by the non-injectiveness of the arg function has no effect in the recovered constellation.

LLO-delayline

Building on the LLO-sequential method proposed in [18, 19], a scheme was proposed in which the pilot and quantum signal pulses are obtained from the same wavefront, employing a delayline scheme to accomplish the necessary time multiplexing. In the reception stage a similar method is used to obtain the LO reference pulses used in the detection of corresponding pilot and quantum signal pulses from the same wavefront. In this

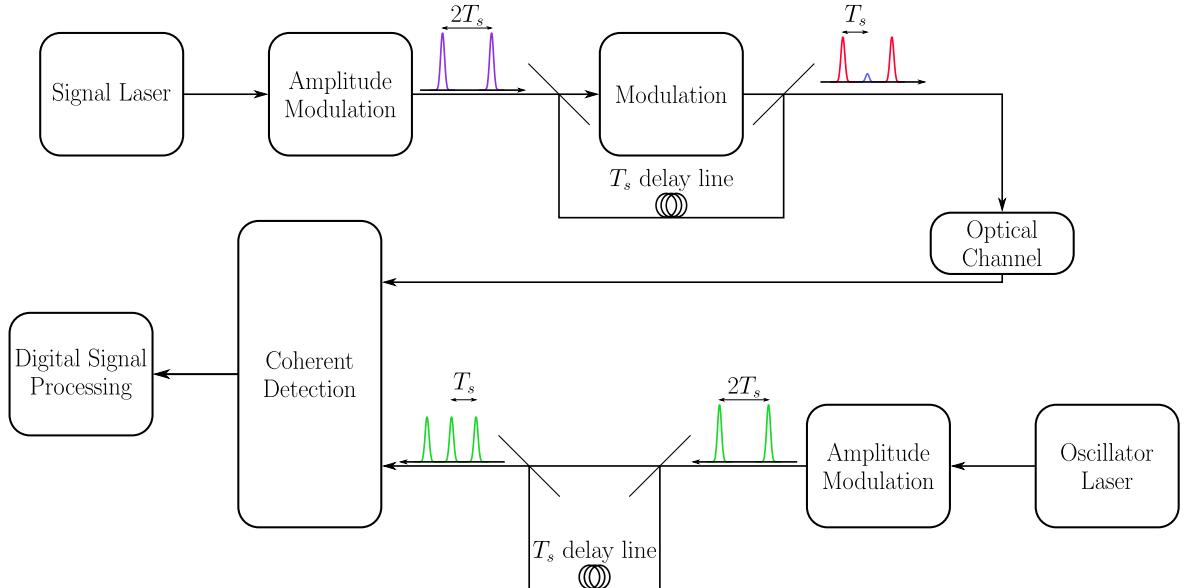


Figure 1.63: Simplified diagram of a partial LLO-delayline CV-QC scheme.

scheme the pilot rate is set to 2, so the frequency mismatch compensated constellations (1.31) simplify to

$$\bar{x}(n) = \begin{cases} \bar{x}_q(n) = |x_q| e^{i(-nk\pi 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq 2j \\ \bar{x}_p(n) = |x_p| e^{i(-nk\pi 2^{-1} + \phi((n+1)T_s) + \phi_0 - \varphi((n+1)T_s) - \varphi_0)}, & n = 2j \end{cases},$$

$$n, j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1. \quad (1.36)$$

The phase mismatch of the n -th quantum signal constellation point is estimated from the previous pilot signal pulse, which yields

$$\Theta_{\text{est}} = -(n-1)\pi k 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0. \quad (1.37)$$

The final recovered constellation is then given by

$$\begin{aligned}\tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} \\ &= |x_q|e^{i(\theta(nT_s)-k\pi 2^{-1})}, \quad n \neq jP, \quad n, j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1.\end{aligned}\quad (1.38)$$

This scheme entirely removes the phase noise introduced by both lasers, apart from phase fluctuations that might occur in the delay lines. In the LLO-delayline CV-QC scheme the uncertainty introduced by the non-injectiveness of the arg function will add a phase of either $\frac{\pi}{2}$, π , $\frac{3\pi}{2}$ or 2π . To allow for this, the proceeding operations must be repeated in order to test all 4 options.

Partial LLO-delayline

An alternate scheme is presented in Figure 1.64, being a combination of the transmission setup of the LLO-delayline with the reception setup of the LLO-sequential. This system removes the phase noise from the transmission laser, while accepting the phase noise of the reception laser. In this scheme the pilot rate is set to 2, so the frequency mismatch

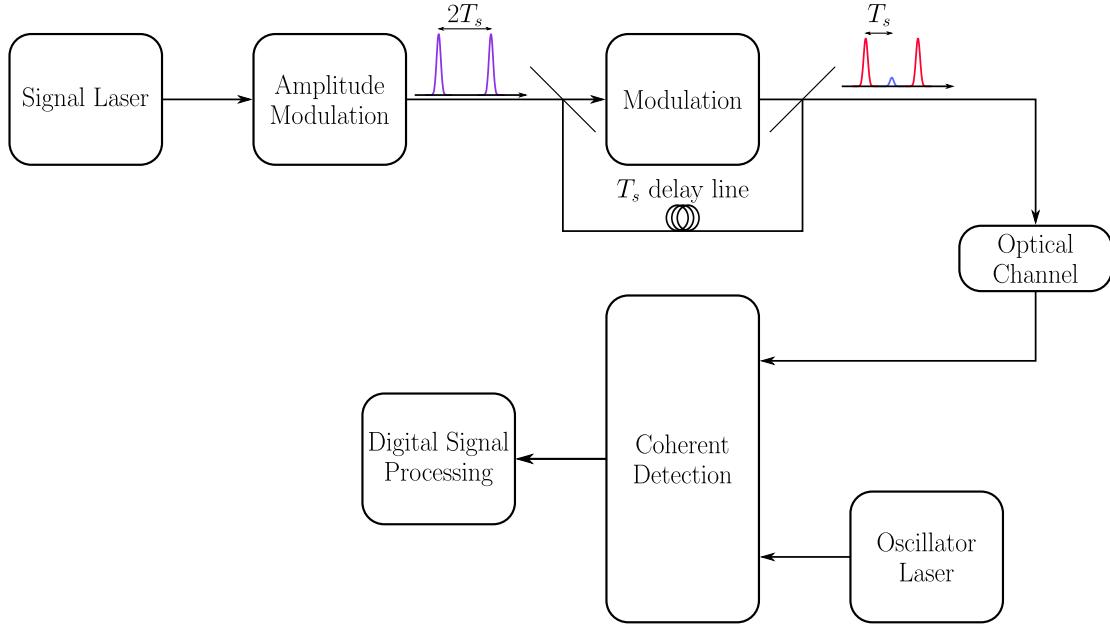


Figure 1.64: Simplified diagram of a partial LLO-delayline CV-QC scheme.

compensated constellations (1.31) simplify to

$$\begin{aligned}\bar{x}(n) &= \begin{cases} \bar{x}_q(n) = |x_q|e^{i(-nk\pi 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq 2j \\ \bar{x}_p(n) = |x_p|e^{i(-nk\pi 2^{-1} + \phi((n+1)T_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = 2j \end{cases}, \\ &n, j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1.\end{aligned}\quad (1.39)$$

The phase mismatch of the n -th quantum signal constellation point is estimated from the previous pilot signal pulse, which yields

$$\Theta_{\text{est}} = -(n-1)\pi k 2^{-1} + \phi(nT_s) + \phi_0 - \varphi((n-1)T_s) - \varphi_0. \quad (1.40)$$

The final recovered constellation is then given by

$$\begin{aligned} \tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} \\ &= |x_q|e^{i(\varphi((n-1)T_s)-\varphi(nT_s)+\theta(nT_s)-k\pi 2^{-1})}, \quad n \neq jP, \quad n, \quad j \in \mathbb{N} \end{aligned} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1, \quad (1.41)$$

The phase noise introduced by the reception laser will have a variance of

$$\sigma^2 = 2\pi\Delta\nu_R T_s, \quad (1.42)$$

where $\Delta\nu_R$ is the linewidth of the reception local oscillator and T_s is the symbol period. As in the LLO-delayline CV-QC scheme, the uncertainty introduced by the non-injectiveness of the arg function will add a phase of either $\frac{\pi}{2}$, π , $\frac{3\pi}{2}$ or 2π . To allow for this, the proceeding operations must be repeated in order to test all 4 options.

1.5.7 Error Vector Magnitude

The figure of merit employed for the comparisons presented in this document will be the Error Vector Magnitude (EVM), where the error is evaluated as the relation between the magnitude of the error vector e_V and the magnitude of the vector of the ideal symbol position ref_V

$$EVM(\%) = 100 \sqrt{\frac{|e_V|}{|\text{ref}_V|}} = 100 \sqrt{\frac{|\text{m}_V - \text{ref}_V|}{|\text{ref}_V|}} \quad (1.43)$$

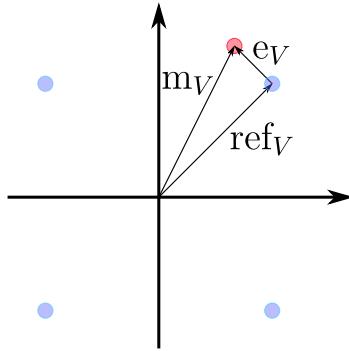


Figure 1.65: Visual representation of the EVM method in a QPSK constellation. The ideal constellation points are presented in blue, while an example for an actual measured symbol is presented in red.

1.5.8 Comparative analysis of the frequency ranging techniques

Three of the frequency mismatch compensation techniques described in Section 1.5.1 were applied on simulated signal and reference constellations, similar to the ones presented in (1.26). The blind frequency search method was applied to the pilot signal following the method described in [3], utilizing a coarse step size of 10 MHz followed by a fine step size of 1 MHz. The spectral analysis method was applied via evaluation of the spectrum of the reference constellation. The alternative blind method was applied as described in Section 1.5.1.

The EVM of the recovered constellation (1.31) was computed in function of the symbol frequency. The results for this study are presented in Figure 1.66. Only our novel method

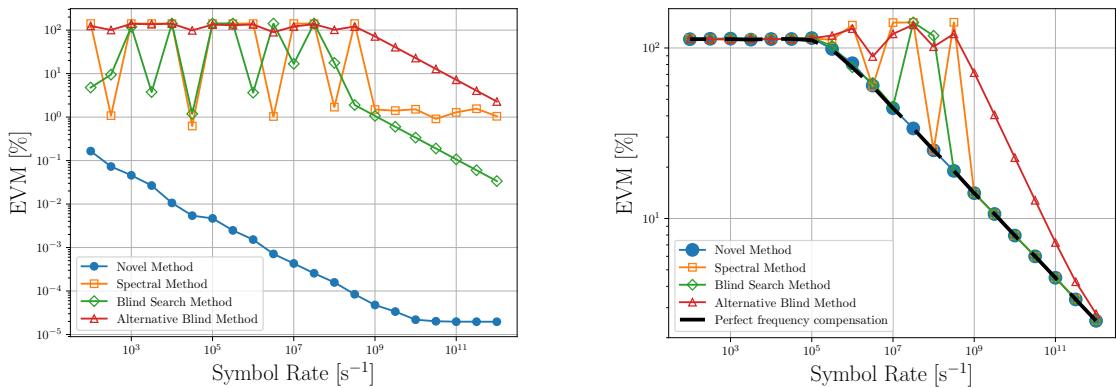


Figure 1.66: EVM in function of the sampling frequency for 4 different frequency ranging techniques without (left) and with (right) phase noise, in the latter case a laser linewidth of 100 kHz was assumed and the partial LLO-delayline phase recovery scheme was applied in order to allow for an easier EVM comparison. In the comparison figure with phase noise, the EVM observed without any frequency mismatch is included as black dashes to indicate the best case scenario. A frequency mismatch of 519213710 Hz (value chosen at random) was assumed.

was able to compensate for the frequency mismatch at the lower sampling frequencies attempted (note the swift increase in EVM for sampling frequencies below 10^9 Hz for the other methods in Figure 1.66-left). In the study without phase noise, for all the tested sampling frequencies, our novel method returns consistently EVM values 4 orders of magnitude below the classical alternatives, thus delivering results of much greater precision. From the study with phase noise we see that our novel method is very resistant to high levels of phase noise, performing very close to perfect frequency mismatch compensation for almost all the symbol rates tested.

A comparison of the EVM performance, in function of the frequency mismatch between the two employed lasers, between our novel frequency compensation scheme and the alternative blind frequency search scheme is presented in Figure 1.67. The alternative blind frequency search method was chosen for this comparison because it allows for a cleaner,

more readable figure. In both situations (with and without phase noise), our novel method

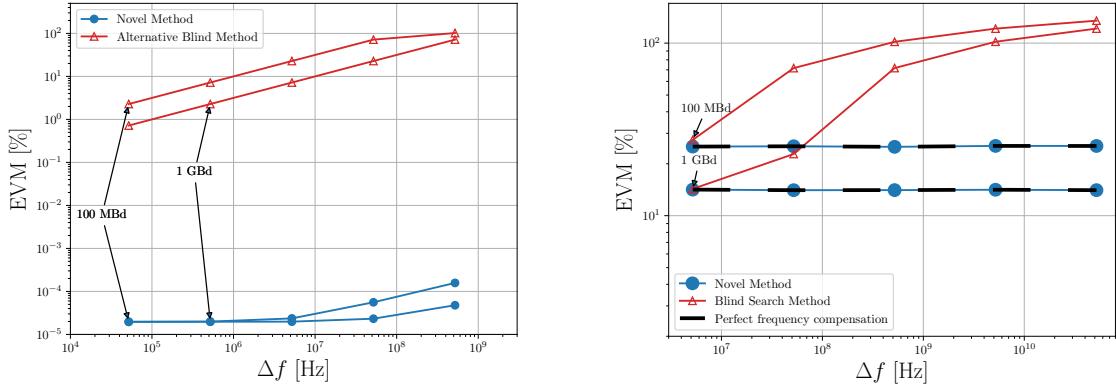


Figure 1.67: EVM in function of the frequency mismatch for 2 different frequency ranging techniques (stars correspond to our novel frequency ranging method and triangles to the alternative blind frequency search technique) and 2 different symbol rates (identified in both figures), without (left) and with (right) phase noise, in the later case a laser linewidth of 100 kHz was assumed and the partial LLO-delayline phase recovery scheme was applied in order to allow for an easier EVM comparison. In the comparison figure with phase noise, the EVM observed without any frequency mismatch is included as black dashes to indicate the best case scenario.

returns results orders of magnitude more precise than the alternative. As observed before, in the situation with phase noise our method performs very close to perfect frequency mismatch compensation, even for very high frequency mismatches.

Our novel technique has a computational complexity of $O(n)$. In comparison, the blind frequency estimation technique requires carrier phase recovery, bit decision and computing of the least mean square error for every test frequency, even assuming a frequency and phase recovery and decision algorithm with a total computational complexity of $O(n)$, the added least mean square error step will push the computational complexity above the one observed in our novel technique. The spectrum aided technique requires a Fourier Transform to be performed, which in itself has a computational complexity of $O(n \log(n))$, this is above the computational complexity of our novel technique for any n greater than e .

1.5.9 Simulation Analysis

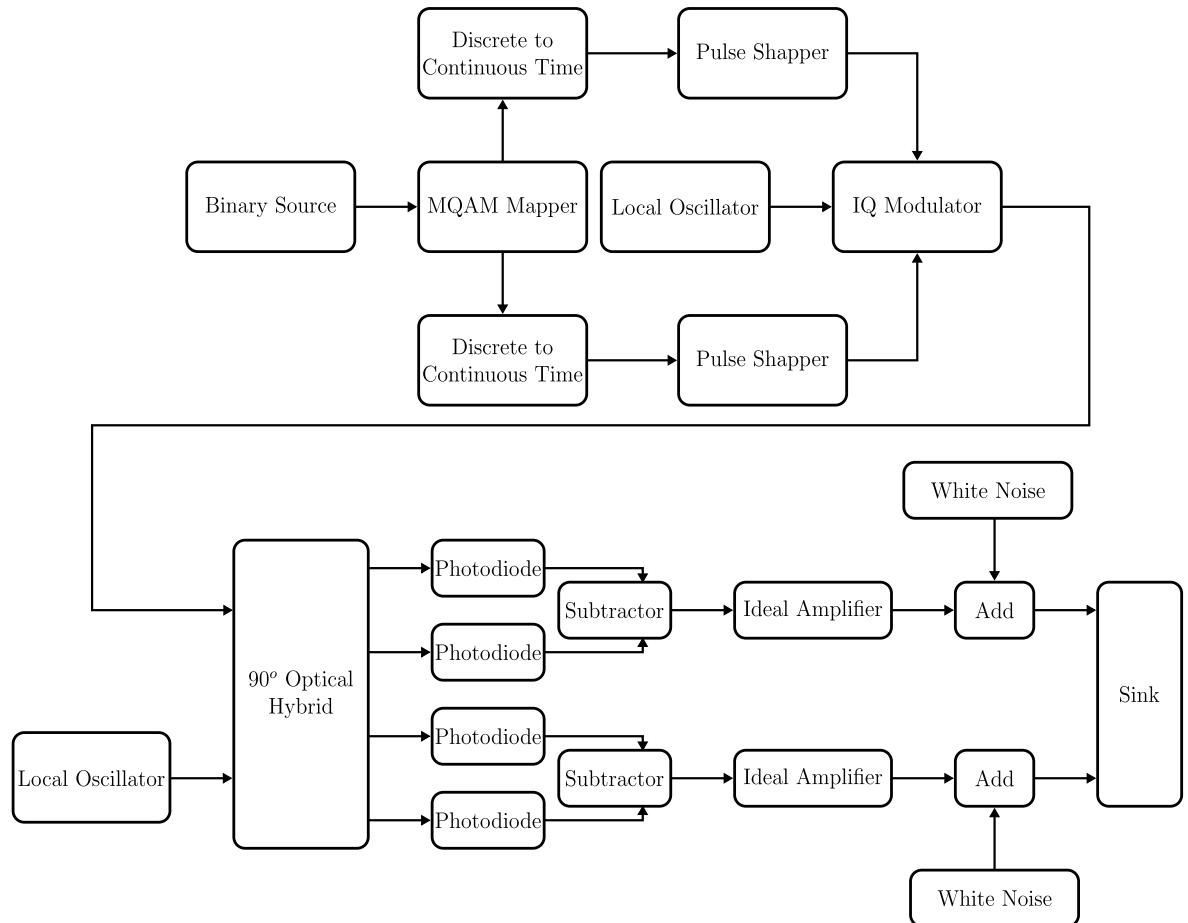


Figure 1.68: Simulation setup for the LLO-sequential CV-QC transmission system employed in testing our novel frequency mismatch compensation system.

The state to bit correspondence, agreed by Alice and Bob beforehand, also presented in Figure ??, is as follows:

Simulation Blocks	
Necessary	Available
Binary Source	✓
MQAM Mapper	✓
Pulse Shaper	✓
Discrete to Continuous Time	✓
Local Oscillator	✓
IQ Modulator	✓
Attenuator	
90° Optical Hybrid	✓
Photodiode	✓
Subtractor	
Ideal Amplifier	✓
Add	✓
White Noise	✓
Sink	✓

1.5.10 Simulation Results

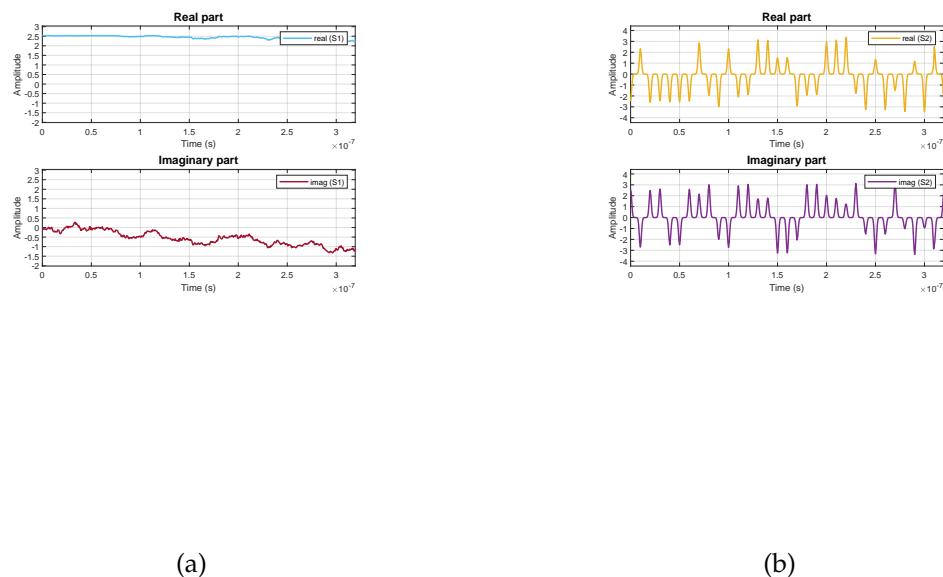


Figure 1.69

1.5.11 New study

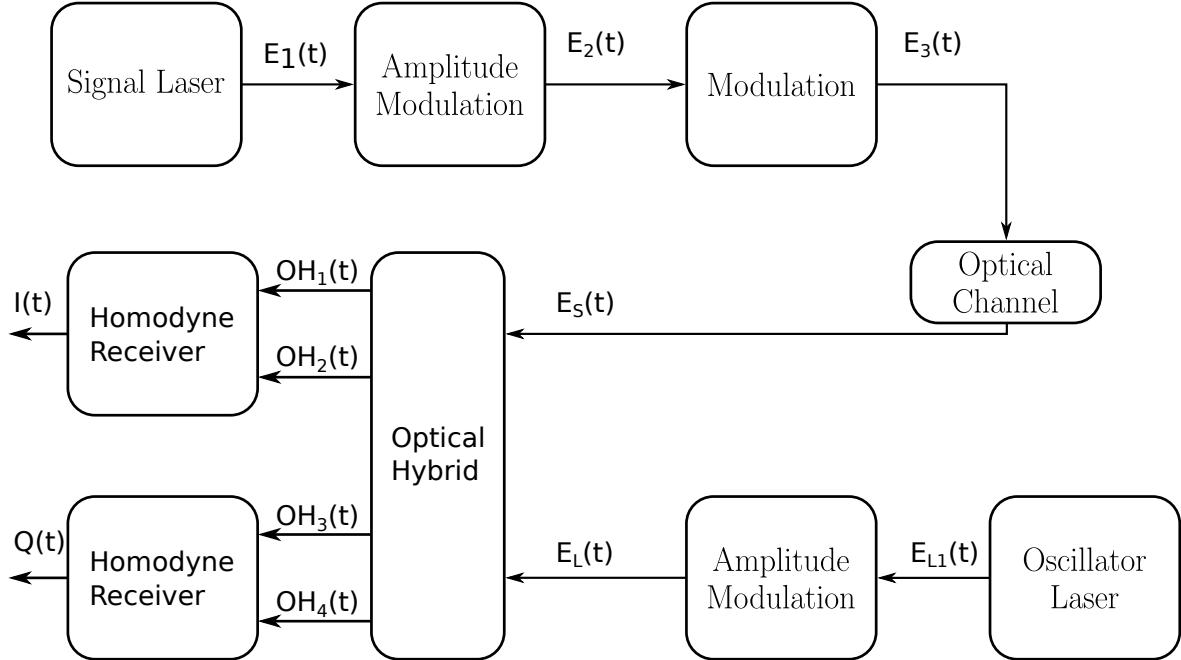


Figure 1.70: Sequential block diagram

Assuming a frequency mismatch between the transmitter and receiver local oscillators of $\Delta\omega$ and a frequency mismatch of $\Delta\epsilon$, this leads to the following sampled signal

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n + \Delta\epsilon(t_n)]}, & n = mR_P \end{cases}, \quad (1.44)$$

where m and $n \in \mathbb{N}$, $x_{q/p}(t_n)$ and $\theta_{q/p}(t_n)$ are, respectively, the amplitude and phase of the quantum/pilot signal at the instant t_n .

The phase mismatch of the n th quantum symbol is compensated by taking a weighted average of the previous and ensuing pilots. For an example, let's consider $R_P = 2$ and that only the two closest pilots are used. The weight for each pilot is the same and the phase estimate is given by

$$\begin{aligned} \hat{\Theta} &= \frac{\Delta\omega(t_n - T_s) + \Delta\epsilon(t_n - T_s) + \Delta\omega(t_n + T_s) + \Delta\epsilon(t_n + T_s)}{2} \\ &= \Delta\omega t_n + \frac{\Delta\epsilon(t_n - T_s) + \Delta\epsilon(t_n + T_s)}{2} \end{aligned} \quad (1.45)$$

The recovered constellation is given by

$$\begin{aligned}\bar{y}_r(t_n) &= x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n)]}e^{-i\Theta} \\ &= x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n) - \Delta\omega t_n - \frac{\Delta\epsilon(t_n - T_s) + \Delta\epsilon(t_n + T_s)}{2}]} \\ &= x_q(t_n)e^{i[\theta_q(t_n) + \frac{\Delta\epsilon(t_n) - \Delta\epsilon(t_n - T_s)}{2} + \frac{\Delta\epsilon(t_n) - \Delta\epsilon(t_n + T_s)}{2}]}\end{aligned}\quad (1.46)$$

Frequency mismatch is fully compensated, some phase mismatch remains.

Delay Line

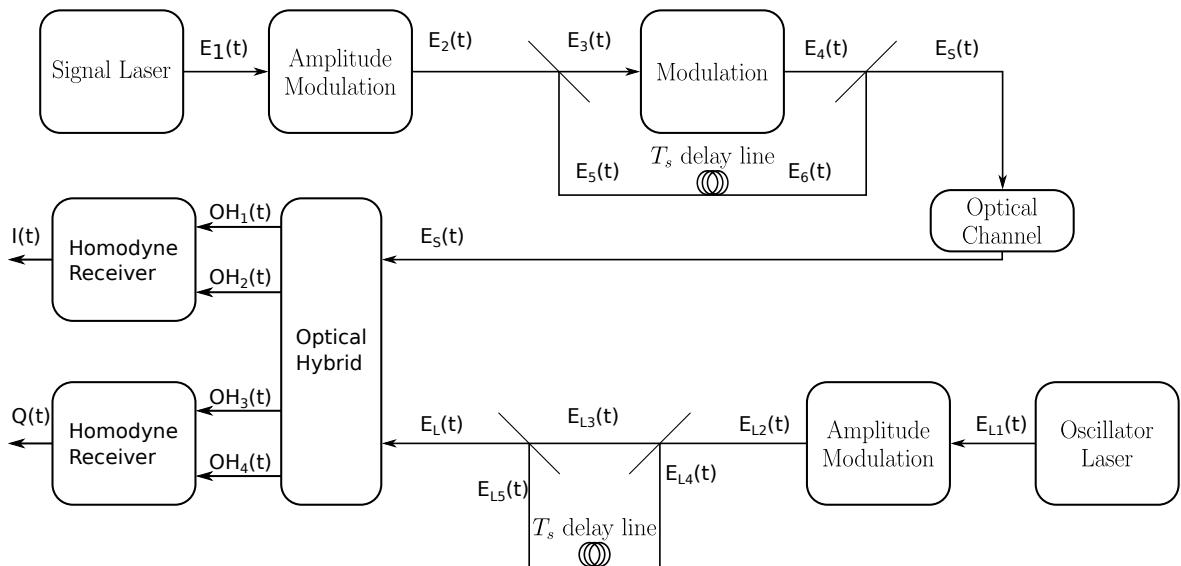


Figure 1.71: Delay line block diagram

The laser output can be described by:

$$E_1(t) = |E_s|e^{i(\omega_{st} + \epsilon_s(t))}. \quad (1.47)$$

This signal is then pulsed by the following function

$$P(t) = \begin{cases} 1, & \text{for } t = 0, 2T_s, 4T_s, 6T_s, \dots \\ 0, & \text{for } t = T, 3T_s, 5T_s, 7T_s, \dots \end{cases}, \quad (1.48)$$

the specific shape of the pulses is not very relevant here, only the values indicated are of interest.

$$E_2(t) = P(t)|E_s|e^{i(\omega_{st} + \epsilon_s(t))}. \quad (1.49)$$

The pulsed signal is split at a beamsplitter, resulting in

$$\begin{bmatrix} E_3(t) \\ E_5(t) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_2(t) \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} P(t)|E_s|e^{i(\omega_{st} + \epsilon_s(t))} \\ P(t)|E_s|e^{i(\omega_{st} + \epsilon_s(t))} \end{bmatrix} \quad (1.50)$$

Signal $E_3(t)$ is then phase modulated, resulting in

$$E_4(t) = E_3(t)e^{i\theta(t)} = P(t)|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} \quad (1.51)$$

Signal $E_5(t)$ is delayed by T_s , resulting in

$$E_6(t) = E_5(t + T_s) = P(t + T_s)|E_s|e^{i(\omega_S(t + T_s) + \epsilon_S(t + T_s))} \quad (1.52)$$

Signals $E_4(t)$ and $E_6(t)$ are combined in a beamsplitter, only one output is monitored, the other is set to $SINK(t)$.

$$\begin{aligned} \begin{bmatrix} E_S(t) \\ SINK(t) \end{bmatrix} &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_4(t) \\ E_6(t) \end{bmatrix} \\ \iff E_S(t) &= \frac{1}{2} \left[P(t)|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + P(t + T_s)|E_s|e^{i(\omega_S(t + T_s) + \epsilon_S(t + T_s))} \right] \end{aligned} \quad (1.53)$$

The local oscillator laser output can be described by

$$E_{L1}(t) = |E_L|e^{i(\omega_L t + \epsilon_L(t))}. \quad (1.54)$$

As previously, this signal is then pulsed

$$E_{L2}(t) = P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))}, \quad (1.55)$$

the pulsed signal is passed through a beam splitter

$$\begin{bmatrix} E_{L3}(t) \\ E_{L4}(t) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_{L2}(t) \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} \\ P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} \end{bmatrix}. \quad (1.56)$$

Signal $E_{L4}(t)$ is then delayed by T_s , resulting in

$$E_{L5}(t) = E_{L4}(t + T_s) = P(t + T_s)|E_L|e^{i(\omega_L(t + T_s) + \epsilon_L(t + T_s))}, \quad (1.57)$$

signals $E_{L5}(t)$ and $E_{L3}(t)$ are then combined, only one output is monitored, the other is set to $SINK(t)$.

$$\begin{aligned} \begin{bmatrix} E_L(t) \\ SINK(t) \end{bmatrix} &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_{L3}(t) \\ E_{L5}(t) \end{bmatrix} \\ \iff E_L(t) &= \frac{1}{2} \left[P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} + P(t + T_s)|E_L|e^{i(\omega_L(t + T_s) + \epsilon_L(t + T_s))} \right] \end{aligned} \quad (1.58)$$

Signals $E_S(t)$ and $E_L(t)$ are then combined in an optical hybrid, yielding

$$\begin{aligned} \begin{bmatrix} OH_1(t) \\ OH_2(t) \\ OH_3(t) \\ OH_4(t) \end{bmatrix} &= \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & i \\ 1 & -i \end{bmatrix} \begin{bmatrix} E_S(t) \\ E_L(t) \end{bmatrix} \\ &= \begin{bmatrix} P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + |E_L|e^{i(\omega_L t + \epsilon_L(t))}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} + |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s))}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} - |E_L|e^{i(\omega_L t + \epsilon_L(t))}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} - |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s))}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + |E_L|e^{i(\omega_L t + \epsilon_L(t) + \frac{\pi}{2})}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} + |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s) + \frac{\pi}{2})}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} - |E_L|e^{i(\omega_L t + \epsilon_L(t) + \frac{\pi}{2})}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} - |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s) + \frac{\pi}{2})}] \end{bmatrix} \quad (1.59) \end{aligned}$$

Finally, these signals are evaluated in balanced homodyne receivers, yielding

$$\begin{aligned} I(t) &= OH_1(t)OH_1^*(t) - OH_2(t)OH_2^*(t) \\ &= \frac{P(t)|E_s||E_L|}{4} \cos(\Delta\omega t + \Delta\epsilon(t) + \theta(t)) + \frac{P(t+T_s)|E_s||E_L|}{4} \cos(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s)) \\ Q(t) &= OH_3(t)OH_3^*(t) - OH_4(t)OH_4^*(t) \\ &= \frac{P(t)|E_s||E_L|}{4} \sin(\Delta\omega t + \Delta\epsilon(t) + \theta(t)) + \frac{P(t+T_s)|E_s||E_L|}{4} \sin(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s)), \end{aligned} \quad (1.60)$$

where $\Delta\omega = \omega_S - \omega_L$ and $\Delta\epsilon(t) = \epsilon_S(t) - \epsilon_L(t)$.

$$IQ(t) = \frac{P(t)|E_s||E_L|}{4} e^{i(\Delta\omega t + \Delta\epsilon(t) + \theta(t))} + \frac{P(t+T_s)|E_s||E_L|}{4} e^{i(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s))} \quad (1.61)$$

Signal

$$IQ(t = 2T, 4T, 6t, \dots) = \frac{|E_s||E_L|}{4} e^{i(\Delta\omega t + \Delta\epsilon(t) + \theta(t))} \quad (1.62)$$

Pilot

$$IQ(t = T, 3T, 5t, \dots) = \frac{|E_s||E_L|}{4} e^{i(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s))} \quad (1.63)$$

Each pilot is used to remove the phase noise of the next signal pulse, for the signal pulse at $t = 2T$ we have, for example

$$\begin{aligned} \widehat{IQ}(2T) &= IQ(2T)IQ^*(T) \\ &= \left(\frac{|E_s||E_L|}{4} \right)^2 e^{i(\Delta\omega 2T_s + \Delta\epsilon(2T_s) + \theta(2T_s) - \Delta\omega(T_s + T_s) - \Delta\epsilon(T_s + T_s))} \\ &= \left(\frac{|E_s||E_L|}{4} \right)^2 e^{i\theta(2T_s)}, \end{aligned} \quad (1.64)$$

Frequency and phase mismatch are fully compensated.

1.5.12 Future work

Comparative analysis of the phase mismatch compensation techniques

The EVM of the recovered constellations (1.34) and (1.41) was evaluated in function of the symbol rate and for multiple laser linewidths and is presented in Figure 1.72. To allow for a better comparison, the pilot rate for the LLO-sequential technique was assumed to take the value $P = 2$, from this choice the remaining variance (1.35) simplifies to

$$\text{Var}[\Delta\Theta(nT_s)] = \pi T_s(\nu_T + \nu_R). \quad (1.65)$$

The EVM was computed by generating the constellations (1.34) and (1.41) with the respective remaining phase noise (1.65) and (1.42) and computing the EVM from these simulated results.

The main experimental setup used is presented in Figure ??.

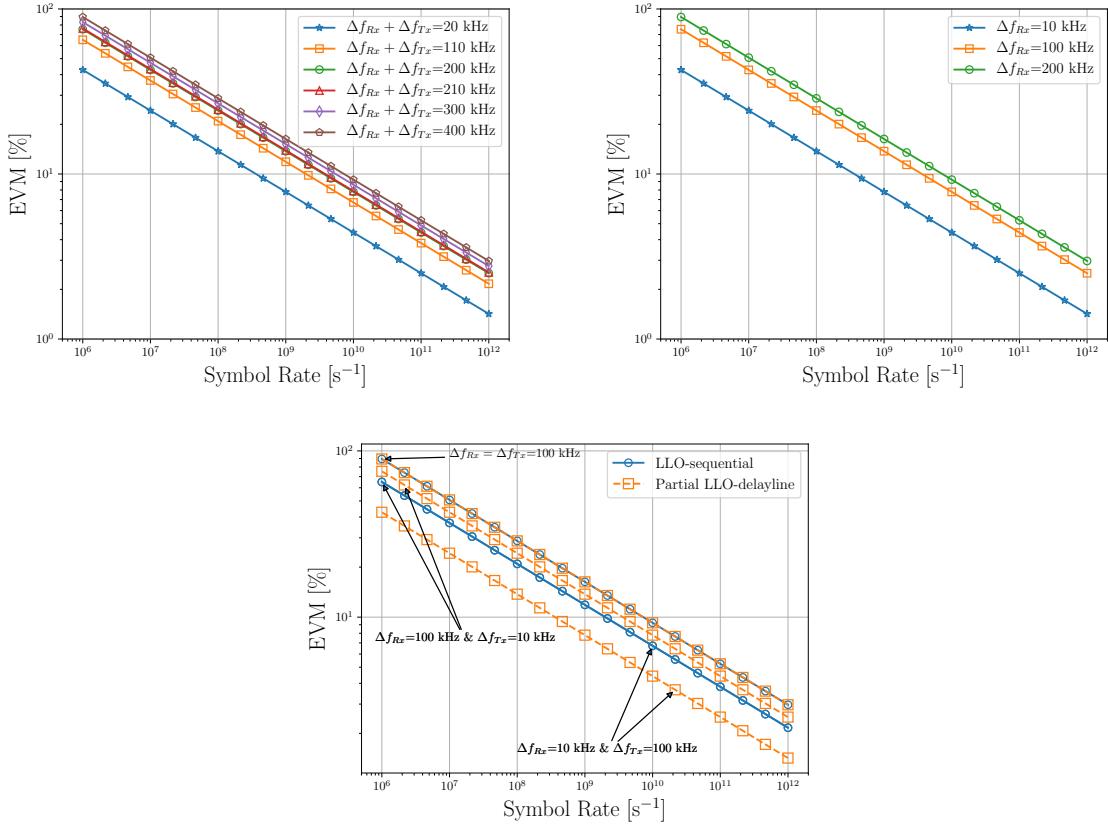


Figure 1.72: EVM in function of the sampling frequency for the LLO-sequential (top left) and partial LLO-delayline (top right) phase mismatch compensation schemes. The bottom figure compares the EVM of the LLO-sequential and the partial LLO-delayline for 3 different combinations of transmission and receptor laser linewidths.

The performance of the LLO-sequential technique does not depend on the individual values of the linewidths, but rather on the value of their sum, i.e. the EVM of a system with $\Delta f_{T_x} = 10$ kHz and $\Delta f_{R_x} = 100$ kHz will be equal to that of a system with $\Delta f_{T_x} = 100$ kHz and $\Delta f_{R_x} = 10$ kHz (as is visible in Figure 1.72-bottom).

The performance of the partial LLO-delayline scheme, in relation to that of the LLO-sequential scheme will be

- equal, if the transmission and reception lasers have the same linewidths;
- worse, if the transmission laser has a narrower linewidth than the reception laser;
- better, if the transmission laser has a wider linewidth than the reception laser.

All of these results are visible in Figure 1.72-bottom

Memory aided sequential referenced LLO CV-QC

We here study the advantages of employing a memory aided sequentially-referenced LLO CV-QC system. Figure 1.73 is included to facilitate this study.

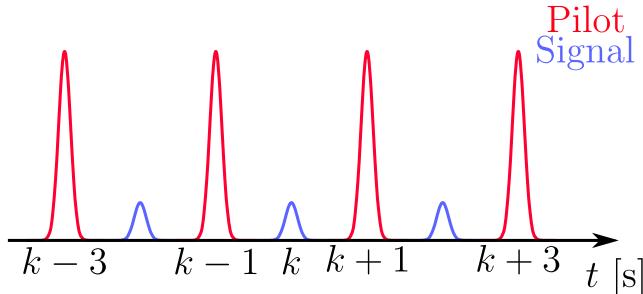


Figure 1.73: Time dependence of a sequential LLO scheme. Indexing in the figure should be considered as relative to the central signal pulse.

Assuming this time dependence to be sampled at the maximum of each pulse, this yields signal and pilot constellations given by

$$x_s(k) = |x_s| e^{i(\theta(t_k) + \phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)}, \quad k \text{ is odd}; \quad (1.66)$$

$$x_p(k) = |x_p| e^{i(\phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)}, \quad k \text{ is even}; \quad (1.67)$$

where $\theta(t_k)$ is the phase encoded in the k th signal symbol, $\phi(t_k)/\varphi(t_k)$ is the instantaneous phase of the transmission/reception laser at the instant t_k and ϕ_0/φ_0 is the initial phase of the transmission/reception laser.

