NetXPTO - LinkPlanner

3 de Janeiro de 2018

Conteúdo

Capítulo 1

Introduction

LinkPlanner is devoted to the simulation of point-to-point links.

Capítulo 2

Simulator Structure

LinkPlanner is a signals open-source simulator.
The major entity is the system.
A system comprises a set of blocks.
The blocks interact with each other through signals.

2.1 System

2.2 Blocks

2.3 Signals

List of available signals:

- Signal

Capítulo 3

Development Cycle

The NetXPTO-LinkPlanner has been developed by several people using git as a version control system. The NetXPTO-LinkPlanner repository is located in the GitHub site http://github.com/netxpto/linkplanner. The more updated functional version of the software is in the branch master. Master should be considered a functional beta version of the software. Periodically new releases are delivered from the master branch under the branch name Release<Year><Month><Day>. The integration of the work of all people is performed by Armando Nolasco Pinto in the branch Develop. Each developer has is how branch with his/her name.

Capítulo 4

Visualizer

visualizer

Capítulo 5

Case Studies

5.1 QPSK Transmitter

2017-08-25, Review, Armando Nolasco Pinto

This system simulates a QPSK transmitter. A schematic representation of this system is shown in figure 5.1.

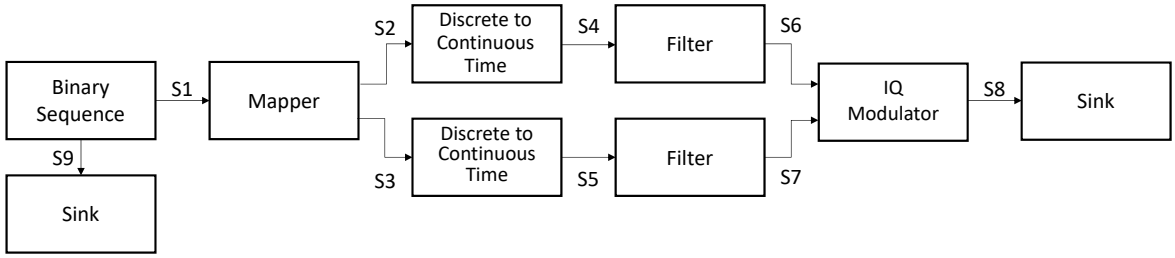

Figura 5.1: QPSK transmitter block diagram.

System Input Parameters

Parameter: *sourceMode*

Description: Specifies the operation mode of the binary source.

Accepted Values: PseudoRandom, Random, DeterministicAppendZeros, DeterministicCyclic.

Parameter: *patternLength*

Description: Specifies the pattern length used my the source in the PseudoRandom mode.

Accepted Values: Integer between 1 and 32.

Parameter: *bitStream*

Description: Specifies the bit stream generated by the source in the DeterministicCyclic and DeterministicAppendZeros mode.

Accepted Values: "XXX..", where X is 0 or 1.

Parameter: *bitPeriod*

Description: Specifies the bit period, i.e. the inverse of the bit-rate.

Accepted Values: Any positive real value.

Parameter: *iqAmplitudes*

Description: Specifies the IQ amplitudes.

Accepted Values: Any four par of real values, for instance { { 1,1 },{ -1,1 },{ -1,-1 },{ 1,-1 } }, the first value correspond to the "00", the second to the "01", the third to the "10"and the forth to the "11".

Parameter: *numberOfBits*

Description: Specifies the number of bits generated by the binary source.

Accepted Values: Any positive integer value.

Parameter: *numberOfSamplesPerSymbol*

Description: Specifies the number of samples per symbol.

Accepted Values: Any positive integer value.

Parameter: *rollOffFactor*

Description: Specifies the roll off factor in the raised-cosine filter.

Accepted Values: A real value between 0 and 1.

Parameter: *impulseResponseTimeLength*

Description: Specifies the impulse response window time width in symbol periods.

Accepted Values: Any positive integer value.

»»»> Romil

5.2 BER of BPSK with additive WGN

Student Name	:	Daniel Pereira (2017/09/01 -)
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 5.2).