The phase difference between the transmission and reception lasers for the k th signal symbol can be estimated by using a weighted average of the preceding and ensuing pilot symbols. The originally proposed sequentially-referenced LLO utilized only the pilot

symbols immediately before and after, corresponding in the Figure 1.73 to utilizing the pilot pulses $k \pm 1$ to compensate the phase difference in the k th signal symbol. In this case it is easy to understand that both pilot symbols should have the same weight in the estimation, since they both have the same time distance from the point being estimated. This method yields the recovered constellation

$$\begin{aligned} x(k) &= |x_s| e^{i(\theta(t_k) + \phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)} e^{-i\left(\frac{\phi(t_{k-1}) + \phi_0 - \varphi(t_{k-1}) - \varphi_0}{2} + \frac{\phi(t_{k+1}) + \phi_0 - \varphi(t_{k+1}) - \varphi_0}{2}\right)} \\ &= |x_s| e^{i\left(\theta(t_k) + \frac{\phi(t_k) - \phi(t_{k-1}) + \phi(t_k) - \phi(t_{k+1})}{2} - \frac{\varphi(t_k) - \varphi(t_{k-1}) + \varphi(t_k) - \varphi(t_{k+1})}{2}\right)}, \quad k \text{ is odd.} \end{aligned} \quad (1.68)$$

The phase variance of this recovered constellation is given by

$$\begin{aligned} \text{Var} [\arg(x(k))] &= \frac{1}{2} \text{Var} [\phi(t_k) - \phi(t_{k-1})] + \frac{1}{2} \text{Var} [\varphi(t_k) - \varphi(t_{k-1})] \\ &= \pi(\Delta\nu_T + \Delta\nu_R)T_s, \end{aligned} \quad (1.69)$$

where $\Delta\nu_T / \Delta\nu_R$ is the transmission/reception laser's linewidth and T_s is the symbol period (note that $t_k - t_{k-1} = T_s$).

Lets now consider that four pilot pulses are used to estimate the phase difference in the k th signal symbol (pulses $k \pm 1$ and $k \pm 3$ in Figure 1.73). The averaging weight of each pulse is not immediately clear, so for now they will be termed μ_1 and μ_2 for the pulses $k \pm 1$ and $k \pm 3$ respectively. The remaining phase variance in this case is given by

$$\begin{aligned} \text{Var} [\arg(x(k))] &= 2\mu_1^2 \text{Var} [\phi(t_k) - \phi(t_{k-1})] + 2\mu_1^2 \text{Var} [\varphi(t_k) - \varphi(t_{k-1})] \\ &\quad + 2\mu_2^2 \text{Var} [\phi(t_k) - \phi(t_{k-3})] + 2\mu_2^2 \text{Var} [\varphi(t_k) - \varphi(t_{k-3})] \\ &= \pi(\Delta\nu_T + \Delta\nu_R)T_s 4(\mu_1^2 + 3\mu_2^2), \quad 2\mu_1 + 2\mu_2 = 1. \end{aligned} \quad (1.70)$$

The values for μ_1 and μ_2 can be optimized within their constrains in order to minimize the value of $\text{Var} [\arg(x(k))]$. The solution for this optimization yields the values $\mu_1 = \frac{3}{8}$ and $\mu_2 = \frac{3}{8}$, which corresponds to a remaining variance of

$$\text{Var} [\arg(x(k))] = \frac{3}{4}\pi(\Delta\nu_T + \Delta\nu_R)T_s, \quad (1.71)$$

which is lower than the remaining variance observed when using only two pilot pulses.

Performing a couple more iterations quickly reveals that, for an arbitrary number of pilot pulses $2n$ (n before and n after the signal symbol) employed in estimating the phase difference in the k th signal symbol yields a remaining phase variance given by

$$\text{Var} [\arg(x(k))] = \pi(\Delta\nu_T + \Delta\nu_R)T_s 4 \sum_{i=1}^n (2i-1)\mu_i^2, \quad 2 \sum_{i=1}^n \mu_i = 1. \quad (1.72)$$

Assuming the properties of the optical system remain the same (i.e. the value of $\pi(\Delta\nu_T + \Delta\nu_R)T_s$ is constant), we have already seen we can reduce the remaining variance by utilizing four pilot pulses instead of only two. We can now prove that we can reduce the remaining

phase variance by an arbitrary amount by employing an appropriate number of pilot pulses n . We can prove this by computing

$$\lim_{n \rightarrow \infty} \min_{2 \sum_{i=1}^n \mu_i = 1} 4 \sum_{i=1}^n (2i - 1) \mu_i^2 = \lim_{n \rightarrow \infty} \min_{g(\mu_i) = 0} f(\mu_i) \quad (1.73)$$

This can be accomplished by using the Lagrangian multiplier method, which can maximize or minimize a function $f(x_i)$ subject to a constraint $g(x_i)$, where x_i is any group of variables, by studying the roots of the Lagrangian

$$\mathcal{L} = f(x_i) \mp \lambda g(x_i), \quad (1.74)$$

where λ is dubbed the Lagrangian multiplier and serves only as an auxiliary variable. The roots of this Lagrangian are evaluated in relation to both its original variables x_i and to the Lagrangian multiplier λ by application of the gradient operator

$$\nabla_{x_i, \lambda} \mathcal{L} = 0 \iff \begin{cases} \frac{\partial \mathcal{L}}{\partial x_i} = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} = 0 \end{cases}. \quad (1.75)$$

Applying this to our optimization problem yields

$$\begin{cases} \frac{\partial(4 \sum_{i=1}^n (2i-1) \mu_i^2 + \lambda(2 \sum_{i=1}^n \mu_i - 1))}{\partial \mu_i} = 0 & \left\{ \begin{array}{l} 4 \sum_{i=1}^n (2i-1) \mu_i + \lambda \sum_{i=1}^n 1 = 0 \\ 2 \sum_{i=1}^n \mu_i = 1 \end{array} \right. \\ \frac{\partial(4 \sum_{i=1}^n (2i-1) \mu_i^2 + \lambda(2 \sum_{i=1}^n \mu_i - 1))}{\partial \lambda} = 0 & \\ \left. \begin{array}{l} 4 \sum_{i=1}^n (2i-1) \mu_i = - \sum_{i=1}^n \lambda \\ - \end{array} \right\} & \left. \begin{array}{l} 4(2i-1) \mu_i = -\lambda \\ - \end{array} \right\} \left. \begin{array}{l} \mu_i = -\frac{\lambda}{4(2i-1)} \\ - \end{array} \right\} \\ \left. \begin{array}{l} - \\ \sum_{i=1}^n \frac{-\lambda}{2(2i-1)} = 1 \end{array} \right\} & \left. \begin{array}{l} - \\ \lambda = -\frac{2}{\sum_{i=1}^n \frac{1}{2i-1}} = -\frac{2}{s} \end{array} \right\} \left. \begin{array}{l} \mu_i = -\frac{\lambda}{2s(2i-1)} \\ - \end{array} \right\} \end{cases} \quad (1.76)$$

Substituting this result into $f(\mu_i)$ yields

$$4 \sum_{i=1}^n (2i-1) \frac{1}{4s^2(2i-1)^2} = \frac{1}{\left(\sum_{i=1}^n \frac{1}{2i-1}\right)^2} \sum_{i=1}^n \frac{1}{2i-1} = \left(\sum_{i=1}^n \frac{1}{2i-1}\right)^{-1}, \quad (1.77)$$

this is the solution to the constrained minimization problem. This result is plotted function of the memory size in Figure 1.74. Finally, inputting this result into (1.73) yields

$$\lim_{n \rightarrow \infty} \left(\sum_{i=1}^n \frac{1}{2i-1}\right)^{-1} = 0, \quad (1.78)$$

which is the result we set out to obtain.

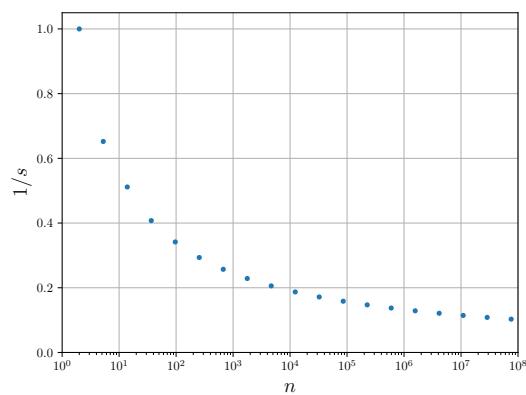


Figure 1.74: Reduction of the observed remaining variance in function of the size of the memory employed.

References

- [1] Ezra Ip and Joseph M Kahn. "Feedforward carrier recovery for coherent optical communications". In: *Journal of Lightwave Technology* 25.9 (2007), pp. 2675–2692.
- [2] Md Saifuddin Faruk and Seb J Savory. "Digital signal processing for coherent transceivers employing multilevel formats". In: *Journal of Lightwave Technology* 35.5 (2017), pp. 1125–1141.
- [3] Xiang Zhou et al. "64-Tb/s, 8 b/s/Hz, PDM-36QAM transmission over 320 km using both pre-and post-transmission digital signal processing". In: *Journal of Lightwave Technology* 29.4 (2011), pp. 571–577.
- [4] Mehrez Selmi, Yves Jaouën, and Philippe Ciblat. "Accurate digital frequency offset estimator for coherent PolMux QAM transmission systems". In: *Optical Communication, 2009. ECOC'09. 35th European Conference on*. IEEE. 2009, pp. 1–2.
- [5] Xian Zhou, Xue Chen, and Keping Long. "Wide-range frequency offset estimation algorithm for optical coherent systems using training sequence". In: *IEEE Photonics Technology Letters* 24.1 (2012), pp. 82–84.
- [6] Michael G Taylor. "Phase estimation methods for optical coherent detection using digital signal processing". In: *Journal of Lightwave Technology* 27.7 (2009), pp. 901–914.
- [7] Maurizio Magarini et al. "Pilot-symbols-aided carrier-phase recovery for 100-G PM-QPSK digital coherent receivers". In: *IEEE Photonics Technology Letters* 24.9 (2012), p. 739.
- [8] Ricardo M Ferreira et al. "Optimized Carrier Frequency and Phase Recovery Based on Blind M th Power Schemes". In: *IEEE Photonics Technology Letters* 28.21 (2016), pp. 2439–2442.
- [9] MB Costa Silva et al. "Homodyne detection for quantum key distribution: an alternative to photon counting in BB84 protocol". In: *Photonics North 2006*. Vol. 6343. International Society for Optics and Photonics. 2006, 63431R.
- [10] John Bertrand Johnson. "Thermal agitation of electricity in conductors". In: *Physical review* 32.1 (1928), p. 97.
- [11] Harry Nyquist. "Thermal agitation of electric charge in conductors". In: *Physical review* 32.1 (1928), p. 110.
- [12] T. C. Ralph. "Continuous variable quantum cryptography". In: *Phys. Rev. A* 61 (1 Dec. 1999), p. 010303. DOI: [10.1103/PhysRevA.61.010303](https://doi.org/10.1103/PhysRevA.61.010303). URL: <https://link.aps.org/doi/10.1103/PhysRevA.61.010303>.
- [13] Mark Hillery. "Quantum cryptography with squeezed states". In: *Physical Review A* 61.2 (2000), p. 022309.
- [14] Xiang-Chun Ma et al. "Wavelength attack on practical continuous-variable quantum-key-distribution system with a heterodyne protocol". In: *Physical Review A* 87.5 (2013), p. 052309.

- [15] Jing-Zheng Huang et al. "Quantum hacking on quantum key distribution using homodyne detection". In: *Physical Review A* 89.3 (2014), p. 032304.
- [16] Paul Jouguet, Sébastien Kunz-Jacques, and Eleni Diamanti. "Preventing calibration attacks on the local oscillator in continuous-variable quantum key distribution". In: *Physical Review A* 87.6 (2013), p. 062313.
- [17] Jing-Zheng Huang et al. "Quantum hacking of a continuous-variable quantum-key-distribution system using a wavelength attack". In: *Physical Review A* 87.6 (2013), p. 062329.
- [18] Daniel BS Soh et al. "Self-referenced continuous-variable quantum key distribution protocol". In: *Physical Review X* 5.4 (2015), p. 041010.
- [19] Bing Qi et al. "Generating the local oscillator "locally" in continuous-variable quantum key distribution based on coherent detection". In: *Physical Review X* 5.4 (2015), p. 041009.
- [20] Adrien Marie and Romain Alléaume. "Self-coherent phase reference sharing for continuous-variable quantum key distribution". In: *Physical Review A* 95.1 (2017), p. 012316.
- [21] S Kleis and CG Schaeffer. "Comparison of frequency estimation methods for heterodyne quantum communications". In: *Photonic Networks; 18. ITG-Symposium; Proceedings of*. VDE. 2017, pp. 1–3.
- [22] Fabian Laudenbach et al. "Pilot-assisted intradyne reception for high-speed continuous-variable quantum key distribution with true local oscillator". In: *arXiv preprint arXiv:1712.10242* (2017).
- [23] *PDB48xC-AC Operation Manual*. Thorlabs. 2018.

Chapter 2

Library

2.1 ADC

Header File	:	adc_*.h
Source File	:	adc_*.cpp
Version	:	20180423 (Celestino Martins)

This super block block simulates an analog-to-digital converter (ADC), including signal resample and quantization. It receives two real input signal and outputs two real signal with the sampling rate defined by ADC sampling rate, which is externally configured using the resample function, and quantized signal into a given discrete values.

Input Parameters

Parameter	Unity	Type	Values	Default
samplingPeriod	–	double	any	—
rFactor	–	double	any	1
resolution	bits	double	any	<i>inf</i>
maxValue	volts	double	any	1.5
minValue	volts	double	any	-1.5

Table 2.1: ADC input parameters

Methods

```

ADC(vector<Signal *> &InputSig, vector<Signal *> &OutputSig);

//void setResampleSamplingPeriod(double sPeriod) B1.setSamplingPeriod(sPeriod);
B2.setSamplingPeriod(sPeriod);;

void setResampleOutRateFactor(double OUTsRate) B01.setOutRateFactor(OUTsRate);
B02.setOutRateFactor(OUTsRate);

void setQuantizerSamplingPeriod(double sPeriod) B03.setSamplingPeriod(sPeriod);
B04.setSamplingPeriod(sPeriod);

void setSamplingPeriod(double sPeriod) B03.setSamplingPeriod(sPeriod);;

void setQuantizerResolution(double nbits) B03.setResolution(nbits);
B04.setResolution(nbits);

void setQuantizer.MaxValue(double maxvalue) B03.setMaxValue(maxvalue);
B04.setMaxValue(maxvalue);

void setQuantizer.MinValue(double minvalue) B03.setMinValue(minvalue);
B04.setMinValue(minvalue);

```

Functional description

This super block is composed of two blocks, resample and quantizer. It can perform the signal resample according to the defined input parameter *rFactor* and signal quantization according to the defined input parameter *nBits*.

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.2 Add

Header File	:	add.h
Source File	:	add.cpp
Version	:	20180118

Input Parameters

This block takes no parameters.

Functional Description

This block accepts two signals and outputs one signal built from a sum of the two inputs. The input and output signals must be of the same type, if this is not the case the block returns an error.

Input Signals

Number: 2

Type: Real, Complex or Complex_XY signal (ContinuousTimeContinuousAmplitude)

Output Signals

Number: 1

Type: Real, Complex or Complex_XY signal (ContinuousTimeContinuousAmplitude)

2.3 Arithmetic Encoder

Header File	:	arithmetic_encoder.h
Source File	:	arithmetic_encoder.cpp
Version	:	20180719 (Diogo Barros)

This block implements the integer version of the arithmetic encoding algorithm, given the symbol counts, the number of bits per symbol and the number of symbols to encode. The block takes a binary input stream and outputs the encoded binary stream.

Input Parameters

Parameter	Type	Values	Default
SeqLen	unsigned int	any	--
BitsPerSymb	unsigned int	any	--
SymbCounts	vector<unsigned int>	any	--

Table 2.2: Arithmetic encoder block input parameters.

Methods

```
bool runBlock(void)

void initialize(void);

void init(const unsigned int& SeqLen, const unsigned int& BitsPerSymb, const
          vector<unsigned int>& SymbCounts);
```

Functional description

This block implements the integer version of the arithmetic encoding algorithm, given the symbol counts, the number of bits per symbol and the number of symbols to encode.

Input Signals

Number: 1

Output Signals

Number: 1

Type: binary

Examples

Sugestions for future improvement

2.4 Arithmetic Decoder

Header File	:	arithmetic_decoder.h
Source File	:	arithmetic_decoder.cpp
Version	:	20180719 (Diogo Barros)

This block implements the integer version of the arithmetic decoding algorithm, given the symbol counts, the number of bits per symbol and the number of symbols to encode. The block takes an encoded binary input stream and outputs the decoded binary stream.

Input Parameters

Parameter	Type	Values	Default
SeqLen	unsigned int	any	--
BitsPerSymb	unsigned int	any	--
SymbCounts	vector<unsigned int>	any	--

Table 2.3: Arithmetic decoding block input parameters.

Methods

```
bool runBlock(void)

void initialize(void);

void init(const unsigned int& SeqLen, const unsigned int& BitsPerSymb, const
          vector<unsigned int>& SymbCounts);
```

Functional description

This block implements the integer version of the arithmetic decoding algorithm, given the symbol counts, the number of bits per symbol and the number of symbols to encode. The block takes an encoded binary input stream and outputs the decoded binary stream.

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.5 Alice QKD

This block is the processor for Alice does all tasks that she needs. This block accepts binary, messages, and real continuous time signals. It produces messages, binary and real discrete time signals.

Input Parameters

- double RateOfPhotons{1e3}
- int StringPhotonsLength{ 12 }

Methods

```
AliceQKD (vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) :  
Block(inputSignals, outputSignals) {};  
void initialize(void);  
bool runBlock(void);  
void setRateOfPhotons(double RPhotons) { RateOfPhotons = RPhotons; }; double const  
getRateOfPhotons(void) { return RateOfPhotons; };  
void setStringPhotonsLength(int pLength) { StringPhotonsLength = pLength; }; int const  
getStringPhotonsLength(void) { return StringPhotonsLength; };
```

Functional description

This block receives a sequence of binary numbers (1's or 0's) and a clock signal which will set the rate of the signals produced to generate single polarized photons. The real discrete time signal **SA_1** is generated based on the clock signal and the real discrete time signal **SA_2** is generated based on the random sequence of bits received through the signal **NUM_A**. This last sequence is analysed by the polarizer in pairs of bits in which each pair has a bit for basis choice and other for direction choice.

This block also produces classical messages signals to send to Bob as well as binary messages to the mutual information block with information about the photons it sent.

Input Signals

Number : 3

Type : Binary, Real Continuous Time and Messages signals.

Output Signals

Number : 3

Type : Binary, Real Discrete Time and Messages signals.

Examples

Sugestions for future improvement

2.6 Alice QKD

Header Files	:	alice_qkd_*.h
Source Files	:	alice_qkd_*.cpp
Version	:	20190410 (Andoni Santos)

This block is the main processor for Alice.

AliceQKD is a superblock intended work as Alice's main processor. It performs a series of functions, including classical channel communication with Bob and data processing for basis reconciliation, error correction and privacy amplification.

Input Signals

- 0 - Binary signal with Alice's key, to be transmitted to Bob.
- 1 - Binary signal with Alice's basis, that will be used to encode the key.
- 2 - Message signal for receiving incoming messages.

Output Signals

- 0 - Binary signal with Alice's key, to be transmitted to Bob.
- 1 - Binary signal with Alice's basis, that will be used to encode the key.
- 2 - Binary signal with Alice's final key, after the post processing.
- 3 - Message signal for sending outgoing messages.
- 4 - (Optional) Binary signal with sifted key.

Input Parameters

Note: all the parameters names are only one word. Some of the parameter's names have been split due to space constraints on the table, but their name is the concatenation of multiple lines shown. This may also be true for some default values.

Parameter	Type	Values	Default	Description
buffSize	unsigned long int	> 0	512	Buffer size for most internal signals
maximumEC SyndromeSize	t_integer	> 0	8192	Buffer size for signals where it is important to transmit large quantities of data.

dispBits Interval	t_integer	> 0	50000	Number of bits transmitted before updating the visual interface
folderName	string	valid path	"signals"	Name of the folder where signals and output files are saved
reportFile Name	string	valid file name	"alice_KeysReport .txt"	Name of the file where the overall report for this block is saved
toPrint	bool	[true, false]	true	If true, it prints the console interface in regular intervals
ipAddress	string	any string	"running locally"	the URL or IP address of the remote computer. Only for the purpose of displaying it in the interface
photonsPer Pulse	double	any	0.1	Average number of photons sent in each transmitted pulse. Only for the purpose of displaying it in the interface.

Methods

```
void initialize(void)
```

```
    bool runBlock(void)
```

```
        AliceQKD(initializer_list<Signal * > inputSignals, initializer_list<Signal * > outputSignals)
```

```
        AliceQKD(initializer_list<Signal*> inputSignals, initializer_list<Signal*> outputSignals,  
        string sFolderName)
```

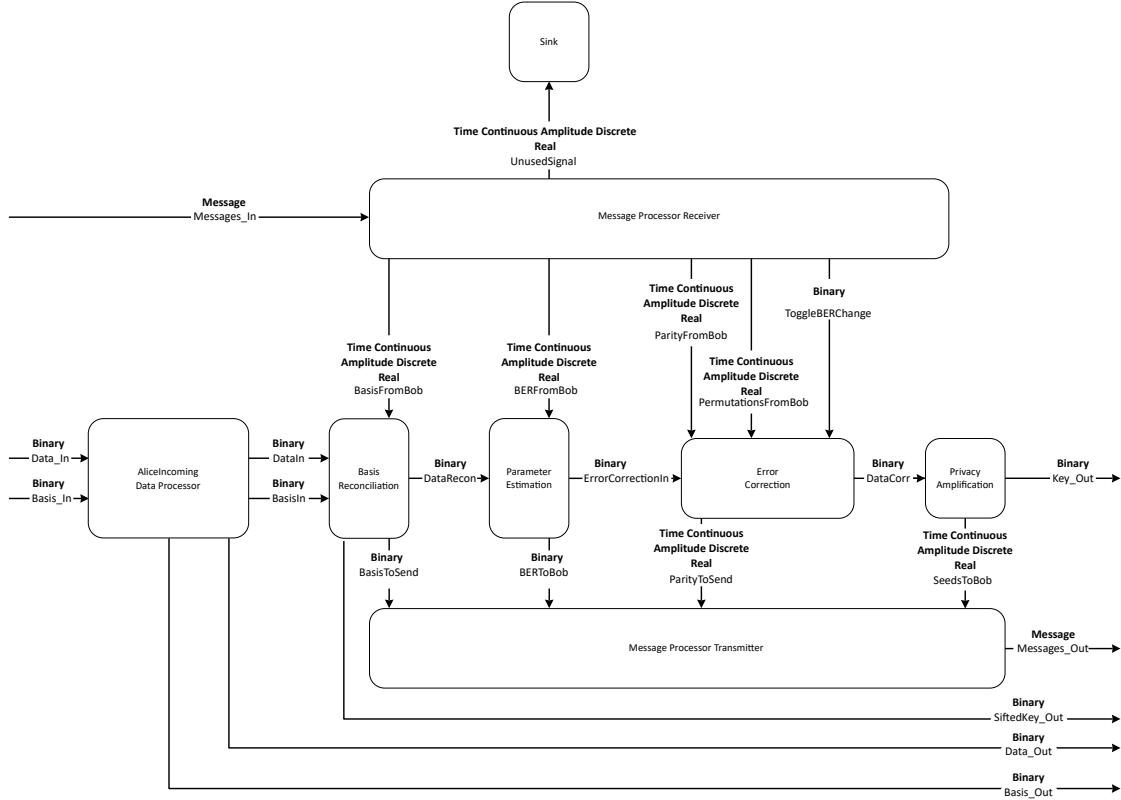
```
        AliceQKD(initializer_list<Signal*> inputSignals, initializer_list<Signal*> outputSignals,  
        t_integer maxSamplesToProcessAtOnce)
```

```
        AliceQKD(initializer_list<Signal*> inputSignals, initializer_list<Signal*> outputSignals,  
        string sFolderName, t_integer maxSamplesToProcessAtOnce)
```

```
        AliceQKD(initializer_list<Signal*> inputSignals, initializer_list<Signal*> outputSignals,  
        string sFolderName, t_integer maxSamplesToProcessAtOnce, unsigned long int buffersSize)
```

```
void setErrorCorrectionPartitionSize(t_integer sz)
void setErrorCorrectionNumberOfPasses(t_integer np)
void setDoublePartitionSize(bool db)
void setMaximumSyndromeSize(t_integer mss)
void setMinimumSyndromeSize(t_integer minss)
void setDoubleClickNumber(t_integer dcn)
void setNoClickNumber(t_integer ncn)
void setBer(t_real berValue)
void setSystematicSecurityParameter(t_integer ssp)
void setBypassHash(bool bh)
void setMinimumNumberOfPartitions(t_integer mnp)
void setParameterEstimationNumberBits(t_integer nb)
void setParameterEstimationNumberBitsToStart(t_integer nbs)
void setPrint(bool print)
void setQberThreshold(const t_real th)
void setPhotonsPerPulse(double ppp)
void setConfidenceInterval(double ci)
void setIpAddress(string ip)
string getIpAddress(void)
```

Functional description



AliceQKD is a superblock intended to do classical channel communication and data processing as required by the BB84 protocol. This superblock requires three different inputs:

- Messages from Bob, which are used to coordinate the required data processing for key extraction;
- A binary signal to be used for the transmitted key;
- A binary signal corresponding to the basis which will be used for encoding the data.

In addition, this block produces a total of four (or five, optionally) output signals:

- Messages intended to be read by Bob, used to coordinate the required data processing for key extraction;
- A binary signal of the data, to be transmitted through the quantum channel;
- A binary signal corresponding to the basis which will be used to encoded the transmitted data;
- A binary signal with the generated key;

- An optional binary signal corresponding to the sifted key.

The process inside the superblock is as follows:

The incoming data and basis binary signals are provided as inputs to the *AliceIncomingDataProcessor*, together with the *transmissionMode* signal. The transmission mode controls the behaviour of the *AliceIncomingDataProcessor*: it either enables it to continue working or tells it to stop reading the input data, while it waits for the current data to be dispatched. The data and basis read at the block are sent twice each to the output signals, keeping their sync. Each pair of basis and data has a different destination: one of the pair is directed to the superblock outputs, in order to transmit the data and control the basis for the transmission; and the other pair is sent to the *BasisReconciliation* block.

The *BasisReconciliation* block in AliceQKD is configured to responding to Bobs messages, instead of sending them first. Therefore, it now waits for a signal from the *MessageProcessorReceiver*. When Bob has enough samples, he should send a message containing the basis he used to measure the received data. This message is received, identified by its type *BasisReconciliation1*, and interpreted at the *MessageProcessorReceiver*. When a message of the correct type arrives, this block sends a signal to the *BasisReconciliation* block, containing the list of basis Bob used to measure the data on his side. The *BasisReconciliation* block then compares the basis received from Bob with the ones that Alice used, and based on this it outputs two signals:

- A binary evaluation of which basis used by Bob are correct. This is sent to the *MessageProcessorTransmitter*, so that a message can be sent to Bob for him to know which basis were correct.
- A binary signal containing the data sent when those basis were used. This is the data that Bob should have measured correctly. This signal is sent to the *ParametersEstimation* block.

The *ParameterEstimation* block receives the key, now free from wrongly measured bits and from the noClicks and doubleClicks. It accumulates the bits it receives until it reaches a certain amount, and then waits for an incoming message from Bob to start processing. The message from Bob contains a seed used to sample a pseudorandom subset of the accumulated bits, and the value of those bits that Bob has. Alice compares them to her own bits, and the ratio of correct bits provides an estimate of the QBER. She then sends her values to Bob, discards those bits and outputs the QBER value and the remaining bits. The QBER value and remaining bits are then sent to the *ErrorCorrection* block.

The *ErrorCorrection* block will be responsible for correcting any errors that occurred during transmission. Similarly to the *BasisReconciliation* block, it is configured to act in response to Bobs messages. As such, it waits for data from the *MessageProcessorReceiver*, and outputs a response to the *MessageProcessorTransmitter*. It also outputs a copy of the signal from basis reconciliation, after it has been used for error correction. The key output by this block is free from errors, or at least contains a very small amount of them, depending on the key size and the chosen parameters.

After each set of bits is corrected, the data is output to the *PrivacyAmplification* block, which does the necessary transformation to ensure the information that any attacker might have gained (Eve) is nullified.

The end result, and final output of the superblock, is a secure key shared known only to Alice and Bob.

Examples

Suggestions for future improvement

2.7 AliceQuantumTx

Header File	:	AliceQTx.h
Source File	:	AliceQTx.cpp
Version	:	20190404

This block is the quantum channel Alice's transmitter. This block accepts binary signals, which comprises the values for basis and bits to be used to encode single photons, as well as another binary value with modeSelection that selects the operation mode that Alice must operate.

- inputSignal[0] → AliceQTxBasis (Binary)
- inputSignal[1] → AliceQTxBits (Binary)
- inputSignal[2] → ModeSelection (Binary)

It produces PhotonStreamXY signal which comprises the encoded single photons.

- outputSignal[0] → AliceQTx_Out (PhotonStreamXY)

It also accepts four input parameters listed below.

Input Parameters

- string controlBasis - This string is the sequence of basis used to control qubits. Its default value is "0".
- string controlBits - This string is the sequence of bits used to control qubits. Its default value is "0".
- double numberOfPhotonsControl - Average number of photons contained in a control coherent pulse. Its default value is 80.
- double numberOfPhotonsData - Average number of photons contained in a data coherent pulse. Its default value is 1.
- int incrementControl - Increment to use in monitoring mode, i.e. for example if incrementControl is 100, Alice will transmit 1 control qubit in each 100 transmitted qubits. Its default value is 100;

Methods

- void setControlBasisSequence(string bSeq) - This method allows to set the control basis sequence accepting a string value.
- void setControlBitsSequence(string bits) - This method allows to set the control bits sequence accepting a string value.

- void setNumberOfPhotonsData(double nData) - This method allows to set the average number of photons contained in data coherent pulses.
- void setNumberOfPhotonsControl(double nControl) - This method allows to set the average number of photons contained in control coherent pulses.
- void setIncrementControl(int inc) - This method allows to set the increment to use in monitoring mode.

Functional description

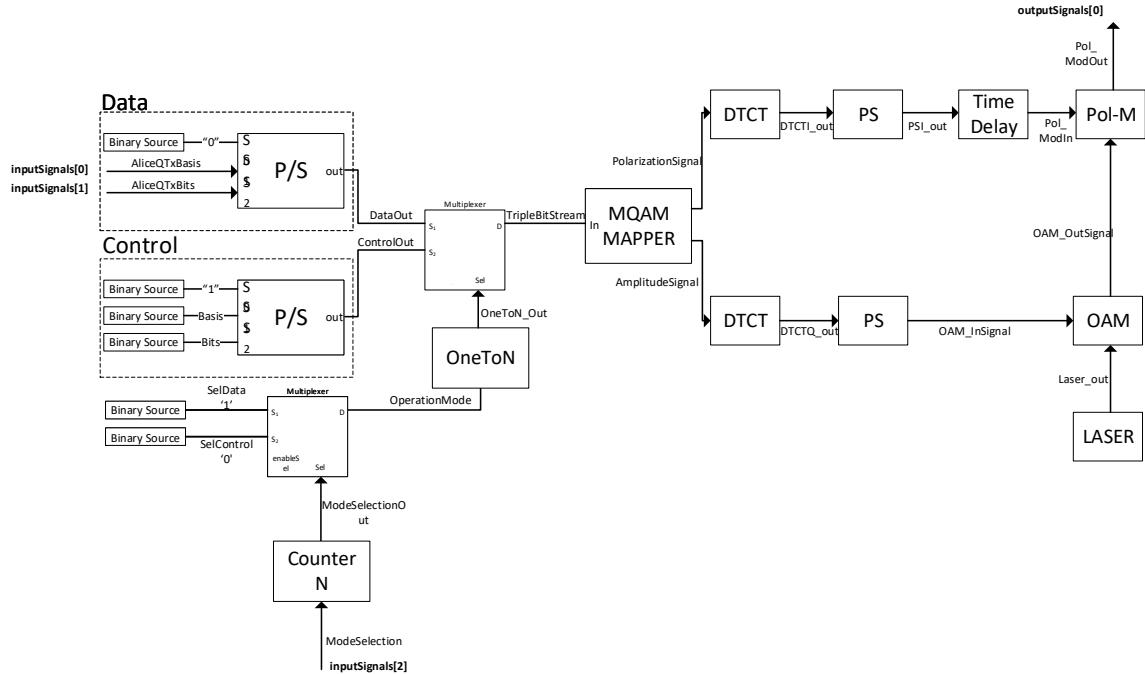


Figure 2.1: Schematic of Alice Quantum Transmitter

This block receives 3 input binary signals.

- "ModeSelection" is a binary signal which gives to AliceQTx the information about operation mode, and so that she will know if she should send control or data qubits. If "ModeSelection" has the value "0" the monitoring mode is selected, and the output of this multiplexer will be a pre-defined sequence that should be "0" when data qubits are sent and "1" when control qubits are sent. This choice is made based on a sequence which should be deterministically made. Otherwise, if ModeSelection has the value "1", the actuation mode is selected and all qubits sent are control qubits.

- "AliceQTxBasis" is a binary signal which provides information about the basis to use to encode single-photons.
- "AliceQTxBits" is a binary signal which provides information about the bits to use to encode single-photons.

"AliceQTxBasis" and "AliceQTxBits" will be responsible for data qubits encoding.

The operationMode signal will feed the other multiplex with an enable signal that should put on the output of the multiplexer, the input s_2 if this signal has the value "0" (control qubits), and the input s_1 if this signal has the value "1"(data qubits).

A triple bit stream binary signal outputs the multiplexer, where the sequence is {Monitoring/Actuation, Basis, Bits}, where the first bit is "0" if the actuation mode was selected, and "1" if the monitoring mode was selected. The MQAM Mapper block receives this bit stream and will be able to encode a constellation with 8 different symbols as shown in Figure 2.2.

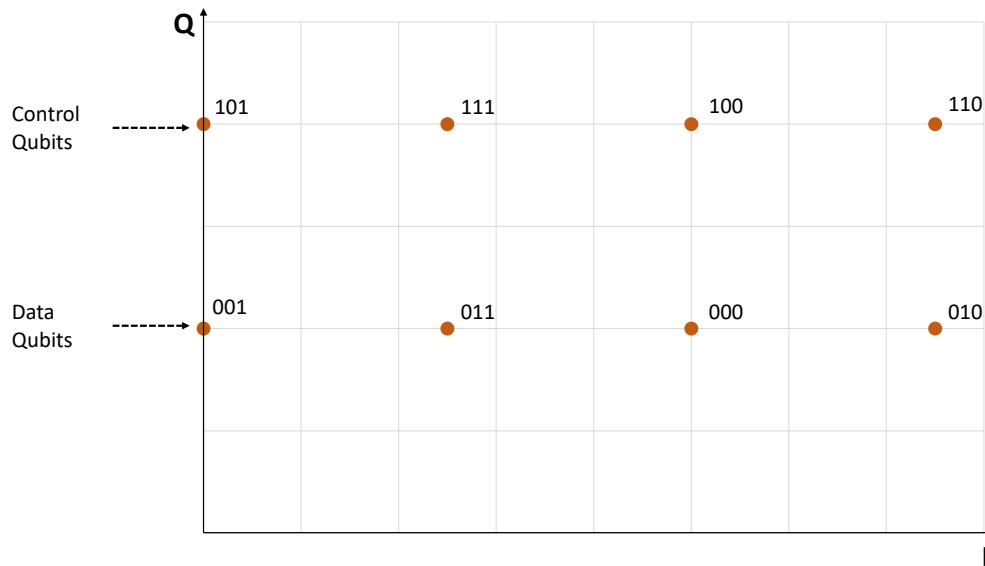


Figure 2.2: 8 symbols constellation for Alice Quantum transmitter single photon codification.

The Quadrature component will provide an input signal for amplitude modulator, and it will modulate the signal according with two different amplitudes depending on the type of qubits to send, i.e. the control qubits will have higher amplitude than data qubits. The Intensity component will provide a signal with 4 different steps that will choose the polarization to encode the single photons. The amplitude modulator outputs a signal that enters in the polarization modulator and outputs with a certain polarization. At the output of Alice quantum transmitter there is a variable optical attenuator that will be responsible to reduce the number of photons per pulse until 0.1 for data qubits.

Input Signals

Number : 3

Type : Binary.

Output Signals

Number : 1

Type : PhotonStreamXY.

Examples

Sugestions for future improvement

2.8 Ascii Source

Header File	:	ascii_source_*.h
Source File	:	ascii_source_*.cpp
Version	:	20180828 (André Mourato)

This block generates an ascii signal and can work in different modes:

- 1. Terminate
- 2. Cyclic
- 3. AppendZeros

This blocks doesn't accept any input signal. It produces any number of output signals.

Input Parameters

Parameter	Type	Values	Default
mode	AsciiSourceMode	Terminate, AppendZeros, Cyclic	Terminate
asciiString	string	any	""
asciiFilePath	string	any	"text_file.txt"
numberOfCharacters	int	$\in [0, \infty[$	1000

Table 2.5: Binary source input parameters

Methods

```
AsciiSource(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
bool runBlock(void);
void setMode(AsciiSourceMode m);
AsciiSourceMode const getMode(void);
void setAsciiString(string s);
string getAsciiFilePath();
void setNumberOfCharacters(int n);
int getNumberOfCharacters();
```

Functional description

The *mode* parameter allows the user to select one of the operation modes of the ascii source.

Terminate The resulting ascii signal will be the *asciiString* sequence of characters.

AppendZeros The resulting ascii signal will be a sequence of characters starting with *asciiString* with zeros (null character) concatenated to the right until *numberOfCharacters* characters have been generated.

Cyclic The resulting ascii signal will be a sequence of characters in which *asciiString* is concatenated to the right of the string until *numberOfCharacters* characters have been generated.

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: Ascii

2.9 Ascii To Binary

Header File	:	ascii_to_binary_*.h
Source File	:	ascii_to_binary_*.cpp
Version	:	20180905 (André Mourato)

Methods

```
AsciiToBinary(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

Functional description

Figure 2.8 shows an example of an input signal that can be passed as argument. This signal contains three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. The values to these three characters are respectively: 01100001, 1100010 and 1100011. The block AsciiToBinary will convert the characters a, b and c to their respective binary codes. The resulting output signal will be of type Binary. The output signal to this example, shown in figure 2.7, is the concatenation of the previous binary codes. The full binary sequence is 0110000111000101100011.