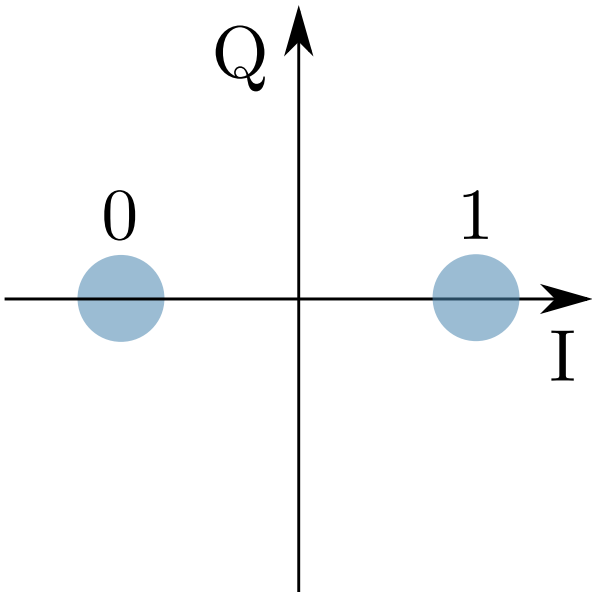

Figura 5.2: BPSK symbol constellation.

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 we know that its spectral density equals its variance. For the purpose of this work, additive WGN is used to model thermal noise typically observed in coherent 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.

5.2.1 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,

$$m_i = 2\sqrt{P_L P_S} G_{ele} \cos(\Delta\theta_i), \tag{5.1}$$

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 transimpedance amplifier 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 (5.1) can be reduced to,

$$m_i = (-1)^{i+1} 2\sqrt{P_L P_S} G_{ele},\ i = 0,\ 1. \tag{5.2}$$

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,

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m_i)^2}{2\sigma^2}}. \tag{5.3}$$

The BER is calculated in the following manner,

$$BER = \frac{1}{2}\int_0^{+\infty} f(x|\Delta\theta = \pi)\mathrm{d}x + \frac{1}{2}\int_{-\infty}^{0} f(x|\Delta\theta = 0)\mathrm{d}x, \tag{5.4}$$

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

$$BER = \int_0^{+\infty} f(x|\Delta\theta = \pi)\mathrm{d}x = \frac{1}{2}\mathrm{erfc}\left(\frac{-m_0}{\sqrt{2}\sigma}\right) \tag{5.5}$$

5.2.2 Simulation Analysis

A diagram of the system being simulated is presented in the Figure 5.3. 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.

Required files

Header Files

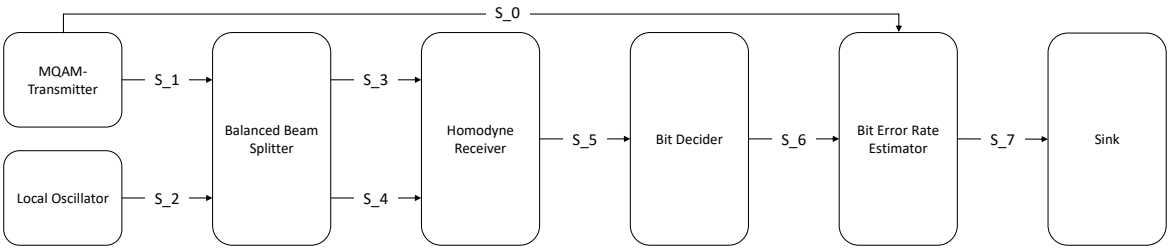

Figura 5.3: Overview of the BPSK system being simulated.

File	Description
add.h	Adds the two input signals and outputs the result
balanced_beam_splitter.h	Mixes the two optical signals set at it's input.
binary_source.h	Generates a sequence of binary values (1 or 0)
bit_decider.h	Decodes the input signal into a binary string.
bit_error_rate.h	Calculates the bit error rate of the decoded string.
discrete_to_continuous_time.h	Converts a signal discrete in time to a signal continuous in time.
netxpto.h	Generic purpose simulator definitions.
m_qam_mapper.h	Maps the binary sequence into the coded constellation.
m_qam_transmitter.h	Generates the signal with coded constellation.
local_oscillator.h	Generates a continuous optical signal with set power and phase.
i_homodyne_reciever.h	Performs coherent detection on the input signal.
ideal_amplififer.h	Multiplies the input signal by a user defined gain factor and outputs the result.
iq_modulator.h	Maps two real valued signal into a complex optical bandpass signal.
photodiode.h	Converts complex optical bandpass signals into real value electrical signals.
pulse_shaper.h	Simulates the impulse response of a circuit.
sampler.h	Samples the input signal at a user defined frequency.
sink.h	Closes any unused signals.
super_block_interface.h	Allows superblocks to output multiple signals.
white_noise.h	Generates white gaussian noise with a user defined spectral density.

Source Files

File	Description
add.cpp	Adds the two input signals and outputs the result
balanced_beam_splitter.cpp	Mixes the two optical signals set at it's input.
binary_source.cpp	Generates a sequence of binary values (1 or 0)
bit_decider.cpp	Decodes the input signal into a binary string.
bit_error_rate.cpp	Calculates the bit error rate of the decoded string.
discrete_to_continuous_time.cpp	Converts a signal discrete in time to a signal continuous in time.
netxpto.cpp	Generic purpose simulator definitions.
m_qam_mapper.cpp	Maps the binary sequence into the coded constellation.
m_qam_transmitter.cpp	Generates the signal with coded constellation.
local_oscillator.cpp	Generates a continuous optical signal with set power and phase.
i_homodyne_reciever.cpp	Performs coherent detection on the input signal.
ideal_amplififer.cpp	Multiplies the input signal by a user defined gain factor and outputs the result.
iq_modulator.cpp	Maps two real valued signal into a complex optical bandpass signal.
photodiode.cpp	Converts complex optical bandpass signals into real value electrical signals.
pulse_shaper.cpp	Simulates the impulse response of a circuit.
sampler.cpp	Samples the input signal at a user defined frequency.
sink.cpp	Closes any unused signals.
super_block_interface.cpp	Allows superblocks to output multiple signals.
white_noise.cpp	Generates white gaussian noise with a user defined spectral density.

System Input Parameters

This system takes into account the following input parameters:

System Parameters	**Description**	**Simulation Value**
numberOfBitsGenerated	Gives the number of bits to be simulated	40000
bitPeriod	Sets the time between adjacent bits	20×10^{-12}
samplesPerSymbol	Establishes the number of samples each bit in the string is given	16
pLength	PRBS pattern length	5
iqAmplitudesValues	Sets the state constellation	$\{\,\{-1,\ 0\}\,,\ \{1,\ 0\}\,\}$
outOpticalPower_dBm	Sets the optical power, in units of dBm, at the transmitter output	Variable
loOutOpticalPower_dBm	Sets the optical power, in units of dBm, of the local oscillator used in the homodyne detector	0
localOscillatorPhase	Sets the initial phase of the local oscillator used in the homodyne detector	0
transferMatrix	Sets the transfer matrix of the beam splitter used in the homodyne detector	$\{\,\{\frac{1}{\sqrt{2}},\ \frac{1}{\sqrt{2}},\ \frac{1}{\sqrt{2}},\ \frac{-1}{\sqrt{2}}\}\,\}$
responsivity	Sets the responsivity of the photodiodes used in the homodyne detector	1
amplification	Sets the amplification of the trans-impedance amplifier used in the homodyne detector	10^3
noiseSpectralDensity	Sets the spectral density of the gaussian thermal noise added in the homodyne detector	$5 \times 10^{-4}\sqrt{2}\ \mathrm{V}^2$
confidence	Sets the confidence interval for the calculated QBER	0.95
midReportSize	Sets the number of bits between generated QBER mid-reports	0

Inputs

This system takes no inputs.

Outputs

This system outputs the following objects:

Parameter: Signals:

Description: Initial Binary String; (S_0)

Description: Optical Signal with coded Binary String; (S_1)

Description: Local Oscillator Optical Signal; (S_2)

Description: Beam Splitter Outputs; (S_3, S_4)

Description: Homodyne Detector Electrical Output; (S_5)

Description: Decoded Binary String; (S_6)

Description: BER result String; (S_7)

Parameter: Other:

Description: Bit Error Rate report in the form of a .txt file. (BER.txt)

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 (5.5), 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 $5 \times 10^{-4}\sqrt{2}\ \mathrm{V}^2$ [1].