```
S1.sgn *  
1 Signal type: Ascii  
2 Symbol Period (s): 1  
3 Sampling Period (s): 1  
4 // ### HEADER TERMINATOR ###  
5 abc  
6  
7
```

Figure 2.3: Ascii signal passed as input to the AsciiToBinary block

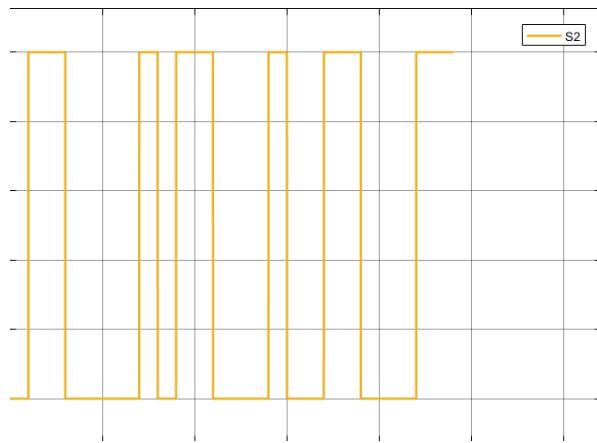


Figure 2.4: Resulting Binary signal from the output of the AsciiToBinary block

Input Signals

Number: 1

Type: Ascii

Output Signals

Number: 1

Type: Ascii

2.10 Balanced Beam Splitter

Header File	:	balanced_beam_splitter.h
Source File	:	balanced_beam_splitter.cpp
Version	:	20180124

Input Parameters

Name	Type	Default Value
Matrix	array <t_complex, 4>	$\left\{ \left\{ \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}} \right\} \right\}$
Mode	double	0

Functional Description

The structure of the beam splitter can be controlled with the parameter mode.

When **Mode = 0** the beam splitter will have one input port and two output ports - **1x2 Beam Splitter**. If Mode has a value different than 0, the splitter will have two input ports and two output ports - **2x2 Beam Splitter**.

Considering the first case, the matrix representing a 2x2 Beam Splitter can be summarized in the following way,

$$M_{BS} = \frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.1)$$

The relation between the values of the input ports and the values of the output ports can be established in the following way

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \begin{bmatrix} A \\ B \end{bmatrix} \quad (2.2)$$

Where, A and B represent the inputs and A' and B' represent the outputs of the Beam Splitter.

Input Signals

Number: 1 or 2

Type: Complex

Output Signals

Number: 2

Type: Complex

2.11 Bit Error Rate

Header File	:	bit_error_rate_*.h
Source File	:	bit_error_rate_*.cpp
Version	:	20180815
Contributors	:	Daniel Pereira
	:	Mariana Ramos
	:	Manuel Neves
	:	Armando Pinto

This block estimates the bit-error rate. It compares two input binary signals and outputs a binary signal with a zero if both input bits are equal and a one if they differ. Additionally, this block does statistical analysis on the BER and outputs a file with the results of this analysis. The statistical analysis consists on the upper and lower bounds of the BER value, as well as its average value.

The block also allows having breakpoints on the BER analysis, outputting mid-reports (instead of only one final report).

Signals

Input Signals

Number: 2

Type: Binary

These two input signals are (interchangeably) the received binary sequence, from the MQAM Receiver output, and the transmitted signal, directly from the Binary Source.

Output Signals

Number: 1

Type: Binary

The output signal is a binary vector of '1's and '0's, with the same length as the input signals, in which a '0' indicates the occurrence of an error and a '1' a successfully transmitted bit.

Input Parameters

Name	Type	Default Value	Description
alpha	double	0.05	Alpha is one minus the confidence level in the BER interval.
m	integer	0	Defines how many bits are analysed in between mid-reports. When null there aren't any mid-reports.
lMinorant	double	1×10^{-10}	Defines the minimum value of the lower bound for the BER.

Block Constructor

- `BitErrorRate(vector<Signal *> InputSig, vector<Signal *> OutputSig) :Block(InputSig,OutputSig){};`

This block's constructor expects, as any *Block* object, an input of two vectors of Signal pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

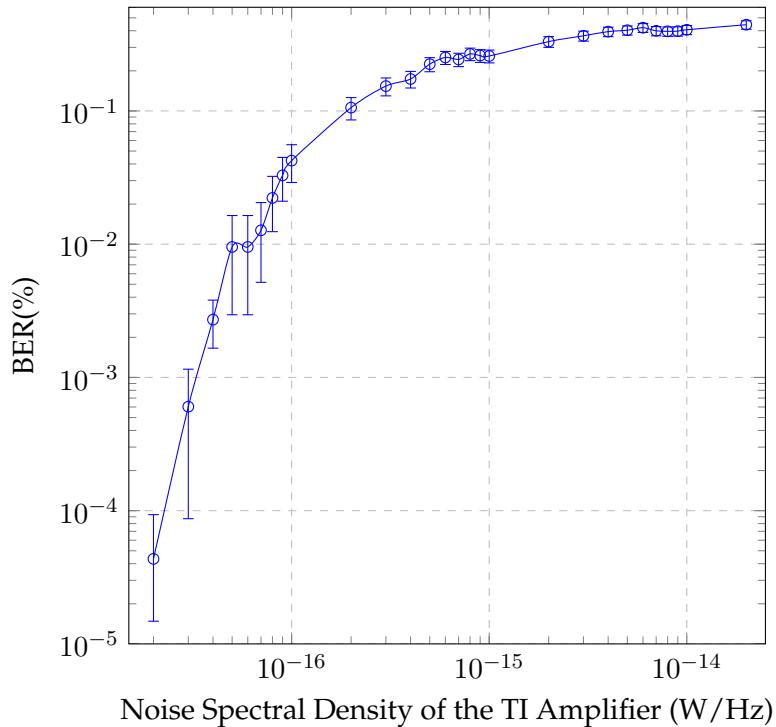
Methods

- `void initialize(void);`
 - Sets symbol and sampling period and the *firstValueToBeSaved* of the output signal, based on the input signals.
- `bool runBlock(void);`
 - Computes output statistics and writes to BER.txt and mid-report#.txt the obtained results.
- `double const getConfidence(double P);`
 - Returns the value of the confidence level based on the private parameter *alpha*.
- `void setConfidence(double P);`
 - Defines the value of the *alpha* parameter, based on the wished confidence.
- `int const getMidReportSize(int M);`
 - Returns the value of the private parameter *m*.
- `void setMidReportSize(int M);`
 - Alters the value of the *m* parameter, from its default.

- double **getLowestMinorant**(double lMinorant);
 - Returns the value of the private parameter *lMinorant*.
- void **setLowestMinorant**(double lMinorant);
 - Alters the value of the *lMinorant* parameter, from its default.

Examples

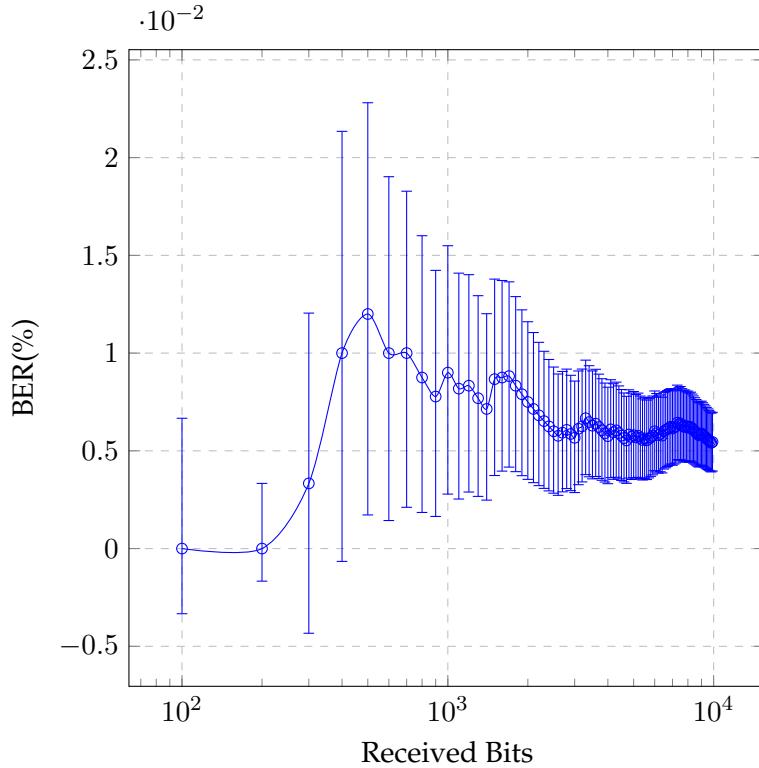
- One interesting test we can do with this block, is analyse how a certain parameter of the system affects de BER. In the following graph we demonstrate an example of how the Noise Spectral Density of the TI Amplifier on the receiver affects the BER:



To obtain this result a Transmitter+Receiver QPSK system was ran several times, applying different values to the spectral density of the noise at the input of the TI amplifier.

- Another interesting aspect facilitated by the BER block is to analyse how the BER tends to its real value as the number of the number of received bits increases. We can do this simply by analysing the multiple mid-reports, by giving a sufficiently low value to the *m* parameter.

The following graph was obtained in the same configuration as the previous graph, for a fixed noise spectral density value of 5e-17W/Hz:



It is interesting to observe how the confidence bounds shrink, and that the BER value is within the previous values of these bounds (in at least $(1 - \alpha)\%$ of the cases).

Output Signals

Number: 1

Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional and Theoretical Description

This block accepts two binary strings and outputs a binary string. For each pair of input bits, it outputs a '1' if the two coincide and '0' if not.

The $\widehat{\text{BER}}$ is thus obtained by counting both the total number received bits, N_{bits} , and the number of coincidences, K , and calculating their relative ratio:

$$\widehat{\text{BER}} = 1 - \frac{K}{N_{bits}}. \quad (2.3)$$

The confidence interval of the $\widehat{\text{BER}}$ value is also calculated. To easily obtain the upper and lower bounds, BER_{UB} and BER_{LB} respectively, an approximation of the Clopper-Pearson

confidence interval [1] is used, which holds valid for $N_{bits} > 40$:

$$\text{BER}_{\text{UB}} = \widehat{\text{BER}} + \frac{z_{\alpha/2}}{\sqrt{N_{bits}}} \sqrt{\widehat{\text{BER}}(1 - \widehat{\text{BER}})} + \frac{1}{3N_{bits}} \left[2 \left(\frac{1}{2} - \widehat{\text{BER}} \right) z_{\alpha/2}^2 + 2 - \widehat{\text{BER}} \right] \quad (2.4)$$

$$\text{BER}_{\text{LB}} = \widehat{\text{BER}} - \frac{z_{\alpha/2}}{\sqrt{N_{bits}}} \sqrt{\widehat{\text{BER}}(1 - \widehat{\text{BER}})} + \frac{1}{3N_{bits}} \left[2 \left(\frac{1}{2} - \widehat{\text{BER}} \right) z_{\alpha/2}^2 - 1 - \widehat{\text{BER}} \right] \quad (2.5)$$

where $z_{\alpha/2}$ is the $100(1 - \frac{\alpha}{2})$ th percentile of a standard normal distribution and this is where the value of the *alpha* parameter influences the result.

Upon finishing running, this block outputs to the *signals* directory a file named *BER.txt* with a report of the estimated $\widehat{\text{BER}}$, the estimated confidence bounds for a given confidence level $(1 - \alpha)$ and the total number of received bits.

The file format is the following:

```
BER= 4.34783e-05
Upper and lower confidence bounds for 95% confidence level
Upper Bound= 9.33303e-05
Lower Bound= 1.47979e-05
Number of received bits =92000
```

Another functionality of the BER block is that it allows for mid-reports to be generated. A mid-report has exactly the same structure of the final report showed above, but it's generated for every m pairs of input bits. These files are saved in the same directory of the running solution and they're saved on files named *midreport#.txt*, where the '#' stands for the number of the respective mid-report and is in the interval $[1; \frac{N_{bits}}{m}]$.

The number of bits between reports is the parameter m , customizable, and if it is set to '0' then the block will only output the final report.

Open Issues

- The output value for the Lower bound is negative when the BER is null, and in these cases it should be set to '0'. This error is hidden in the *BER.txt* file, because when the Lower Bound value is lower than the *lMinorant* parameter, an *if* statement sets it to the value of *lMinorant*. This correction can however lead to errors in the cases that the *lMinorant* is smaller than the actual BER we're attempting to measure.
- Mid-reports are not formatted exactly like the BER output file. And they are stored in the current directory, instead of being saved in the same directory as the *BER.txt* file, which would probably be more desirable.

Future Improvements

- fixing the Lower Bound issue mentioned above;
- the printing of all the mid-reports could be done in the same file, for ease of BER convergence analysis;
- the method *getLowestMinorant()* should have a *const* return;

- there could be only one private method for printing an output file, thus avoiding having two times the same code repeated for the mid-report printing and for the BER final report printing;
- exactness required for the convergence of the z-score should also be a parameter, for it may be desirable to have a different value of exactness.

References

- [1] Álvaro J Almeida et al. “Continuous Control of Random Polarization Rotations for Quantum Communications”. In: *Journal of Lightwave Technology* 34.16 (2016), pp. 3914–3922.

2.12 Binary Source

Header File	:	binary_source_*.h
Source File	:	binary_source_*.cpp
Version	:	20180118 (Armando Pinto)
	:	20180523 (André Mourato)

This block generates a sequence of binary values (1 or 0) and it can work in four different modes:

- | | | |
|-----------------|-----------------------------|-------------------------|
| 1. Random | 3. DeterministicCyclic | 5. AsciiFileAppendZeros |
| 2. PseudoRandom | 4. DeterministicAppendZeros | 6. AsciiFileCyclic |

Signals

Number of Input Signals	0
Type of Input Signals	-
Number of Output Signals	\geq
Type of Output Signals	Binary

Table 2.6: Binary source signals

Input Parameters

Parameter	Type	Values	Default
mode	string	Random, PseudoRandom, DeterministicCyclic, DeterministicAppendZeros, AsciiFileAppendZeros, AsciiFileCyclic	PseudoRandom
probabilityOfZero	real	$\in [0,1]$	0.5
patternLength	int	Any natural number	7
bitStream	string	sequence of 0's and 1's	0100011101010101
numberOfBits	long int	any	-1
bitPeriod	double	any	1.0/100e9
asciiFilePath	string	any	"file_input_data.txt"

Table 2.7: Binary source input parameters

Methods

```
BinarySource(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};

void initialize(void);

bool runBlock(void);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)
```

Functional description

The *mode* parameter allows the user to select between one of the four operation modes of the binary source.

Random Mode Generates a 0 with probability *probabilityOfZero* and a 1 with probability $1 - \text{probabilityOfZero}$.

Pseudorandom Mode Generates a pseudorandom sequence with period $2^{\text{patternLength}} - 1$.

DeterministicCyclic Mode Generates the sequence of 0's and 1's specified by *bitStream* and then repeats it.

DeterministicAppendZeros Mode Generates the sequence of 0's and 1's specified by *bitStream* and then it fills the rest of the buffer space with zeros.

Input Signals

Number: 0

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1 or more

Type: Binary (DiscreteTimeDiscreteAmplitude)

Illustrative Examples

Random Mode

PseudoRandom Mode Consider a pseudorandom sequence with *patternLength*=3 which contains a total of 7 ($2^3 - 1$) bits. In this sequence it is possible to find every combination of 0's and 1's that compose a 3 bit long subsequence with the exception of 000. For this example the possible subsequences are 010, 110, 101, 100, 111, 001 and 100 (they appear in figure 2.5 numbered in this order). Some of these require wrap.

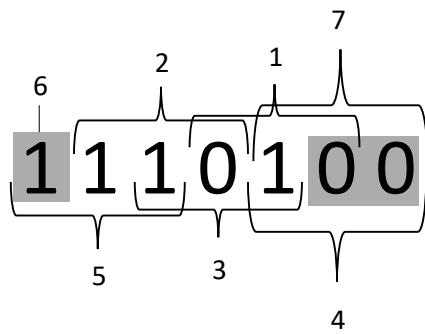


Figure 2.5: Example of a pseudorandom sequence with a pattern length equal to 3.

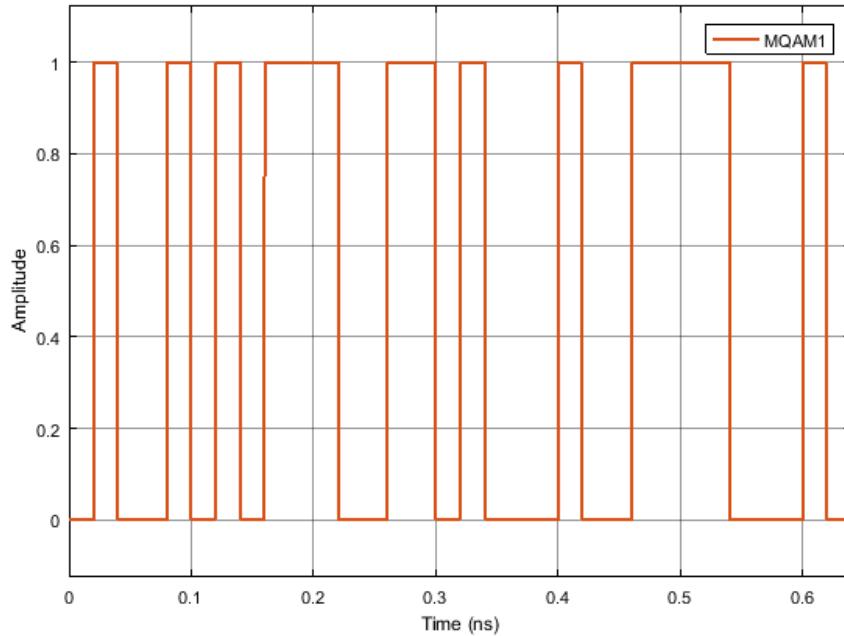


Figure 2.6: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

DeterministicCyclic Mode Take the *bit stream* '0100011101010101'. The generated binary signal is displayed in.

DeterministicAppendZeros Mode Take the *bit stream* '0100011101010101'. The generated binary signal is displayed in 2.6.

Suggestions for future improvement

Implement an input signal that can work as trigger.

2.13 Binary To Ascii

Header File	:	binary_to_ascii_*.h
Source File	:	binary_to_ascii_*.cpp
Version	:	20180905 (André Mourato)

Methods

```
BinaryToAscii(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

Functional description

Figure 2.7 shows an example of an input signal that can be passed as argument. This signal contains the binary representation of three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. This signal contains the following stream of bits: 0110000111000101100011. There are 24 bits in this stream, therefore we can divide it into three segments of 8 bits each. The resulting segments are: 01100001, 1100010 and 1100011. Each of these segments is the binary code of a character. 01100001 represents the character *a*, 1100010 represents the character *b* and 1100011 represents the character *c*. The block BinaryToAscii will convert the binary codes into the respective characters. The resulting output signal will be of type Ascii. The output signal to this example, shown in figure 2.8, contains the characters that can be represented with the previous binary codes.

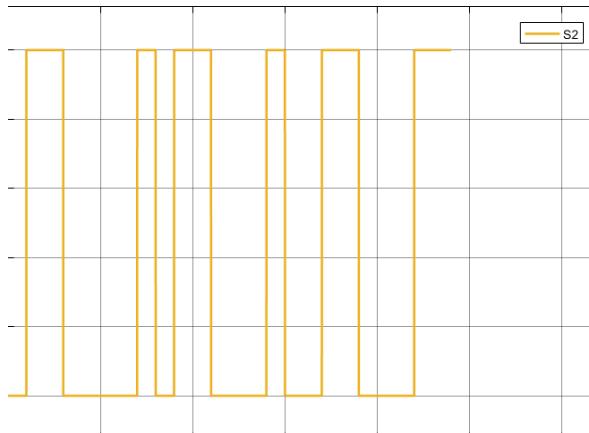
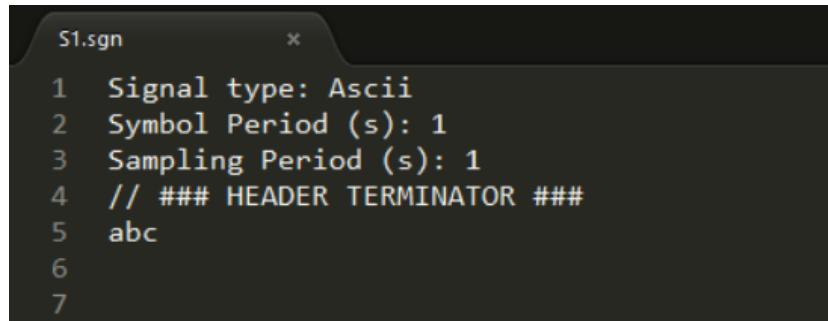


Figure 2.7: Binary signal passed as input to the BinaryToAscii block



The screenshot shows a terminal window titled "S1.sgn" with the following text content:

```
1 Signal type: Ascii
2 Symbol Period (s): 1
3 Sampling Period (s): 1
4 // ### HEADER TERMINATOR ###
5 abc
6
7
```

Figure 2.8: Resulting Ascii signal from the output of the BinaryToAscii block

Input Signals

Number: 1

Type: Ascii

Output Signals

Number: 1

Type: Ascii

2.14 Bob QKD

This block is the processor for Bob does all tasks that she needs. This block accepts and produces:

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

2.15 BobQuantumRx

Header File	:	BobQRx.h
Source File	:	BobQRx.cpp
Version	:	20190410

This block is quantum channel Bob's receiver. This block accepts a binary signal, which comprises the values of basis to measure the encode single photons. These values must be randomly chosen between two different basis corresponding to the two non-orthogonal basis needed. Furthermore, this block also accepts a PhotonStreamXY signal corresponding to the single photon that arrives from quantum channel. It produces a binary signal which contains the modeSelection.

- inputSignal[0] → BobQRxBasis (Binary)
- inputSignal[1] → BobQRx_In (PhotonStreamXY)

This block produces the following output signals,

- outputSignal[0] → BobQRx_DataOut (TimeDiscreteAmplitudeContinuousReal)
- outputSignal[1] → ModeSelection (Binary)
- outputSignal[2] → QBER_QChannel (TimeDiscreteAmplitudeContinuousReal)

Input Parameters

- string controlBasisSequence - This string is the sequence of basis used to control qubits. Its default value is "0".
- string controlSequence - This string is the sequence of bits used to control qubits. Its default value is "0".

Methods

- void setControlBasisMeasurement(string cBasis) - This method allows to set the control basis measurement sequence accepting a string value.
- void setControlSequenceBits(string cSeq) - This method allows to set the control bits sequence to use for quantum bit error rate estimation, taking into account this sequence. It accepts a string value.

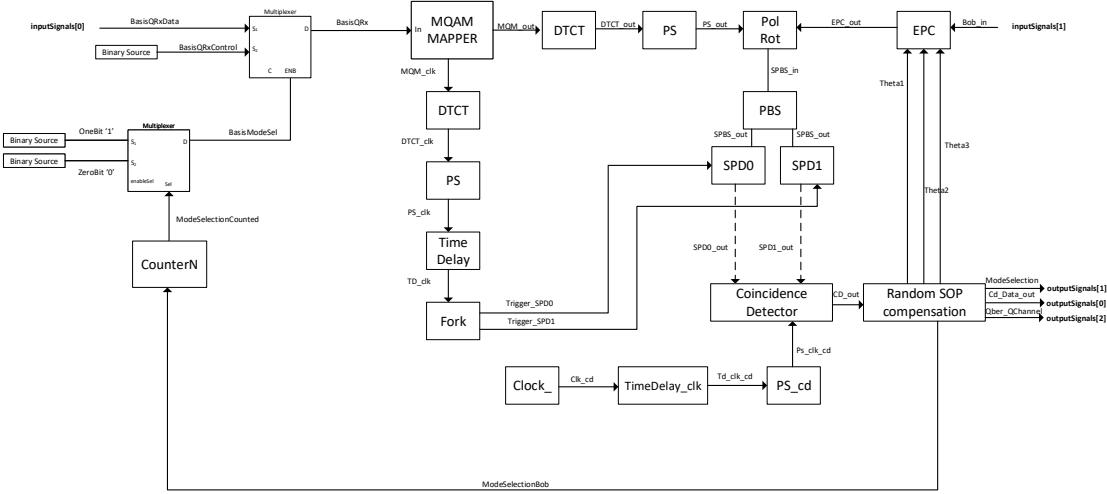


Figure 2.9: Schematic of Bob Quantum Receiver

Functional description

This block receives 1 input binary signal that will have the information about basis ("BobQRxBasis") to measure the single-photons that arrives from the quantum channel. This signal must comprise two possible values corresponding to two non-orthogonal basis used in the communication protocol. It also accepts a PhotonStreamXY signal corresponding to the encoded single photon that arrives from quantum channel sent by AliceQTx.

The mode selection signal that outputs this superblock will inform Alice about which mode she should selects, and so that if she should send data or control qubits.

Apart from the detection scheme, this block also comprises an active random polarization drift compensation scheme. This scheme includes a randomSopCompensation block that is responsible for all post processing tasks, including QBER estimation, and sends three feedback signals with information about the rotations to be performed by an electronic polarization controller (EPC). In this way, the random polarization drift is compensated before the single photons reach the detection scheme.

Input Signals

Number : 2

Type : Binary, PhotonStreamXY.

Output Signals

Number : 3

Type : TimeDiscreteAmplitudeContinuousReal, Binary.

Examples

Sugestions for future improvement

2.16 Bit Decider

Header File	:	bit_decider.h
Source File	:	bit_decider.cpp
Version	:	20170818

Input Parameters

Name	Type	Default Value
decisionLevel	double	0.0

Functional Description

This block accepts one real discrete signal and outputs a binary string, outputting a 1 if the input sample is greater than the decision level and 0 if it is less or equal to the decision level.

Input Signals

Number: 1

Type: Real signal (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1

Type: Binary (DiscreteTimeDiscreteAmplitude)

2.17 Clock

Header File	:	clock.h
Source File	:	clock.cpp

This block doesn't accept any input signal. It outputs one signal that corresponds to a sequence of Dirac's delta functions with a user defined *period*.

Input Parameters

Parameter	Type	Values	Default
period	double	any	0.0
samplingPeriod	double	any	0.0

Table 2.8: Binary source input parameters

Methods

Clock()

```
Clock(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig)
```

```
void initialize(void)
bool runBlock(void)
void setClockPeriod(double per)
void setSamplingPeriod(double sPeriod)
```

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

Type: Sequence of Dirac's delta functions.
(TimeContinuousAmplitudeContinuousReal)

Examples**Sugestions for future improvement**

2.18 Clock_20171219

This block doesn't accept any input signal. It outputs one signal that corresponds to a sequence of Dirac's delta functions with a user defined *period*, *phase* and *sampling period*.

Input Parameters

- period{ 0.0 };
- samplingPeriod{ 0.0 };
- phase {0.0};

Methods

Clock()

```
Clock(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,  
OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setClockPeriod(double per) double getClockPeriod()
```

```
void setClockPhase(double pha) double getClockPhase()
```

```
void setSamplingPeriod(double sPeriod) double getSamplingPeriod()
```

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

Type: Sequence of Dirac's delta functions.
(TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

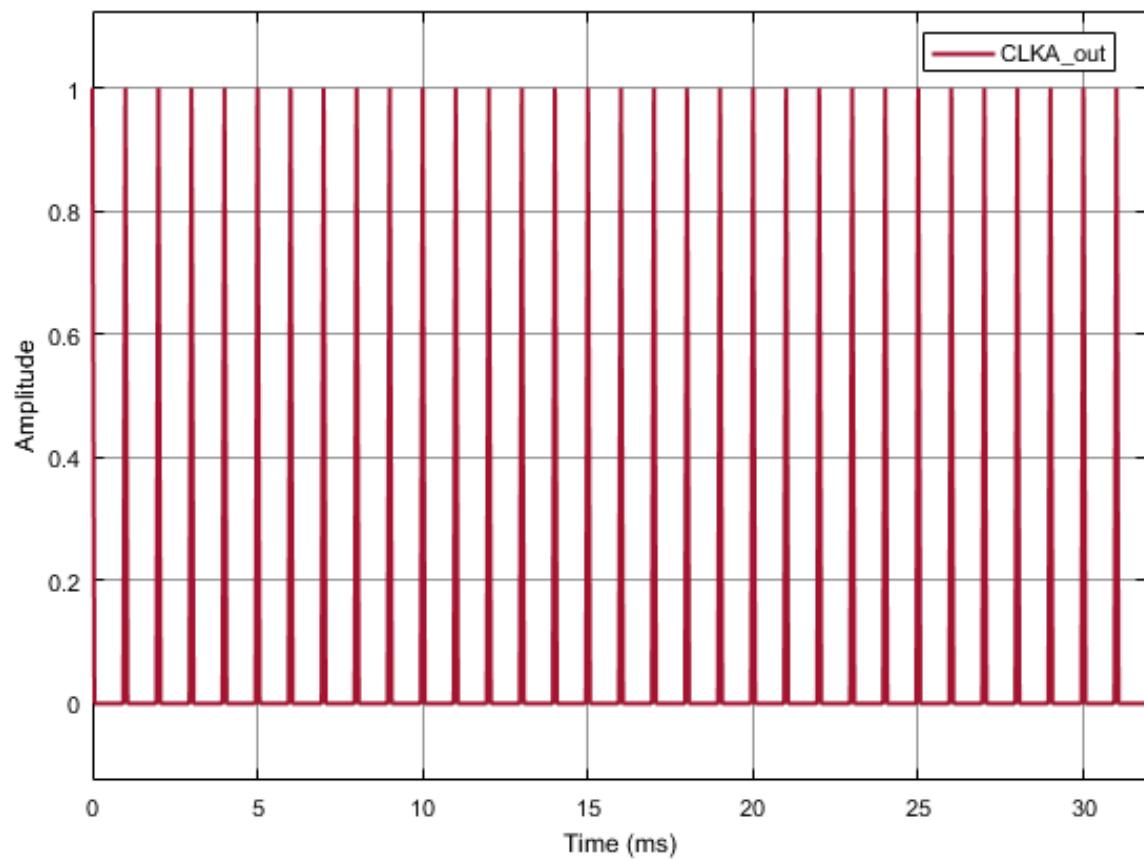


Figure 2.10: Example of the output signal of the clock without phase shift.

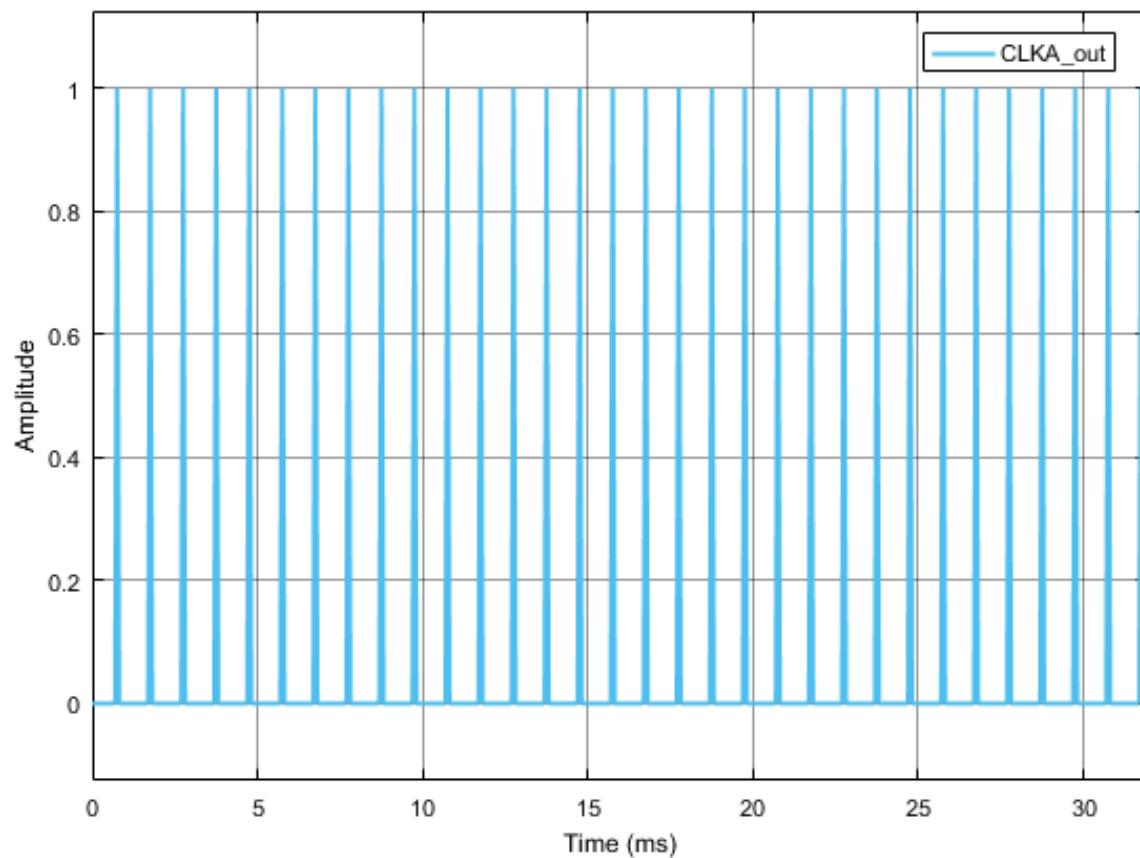


Figure 2.11: Example of the output signal of the clock with phase shift.

2.19 Complex To Real

Header File	:	complex_to_real_*.h
Source File	:	complex_to_real_*.cpp
Version	:	20180717 (Celestino Martins)

This super block converts a complex input signal into two real signals.

Input Parameters

Methods

- ComplexToReal();
- ComplexToReal(vector<Signal * > &InputSig, vector<Signal * > &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);

Functional description

This super block converts a complex input signal into two real signals.

Input Signals

Number: 1

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

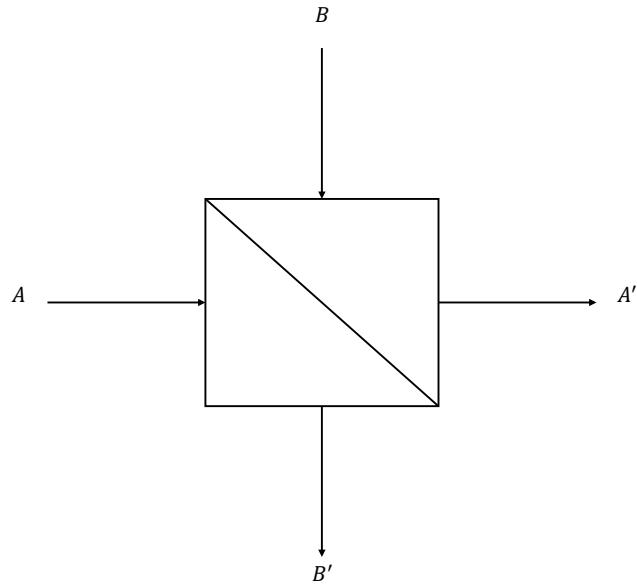


Figure 2.12: 2x2 coupler

2.20 Coupler 2 by 2

In general, the matrix representing 2x2 coupler can be summarized in the following way,

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \quad (2.6)$$

Where, A and B represent inputs to the 2x2 coupler and A' and B' represent output of the 2x2 coupler. Parameters T and R represent transmitted and reflected part respectively which can be quantified in the following form,

$$T = \sqrt{1 - \eta_R} \quad (2.7)$$

$$R = \sqrt{\eta_R} \quad (2.8)$$

Where, value of the $\sqrt{\eta_R}$ lies in the range of $0 \leq \sqrt{\eta_R} \leq 1$.

It is worth to mention that if we put $\eta_R = 1/2$ then it leads to a special case of "Balanced Beam splitter" which equally distribute the input power into both output ports.

2.21 Carrier Phase Compensation

Header File	:	carrier_phase_estimation_*.h
Source File	:	carrier_phase_estimation_*.cpp
Version	:	20180423 (Celestino Martins)

This block performs the laser phase noise compensation using either Viterbi-Viterbi (VV) algorithm or blind phase search algorithm (BPS). For both cases, it receives one input complex signal and outputs one complex signal.

Input Parameters For VV Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
methodType	string	VV	VV
mQAM	int	any	4

Table 2.9: CPE input parameters

Input Parameters For BPS Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
NtestPhase	int	any	32
methodType	string	BPS	VV
mQAM	int	any	4

Table 2.10: CPE input parameters

Methods

```

CarrierPhaseCompensation() ;

    CarrierPhaseCompensation(vector<Signal    *>    &InputSig,    vector<Signal    *>
&OutputSig) :Block(InputSig, OutputSig){};

    void initialize(void);

    bool runBlock(void);

    void setnTaps(int ntaps) nTaps = ntaps;

```

```

double getnTaps() return nTaps;

void setmQAM(int mQAMs) mQAM = mQAMs;

double getmQAM() return mQAM;

void setTestPhase(int nTphase) nTestPhase = nTphase;

double getTestPhase() return nTestPhase;

void setmethodType(string mType) methodType = mType;

string getmethodType() return methodType;

void setBPStype(string tBPS) BPStype = tBPS;

string getBPStype() return BPStype;

```

Functional description

This block can perform the carrier phase noise compensation originated by the laser source and local oscillator in coherent optical communication systems. For the sake of simplicity, in this simulation we have restricted all the phase noise at the transmitter side, in this case generated by the laser source, which is then compensated at the receiver side using DSP algorithms. In this simulation, the carrier phase noise compensation can be performed by applying either the well known Viterbi-Viterbi (VV) algorithm or blind phase search algorithm (BPS), by configuring the parameter *methodType*. The parameter *methodType* is defined as a string type and it can be configured as: i) When the parameter *methodType* is *VV* it is applied the VV algorithm; When the parameter *methodType* is *BPS* it is applied the BPS algorithm.

Viterbi-Viterbi Algorithm

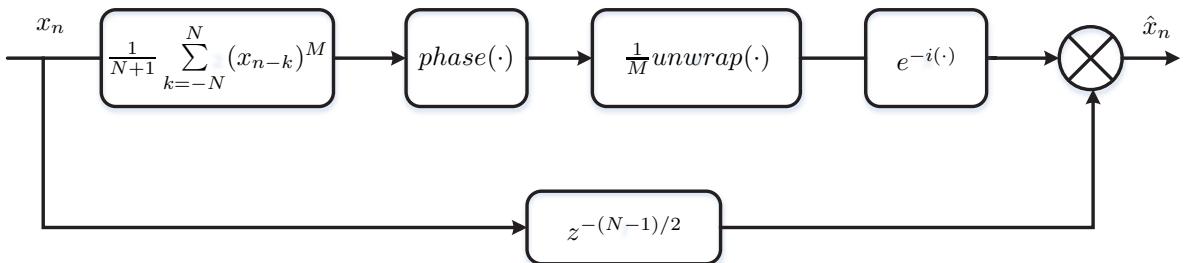


Figure 2.13: Block diagram of Viterbi-Viterbi algorithm for carrier phase recovery.

VV algorithm is a n -th power feed-forward approach employed for uniform angular distribution characteristic of m -PSK constellations, where the information of the modulated

phase is removed by employing the n-th power operation on the received symbols. The algorithm implementation diagram is shown in Figure 2.13, starting with M-th power operation on the received symbols. In order to minimize the impact of additive noise in the estimation process, a sum of $2N + 1$ symbols is considered, which is then divided by M. The resulting estimated phase noise is then submitted to a phase unwrap function in order to avoid the occurrence of cycle slip. The final phase noise estimator is then used to compensate for the phase noise of the original symbol in the middle of the symbols block.

Blind Phase Search Algorithm

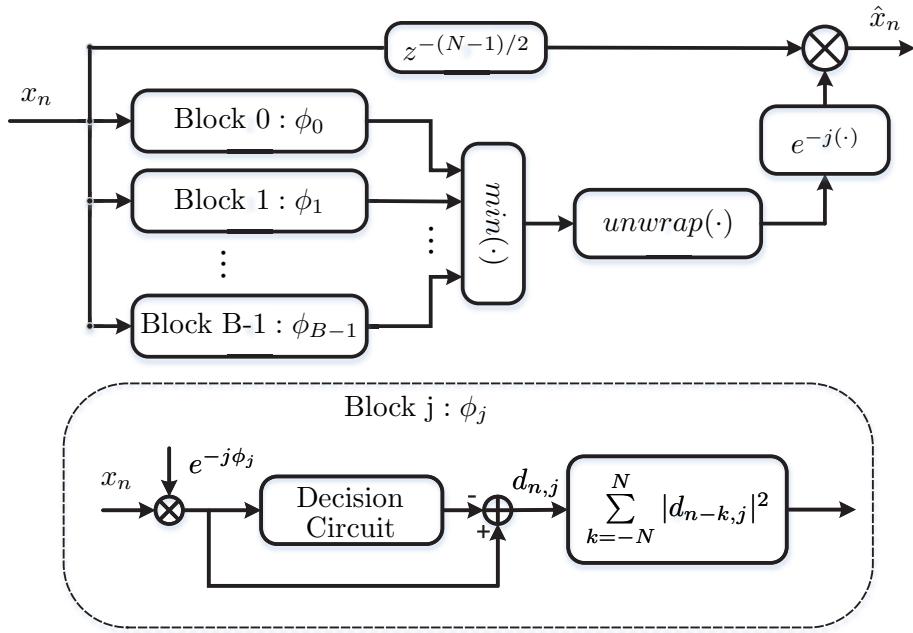


Figure 2.14: Block diagram of blind phase search algorithm for carrier phase recovery.