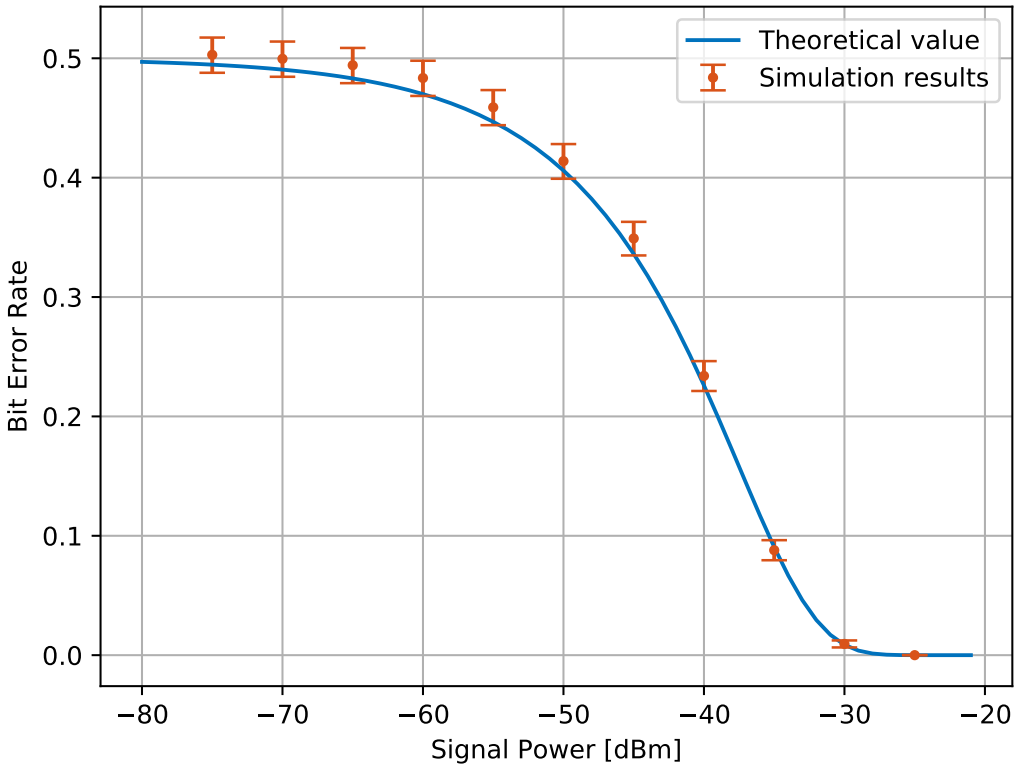

Figura 5.4: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 6.2

Bibliografia

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

[2] Álvaro J Almeida, Nelson J Muga, Nuno A Silva, João M Prata, Paulo S André, and Armando N Pinto. Continuous control of random polarization rotations for quantum communications. *Journal of Lightwave Technology*, 34(16):3914–3922, 2016.

5.3 M-QAM Transmission System

Student Name	:	Ana Luisa Carvalho
Goal	:	M-QAM system implementation with BER measurement and comparison with theoritical values.

5.3.1 Introduction

M-QAM, which stands for Quadrature Amplitude Modulation, is a modulation scheme that takes advantage of two carriers (usually sinusoidal waves) with a phase difference of $\frac{\pi}{2}$. The resultant output consists of a signal with both amplitude and phase variations. The two carriers, refered to as I (In-phase) and Q (Quadrature), can be represented as

$$I = A\cos(\phi) \tag{5.6}$$
$$Q = A\sin(\phi) \tag{5.7}$$

which means that any sinusoidal wave can be decomposed in its I and Q components:

$$A\cos(\omega\, t + \phi) = A\left(\cos(\omega\, t)\cos(\phi) - \sin(\omega\, t)\sin(\phi)\right) \tag{5.8}$$
$$= I\cos(\omega\, t) - Q\sin(\omega\, t), \tag{5.9}$$

where we have used the expression for the cossine of a sum and the definitions of I and Q.

For M$= 4$ the symbol constellation is shown figure **??**.

M can take several values: $2, 4, 16, 32, ...$ The first two correspond to BPSK and QPSK modulation, respectively.