An alternative to the VV phase noise estimator is the so-called BPS algorithm, in which the operation principle is shown in the Figure 2.14. Firstly, a block of $2N + 1$ consecutive received symbols is rotated by a number of B uniformly distributed test phases defined as,

$$\phi_b = \frac{b}{B} \frac{\pi}{2}, b \in \{0, 1, \dots, B - 1\}. \quad (2.9)$$

Then, the rotated blocks symbols are fed into decision circuit, where the square distance to the closest constellation points in the original constellation is calculated for each block. Each resulting square distances block is summed up to minimize the noise distortion. After average filtering, the test phase providing the minimum sum of distances is considered to be the phase noise estimator for the symbol in the middle of the block. The estimated phase noise is then unwrapped to reduce cycle slip occurrence, which is then used employed for the compensation for the phase noise of the original symbols.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.22 Decision Circuit

Header File	:	decision_circuit_*.h
Source File	:	decision_circuit_*.cpp
Version	:	20181012 (Celestino Martins)

This block performs the symbols decision, by calculating the minimum distance between the received symbol relatively to the constellation map. It receives one input complex signal and outputs one complex signal.

Input Parameters

Parameter	Type	Values	Default
mQAM	int	any	4

Table 2.11: Decision circuit input parameters

Methods

```
DecisionCircuitMQAM() ;

DecisionCircuitMQAM(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setmQAM(int mQAMs) mQAM = mQAMs;

double getmQAM() return mQAM;
```

Functional description

This block performs the symbols decision, by minimizing the distance between the received symbol relatively to the constellation map. It perform the symbol decision for the modulations formats, 4QAM and 16QAM. The order of modulation format is defined by the parameter *mQAM*.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

Extend the decision to higher order modulation format.

2.23 Decoder

Header File	:	decoder.h
Source File	:	decoder.cpp

This block accepts a complex electrical signal and outputs a sequence of binary values (0's and 1's). Each point of the input signal corresponds to a pair of bits.

Input Parameters

Parameter	Type	Values	Default
m	int	≥ 4	4
iqAmplitudes	vector<t_complex>	—	{ { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }

Table 2.12: Binary source input parameters

Methods

Decoder()

```
Decoder(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setM(int mValue)
```

```
void getM()
```

```
void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)
```

```
vector<t_iqValues>getIqAmplitudes()
```

Functional description

This block makes the correspondence between a complex electrical signal and pair of binary values using a predetermined constellation.

To do so it computes the distance in the complex plane between each value of the input signal and each value of the *iqAmplitudes* vector selecting only the shortest one. It then converts the point in the IQ plane to a pair of bits making the correspondence between the input signal and a pair of bits.

Input Signals

Number: 1

Type: Electrical complex (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Binary

Examples

As an example take an input signal with positive real and imaginary parts. It would correspond to the first point of the *iqAmplitudes* vector and therefore it would be associated to the pair of bits 00.

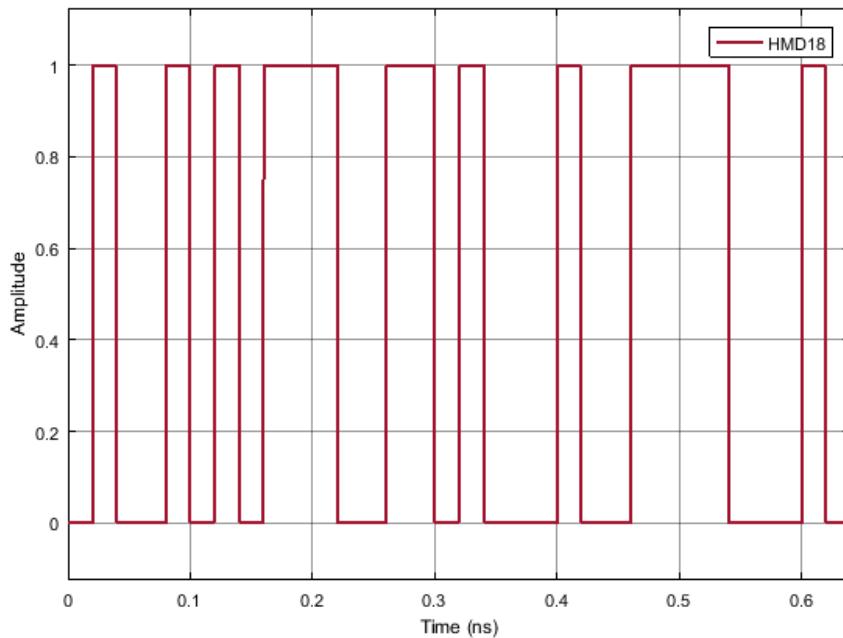


Figure 2.15: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

2.24 Discrete To Continuous Time

Header File	:	discrete_to_continuous_time.h
Source File	:	discrete_to_continuous_time.cpp

This block converts a signal discrete in time to a signal continuous in time. It accepts one input signal that is a sequence of 1's and -1's and it produces one output signal that is a sequence of Dirac delta functions.

Input Parameters

Parameter	Type	Values	Default
numberOfSamplesPerSymbol	int	any	8

Table 2.13: Binary source input parameters

Methods

```
DiscreteToContinuousTime(vector<Signal * > &inputSignals, vector<Signal * > &outputSignals) :Block(inputSignals, outputSignals){};

void initialize(void);

bool runBlock(void);

void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)

int const getNumberOfSamplesPerSymbol(void)
```

Functional Description

This block reads the input signal buffer value, puts it in the output signal buffer and it fills the rest of the space available for that symbol with zeros. The space available in the buffer for each symbol is given by the parameter *numberOfSamplesPerSymbol*.

Input Signals

Number : 1

Type : Sequence of 1's and -1's. (DiscreteTimeDiscreteAmplitude)

Output Signals

Number : 1

Type : Sequence of Dirac delta functions (ContinuousTimeDiscreteAmplitude)

Example

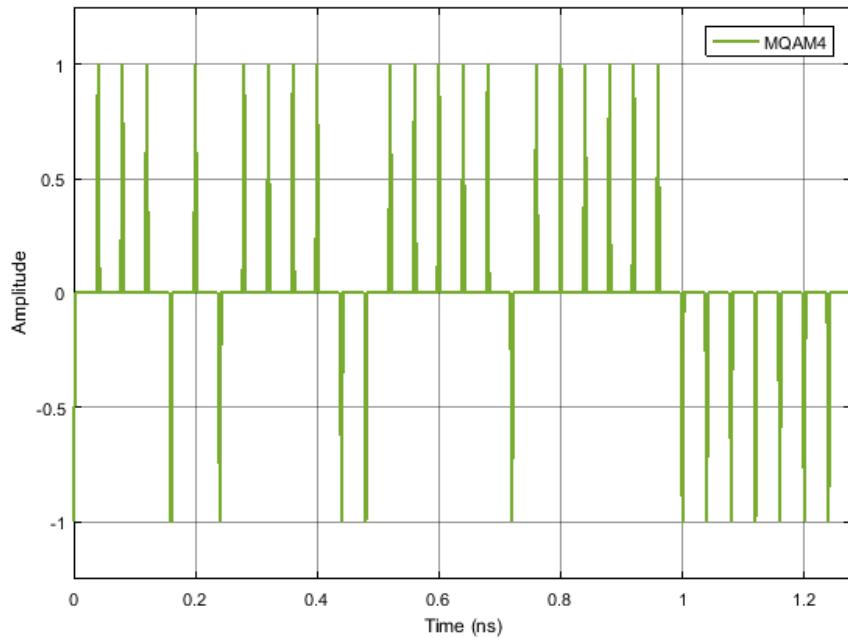


Figure 2.16: Example of the type of signal generated by this block for a binary sequence 0100...

2.25 DownSampling

Header File	:	down_sampling_*.h
Source File	:	down_sampling_*.cpp
Version	:	20180917 (Celestino Martins)

This block simulates the down-sampling function, where the signal sample rate is decreased by integer factor.

Input Parameters

Parameter	Type	Values	Default
downSamplingFactor	int	any	2

Table 2.14: DownSampling input parameters

Methods

```

DownSampling();

DownSampling(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSamplingFactor(unsigned int dSamplingfactor)    downSamplingFactor =
dSamplingfactor;

unsigned int getSamplingFactor() return downSamplingFactor;

```

Functional description

This block perform decreases the sample rate of input signal by a factor of *downSamplingFactor*. Given a down-sampling factor, *downSamplingFactor*, the output signal correspond to the first sample of input signal and every *downSamplingFactor*th sample after the first.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical real signal

Examples

Sugestions for future improvement

2.26 DSP

Header File	:	dsp_*.h
Source File	:	dsp_*.cpp
Version	:	20180423 (Celestino Martins)

This super block simulates the digital signal processing (DSP) algorithms for system impairments compensation in digital domain. It includes the real to complex block, carrier phase recovery block (CPE) and complex to real block. It receives two real input signal and outputs two real signal.

Input Parameters

Parameter	Type	Values	Default
nTaps	int	any	25
NtestPhase	int	any	32
methodType	int	any	[0, 1]
samplingPeriod	double	any	0.0

Table 2.15: DSP input parameters

Methods

```
DSP(vector<Signal *> &InputSig, vector<Signal *> &OutputSig);

void setCPEnTaps(double nTaps) B02.setnTaps(nTaps);

void setCPETestPhase(double TestPhase) B02.setTestPhase(TestPhase);

void setCPESamplingPeriod(double sPeriod) B02.setSamplingPeriod(sPeriod);

void setCPEmethodType(string mType) B02.setmethodType(mType);

void setSamplingPeriod(double sPeriod) B02.setSamplingPeriod(sPeriod); ;
```

Functional description

This super block is composed of three blocks, real to complex block, carrier phase recovery block and complex to real block. The two real input signals are combined into a complex signal using real to complex block. The obtained complex signal is then fed to the CPE block, where the laser phase noise compensation is performed. Finally, the complex output of CPE block is converted into two real signal using complex to real block.

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.27 EDFA

Header File	:	m_qam_receiver.h
Source File	:	m_qam_receiver.cpp

This block mimics an EDFA in the simplest way, by accepting one optical input signal and outputting an amplified version of that signal, affected by white noise.

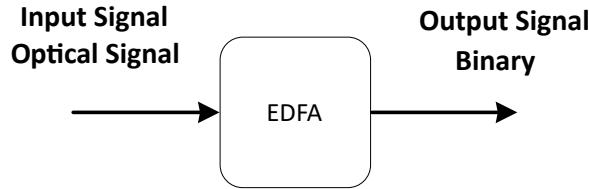


Figure 2.17: Basic configuration of the MQAM receiver

Functional description

This block of code simulates the basic functionality of an EDFA: it amplifies the optical signal by a given gain, and adds noise according to a certain noise figure. Currently the only parameters are the gain and noise figure, and it's assumed that the output power is always far below the EDFA's saturation power. Therefore, the gain and noise spectral density are independent of the signal. The noise spectral density is calculated from the noise figure, gain and wavelength.

This block is made of smaller blocks, and its internal constitution is shown in Figure 2.18.

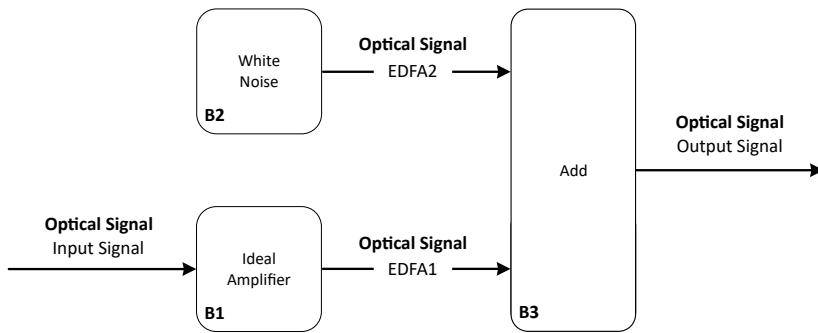


Figure 2.18: Schematic representation of the block homodyne receiver.

Input parameters

Input parameters	Function	Type
powerGain_dB	setGain_dB	t_real
noiseFigure	setNoiseFigure	t_real
samplingPeriod	setNoiseFigure	t_real
wavelength	setWavelength	t_real
dirName	setDirName	string

Table 2.16: List of input parameters of the EDFA block

Methods

Edfa(vector<Signal *> &inputSignal, vector<Signal *> &outputSignal); (**constructor**)0

```

void setGain_dB(t_real newGain)

t_real getGain_dB(void)

void setNoiseFigure(t_real newNoiseFigure)

t_real getNoiseFigure(void)

void setNoiseSamplingPeriod(t_real newSamplingPeriod)

t_real getNoiseSamplingPeriod(void)

void setWavelength(t_real newWavelength)

t_real getWavelength(void)

void setDirName(string newDirName);

string getDirName(void)

```

Input Signals

Number: 1

Type: Optical signal

Output Signals

Number: 1

Type: Optical signal

Example

Sugestions for future improvement

2.28 Electrical Signal Generator

This block generates time continuous amplitude continuous signal, having only one output and no input signal.

2.28.1 ContinuousWave

Continuous Wave the function of the desired signal. This must be introduce by using the function `setFunction(ContinuousWave)`. This function generates a continuous signal with value 1. However, this value can be multiplied by a specific gain, which can be set by using the function `setGain()`. This way, this block outputs a continuous signal with value $1 \times \text{gain}$.

Input Parameters

- `ElectricalSignalFunction` `signalFunction`
`(ContinuousWave)`
- `samplingPeriod{} (double)`
- `symbolPeriod{} (double)`

Methods

```
ElectricalSignalGenerator() {};
void initialize(void);
bool runBlock(void);
void setFunction(ElectricalSignalFunction fun) ElectricalSignalFunction getFunction()
void setSamplingPeriod(double speriod) double getSamplingPeriod()
void setSymbolPeriod(double speriod) double getSymbolPeriod()
void setGain(double gvalue) double getGain()
```

Functional description

The `signalFunction` parameter allows the user to select the signal function that the user wants to output.

Continuous Wave Outputs a time continuous amplitude continuous signal with amplitude 1 multiplied by the gain inserted.

Input Signals

Number: 0

Type: No type

Output Signals

Number: 1

Type: TimeContinuousAmplitudeContinuous

Examples

Sugestions for future improvement

Implement other functions according to the needs.

2.29 Entropy Estimator

Header File	:	entropy_estimator_*.h
Source File	:	entropy_estimator_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal,a binary, and it produces an output signal with the entropy value. No input variables in this block.

Functional Description

This block calculates the entropy of a binary source code composed 0 and 1.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 1

Type: Real (TimeContinuousAmplitudeContinuousReal)

2.30 Entropy Estimator

Functional Description

```
entropyEst(vector<Signal *> &InputSig, int window)
```

The estimator sweeps the full range of the binary input computing an entropy estimation for each window. Then, the entropy mean, the variance and the individual entropy estimations are outputted to a file.

Input Parameters

The block accepts as input parameter the window size, which defines the length over which each entropy estimation is computed.

Note: If the length of the binary stream is not an integer multiple of the window size, the estimator considers the window size equal to the full length of the input signal.

Input Signals

Number: 1

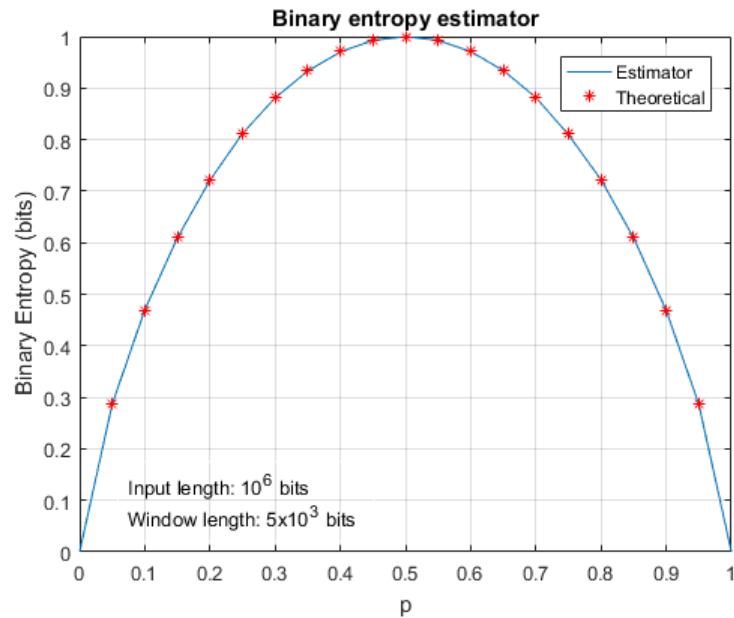
Type: Binary Stream

Output Data

Entropy mean, entropy variance and entropy estimations.

The entropy estimator generates a file with the name "entropy_est.txt" where the output data is written.

Results



2.31 Fork

Header File	:	fork_20171119.h
Source File	:	fork_20171119.cpp
Version	:	20171119 (Student Name: Romil Patel)

Input Parameters

— NA —

Input Signals

Number: 1

Type: Any type (BinaryValue, IntegerValue, RealValue, ComplexValue, ComplexValueXY, PhotonValue, PhotonValueMP, Message)

Output Signals

Number: 2

Type: Same as applied to the input.

Number: 3

Type: Same as applied to the input.

Functional Description

This block accepts any type signal and outputs two replicas of the input signal.

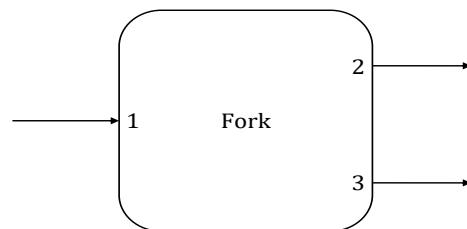


Figure 2.19: Fork

2.32 Gaussian Source

Header File	:	gaussian_source.h
Source File	:	gaussian_source.cpp

This block simulates a random number generator that follows a Gaussian statistics. It produces one output real signal and it doesn't accept input signals.

Input Parameters

Parameter	Type	Values	Default
mean	double	any	0
Variance	double	any	1

Table 2.17: Gaussian source input parameters

Methods

GaussianSource()

```
GaussianSource(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setAverage(double Average);
```

Functional description

This block generates a complex signal with a specified phase given by the input parameter *phase*.

Input Signals

Number: 0

Output Signals

Number: 1

Type: Continuous signal (TimeDiscreteAmplitudeContinuousReal)

Examples

Sugestions for future improvement

2.33 Hamming Decoder

Header File	:	hamming_decoder_*.h
Source File	:	hamming_decoder_*.cpp
Version	:	20180806 (Luís Almeida)

Input Parameters

This block accepts two input parameters (n Bits and k Bits) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair (n, k) are present in columns n and k .

Parity Bits	n (bits)	k (bits)	Hamming Code	Rate
2	3	1	(3, 1)	1/3
3	7	4	(7, 4)	4/7
4	15	11	(15, 11)	11/15
5	31	26	(31, 26)	26/31
6	63	57	(63, 57)	57/63
7	127	120	(127, 120)	120/127
8	255	247	(255, 247)	247/255

Functional Description

This block performs the decoding of the input signal using the selected Hamming Algorithm and outputs the decoded signal.

Input Signals

Number: 1

Type: Binary Signal

Output Signals

Number: 1

Type: Binary Signal

2.34 Hamming Encoder

Header File	:	hamming_encoder_*.h
Source File	:	hamming_encoder_*.cpp
Version	:	20180806 (Luís Almeida)

Input Parameters

This block accepts two input parameters (n Bits and k Bits) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair (n, k) are present in columns n and k .

Parity Bits	n (bits)	k (bits)	Hamming Code	Rate
2	3	1	(3, 1)	1/3
3	7	4	(7, 4)	4/7
4	15	11	(15, 11)	11/15
5	31	26	(31, 26)	26/31
6	63	57	(63, 57)	57/63
7	127	120	(127, 120)	120/127
8	255	247	(255, 247)	247/255

Functional Description

This block performs the encoding of the input signal using the selected Hamming Algorithm and outputs the encoded signal.

Input Signals

Number: 1

Type: Binary Signal

Output Signals

Number: 1

Type: Binary Signal

2.35 MQAM Receiver

Header File	:	m_qam_receiver.h
Source File	:	m_qam_receiver.cpp

Warning: *homodyne_receiver* is not recommended. Use *m_qam_homodyne_receiver* instead.

This block of code simulates the reception and demodulation of an optical signal (which is the input signal of the system) outputing a binary signal. A simplified schematic representation of this block is shown in figure 2.20.



Figure 2.20: Basic configuration of the MQAM receiver

Functional description

This block accepts one optical input signal and outputs one binary signal that corresponds to the M-QAM demodulation of the input signal. It is a complex block (as it can be seen from figure 2.21) of code made up of several simpler blocks whose description can be found in the *lib* repository.

It can also be seen from figure 2.21 that there's an extra internal (generated inside the homodyne receiver block) input signal generated by the *Clock*. This block is used to provide the sampling frequency to the *Sampler*.

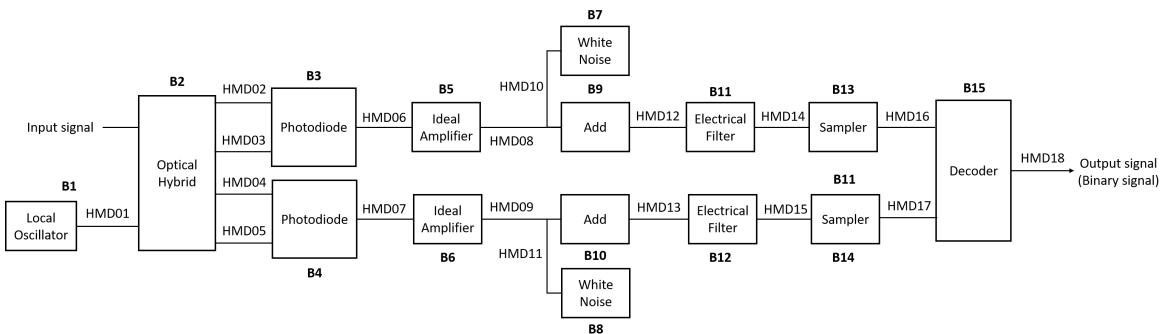


Figure 2.21: Schematic representation of the block homodyne receiver.

Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 2.39) the input parameters and corresponding functions are summarized.

Input parameters	Function	Type	Accepted values
IQ amplitudes	setIqAmplitudes	Vector of coordinate points in the I-Q plane	Example for a 4-QAM mapping: { { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }
Local oscillator power (in dBm)	setLocalOscillatorOpticalPower_dBm	double(t_real)	Any double greater than zero
Local oscillator phase	setLocalOscillatorPhase	double(t_real)	Any double greater than zero
Responsivity of the photodiodes	setResponsivity	double(t_real)	$\in [0,1]$
Amplification (of the TI amplifier)	setAmplification	double(t_real)	Positive real number
Noise amplitude (introduced by the TI amplifier)	setNoiseAmplitude	double(t_real)	Real number greater than zero
Samples to skip	setSamplesToSkip	int(t_integer)	
Save internal signals	setSaveInternalSignals	bool	True or False
Sampling period	setSamplingPeriod	double	Given by $symbolPeriod / samplesPerSymbol$

Table 2.18: List of input parameters of the block MQAM receiver

Methods

HomodyneReceiver(vector<Signal *> &inputSignal, vector<Signal *> &outputSignal)
(constructor)

```
void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)
vector<t_iqValues> const getIqAmplitudes(void)
void setLocalOscillatorSamplingPeriod(double sPeriod)
void setLocalOscillatorOpticalPower(double opticalPower)
void setLocalOscillatorOpticalPower_dBm(double opticalPower_dBm)
void setLocalOscillatorPhase(double lOscillatorPhase)
void setLocalOscillatorOpticalWavelength(double lOscillatorWavelength)
void setSamplingPeriod(double sPeriod)
void setResponsivity(t_real Responsivity)
void setAmplification(t_real Amplification)
void setNoiseAmplitude(t_real NoiseAmplitude)
void setImpulseResponseTimeLength(int impResponseTimeLength)
void setFilterType(PulseShaperFilter fType)
void setRollOffFactor(double rOffFactor)
void setClockPeriod(double per)
void setSamplesToSkip(int sToSkip)
```

Input Signals

Number: 1

Type: Optical signal

Output Signals

Number: 1

Type: Binary signal

Example

Sugestions for future improvement

2.36 Huffman Decoder

Header File	:	huffman_decoder_*.h
Source File	:	huffman_decoder_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal, a binary signal with the message to decode, and it produces an output signal (message decoded). Two inputs are required, the probabilityOfZero and the sourceOrder.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 2.19: Huffman Decoder input parameters

Functional Description

This block decodes a message using Huffman method for a source order of 2, 3 and 4.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 1

Type: Binary

2.37 Huffman Encoder

Header File	:	huffman_encoder_*.h
Source File	:	huffman_encoder_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal,a binary signal with the message to encode, and it produces an output signal (message encoded). Two inputs are required, the probabilityOfZero and the sourceOrder.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 2.20: Huffman Encoder input parameters

Functional Description

This block encodes a message using Huffman method for a source order of 2, 3 and 4.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 1

Type: Binary

2.38 Ideal Amplifier

This block has one input signal and one output signal both corresponding to electrical signals. The output signal is a perfect amplification of the input signal.

Input Parameters

Parameter	Type	Values	Default
gain	double	any	1×10^4

Table 2.21: Ideal Amplifier input parameters

Methods

IdealAmplifier()

```
IdealAmplifier(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig);

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;
```

Functional description

The output signal is the product of the input signal with the parameter *gain*.

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

2.39 IQ Modulator

Header File	:	iq_modulator.h
Source File	:	iq_modulator.cpp
Source File	:	20180130
Source File	:	20180828 (Romil Patel)

Version 20180130

This blocks accepts one input signal continuous in both time and amplitude and it can produce either one or two output signals. It generates an optical signal and it can also generate a binary signal.

Input Parameters

Parameter	Type	Values	Default
outputOpticalPower	double	any	$1e - 3$
outputOpticalWavelength	double	any	$1550e - 9$
outputOpticalFrequency	double	any	speed_of_light/outputOpticalWavelength

Table 2.22: Binary source input parameters

Methods

```
IqModulator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig, OutputSig){};
```

```
void initialize(void);
bool runBlock(void);
void setOutputOpticalPower(double outOpticalPower)
void setOutputOpticalPower_dBm(double outOpticalPower_dBm)
void setOutputOpticalWavelength(double outOpticalWavelength)
void setOutputOpticalFrequency(double outOpticalFrequency)
```

Functional Description

This block takes the two parts of the signal: in phase and in amplitude and it combines them to produce a complex signal that contains information about the amplitude and the phase. This complex signal is multiplied by $\frac{1}{2}\sqrt{\text{outputOpticalPower}}$ in order to reintroduce the information about the energy (or power) of the signal. This signal corresponds to an optical signal and it can be a scalar or have two polarizations along perpendicular axis. It is the signal that is transmitted to the receptor. The binary signal is sent to the Bit Error Rate (BER) measurement block.

Input Signals

Number : 2

Type : Sequence of impulses modulated by the filter
(ContinuousTimeContinuousAmplitude)

Output Signals

Number : 1 or 2

Type : Complex signal (optical) (ContinuousTimeContinuousAmplitude) and binary signal (DiscreteTimeDiscreteAmplitude)

Example

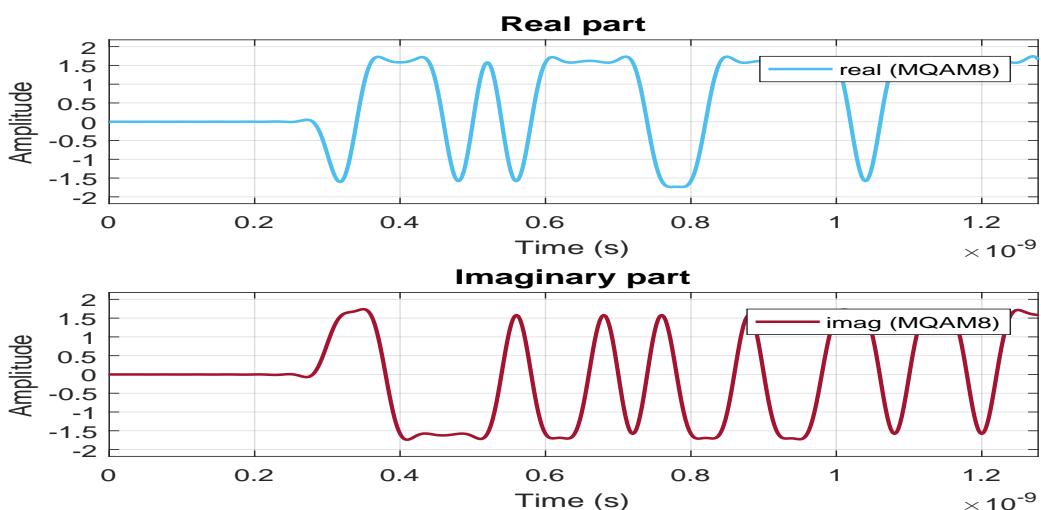


Figure 2.22: Example of a signal generated by this block for the initial binary signal 0100...

Version 20180828

Input Parameters:

—NA—

Input Signals:

Number: 1, 2, 3

Type: RealValue

Output Signals:

Number: 4

Type: RealValue

Functional Description

This blocks has three inputs and one output. Port number 1 and 2 accept the real and imaginary data respectively and port 3 accepts the local oscillator as an input to the IQ modulator. This model serves as an ideal IQ modulator without noise and introduction of nonlinearity.

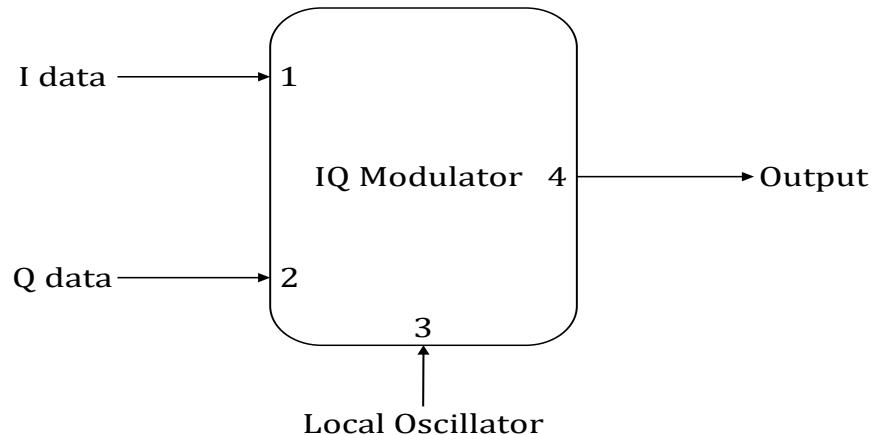


Figure 2.23: IQ Modulator block

IQ MZM Description

The detailed expatiation of the MZM starts with the phase modulator (see Figure ??). The transfer function of the phase modulator can be given as,

$$E_{out}(t) = E_{in}(t) \cdot e^{j\phi_{PM}(t)} = E_{in}(t) \cdot e^{j \frac{u(t)}{V\pi} \pi}$$

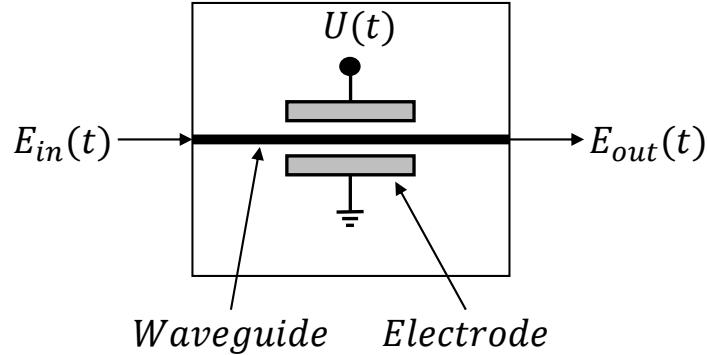


Figure 2.24: Phase Modulator

Two phase modulators can be placed in parallel using an interferometric structure as shown in Figure ???. The incoming light is split into two branches, different phase shifts applies to each path, and then recombined. The output is a result of interference, ranging from constructive (the phase of the light in each branch is the same) to destructive (the phase in each branch differs by π). The transfer function of the structure can be given as,

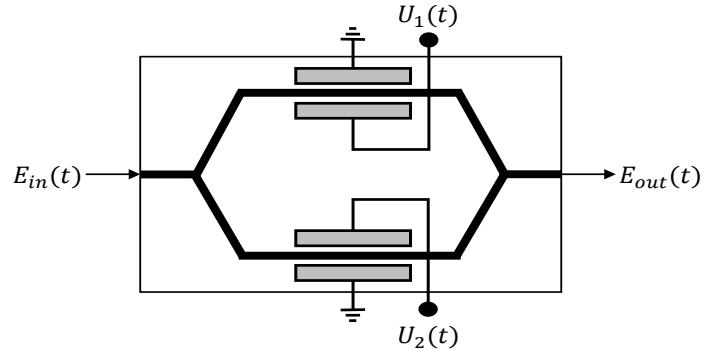


Figure 2.25: Mach-Zehnder Modulator

$$\frac{E_{out}(t)}{E_{in}(t)} = \frac{1}{2} \cdot (e^{j\phi_1(t)} + e^{j\phi_2(t)}) \quad (2.10)$$

Where, $\phi_1(t) = \frac{u_1(t)}{V_{\pi_1}}\pi$ and $\phi_2(t) = \frac{u_2(t)}{V_{\pi_2}}\pi$. if the inputs are set to $u_1 = u_2$ (push-push operation) then it provides the pure phase modulation at the output. Alternatively, if the inputs are set to $u_1 = -u_2$ (push-pull operation) then it provides pure amplitude modulation at the output.

The structure of the IQ MZM can be represented shown in Figure ?? where the incoming source light spitted into two portions. The first portion will drive the MZM of the I-channel and other portion will drive MZM Q-channel data. In the Q-channel, before feeding it to the MZM, it passed though the phase modulator to provide a $\pi/2$ phase shift to the carrier. The output of the MZM combined to form the electrical field $E_{out}(t)$ [NPTEL]. The transfer

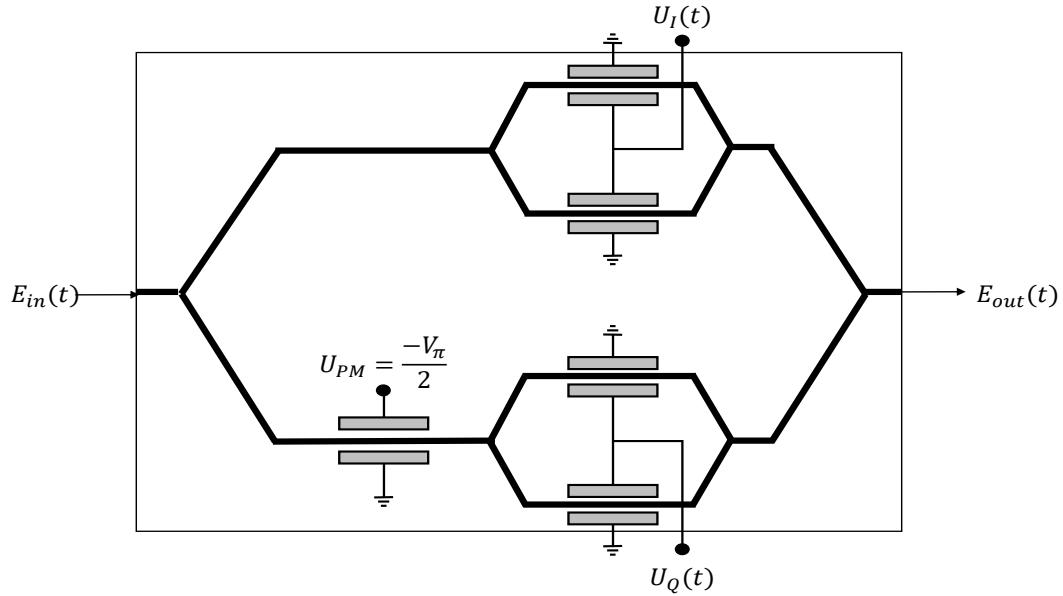


Figure 2.26: IQ Mach-Zehnder Modulator

function of the IQ MZM can be written as,

$$E_{out}(t) = \frac{1}{2}E_{in}(t) \left[\cos\left(\frac{\pi U_I(t)}{2V_\pi}\right) + j \cdot \cos\left(\frac{\pi U_Q(t)}{2V_\pi}\right) \right] \quad (2.11)$$

The black box model of the IQ MZM in the simulator can be depicted as,

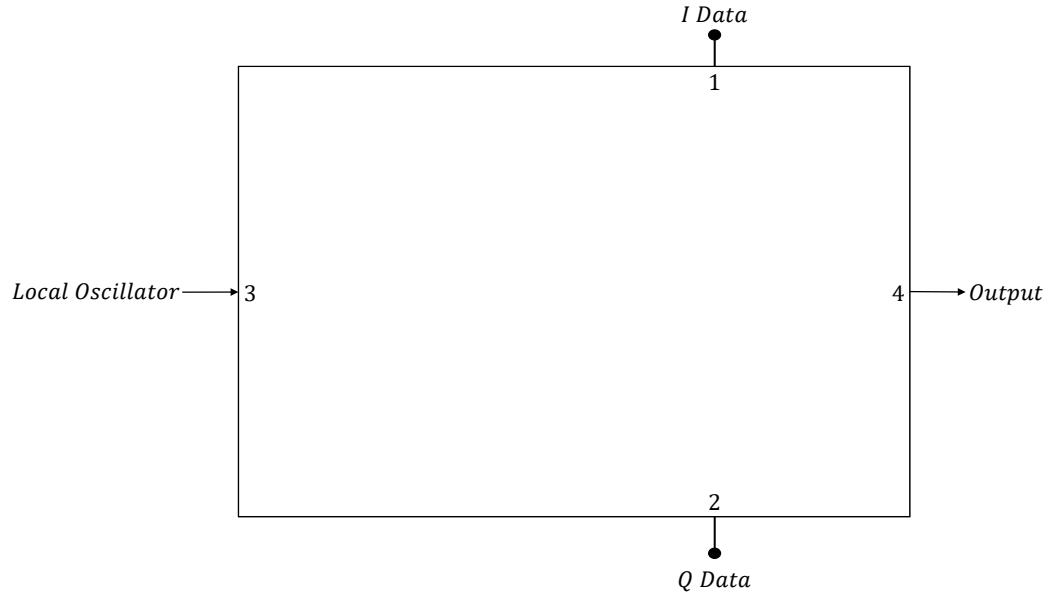


Figure 2.27: Simulation model of the IQ Mach-Zehnder Modulator

2.40 IIR Filter

Header File	:	iir_filter_*.h
Source File	:	iir_filter_*.cpp
Version	:	20180718 (Andoni Santos)

Input Parameters

Name	Type	Default Value

Methods

IIR_Filter()

```
IIR_Filter(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
void initialize(void)
bool runBlock(void)
void setBCoeff(vector<double> newBCoeff)
void setACoeff(vector<double> newACoeff)
int getFilterOrder(void)
```

Input Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This method implements Infinite Impulse Response Filters. Currently it does so by Canonic Realization [jeruchim06].

Theoretical Description

Known Issues

2.41 IP Tunnel MS Windows

Header File	:	ip_tunnel_ms_windows_*.h
Source File	:	ip_tunnel_ms_windows_*.cpp
Version	:	20180815 (João Coelho)

This block works in two ways: one way with one input and zero output signals (as client/sender) and the other one with zero input and one output signal (as server/receiver). IP Tunnel MS Windows is duplicated onto two machines; the first takes samples out of the input buffer until it's empty and transmits them to the IP Tunnel on the second machine through a TCP/IP connection. The IP Tunnel on the second machine receives the signal and outputs it to the output buffer. It's only necessary to specify the ip address of the remote machine and its TCP port.

After some tests on the computer lab, from a total of 100 Mbps of bandwidth on the ethernet port, the average bandwidth used on the simulation is 73 Mbps (around 73%).

Currently, the IP Tunnel works between two machines on the same network or within the same VPN setup. To work between two different networks without a VPN setup, a TCP Nat Traversal/Hole Punching structure would have to be implemented. This consists in a new module, that works as an intermediary between the client and the server. These, connect to the mediator (getting their IP addresses), following with the transmission of each other's addresses and its connection.

Only works on windows due to the Ws2tcpip.h header and ws2_32 library and it is compatible and tested with SDK v8.1 and v10.0.18362.0.

Input Parameters

Parameter	Type	Values	Default	Brief Description
displayNumberOfSamples	bool	true/false	true	If true, number of samples sent and received are displayed.
numberOfTrials	int	any	10	Number of trials after a connection has been refused.
remoteMachineIpAddress	string	any	"127.0.0.1"	IP Address of Remote Machine.
tcpPort	int	any	54000	TCP port used to connect to server.
timeIntervalSeconds	int	any	3	Time interval before trying to connect again after a connection has been refused (seconds).

Table 2.23: IP Tunnel MS Windows input parameters

Methods

```
IPTunnel(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)

void initialize(void);

bool runBlock(void);

void terminate(void);

void ipTunnelSendInt(int space);

int ipTunnelRecvInt();

int ipTunnelPut(T object);

void setDisplayNumberOfSamples(bool opt);

bool getDisplayNumberOfSamples();

bool server();

bool client();

int ipTunnelPut(T object, int objectSize);

int ipTunnelPut(T(&object)[N], int elementSize, int processf);

bool ipTunnelRecvValues(vector <Signal*> outputS, int processf, signal_value_type
stypef);

bool ipTunnelRecvMessages(vector <Signal*> outputS, int processf);

bool ipTunnelOutputData(vector <Signal*> outputS, signal_value_type sType);

void setRemoteMachineIpAddress(string rMachineIpAddress);

string getRemoteMachineIpAddress();

void setTcpPort(int tcpP);

int getTcpPort();

void setNumberOfTrials(int nOfTrials);

int getNumberOfTrials();

void setTimeIntervalSeconds(int tIntervalSeconds);

int getTimeIntervalSeconds();
```

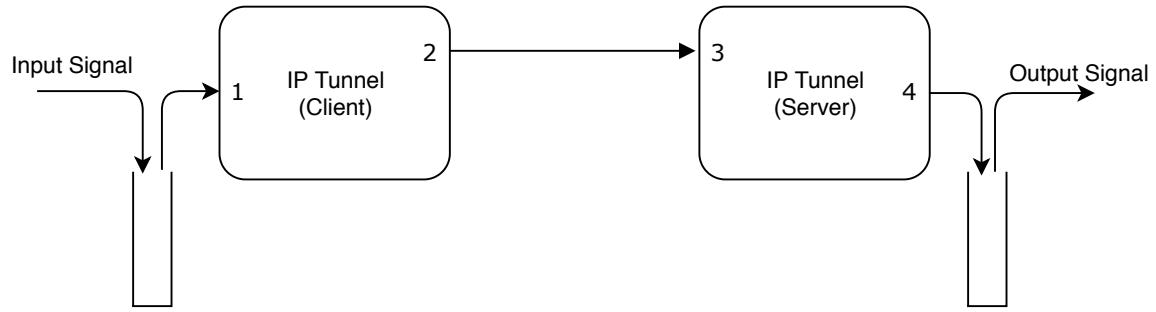


Figure 2.28: IP Tunnel MS Windows structure

Functional Description

The IP Tunnel block transfers signals from one machine to another so it continues the simulation in two different computers. This block is duplicated onto two machines, one with only input (client) and other with only output signals (server). After being executed, the input signal's buffer (1) will be empty and this block will transmit this signal to the other block. The second block will output this signal to the output buffer (4). An architecture "Server - Client" is used to establish a TCP/IP channel between the two blocks (2 and 3). The block without input signal is the server (receiver) and the block with input signal is the client (transmitter). After the connection is established, server sends the "space" of its buffer (maximum signal size that can be received) and client responds with the "process" (signal size that is going to be transmitted) and with an integer representing the type of signal. Subsequently, the transmission of the signal starts and the simulation continues normally on both machines.

The method to send the signal "t_message" is quite different from the other signals: client first deconstructs the object "t_message" into its 3 parameters (message type, message data length and message data). Then, after converting the message data to a char array, the size of the array and each message parameter is sent to the server. On the server side, the message is constructed with its parameters so it can be sent to the output buffer.

Bandwidth testing

This section was created to test the efficiency of the module in two environments: with Ethernet and with VPN. In the folder "sdf" there are two other folders with the names "ip_tunnel_ms_windows_bandwidth_test_tx" for the client and "ip_tunnel_ms_windows_bandwidth_test_rx" for the server corresponding to simulations between the two different computers.

The first experiment is between two computers in the same network with ethernet and bandwidth of 100Mbps. The test occurs running the two simulations at the same time, being necessary to configure the number of bits to be sent. The default is 8 million bits. Transmitting this size of bits (1MB) in this setting, took about 148 seconds.

The second experiment is between two computers on different networks using VPN. This

experiment was conducted in two machines running windows 10 using the windows VPN configuration. On one machine, it was initiated the outgoing VPN connection and on the second machine, the incoming VPN connection was established. After this, the simulations are done the same way of the first experiment. The results are very similar with the first experiment, as the module doesn't lose bandwidth taking about 153 seconds to transfer 1MB.

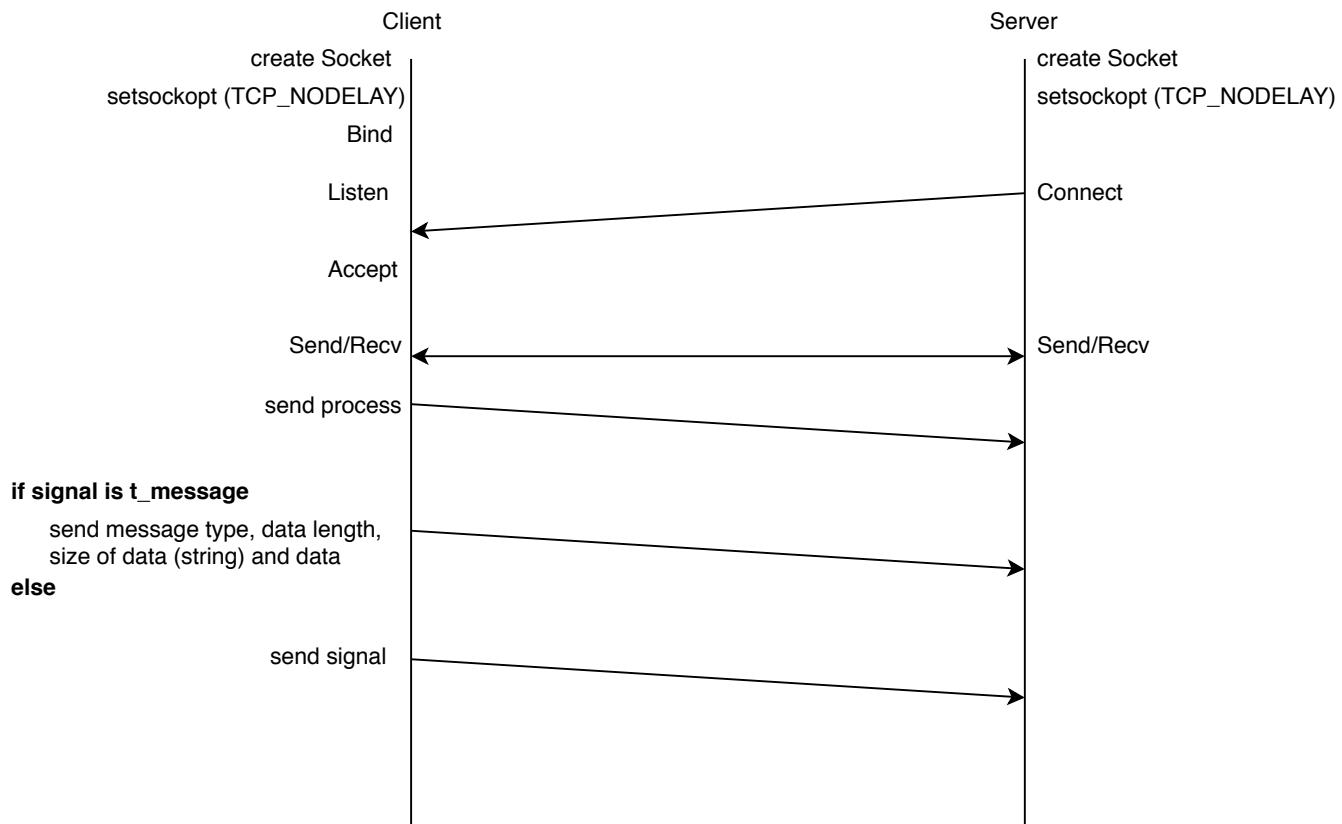


Figure 2.29: IP Tunnel MS Windows fluxogram

2.42 Local Oscillator

Header File	:	local_oscillator.h
Source File	:	local_oscillator.cpp
Version	:	20180130
Version	:	20180828 (Romil Patel)

Version 20180130

This block simulates a local oscillator with constant power and initial phase. It produces one output complex signal and it doesn't accept input signals.

Input Parameters

Parameter	Type	Values	Default
opticalPower	double	any	1e - 3
outputOpticalWavelength	double	any	1550e - 9
outputOpticalFrequency	double	any	SPEED_OF_LIGHT / outputOpticalWavelength
phase	double	$\in [0, \frac{\pi}{2}]$	0
samplingPeriod	double	any	0.0
symbolPeriod	double	any	0.0
signaltoNoiseRatio	double	any	0.0
laserLineWidth	double	any	0.0
laserRIN	double	any	0.0

Table 2.24: Binary source input parameters

Methods

LocalOscillator()

```
LocalOscillator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setSymbolPeriod(double sPeriod);

void setOpticalPower(double oPower);
```

```
void setOpticalPower_dBm(double oPower_dBm);  
void setWavelength(double wlength);  
void setFrequency(double freq);  
void setPhase(double lOscillatorPhase);  
void setSignaltoNoiseRatio(double sNoiseRatio);  
void setLaserLinewidth(double laserLinewidth);  
void setLaserRIN(double laserRIN);
```

Functional description

This block generates a complex signal with a specified phase given by the input parameter *phase*.

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal

Version 20180828

2.43 Local Oscillator

Header File	:	local_oscillator.h
Source File	:	local_oscillator.cpp

This block simulates a local oscillator with constant power and initial phase. It produces one output complex signal and it doesn't accept input signals.

Input Parameters

Parameter	Type	Values	Default
opticalPower	double	any	1e - 3
outputOpticalWavelength	double	any	1550e - 9
outputOpticalFrequency	double	any	SPEED_OF_LIGHT / outputOpticalWavelength
phase0	double	$\in [0, \frac{\pi}{2}]$	0
samplingPeriod	double	any	0.0
laserLW	double	any	0.0
laserRIN	double	any	0.0

Table 2.25: Local oscillator input parameters

Methods

LocalOscillator()

```
LocalOscillator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setPhase(double lOscillatorPhase);

void setLaserLinewidth(double laserLinewidth);
```

```
double getLaserLinewidth();  
  
void setLaserRIN(double LOlaserRIN);  
  
double getLaserRIN();
```

Functional description

This block generates a complex signal with a specified initial phase given by the input parameter *phase0*. The phase noise can be simulated by adjusting the laser linewidth in parameter *laserLW*. The relative intensity noise (RIN) can be also adjusting according to the parameter *laserRIN*.

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal

Examples

Sugestions for future improvement

2.44 Mutual Information Estimator

Header File	:	mutual_information_estimator_20180723.h
Source File	:	mutual_information_estimator_20180723.cpp

This block estimates the mutual information between the input and output channel symbols X and Y, respectively. Each input signal x_j ($j = 1, 2, \dots, J$) has a specific probability being possible calculate the entropy of the alphabet X, $H(X)$. The uncertainty of the observation regarding with the occurrence of the input symbol is measured by the entropy $H(X)$, which is maximum when all inputs have the same probability. Another concept important to estimate the mutual information is the conditional entropy $H(X|Y = y_k)$, which represents the uncertainty related with the channel input symbols X that stays after the observation of the output symbols Y. This way, the difference $H(X) - H(X|Y)$ is on average the amount of information gained by the observer with respect with the channel input based on the channel output symbol. This difference is called the mutual information:

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^K \sum_{j=1}^J P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \quad (2.12)$$

where K corresponds to the number of possible output symbols and J corresponds to the number of possible input symbols.

This block uses the property 1.9.3 in [1] of Mutual information which tells that it can be determined using the formula:

$$I(X; Y) = H(Y) - H(Y|X). \quad (2.13)$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0 $P(X = 0)$ and assumes that the complementary of this probability corresponds to the probability of the input bit is 1 $P(X = 1)$. $P(X = 0)$ corresponds to the α probability calculated in this block. Furthermore, another probability of interest is p which corresponds to the error probability of the channel. Both α and p are estimated in this block.

In order to calculate the mutual information, from equation 2.31 we should calculate the conditional entropy $H(Y|X)$ and the entropy of the channel outputs $H(Y)$. First, lets calculate $H(Y|X)$:

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^K \sum_{j=1}^J P(y_k|x_j) P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) + \bar{\alpha}(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= H(p), \end{aligned} \quad (2.14)$$

which means that the conditional entropy depends only on the channel properties and it does not depend on the channel input statistics. Now, to calculate the entropy of the channel output symbols we need to calculate the channel output probabilities, i.e $P(Y = 0)$ and $P(Y = 1)$.

$$\begin{aligned} P(Y = 0) &= P(Y = 0|X = 0)P(X = 0) + P(Y = 0|X = 1)P(X = 1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \quad (2.15)$$

Since the output channel symbol only has two possible values (0 or 1),

$$\begin{aligned} P(Y = 1) &= P(Y = 1|X = 1)P(X = 1) + P(Y = 1|X = 0)P(X = 0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y = 0). \end{aligned} \quad (2.16)$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \quad (2.17)$$

and the mutual information should be calculated using the following formula:

$$I(X; Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \quad (2.18)$$

The upper and lower bounds, $I(X;Y)_{UB}$ and $I(X;Y)_{LB}$ respectively, are calculated using the method of Coppler-Pearson as described in section 2.11 for bit error rate calculation. This way, it returns the simplified expression:

$$I(X;Y)_{UB} = I(X;Y) + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X;Y)(1 - I(X;Y))} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - I(X;Y) \right) z_{\alpha/2}^2 + (2 - I(X;Y)) \right] \quad (2.19)$$

$$I(X;Y)_{LB} = I(X;Y) - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X;Y)(1 - I(X;Y))} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - I(X;Y) \right) z_{\alpha/2}^2 - (1 + I(X;Y)) \right], \quad (2.20)$$

where $z_{\alpha/2}$ is the $100(1 - \frac{\alpha}{2})$ th percentile of a standard normal distribution and N_T the total number of bits used to calculate the mutual information.

Input Parameters

Name	Type	Default Value
m	integer	0
alpha_bounds	double	0.05

Methods

- MutualInformationEstimator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig) :Block(InputSig,OutputSig){};

- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setConfidence(double P) { alpha = 1-P; }

Input Signals

Number: 2

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1

Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional Description

This block accepts two binary strings and outputs a binary string, outputting a 0 if the two input samples are equal to each other and 1 if not. This block also outputs .txt files with a report of the estimated Mutual Information, as well as the error probability of the channel estimated p and the estimated probability of $X = 0$, α . Furthermore, the mutual information estimator block can output middle report files with size m set by the user using the method `setMidReportSize(int M)`, i.e the mutual information calculated uses m input symbols in its calculation. However, a final report is always outputted using all symbols transmitted.

The block receives two input binary strings, one with the sequence of input channel symbols and from this it calculates the probability of the input symbol is equals to 0, α , and other sequence with the output channel symbols and it compares the bit from this sequence with the correspondent bit from the first sequence (i.e the sequence with the input channel symbols). If the bit in the output channel symbol is different from the correspondent bit in the input channel symbol sequence, it counts as an error and the error probability of the channel is calculated based on the final number of errors, p . Both probabilities α and p allow the block to estimate the conditional entropy and the entropy of the output channel symbols, allowing the calculation of the mutual information.

References

- [1] Krzysztof Wesolowski. *Introduction to digital communication systems*. John Wiley & Sons, 2009.

2.45 MQAM Mapper

Header File	:	m_qam_mapper.h
Source File	:	m_qam_mapper.cpp

This block does the mapping of the binary signal using a m -QAM modulation. It accepts one input signal of the binary type and it produces two output signals which are a sequence of 1's and -1's.

Input Parameters

Parameter	Type	Values	Default
m	int	2^n with n integer	4
iqAmplitudes	vector<t_complex>	—	{ { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }

Table 2.26: Binary source input parameters

Methods

```
MQamMapper(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig) {}

void initialize(void);

bool runBlock(void);

void setM(int mValue);

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues);
```

Functional Description

In the case of $m=4$ this block attributes to each pair of bits a point in the I-Q space. The constellation used is defined by the *iqAmplitudes* vector. The constellation used in this case is illustrated in figure 2.30.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

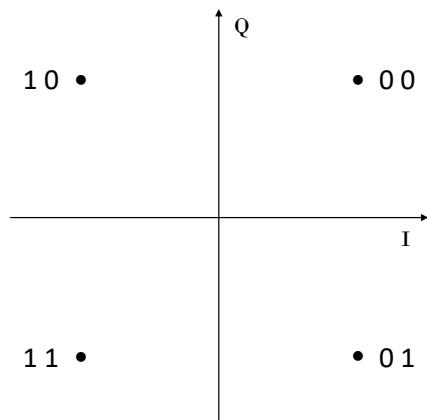


Figure 2.30: Constellation used to map the signal for $m=4$

Output Signals

Number : 2

Type : Sequence of 1's and -1's (DiscreteTimeDiscreteAmplitude)

Example

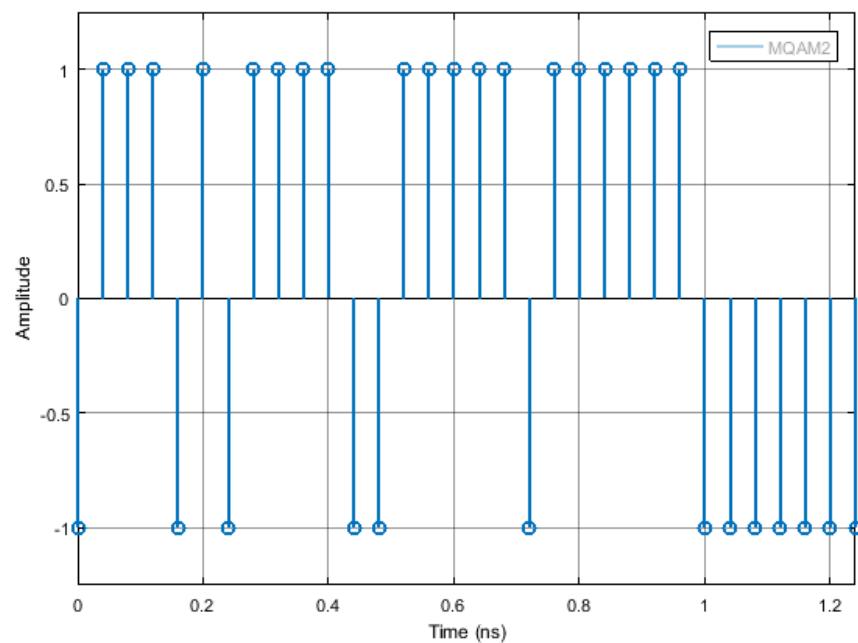


Figure 2.31: Example of the type of signal generated by this block for the initial binary signal 0100...

2.46 M-QAM Receiver

Header File	:	m_qam_receiver.h
Source File	:	m_qam_receiver.cpp
Version	:	20180815 (Manuel Neves)

This block simulates the reception and demodulation of an optical signal (which is the input signal of the system) and outputs a binary signal corresponding to the reconstructed transmitted bitstream. A simplified schematic representation of this block is shown in figure 2.32.

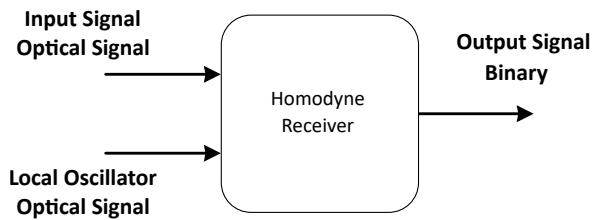


Figure 2.32: Simplified model of the MQAM receiver

It also optionally logs in a file the information about the system flow and stores all internal signals for posterior analysis.

Signals

Input Signals

Number: 2

Type: Optical

These two input signals are the optical signal modulated with the information, from the optical fiber channel, and the optical signal from the local oscillator of the receiver.

Output Signals

Number: 1

Type: Binary

The output signal is the binary sequence demodulated from the optical signal.

Input Parameters

Name	Type	Default Value	Description
logValue	bool	true	If true, the log file will be printed.
logFileName	string	"SuperBlock_MQamReceiver.txt"	Name of the log file.
signalsFolderName	string	"signals/SuperBlock_MQamReceiver"	Name of the directory where the internal signals are saved.

Note that, due to the big amount of parameters involved in this super block, here there's only mention of the input parameters that are strictly related with the M-QAM Receiver super block. To address input parameters related to a block included in the receiver, please refer to the documentation of such block.

You can however, by having a look on the available methods, infer which parameters can be controlled from the scope of the M-QAM Receiver.

Methods

Block Constructor

- **MQamReceiver(initializer_list<Signal * > inputSig, initializer_list<Signal * > outputSig);**

This block's constructor expects, as any *Block* object, an input of two vectors of Signal pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

Methods

Methods to start and run the block:

- void **initialize(void);**
 - Sets up the input and output signals and updates the parameters related to the super block.
- bool **runBlock(void);**

- This method is responsible for iterating over the several blocks contained on the receiver and running them until the signal buffers are full or all data has been processed.

Methods to configure the Photodiodes:

- void **setPhotodiodesResponsivity**(t_real Responsivity);

Methods to configure the TI Amplifiers:

- void **setGain**(t_real gain);
- t_real **getGain**(void);
- void **setAmplifierInputNoisePowerSpectralDensity**(t_real NoiseSpectralDensity);
- t_real **getAmplifierInputNoisePowerSpectralDensity**(void);
- void **setTiAmplifierFilterType**(Filter fType);
- void **setTiAmplifierCutoffFrequency**(double ctfFreq);
- void **setTiAmplifierImpulseResponseTimeLength_symbolPeriods**(int irl);
- void **setElectricalFilterImpulseResponse**(vector<t_real> ir);
- void **setElectricalImpulseResponseFilename**(string fName);
- void **setElectricalSeeBeginningOfImpulseResponse** (bool sBeginningOfImpulse Response);
- double const **getElectricalSeeBeginningOfImpulseResponse**(void);

Methods to configure the General Noise:

- void **setNoiseSamplingPeriod**(t_real SamplingPeriod);
- void **setNoiseSymbolPeriod**(t_real nSymbolPeriod);

Methods to configure the Thermal Noise:

- void **setThermalNoiseSpectralDensity**(t_real NoiseSpectralDensity);
- void **setThermalNoisePower**(t_real NoiseSpectralDensity);
- void **setThermalConstantPower**(bool cp);
- void **setSeeds**(array<int, 2> noiseSeeds);
- void **setSeedType**(SeedType seedType);

Methods to configure the Pulse Shapers:

- void **setImpulseResponseTimeLength**(int impResponseTimeLength);
- void **setFilterType**(pulse_shapper_filter_type fType);
- void **setRollOffFactor**(double rOffFactor);
- void **usePassiveFilterMode**(bool pFilterMode);
- void **setRrcNormalizeEnergy**(bool ne);
- void **setMFImpulseResponseFilename**(string fName);
- void **setMFSeeBeginningOfImpulseResponse** (bool sBeginningOfImpulse Response);
- double const **getMFSeeBeginningOfImpulseResponse**(void);

Methods to configure the Samplers:

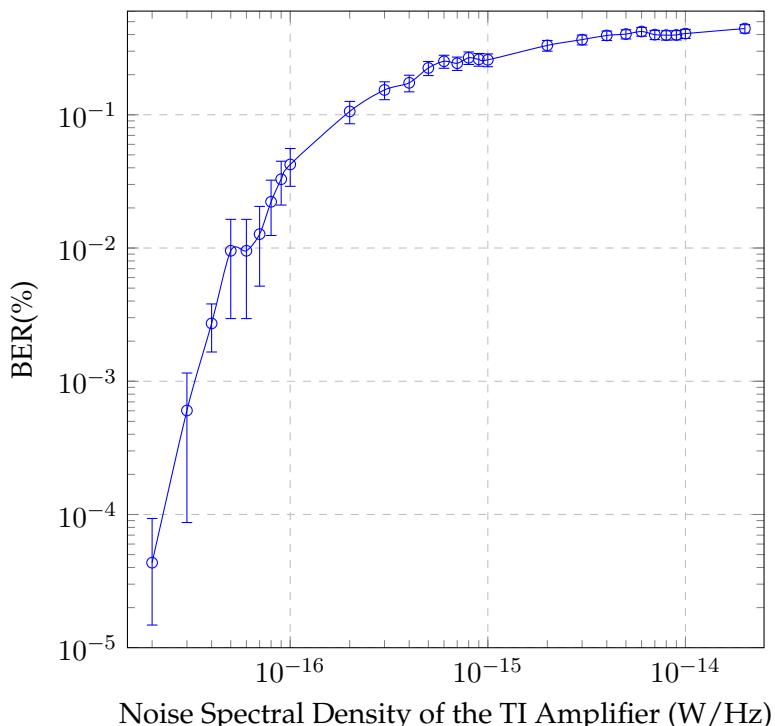
- void **setSamplesToSkip**(int sToSkip);

Methods to configure the Decoder:

- void **setIqAmplitudes**(vector<t_iqValues> iqAmplitudesValues);
- vector<t_iqValues> const **getIqAmplitudes**(void);

Examples

- Impact of the noise on the TI Amplifier on the BER:



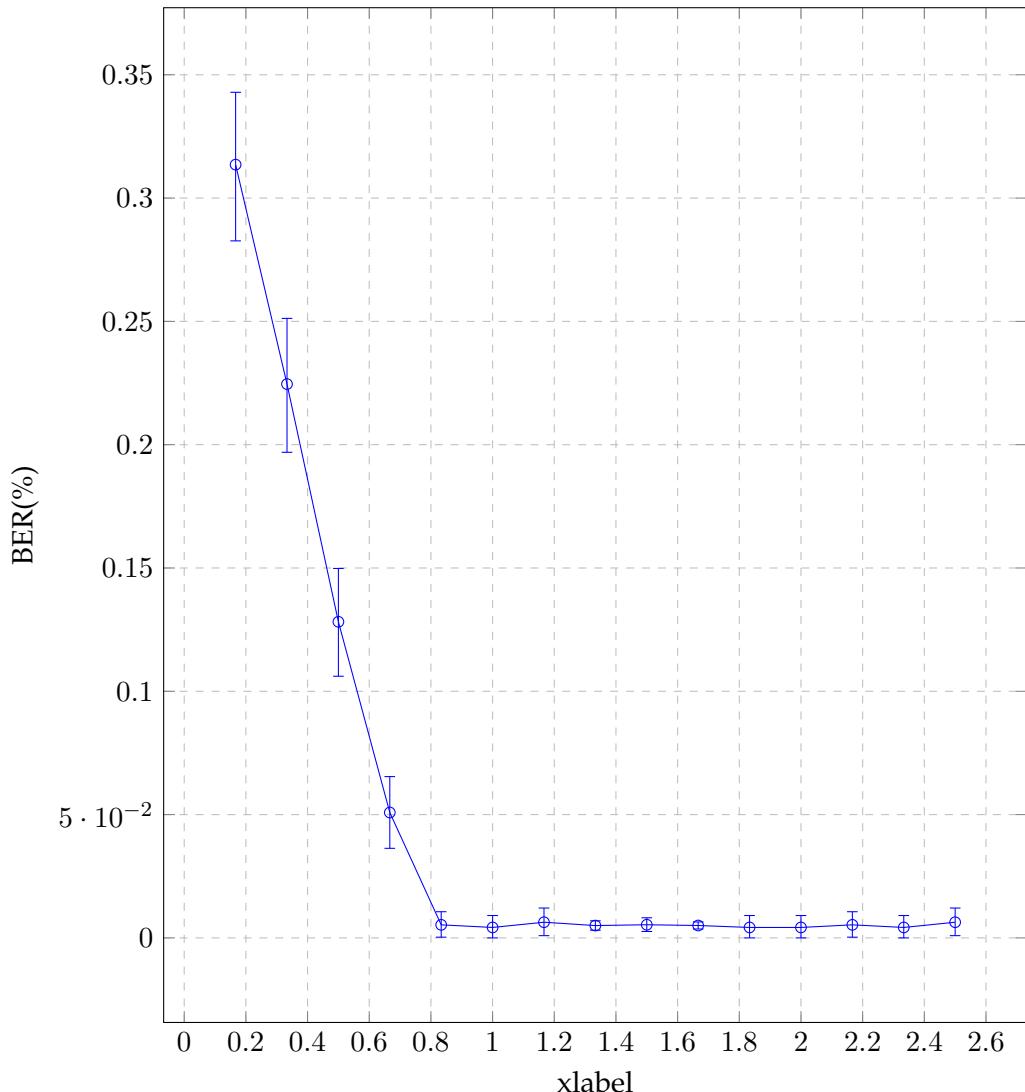
We can observe that the BER decreases at an enlarging rate, as we augment the spectral density of the noise on the TI Amplifier. Giving us an estimate of the sensitivity of the receiver to thermal noise.

- Impact of the bandwidth of the TI Amplifier:

On the following graph we can see the impact of the bandwidth of the electrical amplifier on the BER.

For the horizontal axes the used metric will be the ratio between the signal bandwidth and the TI Amplifier bandwidth, to better understand how these to relate, and the minimum requirements for a well suited electrical amplifier.

For the test, thermal noise will also be inserted, to also evaluate the impact of a too broad bandwidth.



We can see that, on the simulator, the bandwidth of the TI Amplifier only has a significant effect when it is smaller than 80% of the transmitted signal bandwidth. For

higher bandwidths of the TI Amplifier (higher than the signal bandwidth) the simulator demonstrates no increase in the BER.

Functional description

This block has an input of two optical signals, the received signal from the transmission channel and the signal from the local oscillator. These signals pass through a set of blocks, as it can be seen in figure 2.33, and then it outputs one binary signal that corresponds to the decoded information transmitted in the input signal from the transmission channel.

With this block we needn't to manually make all the connections between the different blocks needed to implement an optical receiver. All we need to do is create an instance of the block and only modify the parameters whose default values are not in accordance with our needs. All the methods available to do so have already been presented above.

It is a super block of higher complexity than the M-QAM Transmitter, making it harder to give a general overview of the signal flow, thus it is important to emphasize that, for any more specific doubts the documentation of the singular blocks should be addressed.

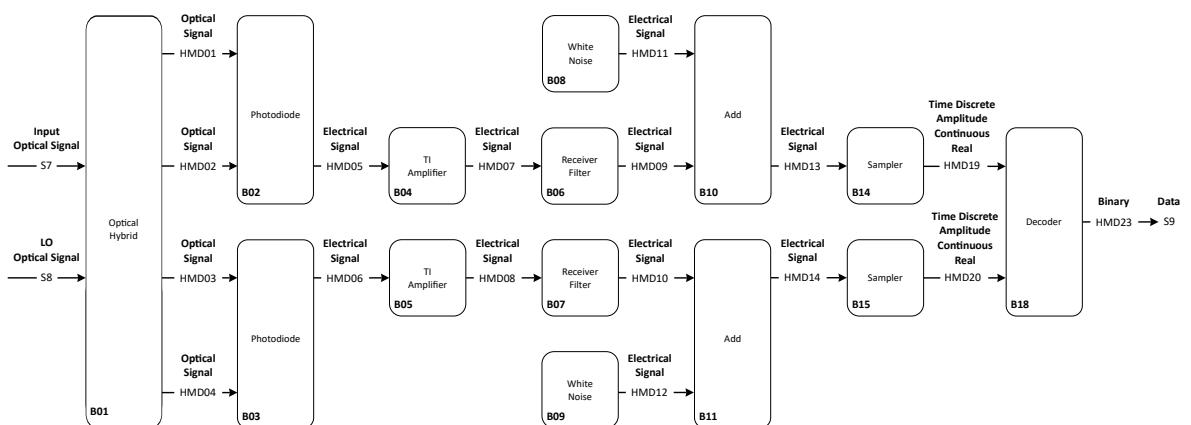


Figure 2.33: Schematic representation of the block homodyne receiver.

The block Optical hybrid alongside with the photodetectors mix the oscillator signal with the input signal in order to convert a complex optical signal into two real electrical signals, these represent the in-phase and the quadrature components of the received signal.

We are now in the electrical domain and the signals are a function of the current in the photodetectors, but still working with very low amplitudes, thus we need to have an electrical transimpedance amplifier, that will increase the signal power. Then, noise is added to simulate thermal noise present in all electronics.

After having strong electrical signals, we need to sample them, for them to be processed by a digital signal processor.

Finally, the in-phase and quadrature signals, discrete in time but continuous in amplitude, are processed by a decoder, which translates pairs of amplitudes to a sequence of binary digits, using the maximum likelihood approach.

Open issues

- The decoder only works for constellations in which all points have the same distance from the origin (example: QPSK). This is due to the fact that the decoder only takes into consideration which point of the constellation is closer to the received coordinates, but not the amplitude of the received signal. This means the decoder doesn't work properly for 16-QAM signals, making it impossible to analyse BER for cases that don't comply with the condition of all points being at the same distance from the origin;
- The formula that is responsible for inserting the effect of the quantum noise on the photodetectors seems to be wrong, since the shot noise power is always much smaller than the signal power, even for values of 100dBm on the receiver's local oscillator;
- There are a lot of parameters from the blocks included in the Receiver super block that can't be accessed/modified through methods of the Receiver, making it hard to configure the various parameters available for each block that constitutes the Receiver block.

Future improvements

- Besides the binary signal, there could be an additional output signal that would easily allow us to observe the received constellation with the noise added by the receiver block;
- Correction of the several issues.

2.47 MQAM Transmitter

Header File	:	m_qam_transmitter.h
Source File	:	m_qam_transmitter.cpp
Contributors	:	Andoni Santos
	:	Manuel Neves
	:	Armando Pinto

This block receives a binary sequence and an optical signal from the local oscillator, with these, it generates a MQAM optical signal. A schematic representation of this block is shown in figure 2.34.

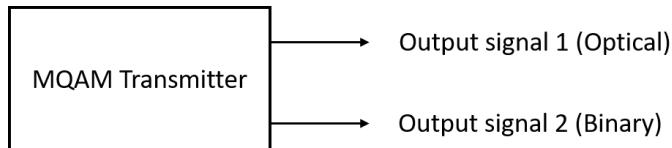


Figure 2.34: Basic configuration of the MQAM transmitter

Signals

Input Signals

Number: 2

Type: Binary and Optical

These two input signals are the binary sequence to be transmitted, from the binary source, and the optical signal, directly from the local oscillator of the transmitter.

Output Signals

Number: 1

Type: Optical

The output signal is an optical signal resultant from the modulation of the binary signal at the input with the optical signal from the local oscillator.

This block generates an optical signal (output signal 1 in figure 2.39). The binary signal generated in the internal block Binary Source (block B1 in figure 2.39) can be used to perform a Bit Error Rate (BER) measurement and in that sense it works as an extra output signal (output signal 2 in figure 2.39).

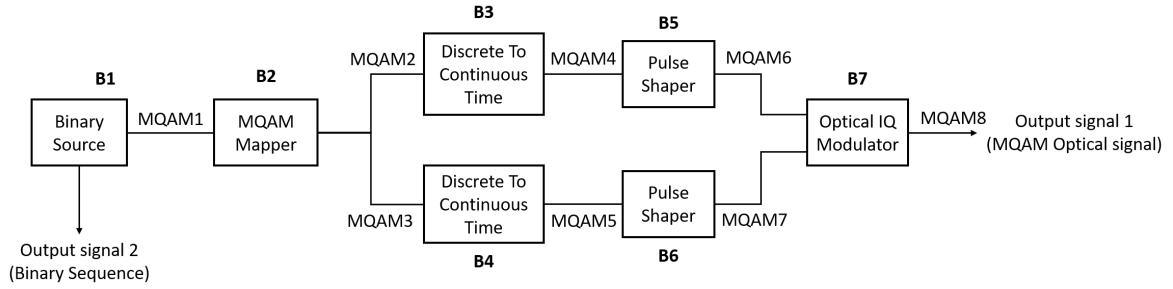


Figure 2.35: Schematic representation of the block MQAM transmitter.

Input Parameters

Name	Type	Default Value	Description
logValue	bool	true	If true, the log file will be printed.
logFileName	string	"SuperBlock_MQamTransmitter.txt"	Name of the log file.
signalsFolderName	string	"signals/SuperBlock_MQamTransmitter"	Name of the directory where the internal signals are saved.

Table 2.27: Parameters related to the super block

Name	Type	Default Value	Description
m	int	4	Cardinality.
iqAmplitudes	vector<vector<t_real>>	$\{\{1,1\}, \{-1,1\}, \{1,-1\}, \{-1,-1\}\}$	Amplitudes of the points of the constellation.
firstTime	bool	true	First time value on the block M-QAM Mapper.
numberOfSamplesPerSymbol	int	8	Number of samples per symbol.
impulseResponseTimeLength	int	16	Temporal length of the impulse response in multiples of the symbol period.
filterType	pulse_shapper_filter_type	RaisedCosine	Type of the shaping filter on the pulse shaper block.
rollOffFactor	double	0.9	Roll-off factor for the raised-cosine filter.
pulseWidth	double	5e-10	Width of the pulse.
passiveFilterMode	bool	false	When true the filter is passive, meaning the amplitudes are normalized.

Table 2.28: Configurable parameters related to the blocks inside the MQAM Transmitter

Block Constructor

- **MQamTransmitter(initializer_list<Signal * > inputSig, initializer_list<Signal * > outputSig);**

This block's constructor expects, as any *Block* object, an input of two vectors of *Signal* pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

Methods

Methods to start and run the block:

- void **initialize(void);**
 - Sets up the input and output signals and updates the parameters related to the super block.
- bool **runBlock(void);**
 - This method is responsible for iterating over the several blocks contained on the transmitter and running them until the signal buffers are full or all data has been processed.

Methods to access/modify system parameters:

- void **setM(int mValue);**
- void **setIqAmplitudes(vector<vector<double>> iqAmplitudesValues);**
- void **setFirstTime(bool fTime);**
- bool **getFirstTime(void);**
- void **setNumberOfSamplesPerSymbol(int nSamplesPerSymbol);**
- int const **getNumberOfSamplesPerSymbol(void);**
- void setImpulseResponseTimeLength_symbolPeriods(int impResponseTime Length);
- int const **getImpulseResponseTimeLength_symbolPeriods(void);**
- void **setFilterType(pulse_shapper_filter_type fType);**
- pulse_shapper_filter_type const **getFilterType(void);**
- void **setRollOffFactor(double rOffFactor);**
- double const **getRollOffFactor(void);**
- void **setPulseWidth(double pWidth);**

- double const **getPulseWidth**(void);
- void **setPassiveFilterMode**(bool pFilterMode);
- bool const **getPassiveFilterMode**(void);

MQamTransmitter(vector<Signal *> &inputSignal, vector<Signal *> &outputSignal);
(constructor)

```
void set(int opt);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

void setM(int mValue) int const getM(void)

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setNumberOfSamplesPerSymbol(int n)

int const getNumberOfSamplesPerSymbol(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor(void)
```

```
void setSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)
double const getSeeBeginningOfImpulseResponse(void)
void setOutputOpticalPower(t_real outOpticalPower)
t_real const getOutputOpticalPower(void)
void setOutputOpticalPower_dBm(t_real outOpticalPower_dBm)
t_real const getOutputOpticalPower_dBm(void)
```

Examples

- Impact of the constellation coding on the BER:

For this example we will analyse how Grey encoding on the constellation optimizes the BER at the receiver in the presence of noise. To achieve that we will use the two following constellations:

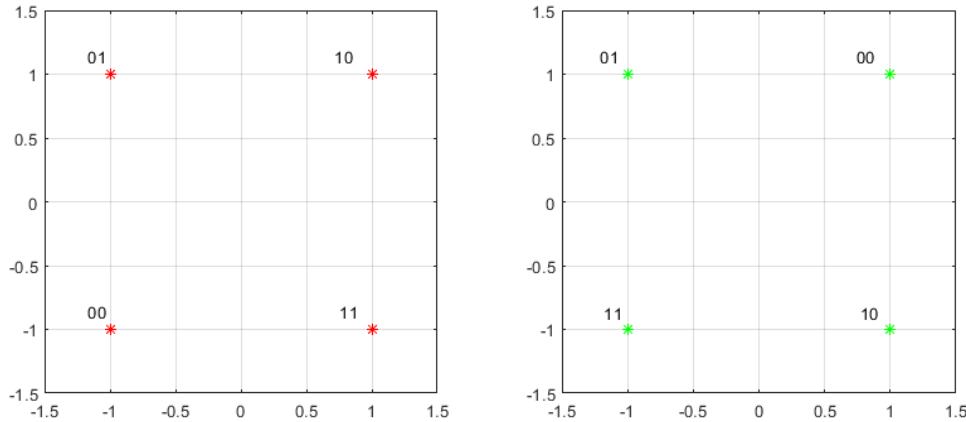


Figure 2.36: Two different constellations for QPSK.

Even though they both seem randomly chosen, after testing these two constellations with all the other parameters constant (the same configuration of the system, apart from the constellation) we get a BER of 10.5% for the constellation on the left and a BER of 8% for the constellation on the right.

This happens because, in the presence of noise, the symbols can be decoded to adjacent symbols. And that obviously has an effect on the BER. What we can see here is how the Symbol Error Rate (SER) affects the BER. It's obvious that, once the only configuration that was changed was the constellation, the SER is constant for both cases.

So, what do these two constellations have different to cause different measured BER?

The difference is that, the constellation on the right has Grey encoding, meaning that adjacent symbols only differ in one bit. In this way, a misevaluated symbol will only result in one out of two bits wrong, thus minimizing the BER of a system.

- Inter-symbolic interference (ISI) for different pulse shapes:

For this example we will analyse the eye diagrams at the receiver side, after the matched receiver filter of the M-QAM Receiver.

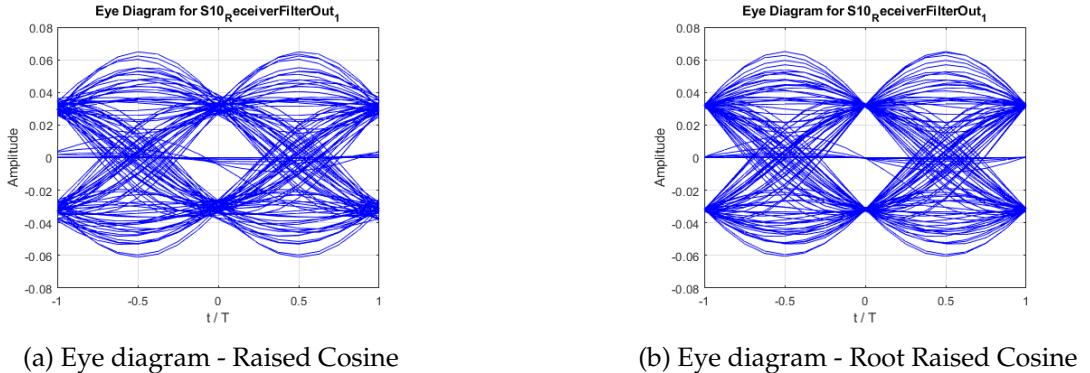


Figure 2.37: Eye diagrams without noise

In the figure 2.37a we have the eye diagram of the received signal, whose pulse was shaped with a Raised Cosine and the matched filter at the receiver had the same filter shape. The result is not what we are expecting, because the Raised Cosine pulse shape is known for having null ISI, but we can clearly see that it is not null. This happens because, thanks to the matched filter, whose shape is the same as the pulse shaper, we actually get a Squared Raised Cosine shape, which does not present the same null ISI as the Raised Cosine impulse. To solve this problem we must instead apply both on the pulse shaper and on the matched filter a Root Raised Cosine filter, in such way that at the receiver we obtain the expected null ISI. The result obtained from this can be observed in figure 2.37b.

Even though that it is clear to see the differences between figures 2.37a and 2.37b, we have to keep in mind that these signals are ideal and have no noise. It is then of interest to observe if the difference is still noticeable when we have a noise figure with the same order of the ISI seen before.

To analyse that we can add noise to the both cases mentioned before, and analyse the differences between both pulse shapes on the eye diagram. The result of doing so can be seen in figures ?? and 2.38b.

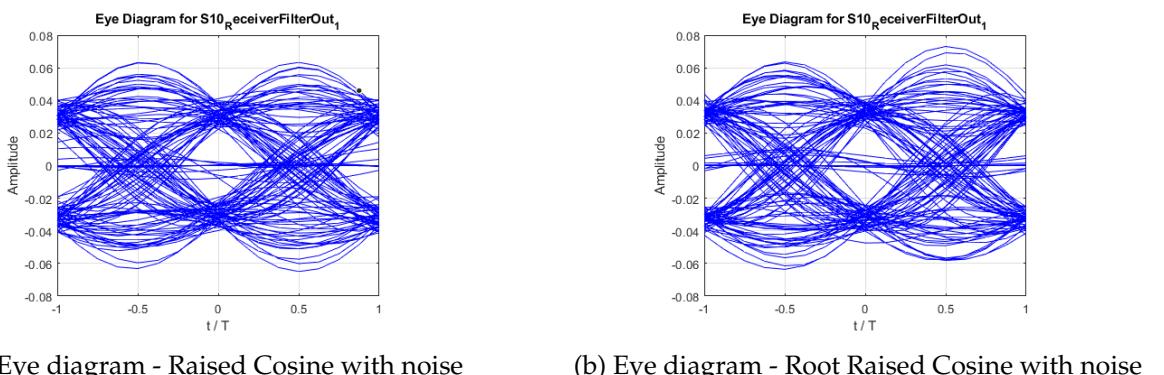


Figure 2.38: Eye diagrams with noise

From this we can see that, in spite of one pulse shape having null ISI at the receiver and the other not having, when the noise is of a significant order we can't differentiate both types

of pulse shapes as one might expect.

Number: 1 optical and 1 binary (optional)

Functional description

The M-QAM Transmitter is a super block that contains all the necessary blocks to generate a modulated optical signal using only as an input the binary code to modulate and the optical signal from the local oscillator, as we can observe in figure 2.39.

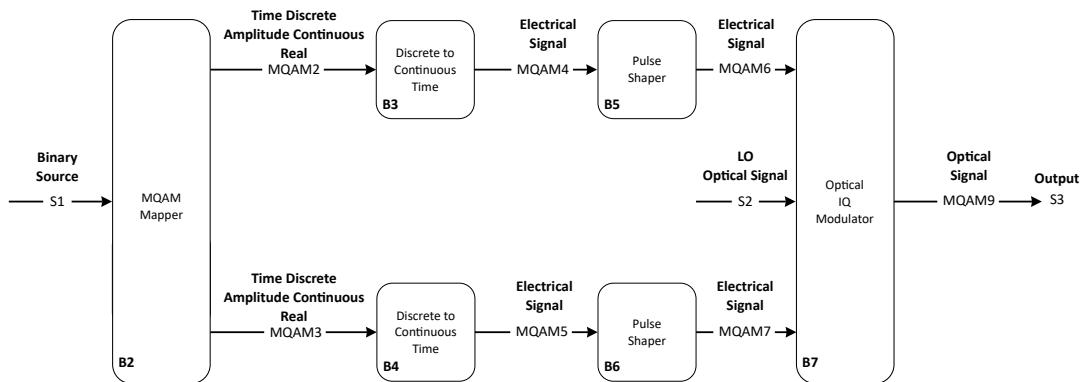


Figure 2.39: Schematic representation of the block MQAM transmitter.

With this block we needn't to manually make all the connections between the different blocks needed to implement an optical transmitter (B1-6). All we need to do is create an instance of the block and only modify the parameters whose default values are not in accordance with our needs. All the methods available to do so have already been presented above.

On the next paragraphs, a general overview of the flow of the signal from the input to the output will be made (please note that this is only a simple high-level description, for a detailed analysis of each block you should resort to the respective's block documentation). Firstly, the input signal passes through the M-QAM Mapper, which converts the groups of $\log_2 M$ consecutive bits into two streams of *TimeDiscreteAmplitudeContinuousReal* signals (MQAM 1 and 2), these signals have for amplitude the values inserted on the *iqAmplitudes* parameter, in such way that together, they map the binary sequence that originated them. Then, these signals are converted to *TimeContinuousAmplitudeContinuousReal* signals, by up-sampling the previous signals of a factor given by *numberOfSamplesPerSymbol*, obtaining (MQAM 3 and 4).

Following, the pulse shaper block uses the previously configured pulse shape (through the input parameters) to turn the pulses into impulses, obtaining signals MQAM 5 and 6, that are the In-phase and Quadrature components used to modulate the optical signal from the input signal 2, henceforth obtaining the output signal 1.

Sugestions for future improvement

Open issues

- the default QPSK constellation isn't the same as the one set if you call the function `setM(4)`, which may originate problems concerning the compatibility with the transmitter;
- the block has an unused parameter called `firstTime`, which may cause confusion with the parameter from the M-QAM Mapper, which is accessed and modified through the methods `set/getFirstTime()`;
- setting a square pulse shape throws a runtime error;
- not all methods available at the pulse shaper block can be accessed from the transmitter block.

Future improvements

- add missing methods from the pulse shaper to the transmitter block;
- add methods to modify the settings of the IQ Modulator;
- it would be interesting to have a noise input at the pulse shaper, to be able to see the constellations with noise, because currently there's no easy way to observe this.

2.48 Netxpto

Header File	:	netxpto.h
	:	netxpto_20180118.h
	:	netxpto_20180418.h
Source File	:	netxpto.cpp
	:	netxpto_20180118.cpp
	:	netxpto_20180418.cpp

The netxpto files define and implement the major entities of the simulator.

Namely the signal value possible types

Signal Value Type	Data Range
BinaryValue	{0,1}
IntegerValue	\mathbb{Z}
RealValue	\mathbb{R}
ComplexValue	\mathbb{C}
ComplexValueXY	(\mathbb{C}, \mathbb{C})
PhotonValue	
PhotonValueMP	
PhotonValueMPXY	
Message	

2.48.1 Version 20180118

Adds the type `t_photon_mp_xy`, to support multi-path photon signals with polarization information.

Changes the signal data type to make private its data structure, only allowing its access through appropriate methods.

2.48.2 Version 20180418

Adds the possibility to include the parameters from an external file.

Sugestions for future improvement

2.49 Alice QKD

This block is the processor for Alice does all tasks that she needs. This block accepts binary, messages, and real continuous time signals. It produces messages, binary and real discrete time signals.

Input Parameters

- double RateOfPhotons{1e3}
- int StringPhotonsLength{ 12 }

Methods

```
AliceQKD (vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) :  
Block(inputSignals, outputSignals) {};  
void initialize(void);  
bool runBlock(void);  
void setRateOfPhotons(double RPhotons) { RateOfPhotons = RPhotons; }; double const  
getRateOfPhotons(void) { return RateOfPhotons; };  
void setStringPhotonsLength(int pLength) { StringPhotonsLength = pLength; }; int const  
getStringPhotonsLength(void) { return StringPhotonsLength; };
```

Functional description

This block receives a sequence of binary numbers (1's or 0's) and a clock signal which will set the rate of the signals produced to generate single polarized photons. The real discrete time signal **SA_1** is generated based on the clock signal and the real discrete time signal **SA_2** is generated based on the random sequence of bits received through the signal **NUM_A**. This last sequence is analysed by the polarizer in pairs of bits in which each pair has a bit for basis choice and other for direction choice.

This block also produces classical messages signals to send to Bob as well as binary messages to the mutual information block with information about the photons it sent.

Input Signals

Number : 3

Type : Binary, Real Continuous Time and Messages signals.

Output Signals

Number : 3

Type : Binary, Real Discrete Time and Messages signals.

Examples

Sugestions for future improvement

2.50 Polarizer

This block is responsible of changing the polarization of the input photon stream signal by using the information from the other real time discrete input signal. This way, this block accepts two input signals: one photon stream and other real discrete time signal. The real discrete time input signal must be a signal discrete in time in which the amplitude can be 0 or 1. The block will analyse the pairs of values by interpreting them as basis and polarization direction.

Input Parameters

- m{4}
- Amplitudes { {1,1}, {-1,1}, {-1,-1}, { 1,-1} }

Methods

```
Polarizer (vector <Signal*> &inputSignals, vector <Signal*>&outputSignals) :  
Block(inputSignals, outputSignals) {};  
void initialize(void);  
bool runBlock(void);  
void setM(int mValue);  
void setAmplitudes(vector <t_iqValues> AmplitudeValues);
```

Functional description

Considering m=4, this block attributes for each pair of bits a point in space. In this case, it is be considered four possible polarization states: 0° , 45° , 90° and 135° .

Input Signals

Number : 2

Type : Photon Stream and a Sequence of 0's and '1s (DiscreteTimeDiscreteAmplitude).

Output Signals

Number : 1

Type : Photon Stream

Examples

Sugestions for future improvement

2.51 Probability Estimator

This blocks accepts an input binary signal and it calculates the probability of having a value "1" or "0" according to the number of samples acquired and according to the z-score value set depending on the confidence interval. It produces an output binary signal equals to the input. Nevertheless, this block has an additional output which is a txt file with information related with probability values, number of samples acquired and margin error values for each probability value.

In statistics theory, considering the results space Ω associated with a random experience and A an event such that $P(A) = p \in]0, 1[$. Lets $X : \Omega \rightarrow \mathbb{R}$ such that

$$\begin{aligned} X(\omega) &= 1 && \text{,if } \omega \in A \\ X(\omega) &= 0 && \text{,if } \omega \in \bar{A} \end{aligned} \tag{2.21}$$

This way, there only are two possible results: success when the outcome is 1 or failure when the outcome is 0. The probability of success is $P(X = 1)$ and the probability of failure is $P(X = 0)$,

$$\begin{aligned} P(X = 1) &= P(A) = p \\ P(X = 0) &= P(\bar{A}) = 1 - p \end{aligned} \tag{2.22}$$

X follows the Bernoulli law with parameter \mathbf{p} , $X \sim \mathbf{B}(p)$, being the expected value of the Bernoulli random value $E(X) = p$ and the variance $\text{VAR}(X) = p(1-p)$ [**probabilitySheldon**].

Assuming that N independent trials are performed, in which a success occurs with probability p and a failure occurs with probability $1-p$. If X is the number of successes that occur in the N trials, X is a binomial random variable with parameters (n, p) . Since N is large enough, X can be approximately normally distributed with mean np and variance $np(1-p)$.

$$\frac{X - np}{\sqrt{np(1-p)}} \sim N(0, 1). \tag{2.23}$$

In order to obtain a confidence interval for p , lets assume the estimator $\hat{p} = \frac{X}{N}$ the fraction of samples equals to 1 with regard to the total number of samples acquired. Since \hat{p} is the estimator of p , it should be approximately equal to p . As a result, for any $\alpha \in [0, 1]$ we have that:

$$\frac{X - np}{\sqrt{np(1-p)}} \sim N(0, 1) \tag{2.24}$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{np(1-p)}} < z_{\alpha/2}\} &\approx 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{np(1-\hat{p})} < np - X < z_{\alpha/2}\sqrt{np(1-\hat{p})}\} &\approx 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}\} &\approx 1 - \alpha \end{aligned} \tag{2.25}$$

This way, a confidence interval for p is approximately $100(1 - \alpha)$ percent.

Input Parameters

- zscore
(double)
- fileName
(string)

Methods

```
ProbabilityEstimator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setProbabilityExpectedX(double probx) double getProbabilityExpectedX()

void setProbabilityExpectedY(double proby) double getProbabilityExpectedY()

void setZScore(double z) double getZScore()
```

Functional description

This block receives an input binary signal with values "0" or "1" and it calculates the probability of having each number according with the number of samples acquired. This probability is calculated using the following formulas:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \quad (2.26)$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \quad (2.27)$$

The error margin is calculated based on the z-score set which specifies the confidence interval using the following formula:

$$ME = z_{\text{score}} \times \sqrt{\frac{\hat{p}(1 - \hat{p})}{N}} \quad (2.28)$$

being \hat{p} the expected probability calculated using the formulas above and N the total number of samples.

This block outputs a txt file with information regarding with the total number of received bits, the probability of 1, the probability of 0 and the respective errors.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 2

Type: Binary

Type: txt file

Examples

Lets calculate the margin error for N of samples in order to obtain X inside a specific confidence interval, which in this case we assume a confidence interval of 99%.

We will use *z-score* from a table about standard normal distribution, which in this case is 2.576, since a confidence interval of 99% was chosen, to calculate the expected error margin,

$$\begin{aligned} ME &= \pm z_{\alpha/2} \frac{\sigma}{\sqrt{N}} \\ ME &= \pm z_{\alpha/2} \sqrt{\frac{\hat{p}(1 - \hat{p})}{N}}, \end{aligned} \quad (2.29)$$

where, ME is the error margin, $z_{\alpha/2}$ is the *z-score* for a specific confidence interval, $\sigma = \sqrt{\text{VAR}(X)} = \sqrt{\hat{p}(1 - \hat{p})}$ is the standard deviation and N the number of samples.

This way, with a 99% confidence interval, between $(\hat{p} - ME) \times 100$ and $(\hat{p} + ME) \times 100$ percent of the samples meet the standards.

Sugestions for future improvement

2.52 Quantum Channel Emulator DV

Header Files	:	quantum_channel_emulator_dv_*.h
Source Files	:	quantum_channel_emulator_dv_*.cpp
Version	:	20190328 (Andoni Santos)

The Quantum Channel Emulator DV block tries to imitate the possible outcomes of the discrete variable quantum channel used in the laboratory. There is another more complex model of this system. However, the additional complexity makes it difficult to test and troubleshoot the software to be used with it. Therefore, this block was created. It is able to simulate the conditions existent on the complex system conditions in a stationary manner. This way, it produces similar results, while being much faster and simpler to interact with.

Signals

Number of Input Signals	3
Type of Input Signals	Binary, Binary, Binary
Number of Output Signals	3
Type of Output Signals	TimeContinuousAmplitudeDiscreteReal

Table 2.29: Quantum Channel Emulator signals

Input Signals

- 0 - Binary signal with Alice's key, to be transmitted to Bob.
- 1 - Binary signal with Alice's basis, used to encode the key
- 2 - Binary signal with Bob's basis, used to measure the transmitted data.

Output Signals

- 0 - TimeContinuousAmplitudeDiscreteReal signal containing the results from measuring the transmitted data on Bob's side.

Input Parameters

Methods

```
QuantumChannelEmulatorDv(initializer_list<Signal *> InputSig, initializer_list<Signal *>
OutputSig) : Block(InputSig, OutputSig) {};
```

```
void initialize(void);
```

Parameter	Type	Values	Default	Description
doubleClickNumber	t_integer	any	2	Number to signal that both detectors clicked
noClickNumber	t_integer	any	3	Number to signal that no detectors clicked
qber	t_real	[0,1]	0	Desired average quantum bit error rate
probabilityOfDoubleClick	t_real	[0,1]	0	Probability of both the detectors clicking
seed	t_integer	any	1	Random number generator seed.
deadtime	double	any	1×10^{-5}	Amount of time the detectors aren't responsive after clicking
detectorEfficiency	double	[0,1]	0.25	Efficiency of the detectors receiving the photons.
fiberAttenuation_dB	double	≥ 0	0.2	Fiber optical attenuation in dB km.
fiberLength	double	≥ 0	50	Length of the fiber in km.
insertionLosses_dB	double	≥ 0	2	Attenuation by insertion losses.
numberOfPhotonsPerInputPulse	double	≥ 0	0.1	Statistical average of the number of photons per input pulse.

Table 2.30: Binary source input parameters

```

bool runBlock(void);

initialize(void);

runBlock(void);

setProbabilityOfBothClick(t_real pbc)

```

```

setProbabilityOfNoClick(t_real pnc)

setProbabilityOfError(t_real pe)

setNoClickNumber(t_integer ncn)

setDoubleClickNumber(t_integer dcn)

setDeadtime(t_integer dt)

setDetectorEfficiency(t_real def)

setFiberAttenuation_dB(t_real fatt)

setFiberLength(t_real fl)

setInsertionLosses_dB(t_real il)

setNumberOfPhotonsPerInputPulse(t_real nPhotons)

```

Functional description

This block receives three main input signals: the key provided by Alice, the basis chosen by Alice to encode the key, and the basis chosen by Bob to decode the key.

The *onDeadtime* and *elapsedDeadtime* variables control whether the detectors are recovering from a click. Each time anything except a *noClick* is output, the variable *onDeadtime* is set to true. From them on, on every iteration, the *symbolPeriod* of the data signal is added to the *elapsedDeadtime* variable. When that number grows higher than *deadtime*, the *elapsedDeadtime* is set to zero and *onDeadtime* is set to false.

The process to determine the output of the Bob's measurement is as follows:

1. If *onDeadtime* is true, a *noClickNumber* is output;
2. A random number between zero and one, *randomRollClicks*, is drawn. If that number is lower than *probabilityOfNoClick*, a *noClickNumber* is output;
3. else, if that number is lower than *probabilityOfNoClick* + *probabilityOfDoubleClick*, a *doubleClickNumber* is output;
4. If none of the above happens, another random number between zero and one, *randomRollError*, is drawn. If the number is lower than *probabilityOfError*, the wrong bit is put in the buffer, indicating an error (independently of the measurement basis);
5. If the *randomRollError* is higher than the *probabilityOfError*, then the outcome depends on whether Alice's and Bob's basis are equal. If they are, the correct bit is output;

6. If the basis are different, a random number *randomRollErrorFromWrongBasis* between zero and one is generated. If the value is lower than 0.5, the correct bit is output. Otherwise, the wrong bit is output.

In addition, the third input signal is copied to the second output signal, at the same rate as the key. The third output signal currently is a placeholder and produces no output.

Suggestions for future improvement

2.53 Bob QKD

This block is the processor for Bob does all tasks that she needs. This block accepts and produces:

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

2.54 Eve QKD

This block is the processor for Eve does all tasks that she needs. This block accepts and produces:

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

2.55 Rotator Linear Polarizer

This block accepts a Photon Stream signal and a Real discrete time signal. It produces a photon stream by rotating the polarization axis of the linearly polarized input photon stream by an angle of choice.

Input Parameters

- m{2}
- axis { {1,0}, { $\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}$ } }

Methods

```
RotatorLinearPolarizer(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) :
    Block(inputSignals, outputSignals) {};
    void initialize(void);
    bool runBlock(void);
    void setM(int mValue);
    void setAxis(vector <t_iqValues> AxisValues);
```

Functional description

This block accepts the input parameter m, which defines the number of possible rotations. In this case m=2, the block accepts the rectilinear basis, defined by the first position of the second input parameter axis, and the diagonal basis, defined by the second position of the second input parameter axis. This block rotates the polarization axis of the linearly polarized input photon stream to the basis defined by the other input signal. If the discrete value of this signal is 0, the rotator is set to rotate the input photon stream by 0°, otherwise, if the value is 1, the rotator is set to rotate the input photon stream by an angle of 45°.

Input Signals

Number : 2

Type : Photon Stream and a Sequence of 0's and '1s (DiscreteTimeDiscreteAmplitude)

Output Signals

Number : 1

Type : Photon Stream

Examples

Sugestions for future improvement

2.56 MS Windows IP Tunnel

Header File	:	ms_windows_ip_tunnel_*.h
Source File	:	ms_windows_ip_tunnel_*.cpp
Version	:	20180815 (João Coelho)

This block works in two ways: one way with one input and zero output signals (as client/sender) and the other one with zero input and one output signal (as server/receiver). MS Windows IP Tunnel is duplicated onto two machines; the first takes samples out of the input buffer until it's empty and transmits them to the IP Tunnel on the second machine through a TCP/IP connection. The IP Tunnel on the second machine receives the signal and outputs it to the output buffer. It's only necessary to specify the ip address of the remote machine and its TCP port. Only works on windows due to the Ws2tcpip.h header and ws2_32 library and it is compatible and tested with SDK v8.1 and v10.0.18362.0.

Input Parameters

Parameter	Type	Values	Default	Brief Description
displayNumberOfSamples	bool	true/false	true	If true, number of samples sent and received are displayed.
numberOfTrials	int	any	10	Number of trials after a connection has been refused.
remoteMachineIpAddress	string	any	"127.0.0.1"	IP Address of Remote Machine.
tcpPort	int	any	54000	TCP port used to connect to server.
timeIntervalSeconds	int	any	3	Time interval before trying to connect again after a connection has been refused (seconds).

Table 2.31: MS Windows IP Tunnel input parameters

Methods

IPTunnel(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)

void initialize(void);

bool runBlock(void);

void terminate(void);

```

void ipTunnelSendInt(int space);

int ipTunnelRecvInt();

int ipTunnelPut(T object);

void setDisplayNumberOfSamples(bool opt);

bool getDisplayNumberOfSamples();

bool server();

bool client();

```

Functional Description

The IP Tunnel block transfers signals from one machine to another so it continues the simulation in two different computers. This block is duplicated onto two machines, one with only input (client) and other with only output signals (server). After being executed, the input signal's buffer (1) will be empty and this block will transmit this signal to the other block. The second block will output this signal to the output buffer (4). An architecture "Server - Client" is used to establish a TCP/IP channel between the two blocks (2 and 3). The block without input signal is the server (receiver) and the block with input signal is the client (transmitter). After the connection is established, server sends the "space" of its buffer (maximum signal size that can be received) and client responds with the "process" (signal size that is going to be transmitted) and with an integer representing the type of signal. Subsequently, the transmission of the signal starts and the simulation continues normally on both machines.

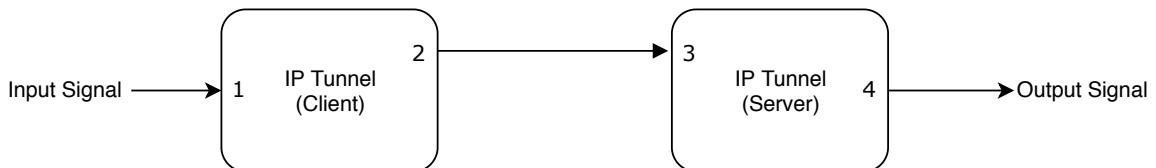


Figure 2.40: MS Windows IP Tunnel block

2.57 Mutual Information Estimator

Header File	:	mutual_information_estimator_20180723.h
Source File	:	mutual_information_estimator_20180723.cpp

This block estimates the mutual information between the input and output channel symbols X and Y, respectively. Each input signal x_j ($j = 1, 2, \dots, J$) has a specific probability being possible calculate the entropy of the alphabet X, $H(X)$. The uncertainty of the observation regarding with the occurrence of the input symbol is measured by the entropy $H(X)$, which is maximum when all inputs have the same probability. Another concept important to estimate the mutual information is the conditional entropy $H(X|Y = y_k)$, which represents the uncertainty related with the channel input symbols X that stays after the observation of the output symbols Y. This way, the difference $H(X) - H(X|Y)$ is on average the amount of information gained by the observer with respect with the channel input based on the channel output symbol. This difference is called the mutual information:

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^K \sum_{j=1}^J P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \quad (2.30)$$

where K corresponds to the number of possible output symbols and J corresponds to the number of possible input symbols.

This block uses the property 1.9.3 in [1] of Mutual information which tells that it can be determined using the formula:

$$I(X; Y) = H(Y) - H(Y|X). \quad (2.31)$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0 $P(X = 0)$ and assumes that the complementary of this probability corresponds to the probability of the input bit is 1 $P(X = 1)$. $P(X = 0)$ corresponds to the α probability calculated in this block. Furthermore, another probability of interest is p which corresponds to the error probability of the channel. Both α and p are estimated in this block.

In order to calculate the mutual information, from equation 2.31 we should calculate the conditional entropy $H(Y|X)$ and the entropy of the channel outputs $H(Y)$. First, lets calculate $H(Y|X)$:

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^K \sum_{j=1}^J P(y_k|x_j) P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) + \bar{\alpha}(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= H(p), \end{aligned} \quad (2.32)$$

which means that the conditional entropy depends only on the channel properties and it does not depend on the channel input statistics. Now, to calculate the entropy of the channel output symbols we need to calculate the channel output probabilities, i.e $P(Y = 0)$ and $P(Y = 1)$.

$$\begin{aligned} P(Y = 0) &= P(Y = 0|X = 0)P(X = 0) + P(Y = 0|X = 1)P(X = 1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \quad (2.33)$$

Since the output channel symbol only has two possible values (0 or 1),

$$\begin{aligned} P(Y = 1) &= P(Y = 1|X = 1)P(X = 1) + P(Y = 1|X = 0)P(X = 0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y = 0). \end{aligned} \quad (2.34)$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \quad (2.35)$$

and the mutual information should be calculated using the following formula:

$$I(X; Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \quad (2.36)$$

The upper and lower bounds, $I(X; Y)_{UB}$ and $I(X; Y)_{LB}$ respectively, are calculated using the method of Coppler-Pearson as described in section 2.11 for bit error rate calculation. This way, it returns the simplified expression:

$$I(X; Y)_{UB} = I(X; Y) + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 + (2 - I(X; Y)) \right] \quad (2.37)$$

$$I(X; Y)_{LB} = I(X; Y) - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 - (1 + I(X; Y)) \right], \quad (2.38)$$

where $z_{\alpha/2}$ is the $100(1 - \frac{\alpha}{2})$ th percentile of a standard normal distribution and N_T the total number of bits used to calculate the mutual information.

Input Parameters

Name	Type	Default Value
m	integer	0
alpha_bounds	double	0.05

Methods

- MutualInformationEstimator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig) :Block(InputSig,OutputSig){};

- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setConfidence(double P) { alpha = 1-P; }

Input Signals

Number: 2

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1

Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional Description

This block accepts two binary strings and outputs a binary string, outputting a 0 if the two input samples are equal to each other and 1 if not. This block also outputs .txt files with a report of the estimated Mutual Information, as well as the error probability of the channel estimated p and the estimated probability of $X = 0$, α . Furthermore, the mutual information estimator block can output middle report files with size m set by the user using the method `setMidReportSize(int M)`, i.e the mutual information calculated uses m input symbols in its calculation. However, a final report is always outputted using all symbols transmitted.

The block receives two input binary strings, one with the sequence of input channel symbols and from this it calculates the probability of the input symbol is equals to 0, α , and other sequence with the output channel symbols and it compares the bit from this sequence with the correspondent bit from the first sequence (i.e the sequence with the input channel symbols). If the bit in the output channel symbol is different from the correspondent bit in the input channel symbol sequence, it counts as an error and the error probability of the channel is calculated based on the final number of errors, p . Both probabilities α and p allow the block to estimate the conditional entropy and the entropy of the output channel symbols, allowing the calculation of the mutual information.

References

- [1] Krzysztof Wesolowski. *Introduction to digital communication systems*. John Wiley & Sons, 2009.

2.58 Optical Switch

This block has one input signal and two input signals. Furthermore, it accepts an additional input binary input signal which is used to decide which of the two outputs is activated.

Input Parameters

No input parameters.

Methods

```
OpticalSwitch(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) :  
Block(inputSignals, outputSignals) {};  
void initialize(void);  
bool runBlock(void);
```

Functional description

This block receives an input photon stream signal and it decides which path the signal must follow. In order to make this decision it receives a binary signal (0's and 1's) and it switch the output path according with this signal.

Input Signals

Number : 1

Type : Photon Stream

Output Signals

Number : 2

Type : Photon Stream

Examples

Sugestions for future improvement

2.59 Optical Hybrid

Header File	:	optical_hybrid.h
Source File	:	optical_hybrid.cpp

This block simulates an optical hybrid. It accepts two input signals corresponding to the signal and to the local oscillator. It generates four output complex signals separated by 90° in the complex plane. Figure 2.41 shows a schematic representation of this block.

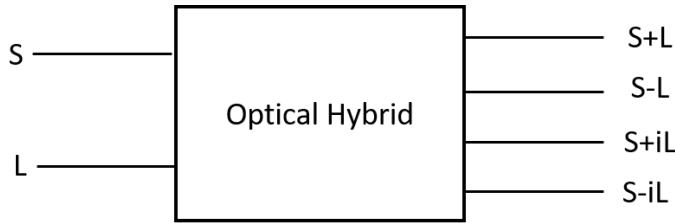


Figure 2.41: Schematic representation of an optical hybrid.

Input Parameters

Parameter	Type	Values	Default
outputOpticalPower	double	any	$1e - 3$
outputOpticalWavelength	double	any	$1550e - 9$
outputOpticalFrequency	double	any	$SPEED_OF_LIGHT / outputOpticalWavelength$
powerFactor	double	≤ 1	0.5

Table 2.32: Optical hybrid input parameters

Methods

OpticalHybrid()

```
OpticalHybrid(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setOutputOpticalPower(double outOpticalPower)
```

```
void setOutputOpticalPower_dBm(double outOpticalPower_dBm)
```

```
void setOutputOpticalWavelength(double outOpticalWavelength)  
void setOutputOpticalFrequency(double outOpticalFrequency)  
void setPowerFactor(double pFactor)
```

Functional description

This block accepts two input signals corresponding to the signal to be demodulated (S) and to the local oscillator (L). It generates four output optical signals given by $powerFactor \times (S + L)$, $powerFactor \times (S - L)$, $powerFactor \times (S + iL)$, $powerFactor \times (S - iL)$. The input parameter $powerFactor$ assures the conservation of optical power.

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 4

Type: Optical (OpticalSignal)

Examples

Sugestions for future improvement

2.60 Photodiode pair

Header File	:	photodiode_old.h
Source File	:	photodiode_old.cpp

This block simulates a block of two photodiodes assembled like in figure 2.42. It accepts two optical input signals and outputs one electrical signal. Each photodiode converts an optical signal to an electrical signal. The two electrical signals are then subtracted and the resulting signals corresponds to the output signal of the block.

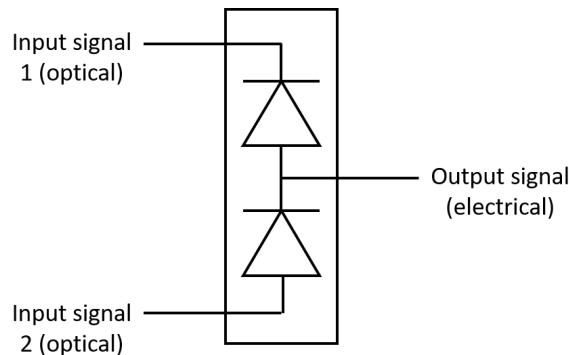


Figure 2.42: Schematic representation of the physical equivalent of the photodiode code block.

Input Parameters

- responsivity{1}
- outputOpticalWavelength{ 1550e-9 }
- outputOpticalFrequency{ SPEED_OF_LIGHT / wavelength }

Methods

Photodiode()

```
Photodiode(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setResponsivity(t_real Responsivity)
```

Functional description

This block accepts two input optical signals. It computes the optical power of the signal (given by the absolute value squared of the input signal) and multiplies it by the *responsivity* of the photodiode. This product corresponds to the current generated by the photodiode. This is done for each of the input signals. The two currents are then subtracted producing a single output current, that corresponds to the output electrical signal of the block.

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

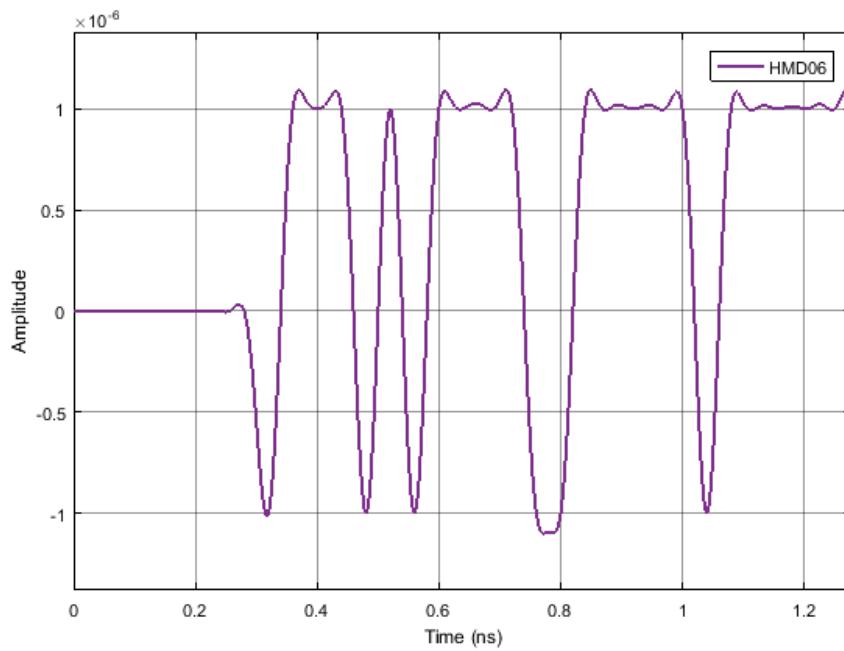


Figure 2.43: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

2.61 Photoelectron Generator

Header File	:	photoelectron_generator_*.h
Source File	:	photoelectron_generator_*.cpp
Version	:	20180302 (Diamantino Silva)

This block simulates the generation of photoelectrons by a photodiode, performing the conversion of an incident electric field into an output current proportional to the field's instantaneous power. It is also capable of simulating shot noise.

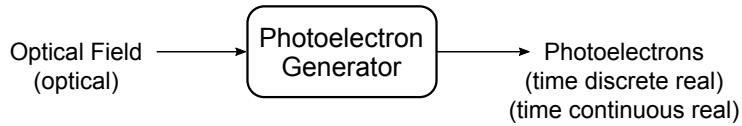


Figure 2.44: Schematic representation of the photoelectron generator code block

Theoretical description

The operation of a real photodiode is based on the photoelectric effect, which consists on the removal of one electron from the target material by a single photon, with a probability η . Given an input beam with an optical power $P(t)$ in which the photons are around the wavelength λ , the flux of photons $\phi(t)$ is calculated as [1]

$$\phi(t) = \frac{P(t)\lambda}{hc} \quad (2.39)$$

therefore, the mean number of photons in a given interval $[t, t + T]$ is

$$\bar{n}(t) = \int_t^{t+T} \phi(\tau) d\tau \quad (2.40)$$

But the actual number of photons in a given time interval, $n(t)$, is random. If we assume that the electric field is generated by an ideal laser with constant power, then $n(t)$ will follow a Poisson distribution

$$p(n) = \frac{\bar{n}^n \exp(-\bar{n})}{n!} \quad (2.41)$$

where $n = n(t)$ and $\bar{n} = \bar{n}(t)$.

For each incident photon, there is a probability η of generating a phototransistor. Therefore, we can model the generation of photoelectrons during this time interval, as a binomial process where the number of events is equal to the number of incident photons, $n(t)$, and the rate of success is η . If we combine the two random processes, binomial photoelectron generation after poissonian photon flux, the number of output photoelectrons in this time interval, $m(t)$, will follow [1]

$$m \sim \text{Poisson}(\eta\bar{n}) \quad (2.42)$$

with $\bar{m} = \eta\bar{n}$ where $m = m(t)$.

Functional description

The input of this block is the electric field amplitude, $A(t)$, with sampling period T . The first step consists on the calculation the instantaneous power. Given that the input amplitude is a baseband representation of the original signal, then $P(t) = 4|A(t)|^2$. From this result, the average number of photons $\bar{n}(t) = TP(t)\lambda/hc$.

If the shot-noise is negleted, then the output number of photoelectrons, $n_e(t)$ in the interval, will be equal to

$$m(t) = \eta \bar{n}(t) \quad (2.43)$$

If the shot-noise is considered, then the output fluctuations will be simulated by generating a value from a Poissonian random number generator with mean $\eta \bar{n}(t)$

$$m(t) \sim \text{Poisson} \left(\eta \bar{n}(t) \right) \quad (2.44)$$

Input Parameters

Parameter	Default Value	Description
efficiency	1.0	Photodiode's quantum efficiency.
shotNoise	false	Shot-noise off/on.

Methods

PhotoelectronGenerator()

```
PhotoelectronGenerator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setEfficiency(t_real efficiency)
```

```
void getEfficiency()
```

```
void setShotNoise(bool shotNoise)
```

```
void getShotNoise()
```

Input Signals

Number: 1

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal
TimeContinuousAmplitudeContinuousReal) or

Examples

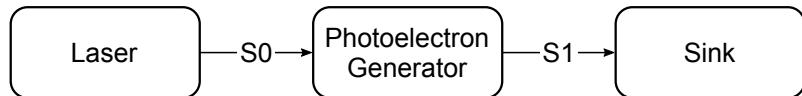


Figure 2.45: Constant power simulation setup

To test the output of this block, we recreated the results of figure 11.2 – 3 in [1]. We started by simulating the constant optical power case, in which the local oscillator power was fixed to a constant value. Two power levels were tested, $P = 1\mu W$ and $P = 1nW$, using a sample period of 20 picoseconds and photoelectron generator efficiency of 1.0. The simulation code is in folder lib \photoelectron_generator \photoelectron_generator_test_constant. The following plots show the number of output electrons per sample when the shot noise is ignored or considered

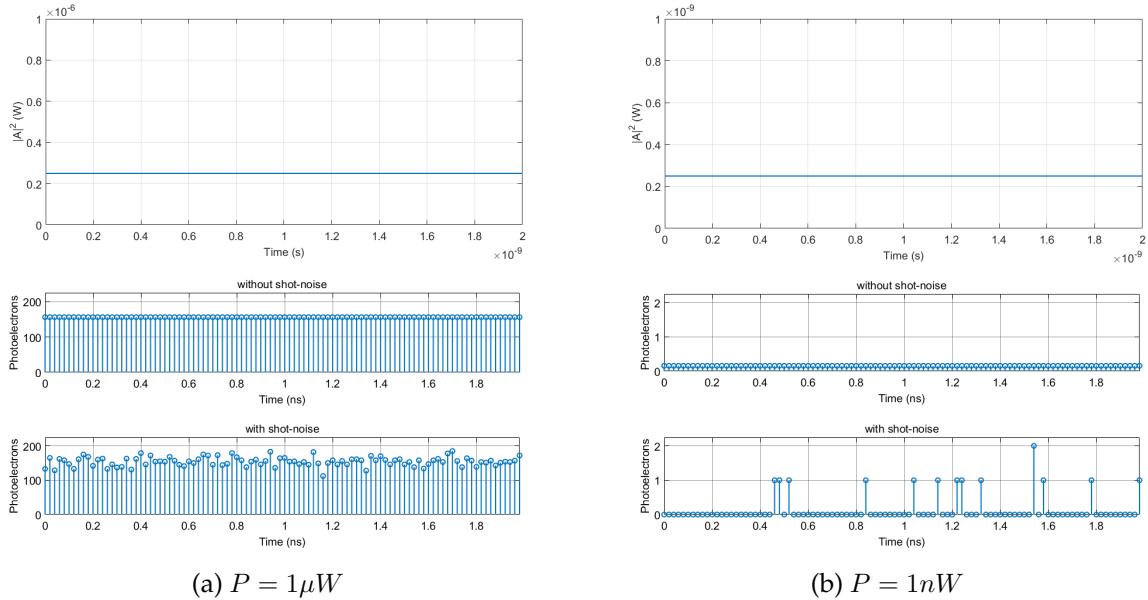


Figure 2.46: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 2.46 shows clearly that turning the quantum noise on or off will produce a signal with or without variance, as predicted. If we compare this result with plot (a) in [1], in particular $P = 1nW$, we see that they are in conformance, with a slight difference, where a sample has more than one photoelectron. In contrast with the reference result, where only single events are represented, the $P = 1\mu W$ case shows that all samples account many photoelectrons. Given it's input power, multiple photoelectron generation events will occur during the sample time window. Therefore, to recreate the reference result, we just need to reduce the sample period until the probability of generating more than 1 photoelectron per sample goes to 0.

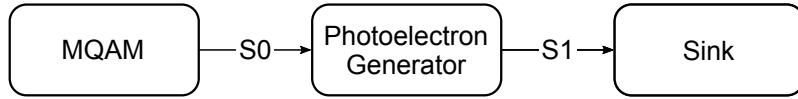


Figure 2.47: Variable power simulation setup

To recreate plot (b) in [1], a more complex setup was used, where a series of states are generated and shaped by a MQAM, creating a input electric field with time-varying power. The simulation code is in folder lib \photoelectron_generator \photoelectron_generator_variable.

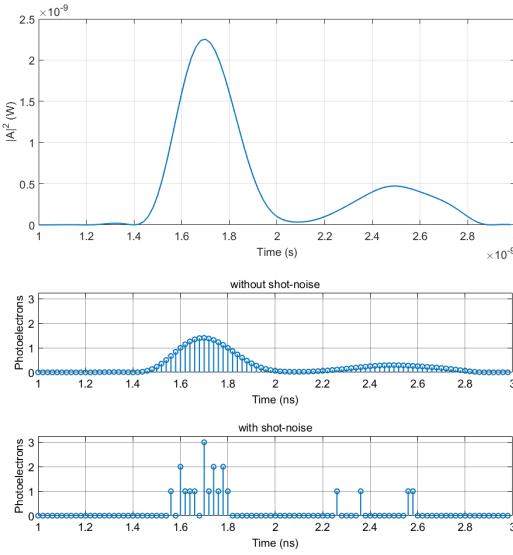


Figure 2.48: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 2.48 shows that the output without shot-noise is following the input power perfectly, apart from a constant factor. In the case with shot-noise, we see that there are only output samples in high power input samples.

These results are not following strictly plot (b) in [1], because has already discussed previously, in the high power input samples, we have a great probability of generating many photoelectrons.

Sugestions for future improvement

References

- [1] B. E. Saleh and M. C. Teich. *Fundamentals of photonics*. Wiley series in pure and applied optics. New York (NY): John Wiley & Sons, 1991.

2.62 Power Spectral Density Estimator

Header File	:	power_spectral_density_estimator_*.h
Source File	:	power_spectral_density_estimator_*.cpp
Version	:	20180704 (Andoni Santos)

Input Parameters

Name	Type	Default Value
method	enum	WelchPgram
ignoreInitialSamples	int	513
windowType	WindowType	Hann
measuredIntervalSize	int	2048
segmentSize	int	512
overlapPercent	double	0.5
overlapCount	int	256
alpha	double	0.05
tolerance	double	1e-6
filename	string	signals/powerSpectralDensity.txt
active	bool	false

Methods

```

PowerSpectralDensityEstimator() ;

    PowerSpectralDensityEstimator(vector<Signal * > &InputSig, vector<Signal * >
&OutputSig) :Block(InputSig, OutputSig) ;

    void initialize(void);

    bool runBlock(void);

    void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

    int getMeasuredIntervalSize(void) return measuredIntervalSize;

    void setSegmentSize(int sz) segmentSize = sz;

    int getSegmentSize(void) return segmentSize;

    void setOverlapCount(int olp) overlapCount = olp; overlapPercent = 
overlapCount/segmentSize;

```

```

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olpP)    overlapPercent = olpP; overlapCount =
segmentSize*overlapPercent;

double getOverlapPercent(void) return overlapPercent;

void setConfidence(double P) alpha = 1-P;

double getConfidence(void) return 1 - alpha;

void setWindowType(WindowType wd) windowType = wd;

WindowType getWindowType(void) return windowType;

void setFilename(string fname) filename = fname;

string getFilename(void) return filename;

void setEstimatorMethod(PowerSpectralDensityEstimatorMethod mtd) method = mtd;

PowerSpectralDensityEstimatorMethod getEstimatorMethod(void) return method;

void setActivityState(bool state) active = state;

bool getActivityState(void) return active;

```

Input Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This block accepts one OpticalSignal or TimeContinuousAmplitudeContinuousReal signal (or two real signals, an In-phase signal and a quadrature signal) and outputs an exact copy of the input signals to the output. As it receives the input signals, it saves until it has enough to perform a power spectral density estimation, in W/Hz. The number of samples required to perform this estimation is defined by the parameter measuredIntervalSize. After it has enough samples, it proceeds to estimating the power spectral density of the signal through the procedure chosen through the method parameter. Currently only one

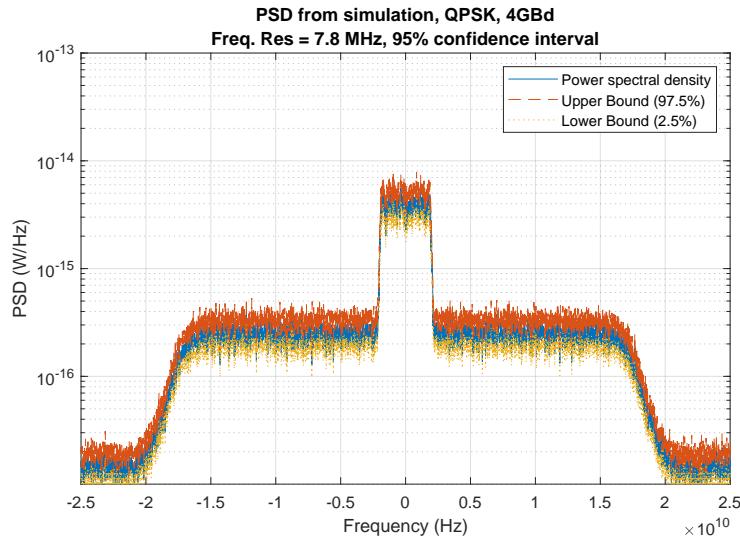


Figure 2.49: Estimated power spectral density for a QPSK signal.

method is available, the Welch periodogram (`WelchPgram`). With this method, when enough samples are gathered, a segment of size `segmentSize` is chosen, windowed with a window chosen with `windowType` and a periodogram obtained from it. Another segment is then selected, with a certain number of samples overlapping with the previous one, as defined by `overlapCount` or `overlapPercent`. This process is repeated until the end of the measured interval, and the average of every periodogram obtained so far is the end result. This is then saved to the target file, specified by the `filename` parameter. This file is updated each time that the chosen number of samples is reached. As the simulation continues to run, more samples are acquired and as the estimation is improved.

The file contains data divided in different lines, separated by "`\n`", to simplify reading it with external programs. The first line identifies the file as a Power Spectral Density (PSD) estimate. The second line lists the number of segments averaged to obtain the estimated. The third line identifies the chosen confidence interval. The remaining lines are grouped in pairs, with the first line describing the contents of the next one. These pairs contain, in order, the center of the frequency bins, the Power Spectral Density estimate, and the corresponding upper and lower confidence bounds.

Theoretical Description

The techniques used for estimating the Power Spectral Density are divided in two, based on different definitions of power spectral density. These are the periodogram (direct method) and the correlogram (indirect method).

Periodogram

It is also possible to define the Power Spectral Density as follows [jeruchim06]:

$$S_{XX}(f) = \lim_{T \rightarrow \infty} E \left[\hat{S}_{XX}(f, T) \right] \quad (2.45)$$

with

$$\hat{S}_{XX}(f, T) = \frac{|X_T(f)|^2}{T} \quad (2.46)$$

and

$$X_T(f) = \int_0^T x(t) \exp(-j2\pi ft) dt \quad (2.47)$$

$tS_{XX}(f, T)$ presents a natural estimation for the Power Spectral Density, if the limiting and expectation operations are omitted. This is called a periodogram. Its discrete version is given by:

$$\tilde{P}(f) = \frac{1}{NT_s} |X_N(f)|^2 \quad (2.48)$$

with

$$X_N(f) = T_s \sum_{n=0}^{N-1} X(n) \exp(-j2\pi fnT_s) \quad (2.49)$$

$X_N(f)$ is the DFT of the observed data sequence, which can easily be determined through FFt calculation.

To obtain the periodogram, the data needs to windowed. Windowing implies a function that selects a certain portions of another function, either as is or transforming it in a certain way. Any finite observation is at least implicitly windowed with a rectangular window, as it is a smaller set of the original function. Windows can also modify the data in a given desired way. Any window necessarily alters the true spectrum. The expected value of the periodograms is therefore different from the true spectrum, instead providing a biased estimate. However, the periodogram is asymptotically unbiased. This is because the effects of windowing, which are the origin of the bias, disappear as the number of samples increases. For any window $W(f)$, including the rectangular window, $|W(f)|^2 \rightarrow \delta(f)$ as the number of observations tends to infinity.

Also, for any periodogram, the standard deviation is at least as large as the expected value of the spectrum, independently of the size of the observation.

Most common methods for obtaining periodograms try to avoid this issue. One operation they resort to is the averaging of periodograms. This simply means to obtain the average periodogram from a given set of periodograms obtained from different signal sequences. The averaging of periodograms turn them into a consistent estimator. In order to average the periodograms, the signal is divided into smaller segments through windowing and periodograms are obtained from different segments, in order to be averaged.

The two most common methods are the Bartlett periodogram and the Welch periodogram. The Bartlett periodogram finds the average of nonoverlapping signal

segments obtained with a rectangular window. The Welch periodogram is calculated by using segments with a certain amount of overlap, which are obtained with other windows, typically Hann or Hamming.

Confidence intervals

The periodogram is Chi-Square distributed [jeruchim06]. Therefore, the confidence interval must be calculated accordingly. The confidence interval for a an estimated Power Spectral Density value $\hat{S}(f)$ is defined by [nsapplication255]

$$P(A < S(f) < B) = 1 - \alpha$$

where A and B are the bounds of the confidence interval, and α is the probability of the true Power Spectral Density being outside the confidence interval. For a given number of degrees of freedom, the interval's bounds are given by

$$A = \frac{\nu \hat{S}(f)}{\chi^2_{1-\alpha/2}}$$

$$B = \frac{\nu \hat{S}(f)}{\chi^2_{\alpha/2}}$$

$\chi^2_{\alpha/2}$ can be obtained by finding the values at which the chi-squared cumulative distribution function, for ν degrees of freedom, equals the desired probabilities. This function is calculated as [weisstein18csd]

$$\text{CDF}(\chi^2) = \frac{\gamma(\nu/2, \chi^2/2)}{\Gamma(\nu/2)} \quad (2.50)$$

where $\gamma(\nu/2, \chi^2/2)$ is an incomplete gamma function and $\Gamma(\nu/2)$ is the gamma function. The values of χ^2 needed for calculating A and B can be found by numerically evaluating 2.50 to find the values which fit the desired probabilities. This is currently done using the bisection method, which provides an approximation better than most available tables ($\frac{|\alpha_{\text{calc}} - \alpha_{\text{desired}}|}{\alpha_{\text{desired}}} \leq 10^{-6}$) in approximately 20 iterations.

The confidence bounds are then calculated using equations 2.62 and 2.62.

Known Issues

The current implementation for obtaining confidence intervals is susceptible to underflow/overflow errors if the degrees of freedom grow too much, due to the calculation of the incomplete gamma functions. This can mostly be avoided by making sure that the segment size for calculating the periodogram is, at least, approximately 1/1000th of the total number of samples.

$$\text{Segment_size} \geq \frac{\text{NumberOfBits} \times \text{SamplesPerSymbol}}{1000 \times \text{BitsPerSymbol}}$$

In order to completely correct this issue, two possibilities exist:

1. Improve the algorithm in order to overcome the overflow limitations, by using increased precision numeric types or some other method;
2. By using a gaussian approximation for the distribution of the periodogram, which is valid for larger degrees of freedom [**jeruchim06**].

2.63 Pulse Shaper

Header File	:	pulse_shaper.h
Source File	:	pulse_shaper.cpp

This block applies an electrical filter to the signal. It accepts one input signal that is a sequence of Dirac delta functions and it produces one output signal continuous in time and in amplitude.

Input Parameters

Parameter	Type	Values	Default
filterType	string	RaisedCosine, Gaussian	RaisedCosine
impulseResponseTimeLength	int	any	16
rollOffFactor	real	$\in [0, 1]$	0.9

Table 2.33: Pulse shaper input parameters

Methods

```
PulseShaper(vector<Signal * > &InputSig, vector<Signal * > OutputSig)
:FIR_Filter(InputSig, OutputSig){};

void initialize(void);

void setImpulseResponseTimeLength(int impResponseTimeLength)

int const getImpulseResponseTimeLength(void)

void setFilterType(PulseShaperFilter fType)

PulseShaperFilter const getFilterType(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor()
```

Functional Description

The type of filter applied to the signal can be selected through the input parameter *filterType*. Currently the only available filter is a raised cosine.

The filter's transfer function is defined by the vector *impulseResponse*. The parameter *rollOffFactor* is a characteristic of the filter and is used to define its transfer function.

Input Signals

Number : 1

Type : Sequence of Dirac Delta functions (ContinuousTimeDiscreteAmplitude)

Output Signals

Number : 1

Type : Sequence of impulses modulated by the filter
(ContinuousTimeContinuousAmplitude)

Example

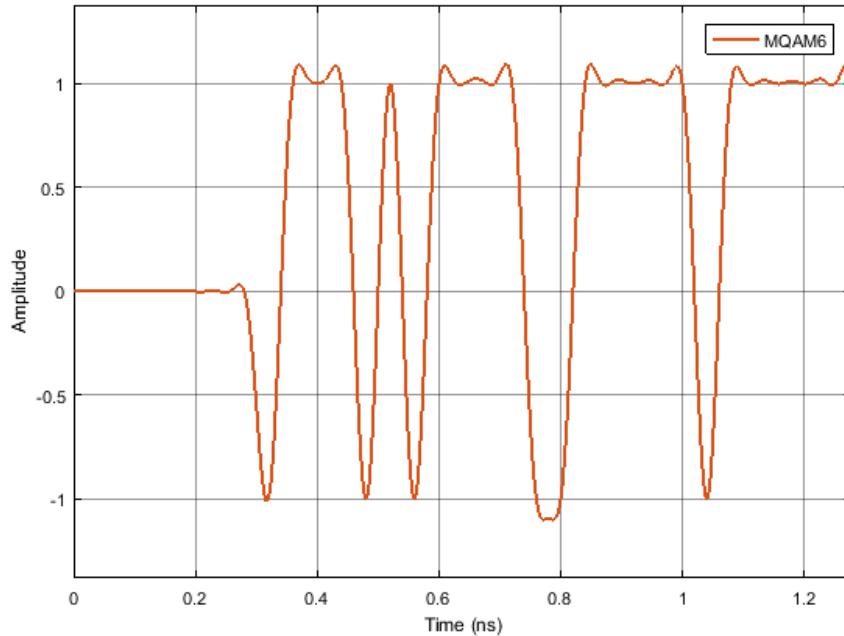


Figure 2.50: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

2.64 Quantizer

Header File	:	quantizer_*.h
Source File	:	quantizer_*.cpp
Version	:	20180423 (Celestino Martins)

This block simulates a quantizer, where the signal is quantized into discrete levels. Given a quantization bit precision, *resolution*, the outputs signal will be comprise $2^{nBits} - 1$ levels.

Input Parameters

Parameter	Unity	Type	Values	Default
resolution	bits	double	any	<i>inf</i>
maxValue	volts	double	any	1.5
minValue	volts	double	any	-1.5

Table 2.34: Quantizer input parameters

Methods

```

Quantizer() ;

    Quantizer(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};

    void initialize(void);

    bool runBlock(void);

    void setSamplingPeriod(double sPeriod) samplingPeriod = sPeriod;

    void setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;

    void setResolution(double nbites) resolution = nbites;

    double getResolution() return resolution;

    void setMinValue(double maxvalue) maxValue = maxvalue;

    double getMinValue() return maxValue;

    void setMaxValue(double maxvalue) maxValue = maxvalue;

    double getMaxValue() return maxValue;

```

Functional description

This block can performs the signal quantization according to the defined input parameter *resolution*.

Firstly, the parameter *resolution* is checked and if it is equal to the infinity, the output signal correspond to the input signal. Otherwise, the quantization process is applied. The input signal is quantized into $2^{\text{resolution}-1}$ discrete levels using the standard *round* function.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.65 Resample

Header File	:	resample_*.h
Source File	:	resample_*.cpp
Version	:	20180423 (Celestino Martins)
	:	20190509 (Daniel Pereira)

This block simulates the resampling of a signal. It receives one input signal and outputs a signal with the sampling rate defined by sampling rate, which is externally configured.

Input Parameters

Parameter	Type	Values	Default
rFactor	double	any	<i>inf</i>
samplingPeriod	double	any	0.0
symbolPeriod	double	any	1.5

Table 2.35: Resample input parameters

Methods

```
Resample() ; Resample(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void); bool runBlock(void);
void setSamplingPeriod(double sPeriod)    samplingPeriod = sPeriod;      void
setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;
void setOutRateFactor(double OUTsRate)    rFactor = OUTsRate;      double
getOutRateFactor() return rFactor;
```

Functional description

This block can performs the signal resample according to the defined input parameter *rFactor*. It resamples the input signal at *rFactor* times the original sample rate.

Firstly, the parameter *nBits* is checked and if it is greater than 1 it is performed a linear interpolation, increasing the input signal original sample rate to *rFactor* times.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Version 20190506

Version 20190509 adds the option to use FIR interpolation using a multiphase interpolation filter similar to the one presented in Figure 2.51.

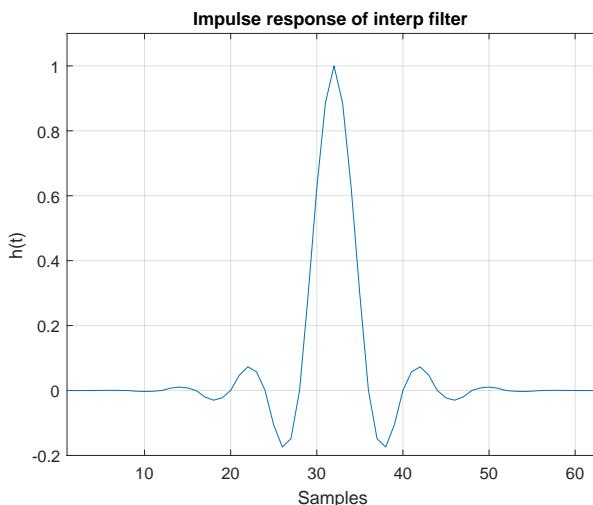


Figure 2.51: Impulse response of a $4 \times$ upsampling filter using 16 taps.

- **Input Parameters**

Name	Type	Values	Default Value
mode	UpSamplingMode	LinearInterpolation, FIRFilterInterpolation	LinearInterpolation
numberOfTaps	int	any	8

- **Methods**

- void setNumberOfTaps(int nTaps) { numberOfTaps = nTaps; }

```
- void setMode(UpSamplingMode m)
{ mode = m; }
```

Sugestions for future improvement

2.66 SNR Estimator

Header File	:	snr_estimator_*.h
Source File	:	snr_estimator_*.cpp
Version	:	20180227 (Andoni Santos)

Input Parameters

Name	Type	Value
Confidence	double	0.95
measuredIntervalSize	int	1024
windowType	WindowType	Hamming
segmentSize	int	512
overlapCount	int	256
active	bool	false

Methods

```

SNREstimator() ;

SNREstimator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig) ;

void initialize(void);

bool runBlock(void);

void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

int getMeasuredIntervalSize(void) return measuredIntervalSize;

void setSegmentSize(int sz) segmentSize = sz;

int getSegmentSize(void) return segmentSize;

void setOverlapCount(int olp) overlapCount = olp;

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olp) overlapCount = floor(segmentSize*olp);

double getOverlapPercent(void) return overlapCount/segmentSize;

void setConfidence(double P) alpha = 1-P;

```

```
double getConfidence(void) return 1 - alpha;  
  
void setWindowType(WindowType wd) windowType = wd;  
  
WindowType getWindowType(void) return windowType;  
  
void setFilename(string fname) filename = fname;  
  
string getFilename(void) return filename;  
  
vector<complex<double>> fftshift(vector<complex<double>> &vec);  
  
void setRollOff(double rollOff) rollOffComp = rollOff;  
  
double getRollOff(void) return rollOffComp;  
  
void setNoiseBw(double nBw) noiseBw = nBw;  
  
double getNoiseBw(void) return noiseBw;  
  
void setEstimatorMethod(SNREstimatorMethod mtd) method = mtd;  
  
SNREstimatorMethod getEstimatorMethod(void) return method;  
  
void setIgnoreInitialSamples(int iis) ignoreInitialSamples = iis;  
  
int getIgnoreInitialSamples(void) return ignoreInitialSamples;  
  
void setActivityState(bool state) active = state;  
  
bool getActivityState(void) return active;
```

Input Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This accepts one OpticalSignal or TimeContinuousAmplitudeContinousReal signal (or two, an In-phase signal and a quadrature signal), estimates the signal-to-noise ratio for a given time interval and outputs an exact copy of the input signal. The estimated SNR value is printed to *cout* after each calculation. The block also writes a .txt file reporting the estimated signal-to-noise ratio, a count of the number of measurements and the corresponding bounds for a given confidence level, based on measurements made on consecutive signal segments.

Theoretical Description

This block can use one of several methods to estimate the signal's SNR. Any of them can be useful for particular use cases.

Currently there are three methods implemented:

- **powerSpectrum** - Spectral analysis [[harris12](#), [xiao10](#), [kashefi12](#)];
- **m2m4** - Moments method [[matzner93](#)];
- **ren** - Modified moments method [[ren05](#)];

Spectral analysis

Several methods exist for SNR estimation, most of them requiring prior knowledge of the sample time or of the signal's conditions. Spectrum-based SNR estimators have been shown to provide accurate results even if these informations are not available, at the cost of being less efficient [[harris12](#), [xiao10](#), [kashefi12](#)].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [[john2007digital](#)]. This averages the values in each bin, providing a smoother estimate of the power spectrum. Using the power spectrum obtained from this sequence, the frequency interval containing the signal is estimated from the sampling rate, symbol rate and modulation type. The power contained in this interval will include both the signal and the white noise in those frequencies.

The power of a signal can be calculated from its power spectrum [[john2007digital](#)]:

$$P = \frac{1}{N} \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N^2} \sum_{k=0}^{N-1} |X(k)|^2 \quad (2.51)$$

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the spectrum in the area without signal. This is used to calculate the total noise power. The signal power is obtained by summing the FFT bins within the signal's frequency interval and subtracting the corresponding superimposed noise, according to what was previously estimated. The power SNR is the ratio between these two values.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \quad (2.52)$$

This

value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [tranter2004principles]. The SNR value and confidence interval are then saved to a text file.

Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal wo estimate the SNR [matzner93].

Let the second and fourth order moments be M_2 and M_4 . In addition, let S_k be the sampled signal, with S_{I_k} and S_{Q_k} as its in-phase and quadrature components.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (2.53)$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{I_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4P_S P_N + k_n P_N^2 \quad (2.54)$$

Here, k_e and k_n are constants depending on the properties of the probability density function of the signal and noise processes. In practice, they are known from the modulation scheme used and the noise characteristics. For signals with constant energy, for instance, $k_e = 1$, and for two dimensional gaussian noise $k_n = 2$. It follows that the signal and noise powers can be defined as follows:

$$P_S = \sqrt{2M_2^2 - M_4} \quad (2.55)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (2.56)$$

and therefore

$$\text{SNR} = \frac{\sqrt{2EM_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \quad (2.57)$$

Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [ren05].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (2.58)$$

Thus, P_S , P_N and the SNR become:

$$P'_S = \sqrt{M_2^2 - M_4'} \quad (2.59)$$

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (2.60)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (2.61)$$

Known Issues

This block currently only works with either one or two *TimeContinuousAmplitudeContinuousReal* signals, depending on the chosen method. It has also only been tested with the in-phase and quadrature components of an MQAM signal with M=4. If the noise's power spectral density is high enough for the signal not to be detected, the block will fail (in the mentioned case, typically values of SNR < -10dB can be unreliable).

2.67 Sampler

Header File	:	sampler.h
Source File	:	sampler_20171119.cpp

This block can work in two configurations: with an external clock or without it. In the latter it accepts two input signals one being the clock and the other the signal to be demodulated. In the other configuration there's only one input signal which is the signal.

The output signal is obtained by sampling the input signal with a predetermined sampling rate provided either internally or by the clock.

Input Parameters

Parameter	Type	Values	Default
samplesToSkip	int	any (smaller than the number of samples generated)	0

Table 2.36: Sampler input parameters

Methods

Sampler()

```
Sampler(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig)
```

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setSamplesToSkip(t_integer sToSkip)
```

Functional description

This block can work with an external clock or without it.

In the case of having an external clock it accepts two input signals. The signal to be demodulate which is complex and a clock signal that is a sequence of Dirac delta functions with a predetermined period that corresponds to the sampling period. The signal and the clock signal are scanned and when the clock has the value of 1.0 the correspondent complex value of the signal is placed in the buffer corresponding to the output signal.

There's a detail worth noting. The electrical filter has an impulse response time length of 16 (in units of symbol period). This means that when modulating a bit the spike in the signal corresponding to that bit will appear 8 units of symbol period later. For this reason there's

the need to skip the earlier samples of the signal when demodulating it. That's the purpose of the *samplesToSkip* parameter.

Between the binary source and the current block the signal is filtered twice which means that this effect has to be taken into account twice. Therefore the parameter *samplesToSkip* is given by $2 * 8 * \text{samplesPerSymbol}$.

Input Signals

Number: 1

Type: Electrical real (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical real (TimeDiscreteAmplitudeContinuousReal)

Examples

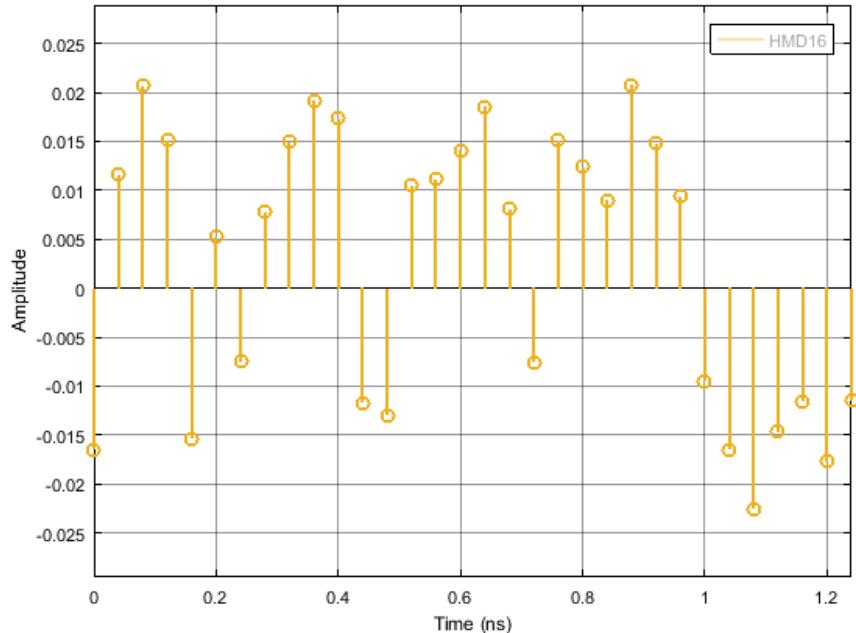


Figure 2.52: Example of the output signal of the sampler

Sugestions for future improvement

2.68 SNR of the Photoelectron Generator

Header File	:	srn_photoelectron_generator_*.h
Source File	:	snr_photoelectron_generator_*.cpp
Version	:	20180309 (Diamantino Silva)

This block estimates the signal to noise ratio (SNR) of a input stream of photoelectrons, for a given time window.

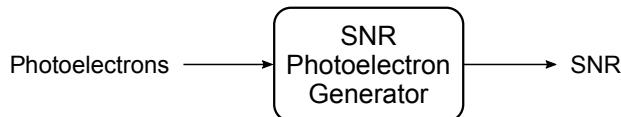


Figure 2.53: Schematic representation of the SNR of the Photoelectron Generator code block

Theoretical description

The input of this block is a stream of samples, y_j , each one of them corresponding to a number of photoelectrons generated in a time interval Δt . These photoelectrons are usually the output of a photodiode (photoelectron generator). To calculate the SNR of this stream, we will use the definition used in [1]

$$\text{SNR} = \frac{\bar{n}^2}{\sigma_n^2} \quad (2.62)$$

in which \bar{n} is the mean value and σ_n^2 is the variance of the photon number in a given time interval T .

To apply this definition to our input stream, we start by separate it's samples in contiguous time windows with duration T . Each time window k is defined as the time interval $[kT, (k + 1)T[$. To estimate the SNR for each time window, we will use the following estimators for the mean, μ_k , and variance, s_k^2 [2]

$$\mu_k = \langle y \rangle_k \quad s_k^2 = \frac{N}{N - 1} \left(\langle y^2 \rangle_k - \langle y \rangle_k^2 \right) \quad (2.63)$$

where $\langle y^n \rangle_k$ is the n moment of the k window given by

$$\langle y^n \rangle_k = \frac{1}{N_k} \sum_{j=j_{\min}(k)}^{j_{\max}(k)} y_j^n \quad (2.64)$$

in which

$$j_{min}(k) = \lceil t_k / \Delta t \rceil \quad (2.65)$$

$$j_{max}(k) = \lceil t_{k+1} / \Delta t \rceil - 1 \quad (2.66)$$

$$N_k = j_{max}(k) - j_{min}(k) + 1 \quad (2.67)$$

$$t_k = kT \quad (2.68)$$

where $\lceil x \rceil$ is the ceiling function.

In our implementation, we define two variables, $S_1(k)$ and $S_2(k)$, corresponding to the sum of the samples and the sum of the squares of the sample in the time interval k . These two sums are related to the moments as

$$S_1(k) = N_k \langle y \rangle_k \quad (2.69)$$

$$S_2(k) = N_k \langle y^2 \rangle_k \quad (2.70)$$

Using these two variables, we can rewrite μ_k and s_k^2 as

$$\mu_k = \frac{S_1(k)}{N_k} \quad s_k^2 = \frac{1}{N_k - 1} \left(S_2(k) - \frac{1}{N_k} (S_1(k))^2 \right) \quad (2.71)$$

The signal to noise ratio of the time interval k , SNR_k , can be expressed as

$$\text{SNR}_k = \frac{\mu_k^2}{\sigma_k^2} = \frac{N_k - 1}{N_k} \frac{(S_1(k))^2}{N_k S_2(k) - (S_1(k))^2} \quad (2.72)$$

One particularly important case is the phototransistor stream resulting from the conversion of a laser photon stream by a photodiode (phototransistor generator). The resulting SNR will be [1]

$$\text{SNR} = \eta \bar{n} \quad (2.73)$$

in which η is the photodiode quantum efficiency.

Functional description

This block is designed to operate in time windows, dividing the input stream in contiguous sets of samples with a duration $t_{\text{Window}} = T$. For each time window, the general process consists in accumulating the input sample values and the square of the input sample values, and calculating the SNR of the time window based on these two variables.

To process this accumulation, the block uses two state variables, `aux_sum1` and `aux_sum2`, which hold the accumulation of the sample values and accumulation of the square of sample values, respectively.

The block starts by calculating the number of samples it has to process for the current time window, using equations 2.66, 2.67 and 2.68. If the duration of t_{Window} is 0, then we assume that this time window has infinite time (infinite samples). The values of `aux_sum1` and `aux_sum2` are set to 0, and the processing of the samples of current window begins.

After processing all the samples of the time window, we obtain $S_1(k)$ and $S_2(k)$ from the

state variables as $S_1(k) = \text{aux_sum1}$ and $S_2(k) = \text{aux_sum2}$, and proceed to the calculation of the SNR_k , using equation 2.72.

If the simulation ends before reaching the end of the current time window, we calculate the SNR_k , using the current values of `aux_sum1`, `aux_sum2` for $S_1(k)$ and $S_2(k)$, and the number of samples already processed, `currentWindowSample`, for N_k .

Input Parameters

Parameter	Default Value	Description
windowTime	0	SNR time window.

Methods

`SnrPhotoelectronGenerator()`

`SnrPhotoelectronGenerator(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)`
`:Block(InputSig, OutputSig)`

`void initialize(void)`

`bool runBlock(void)`

`void setTimeWindow(t_real timeWindow)`

Input Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Examples

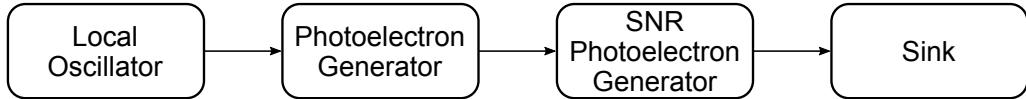


Figure 2.54: Simulation setup

To confirm the block's correct output, we have designed a simulation setup which calculates the SNR of a stream of photoelectrons generated by the detection of a laser photon stream by a photodiode.

The simulation has three main parameters, the power of the local oscillator, P_{LO} , the duration of the time window, T , and the photodiode's quantum efficiency, η . For each combination of these three parameters, the simulation generates 1000 SNR samples, during which all parameters stay constant. The final result is the average of these SNR samples. The simulations were performed with a sample time $\Delta t = 10^{-10}s$.

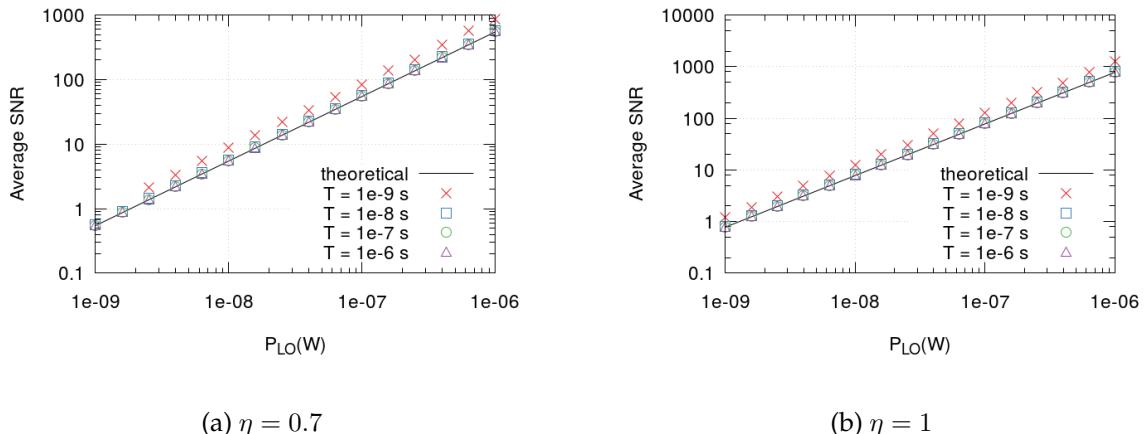


Figure 2.55: Theoretical and simulated results of the average SNR, for two photodiode efficiencies.

The plots in 2.55 show the comparison between the theoretical result 2.73 and the simulation results. We see that for low values of T , the average SNR shows a systematic deviation from the theoretical result, but for $T > 10^{-6}s$ (10000 samples per time window), the simulation result shows a very good agreement with the theoretical result.

The simulations also show a lack of average SNR results when low power, low efficiency and small time window are combined (see plot 2.55a). This is because in those conditions, the probability of having a time windows with no photoelectrons, creating an invalid SNR, is very high, which will prevent the calculation of the average SNR.

We can estimate the probability of calculating a valid SNR average by calculating the probability of no time window having 0 phototelectrons, $p_{ave} = (1 - q)^M$, in which q is the probability of a time window having all its samples equal to 0 and M is the number of time windows. We know that the input stream follows a Poisson distribution with mean \bar{m} , therefore $q = (\exp(-\bar{m}))^N$, in which $\bar{m} = \eta P \lambda / hc$ and $N = T/\Delta t$, is the average number of samples per time window. Using this result, we obtain the probability of calculating a valid SNR average as

$$p_{ave} = (1 - \exp(-N\bar{m}))^M \quad (2.74)$$

Block problems

Future work

The block could also output a confidence interval for the calculated SNR. Given that the output of the Photoelectron Generator follows a Poissonian distribution when the shot noise is on, the article "Confidence intervals for signal to noise ratio of a Poisson distribution" by Florence George and B.M. Kibria [3], could be used as a reference to implement such feature.

References

- [1] B. E. Saleh and M. C. Teich. *Fundamentals of photonics*. Wiley series in pure and applied optics. New York (NY): John Wiley & Sons, 1991.
- [2] S. W. Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.
- [3] G. Florence and K. B. Golam. "Confidence intervals for signal to noise ratio of a Poisson distribution". In: *American Journal of Biostatistics* 2.2 (2011), p. 44.

2.69 Sink

Header File	:	sink_*.h
Source File	:	sink_*.cpp
Version	:	20180523 (André Mourato)

This block accepts one input signal and it does not produce output signals. It takes samples out of the buffer until the buffer is empty. It has the option of displaying the number of samples still available.

Input Parameters

Parameter	Type	Values	Default
numberOfSamples	long int	any	-1
displayNumberOfSamples	bool	true/false	true

Table 2.37: Sink input parameters

Methods

```

Sink(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)

bool runBlock(void)

void setAsciiFilePath(string newName)

string getAsciiFilePath()

void setNumberOfBitsToSkipBeforeSave(long int newValue)

long int getNumberOfBitsToSkipBeforeSave()

void setNumberOfBytesToFile(long int newValue)

long int getNumberOfBytesToFile()

void setNumberOfSamples(long int nOfSamples)

long int getNumberOfSamples const()

void setDisplayNumberOfSamples(bool opt)

bool getDisplayNumberOfSamples const()

```

Functional Description

The Sink block discards all elements contained in the signal passed as input. After being executed the input signal's buffer will be empty.

2.70 SNR Estimator

Header File	:	snr_estimator_*.h
Source File	:	snr_estimator_*.cpp
Version	:	20180227 (Andoni Santos)

Input Parameters

Name	Type	Value
Confidence	double	0.95
measuredIntervalSize	int	1024
windowType	WindowType	Hamming
segmentSize	int	512
overlapCount	int	256
active	bool	false

Methods

```

SNREstimator() ;

SNREstimator(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig) ;

void initialize(void);

bool runBlock(void);

void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

int getMeasuredIntervalSize(void) return measuredIntervalSize;

void setSegmentSize(int sz) segmentSize = sz;

int getSegmentSize(void) return segmentSize;

void setOverlapCount(int olp) overlapCount = olp;

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olp) overlapCount = floor(segmentSize*olp);

double getOverlapPercent(void) return overlapCount/segmentSize;

void setConfidence(double P) alpha = 1-P;

```

```
double getConfidence(void) return 1 - alpha;  
  
void setWindowType(WindowType wd) windowType = wd;  
  
WindowType getWindowType(void) return windowType;  
  
void setFilename(string fname) filename = fname;  
  
string getFilename(void) return filename;  
  
vector<complex<double>> fftshift(vector<complex<double>> &vec);  
  
void setRollOff(double rollOff) rollOffComp = rollOff;  
  
double getRollOff(void) return rollOffComp;  
  
void setNoiseBw(double nBw) noiseBw = nBw;  
  
double getNoiseBw(void) return noiseBw;  
  
void setEstimatorMethod(SNREstimatorMethod mtd) method = mtd;  
  
SNREstimatorMethod getEstimatorMethod(void) return method;  
  
void setIgnoreInitialSamples(int iis) ignoreInitialSamples = iis;  
  
int getIgnoreInitialSamples(void) return ignoreInitialSamples;  
  
void setActivityState(bool state) active = state;  
  
bool getActivityState(void) return active;
```

Input Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2

Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This accepts one OpticalSignal or TimeContinuousAmplitudeContinousReal signal (or two, an In-phase signal and a quadrature signal), estimates the signal-to-noise ratio for a given time interval and outputs an exact copy of the input signal. The estimated SNR value is printed to *cout* after each calculation. The block also writes a .txt file reporting the estimated signal-to-noise ratio, a count of the number of measurements and the corresponding bounds for a given confidence level, based on measurements made on consecutive signal segments.

Theoretical Description

This block can use one of several methods to estimate the signal's SNR. Any of them can be useful for particular use cases.

Currently there are three methods implemented:

- **powerSpectrum** - Spectral analysis [harris12, xiao10, kashefi12];
- **m2m4** - Moments method [matzner93];
- **ren** - Modified moments method [ren05];

Spectral analysis

Several methods exist for SNR estimation, most of them requiring prior knowledge of the sample time or of the signal's conditions. Spectrum-based SNR estimators have been shown to provide accurate results even if these informations are not available, at the cost of being less efficient [harris12, xiao10, kashefi12].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [john2007digital]. This averages the values in each bin, providing a smoother estimate of the power spectrum. Using the power spectrum obtained from this sequence, the frequency interval containing the signal is estimated from the sampling rate, symbol rate and modulation type. The power contained in this interval will include both the signal and the white noise in those frequencies.

The power of a signal can be calculated from its power spectrum [john2007digital]:

$$P = \frac{1}{N} \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N^2} \sum_{k=0}^{N-1} |X(k)|^2 \quad (2.75)$$

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the spectrum in the area without signal. This is used to calculate the total noise power. The signal power is obtained by summing the FFT bins within the signal's frequency interval and subtracting the corresponding superimposed noise, according to what was previously estimated. The power SNR is the ratio between these two values.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \quad (2.76)$$

This

value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [tranter2004principles]. The SNR value and confidence interval are then saved to a text file.

Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal wo estimate the SNR [matzner93].

Let the second and fourth order moments be M_2 and M_4 . In addition, let S_k be the sampled signal, with S_{I_k} and S_{Q_k} as its in-phase and quadrature components.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (2.77)$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{I_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4P_S P_N + k_n P_N^2 \quad (2.78)$$

Here, k_e and k_n are constants depending on the properties of the probability density function of the signal and noise processes. In practice, they are known from the modulation scheme used and the noise characteristics. For signals with constant energy, for instance, $k_e = 1$, and for two dimensional gaussian noise $k_n = 2$. It follows that the signal and noise powers can be defined as follows:

$$P_S = \sqrt{2M_2^2 - M_4} \quad (2.79)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (2.80)$$

and therefore

$$\text{SNR} = \frac{\sqrt{2EM_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \quad (2.81)$$

Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [ren05].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (2.82)$$

Thus, P_S , P_N and the SNR become:

$$P'_S = \sqrt{M_2^2 - M_4'} \quad (2.83)$$

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (2.84)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (2.85)$$

Known Issues

This block currently only works with either one or two *TimeContinuousAmplitudeContinuousReal* signals, depending on the chosen method. It has also only been tested with the in-phase and quadrature components of an MQAM signal with M=4. If the noise's power spectral density is high enough for the signal not to be detected, the block will fail (in the mentioned case, typically values of SNR < -10dB can be unreliable).

2.71 Single Photon Receiver

This block of code simulates the reception of two time continuous signals which are the outputs of single photon detectors and decode them in measurements results. A simplified schematic representation of this block is shown in figure 2.56.

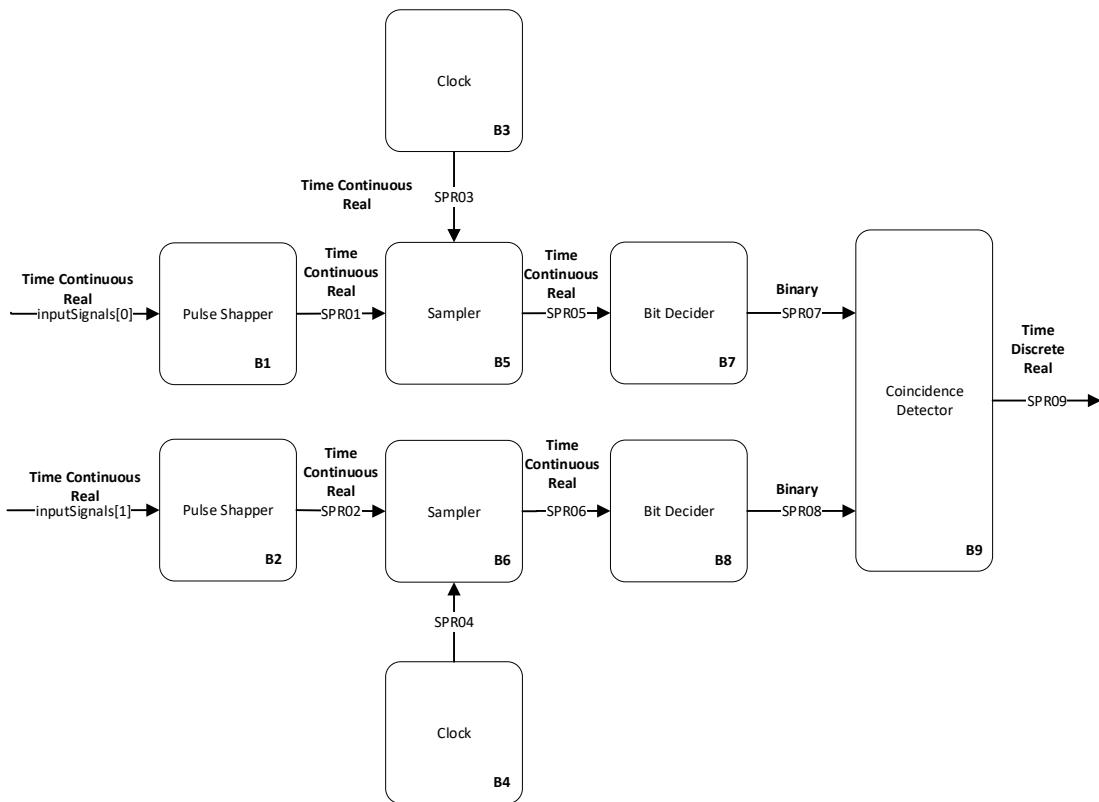


Figure 2.56: Basic configuration of the SPR receiver

Functional description

This block accepts two time continuous input signals and outputs one time discrete signal that corresponds to the single photon detection measurements demodulation of the input signal. It is a complex block (as it can be seen from figure 2.56 of code made up of several simpler blocks whose description can be found in the *lib* repository).

It can also be seen from figure 2.56 that there are two extra internal input signals generated by the *Clock* in order to keep all blocks synchronized. This block is used to provide the sampling frequency to the *Sampler* blocks.

Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 2.39) the input parameters and corresponding functions are summarized.

Input parameters	Function	Type	Accepted values
samplesToSkip	Samples to skip in sampler block	int values	
filterType	Type of the filter applied in pulse shapper	PulseShaperFilter	

Table 2.38: List of input parameters of the block SP receiver

Methods

```
SinglePhotonReceiver(vector <Signal*> &inputSignals, vector <Signal*>&outputSignals)({constructor})
```

```
void setPulseShaperFilter(PulseShaperFilter fType)
```

```
void setPulseShaperWidth(double pulseW)
```

```
void setClockBitPeriod(double period)
```

```
void setClockPhase(double phase)
```

```
void setClockSamplingPeriod(double sPeriod)
```

```
void setThreshold(double threshold)
```

Input Signals

Number: 2

Type: Time Continuous Amplitude Continuous Real

Output Signals

Number: 1

Type: Time Discrete Amplitude Discrete Real

Example

Sugestions for future improvement

2.72 SOP Modulator

Header File	:	sop_modulator_*.h
Source File	:	sop_modulator_*.cpp
Version	:	20180514 (Mariana Ramos)

This block of code simulates a modulation of the State Of Polarization (SOP) in a quantum channel, which intends to insert possible errors occurred during the transmission due to the polarization rotation of single photons. These SOP changes can be simulated using deterministic or stochastic methods. The type of simulation is one of the input parameters when the block is initialized.

Functional description

This block intends to simulate SOP changes using deterministic and stochastic methods. The required function mode must be set when the block is initialized. Furthermore, other input parameters should be also set at initialization. If a deterministic method was set by the user, he also needs to set the θ and ϕ angles in degrees, which corresponds to the two parameters of Jones Space in Poincare Sphere thereby being the rotation and elevation angle, respectively. On the other hand, if a stochastic method is set by the user a model proposed in [1] was implemented.

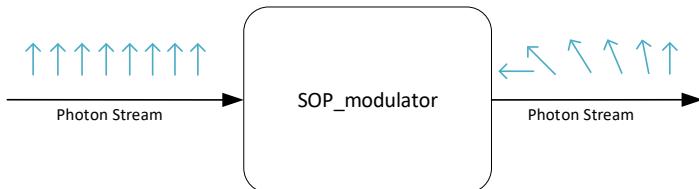


Figure 2.57: Diagram block of SOP block input and outputs.

The block in figure 2.57 was implemented based on a continuous matrix calculation. Lets assume that the input photon stream can be represented by the matrix:

$$S_{in} = \begin{bmatrix} A_x \\ A_y \end{bmatrix}, \quad (2.86)$$

where A_x and A_y are complex numbers. The SOP block will implement the multiplication operation between the input photon S_{in} and the random matrix J_k obtaining the output photon S_{out} with a different polarization:

$$S_{out} = S_{in}J_k, \quad (2.87)$$

where,

$$J_k = J(\dot{\alpha})J_{k-1}. \quad (2.88)$$

$J(\alpha)$ is a random matrix which can be expressed using the matrix exponential parameterized by $\alpha = (\alpha_1, \alpha_2, \alpha_3)$:

$$\begin{aligned} J(\alpha) &= e^{-i\alpha \cdot \vec{\sigma}} \\ &= \mathbf{I}_2 \cos(\theta) - i\mathbf{a} \cdot \vec{\sigma} \sin(\theta), \end{aligned} \quad (2.89)$$

where $\vec{\sigma} = (\sigma_1, \sigma_2, \sigma_3)$ is the tensor of Pauli Matrices:

$$\sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}; \sigma_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \sigma_3 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \quad (2.90)$$

\mathbf{I}_2 is the 2×2 identity matrix. In addition, $\alpha = \theta\mathbf{a}$, having the length $\theta = \|\alpha\|$ and the direction on the unit sphere $\mathbf{a} = (a_1, a_2, a_3)$, where $\|\cdot\|$ denotes the Euclidian norm. Furthermore, the randomness is achieved by α parameters:

$$\dot{\alpha} \sim \mathcal{N}(0, \sigma_p^2 \mathbf{I}_3), \quad (2.91)$$

where $\sigma_p^2 = 2\pi\Delta p T$, being T the symbol period and Δp the polarization linewidth, which is a parameter dependent on the fiber installation.

This method allows to analyse the polarization drift over time for a fixed fiber length.

Input parameters

SOP modulator block must have an input signals to set the clock of the operations. This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the SOP modulator. Each parameter has associated a function that allows for its change. In the following table (table 2.39) the input parameters and corresponding functions are summarized.

Input parameters	Function	Accepted values
sopType	Simulation type that the user intends to simulate.	Deterministic OR Stochastic
theta	Rotation Angle in Jones space for deterministic rotation in degrees	double
phi	Elevation Angle in Jones space for deterministic rotation in degrees	double
deltaP	Polarization Linewidth	double

Table 2.39: List of input parameters of the block sop modulator.

Methods

```
SOPModulator(vector<Signal*> &inputSignals, vector<Signal*> &outputSignals)(constructor)
```

```
void initialize(void)  
bool runBlock(void)  
void setSOPType(SOPType sType)  
void setRotationAngle(double angle)  
void setElevationAngle(double angle)  
void setDeltaP(long double deltaP)  
long double getDeltaP()
```

Input Signals

Number: 1

Type: PhotonStreamXY

Output Signals

Number: 1

Type: PhotonStreamXY

Example

Sugestions for future improvement

References

- [1] Cristian B Czegledi et al. "Polarization drift channel model for coherent fibre-optic systems". In: *Scientific reports* 6 (2016), p. 21217.

2.73 Source Code Efficiency

Header File	:	source_code_efficiency_*.h
Source File	:	source_code_efficiency_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

This block accepts one input signal and it produces one output signal. To perform this block, two input variables are required, probabilityOfZero and sourceOrde.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 2.40: Source Code Efficiency input parameters

Functional Description

This block estimates the efficiency of the message, by calculating the entropy and the length of the message.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 1

Type: Real (TimeContinuousAmplitudeContinuousReal)

2.74 White Noise

Header File	:	white_noise_20180420.h
Source File	:	white_noise_20180420.cpp

This block generates a gaussian pseudo-random noise signal with a given spectral density. It can be initialized with three different seeding methods to allow control over correlation and reproducibility:

1. DefaultDeterministic
2. RandomDevice
3. Selected

This block does not accept any input signal. It produces can produce a real or complex output, depending on the used output signal.

Input Parameters

Parameter	Type	Values	Default
seedType	enum	DefaultDeterministic, RandomDevice, Selected	RandomDevice
spectralDensity	real	> 0	1.5×10^{-17}
seed	int	$\in [1, 2^{32} - 1]$	1
samplingPeriod	double	> 0	1.0

Table 2.41: White noise input parameters

Methods

```
WhiteNoise(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig, OutputSig);
```

```
void initialize(void);

bool runBlock(void);

void setNoiseSpectralDensity(double SpectralDensity) spectralDensity = SpectralDensity;

double const getNoiseSpectralDensity(void) return spectralDensity;

void setSeedType(SeedType sType) seedType = sType; ;
```

```

SeedType const getSeedType(void) return seedType; ;

void setSeed(int newSeed) seed = newSeed;

int getSeed(void) return seed;

```

Functional description

The *seedType* parameter allows the user to select between one of the three seeding methods to initialize the pseudo-random number generators (PRNGs) responsible for generating the noise signal.

DefaultDeterministic: Uses default seeds to initialize the PRNGs. These are different for all generators used within the same block, but remain the same for sequential runs or different *white_noise* blocks. Therefore, if more than one *white_noise* block is used, another seeding method should be chosen to avoid producing the exact same noise signal in all sources.

RandomDevice: Uses randomly chosen seeds using *std::random_device* to initialize the PRNGs.

SingleSelected: Uses one user selected seed to initialize the PRNGs. The selected seed is passed through the variable *seed*. If more than one generator is used, additional seeds are created by choosing the next sequential integers. For instance, if the user selected seed is 10, and all the four PRNGs are used, the used seeds will be [10, 11, 12, 13].

The noise is obtained from a gaussian distribution with zero mean and a given variance. The variance is equal to the noise power, which can be calculated from the spectral density n_0 and the signal's bandwidth B , where the bandwidth is obtained from the defined sampling time T .

$$N = n_0 B = n_0 \frac{2}{T} \quad (2.92)$$

If the signal is complex, the noise is calculated independently for the real and imaginary parts, and the spectral density value is divided by two, to account for the two-sided noise spectral density.

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: RealValue, ComplexValue or ComplexValueXY

Examples

Random Mode

Suggestions for future improvement

2.75 Ideal Amplifier

This block has one input signal and one output signal both corresponding to electrical signals. The output signal is a perfect amplification of the input signal.

Input Parameters

Parameter	Type	Values	Default
gain	double	any	1×10^4

Table 2.42: Ideal Amplifier input parameters

Methods

IdealAmplifier()

```
IdealAmplifier(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig);

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;
```

Functional description

The output signal is the product of the input signal with the parameter *gain*.

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

2.76 Phase Mismatch Compensation

Header File	:	phase_mismatch_compensation_*.h
Source File	:	phase_mismatch_compensation_*.cpp
Version	:	20190114 (Daniel Pereira)
	:	20190506 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
numberOfSamplesForEstimation	integer	21
pilotRate	integer	2

Methods

- PhaseMismatchCompensation(vector<Signal * > &InputSig, vector<Signal * > &OutputSig) :Block(InputSig, OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setPilotRate(int pRate) { pilotRate = pRate; };
- void setNumberOfSamplesForEstimation(int nSamplesEstimation) {
 numberOfSamplesForEstimation = nSamplesEstimation;
 samplesForEstimation.resize(nSamplesEstimation); };

Input Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts a complex constellation signal and outputs another complex constellation built from its input. The block assumes pilot-aided phase mismatch compensation is being performed, takes the pilot signals before and after each signal, makes an average of the phase in each pilot and uses that estimation to compensate the phase mismatch.

Theoretical Description

The pilot-assisted phase mismatch compensation scheme employed in this block is based on the schemes proposed in [1, 2]. A representation of the output of the modulation stage of a pilot-assisted LLO CV-QC scheme is presented in Figure 2.58. An unmodulated pilot signal

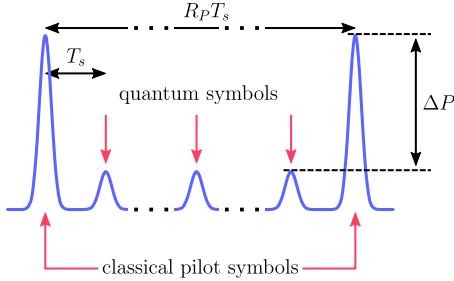


Figure 2.58: Representative time dependence of a pilot assisted phase mismatch compensation signal.

is sent, time-multiplexed with the information carrying pulses, with the pilot signal being used to estimate the phase mismatch between the two lasers. At the receiver, the quantum and the pilot constellation are given, respectively, by

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\epsilon(t_n)]}, & n = mR_P \end{cases}, \quad n, m \in \mathbb{N}. \quad (2.93)$$

The phase mismatch for the nth pulse is estimated from an average of the phase of the previous and the later pilots, resulting in

$$\hat{\epsilon}(t_n) = \frac{\epsilon(t_{(m+1)R_P}) + \epsilon(t_{mR_P})}{2}, \quad mR_P < n < (m+1)R_P, \quad (2.94)$$

with the mismatch compensation being accomplished by multiplying the constellation by $e^{-i(\hat{\epsilon}(t_n))}$, resulting in

$$x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)-\hat{\epsilon}(t_n)]}. \quad (2.95)$$

Version 20190506

Version 20190506 allows setting different pilot constellations and orders. Associated with this additional capability, the new version also allows setting the state of the first pilot symbol.

- **Input Parameters**

Name	Type	Default Value
pilotStateIndex	int	1
pilotConstellation	vector<vector<t_real>>	{ {1.0, 0.0} }

- **Methods**

- void setFirstPilotState(int fPilotState) { pilotStateIndex = fPilotState; }
- void setPilotConstellation(vector<vector<t_real>> pConstellation){
 pilotConstellationPhase.resize(pConstellation.size());
 pilotConstellation.resize(pConstellation.size());
 pilotConstellation = pConstellation; }

References

- [1] Daniel BS Soh et al. "Self-referenced continuous-variable quantum key distribution protocol". In: *Physical Review X* 5.4 (2015), p. 041010.
- [2] Bing Qi et al. "Generating the local oscillator "locally" in continuous-variable quantum key distribution based on coherent detection". In: *Physical Review X* 5.4 (2015), p. 041009.

2.77 Frequency Mismatch Compensation

Header File	:	frequency_mismatch_compensation_*.h
Source File	:	frequency_mismatch_compensation_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
pilotRate	integer	0
numberOfSamplesForEstimation	integer	21

Methods

- FrequencyMismatchCompensation(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setPilotRate(int pRate) { pilotRate = pRate; };
- void setNumberOfSamplesForEstimation(int nSamplesEstimation) { numberOfSamplesForEstimation = nSamplesEstimation; samplesForEstimation.resize(nSamplesEstimation); };
- void setMode(FrequencyCompensationMode m) { mode = m; }

Input Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts a complex constellation and outputs another built from its input. The block attempts to perform frequency mismatch compensation on the input constellation by one of three different methods:

- Pilot aided method
- Blind estimation method
- Spectral method

Theoretical Description

The signal at the receiver for a system with frequency mismatch can be assumed to take the form

$$y(t) = |y(t)|e^{i[\Delta\omega t + \theta(t) + \epsilon(t)]}, \quad (2.96)$$

where $\Delta\omega$ is the frequency mismatch, $\theta(t_n)$ is the phase encoded in the signal and $\epsilon(t_n)$ is the phase noise contribution. Frequency mismatch compensation is accomplished by first estimating the value of $\Delta\omega$ and then removing via

$$y(t)' = y(t) * e^{-i\Delta\hat{\omega}}. \quad (2.97)$$

Three methods for estimating $\Delta\omega$ are presented here.

Pilot aided frequency mismatch compensation

This method uses a pilot signal similar to the ones employed in pilot assisted phase mismatch compensation techniques. In a pilot aided technique a reference signal (the pilot), composed of pre-agreed on symbols, is inserted, time multiplexed, with the data payload at a pre-agreed on rate. A visual representation of the output of the modulation stage of a pilot-assisted scheme is presented in Figure 2.59. At the receiver stage, after coherent detection,

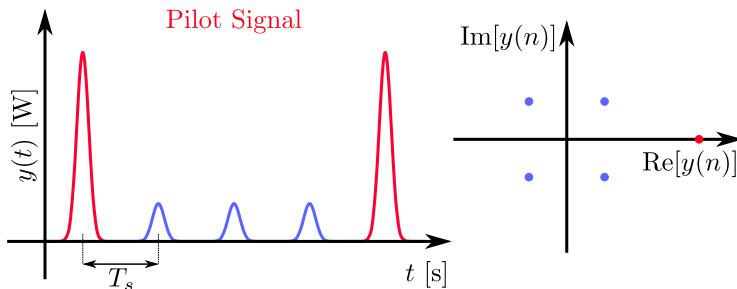


Figure 2.59: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted scheme. Pilot pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

the signal is described by

$$y(t_n) = \begin{cases} x_s(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n + \epsilon(t_n)]}, & n = mR_P \end{cases}, \quad (2.98)$$

where $x_s(t)/x_p(t)$ represents the signal/pilot amplitude, R_P is the pilot rate, $\Delta\omega$ is the offset frequency, $\epsilon(t)$ is the instantaneous phase noise contribution and $\theta(t)$ is the phase

modulation at instant t . The first step in this technique is to obtain the auxiliary signal $x_f(m)$, which is defined as

$$\begin{aligned} x_f(m) &= y_r^*(t_{mR_p})y_r(t_{(m+1)R_p}) \\ &= x_p(t_{mR_p})x_p(t_{(m+1)R_p})e^{i(\Delta\omega R_p T_s + \Delta\epsilon(m))}, \end{aligned} \quad (2.99)$$

where

$$\Delta\epsilon(m) = \epsilon(t_{mR_p}) - \epsilon(t_{(m+1)R_p}). \quad (2.100)$$

The frequency estimation is accomplished by taking the expected value of the complex argument of (2.99) and dividing it by $R_p T_s$. This returns

$$\begin{aligned} \hat{\Delta\omega} &= E\left[\frac{1}{R_p T_s} \arg(x_f(m))\right] \\ &= E\left[\Delta\omega + 2k\pi(R_p T_s)^{-1} + \frac{\Delta\epsilon(m)}{R_p T_s}\right] \\ &= \Delta\omega + 2k\pi(R_p T_s)^{-1}. \end{aligned} \quad (2.101)$$

Blind estimation frequency mismatch compensation

In this method the frequency is scanned over a predetermined range, symbol decisions are made and the minimum square error used as the frequency-selection criteria. Scanning is first done with a large step to find a rough value for $\Delta\omega$, this is then repeated with a smaller step to find a more exact estimate [1].

Spectral evaluation frequency mismatch compensation

In this method the frequency is estimated by evaluating the spectrum of the m th power of the input signal [2], which exhibits a peak at the frequency $m\Delta\Omega$, this value is used as the estimate.

References

- [1] Xiang Zhou et al. "64-Tb/s, 8 b/s/Hz, PDM-36QAM transmission over 320 km using both pre-and post-transmission digital signal processing". In: *Journal of Lightwave Technology* 29.4 (2011), pp. 571–577.
- [2] Mehrez Selmi, Yves Jaouën, and Philippe Ciblat. "Accurate digital frequency offset estimator for coherent PolMux QAM transmission systems". In: *Optical Communication, 2009. ECOC'09. 35th European Conference on*. IEEE. 2009, pp. 1–2.

2.78 Cloner

Header File	:	cloner_*.h
Source File	:	cloner_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

This block takes no input parameters.

Methods

- `Cloner(vector<Signal * > &InputSig, vector<Signal * > &OutputSig) :Block(InputSig,OutputSig){};`
- `void initialize(void);`
- `bool runBlock(void);`

Input Signals

Number: 1

Type: Real, Complex, Complex_XY, Binary

Output Signals

Number: Arbitrary

Type: Same as input

Functional Description

This block accepts a signal and outputs a number of copies of the input. The number of the copies is set by the number of output signals given to the block. The block adapts dynamically.

Theoretical Description

2.79 Error Vector Magnitude

Header File	:	error_vector_magnitude_*.h
Source File	:	error_vector_magnitude_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
referenceAmplitude	double	1.0
m	integer	0
midRepType	enum	Cumulative

Methods

- ErrorVectorMagnitude(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setMidReportType(MidReportType mrt) { midRepType = mrt; }
- void setReferenceAmplitude(double rAmplitude) { referenceAmplitude = rAmplitude; }

Input Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1

Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts one binary string and outputs copy of the input binary string. This block also outputs .txt files with a report of the estimated Error Vector Magnitude (EVM), \widehat{EVM} .

The block allows for mid-reports to be generated, the number of bits between reports is customizable, if it is set to 0 then the block will only output the final report. This block can operate mid-reports using CUMULATIVE mode, in which \widehat{EVM} is calculated taking into account all received points, or in a RESET mode, in which \widehat{EVM} is computed from only the points after the previous mid-report.

Theoretical Description

The \widehat{EVM} is obtained by evaluating the relation between the magnitude of the error vector \vec{e}_v and the magnitude of the vector of the ideal symbol position \vec{ref}_v . This process is presented visually in Figure 2.60 and is described by

$$\widehat{EVM} = 100 \sqrt{\frac{|\vec{e}_v|}{|\vec{ref}_v|}} = 100 \sqrt{\frac{|\vec{m}_v - \vec{ref}_v|}{|\vec{ref}_v|}}. \quad (2.102)$$

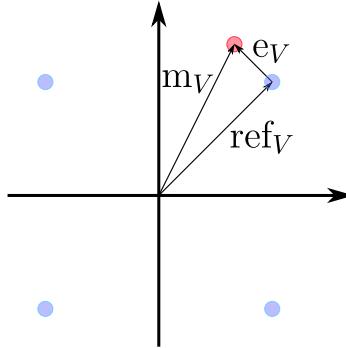


Figure 2.60: Visual representation of the EVM method in a QPSK constellation. The equal constellation points are presented in blue, while an example for the actual measured symbol is presented in red.

2.80 Load Ascii

Header File	:	load_ascii.h
Source File	:	load_ascii.cpp
Version	:	20190205 (Daniel Pereira)

This block loads signals from any ascii file into the netxpto simulation environment.

Input Parameters

Parameter	Type	Values	Default
samplingPeriod	double	any	1.0
symbolPeriod	double	any	1.0
asciiFileName	string	any	InputFile.txt
delimiterType	delimiter_type	CommaSeperatedValues, ConcatenatedValues	CommaSeperatedValues
dataType	signal_value_type	BinaryValue, RealValue, ComplexValue, ComplexValueXY	BinaryValue

Table 2.43: Load ascii input parameters

Methods

LoadAscii()

```
LoadAscii(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
bool runBlock(void);
void setSamplingPeriod(double sPeriod);
void setSymbolPeriod(double sPeriod);
void setDataType(signal_value_type dType);
void setAsciiFileName(string aFileName);
```

Functional description

This block loads signals from a .txt file and generates a signal with a specified data type given by the input parameter *dataType* and with a specified delimiter type given by the input parameter *delimiterType*.

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal, Electrical signal, Complex signal, Binary signal (user defined)

2.81 Load Signal

Header File	:	load_signal.h
Source File	:	load_signal.cpp
Version	:	20190205 (Daniel Pereira)
	:	20190503 (Daniel Pereira)

This block loads signals from a .sgn file into the netxpto simulation environment.

Input Parameters

Parameter	Type	Values	Default
sgnFileName	string	any	InputFile.sgn

Table 2.44: Load signal input parameters

Methods

LoadSignal()

```
LoadSignal(vector<Signal * > &InputSig, vector<Signal * > &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSgnFileName(string sFileName);
```

Functional description

This block loads signals from a .sgn file into a signal with symbol and sampling period set by the input .sgn file and type set by the output signal (make sure that the output signal has the correct format, otherwise it will be corrupted).

Input Signals

Number: 0

Output Signals

Number: 1

Type: Binary signal, Integer signal, Complex signal, Complex XY signal, Photon signal, Photon Multipath signal, Photon Multipath XY signal

Version 20190503

Version 20190503 allows the user to skip a user determined number of samples at the beginning of the file.

- **Input Parameters**

Name	Type	Default Value
samplesToSkip	int	0

- **Methods**

- void setSamplesToSkip(int sToSkip) { samplesToSkip = sToSkip; };

2.82 Matched Filter

Header File	:	matched_filter.h
Source File	:	matched_filter.cpp
Version	:	20190205 (Daniel Pereira)

This block applies a matched filter to the input signal.

Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	64
rollOffFactor	double	any	0.5

Table 2.45: Matched filter input parameters

Methods

```
MatchedFilter(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};
```

```
void initialize(void);
bool runBlock(void);
void setNumberOfTaps(int numberOfTaps);
void setRollOffFactor(double rollOffFactor);
```

Functional description

This block implements a matched filter, that coincides with the modulation applied to the signal at the transmitter stage. The filter is applied in frequency domain, where it takes the form of a simple multiplication.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

2.83 Timing Deskew

Header File	:	timing_deskew.h
Source File	:	timing_deskew.cpp
Version	:	20190225 (Daniel Pereira)

This block allows for the removal of the timing mismatches between the real and imaginary components of a complex signal.

Input Parameters

Parameter	Type	Values	Default
skew	vector<double>	any	[0, 0]

Table 2.46: Timing deskew input parameters

Methods

```
TimingDeskew(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSkew(vector<double> s) { skew.resize(s.size()); skew[0] = s[0]; skew[1] = s[1]; }
```

Functional description

This DSP step takes a complex input signal and removes a given amount of timing skew between its real (in-phase) and imaginary (quadrature) parts. This DSP step follows the topology presented in Figure 2.61, in which the input signal $S_{\text{in}}(n)$ is separated into its in-

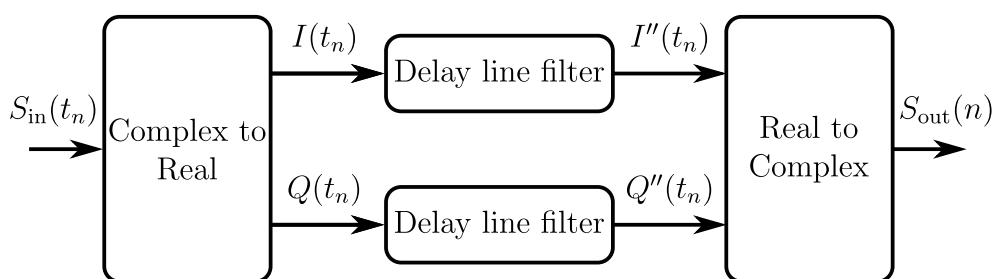


Figure 2.61: Block diagram representation of the deskeew procedure.

phase and in-quadrature components, $I(n)$ and $Q(n)$ and are each sent through independent delay line filters. In turn, the delay line filters function as shown in Figure 2.62. The signal

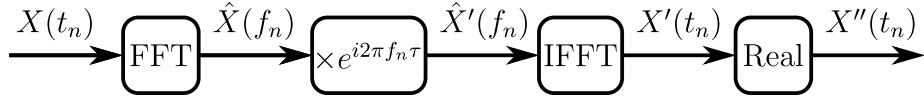


Figure 2.62: Block diagram representation of the implemented delay line filter.

at the input, $S_{in}(t_n)$, is transformed to Fourier space, where it is multiplied by a deskew element τ

$$\hat{S}'(f_n) = \hat{S}(f_n)e^{i2\pi f_n \tau}. \quad (2.103)$$

The value of τ here is set by trial and error and is characteristic of a certain receiver. For these results the timing skew was set at 20 ps.

This signal is then transformed back to time domain and the imaginary part is discarded. After both components are passed through the delay line, they are combined into a complex signal as the output

$$S_{out}(t_n) = \text{real}(I''(t_n)) + i\text{real}(Q''(t_n)). \quad (2.104)$$

Input Signals

Number: 1

Type: Complex signal

Output Signals

Number: 1

Type: Complex signal

2.84 DC component removal

Header File	:	dc_component_removal.h
Source File	:	dc_component_removal.cpp
Version	:	20190225 (Daniel Pereira)

This block removes the DC/low frequency component of a complex time continuous signal.

Input Parameters

This block has no input parameters.

Methods

```
DCComponentRemoval(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

Functional description

This DSP step removes the DC contribution on the input signal, $S_{\text{in}}(t_n)$. This DC contribution can either be estimated by taking the average of the full input signal or from the average of a moving window. This contribution is then subtracted from the input

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n) - \text{mean}(S_{\text{in}}(t_n)) \quad (2.105)$$

The effect of this DSP step is presented in Figure ???. To better show the effect of this DSP, an exaggerated DC removal is presented in Figure 2.63, where it can be seen that the DC component is removed without introducing obvious discontinuities in the signal.

Input Signals

Number: 1

Type: Complex signal

Output Signals

Number: 1

Type: Complex signal

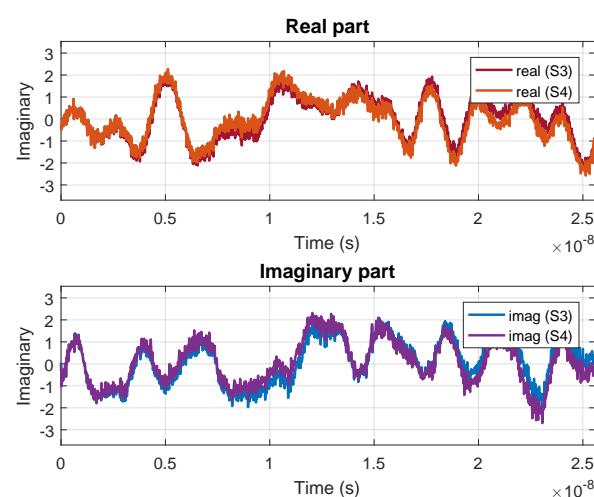


Figure 2.63: Exaggerated DC component removal.

2.85 Orthonormalization

Header File	:	orthonormalization.h
Source File	:	orthonormalization.cpp
Version	:	20190225 (Daniel Pereira)

This block applies a Gram-Schmidt orthonormalization procedure to the input signal.

Input Parameters

This block has no input parameters.

Methods

```
Orthonormalization(initializer_list<Signal * > &InputSig, initializer_list<Signal * > &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);
```

Functional description

This block orthonormalizes the data by implementing a Gram-Schmidt algorithm [arfken99]. This implementation follows the topology presented in Figure 2.64. The input signal $S_{in}(t_n)$ is

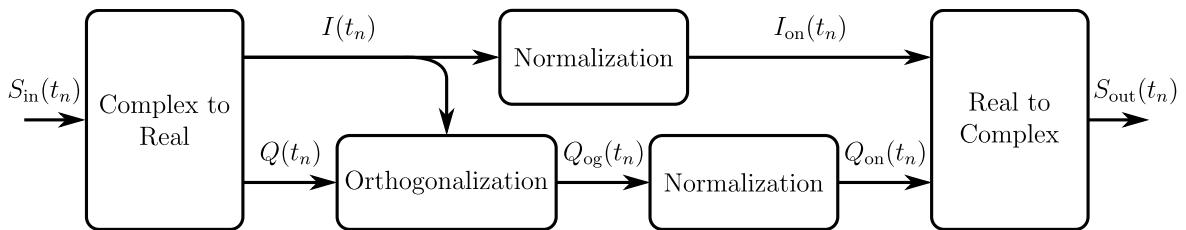


Figure 2.64: Block diagram representation of the Gram-Schmidt orthonormalization procedure.

separated into its in-phase and in-quadrature components, $I(t_n)$ and $Q(t_n)$, the amplitude of the in-phase component, P_I , is estimated as the average of its square, ie.:

$$P_I = \text{mean}(I^2(t_n)), \quad (2.106)$$

the in-phase signal is then normalized, ie.:

$$I_{on}(t_n) = \frac{I(t_n)}{\sqrt{P_I}}. \quad (2.107)$$

The in-quadrature component is then orthogonalized in relation to the in-phase component

$$Q_{\text{og}}(t_n) = Q(t_n) - \frac{I(t_n)\text{mean}(I(t_n)Q(t_n))}{P_I}, \quad (2.108)$$

which is then normalized in a manner similar to what was done for the in-phase component

$$P_Q = \text{mean}(Q_{\text{og}}^2(t_n)), \quad (2.109)$$

$$Q_{\text{on}}(t_n) = \frac{Q_{\text{og}}(t_n)}{\sqrt{P_Q}}. \quad (2.110)$$

Finally, the two orthonormalized components are combined to form the output signal

$$S_{\text{out}}(t_n) = I_{\text{on}}(t_n) + iQ_{\text{on}}(t_n) \quad (2.111)$$

Input Signals

Number: 1 **Type:** Complex signal

Output Signals

Number: 1

Type: Complex signal

Suggestions for future improvement

- Add the optimized recursive Gram-Schmidt algorithm as an option.