5.3.2 Bit Error Rate for 4-QAM with Additive White Gaussian Noise (AWGN)

When demodulating a signal it is necessary to associate the received signal to the corresponding signal. The existence of noise in the channel means that we can only compute the probability that a given signal corresponds to a certain carrier and that's why we need to define the Bit Error Rate (BER). Using

$$P_i f(s|c_i) > P_j f(s|c_j), \qquad i \neq j \tag{5.10}$$

where $f(s|c_i)$ stands for the probability of detecting the signal s given that c_i was emmited. This inequallity can be rewritten in the following way

$$P(c_i|s) > P(c_j|s) \tag{5.11}$$

where $P(c_i|s)$ and $P(c_j|s)$ are called *a posteriori* probabilities and represent the probability that c_i or c_j were transmitted given that s was received. In terms of the systems this simply means that we should select the signal most likely to have been transmitted.

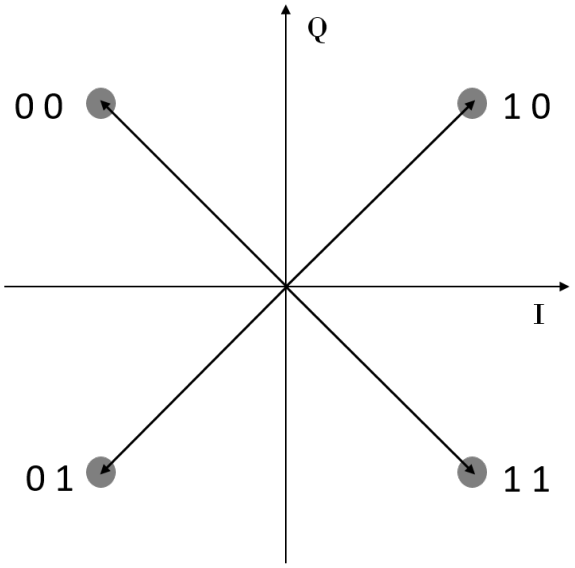

Figura 5.5: 4-QAM symbol constellation

In the case of additive white gaussian noise the f function is simply given by

$$f(s|c_i) = \frac{e^{-x^2/n_0}}{(\pi n_0)^{N/2}} \tag{5.12}$$

where x is the Euclidean distance in the I-Q plane between the signal received and carrier i and N is the number of noise samples.

When using 4-QAM modulation all points are at an equal distance from the origin (in the I-Q plane) so they all have the same energy given by

$$E = \frac{d^2}{2} \tag{5.13}$$

where d is the side of the square formed bye the constellation points.

The probability that a given signal is identified correctly is given by

$$P_c = r^2 \tag{5.14}$$

where $n_0/2$ is the noise variance for AWGN and

$$r = \int_{-d/2}^{\infty} \frac{e^{-x^2/n_0}}{\sqrt{\pi n_0}} dx. \tag{5.15}$$

The error probability, P_e, given by $1 - P_c$ is given by

$$P_e = erfc\sqrt{\frac{E}{2n_0}}. \tag{5.16}$$

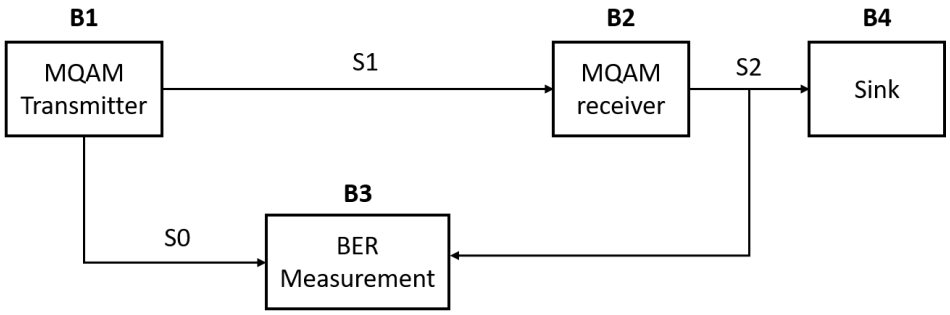

Figura 5.6: Schematic representation of the MQAM system.

5.3.3 Simulation setup

The M-QAM system transmission system is a complex block of code that simulates the modulation, transmission and demodulation of an optical signal using M-QAM modulation. It is composed of four blocks: a transmitter, a receiver, a sink and a block that performs a Bit Error Rate (BER) measurement.

Current state: The system currently being implement is a QPSK system (M=4).

Future work: Extend this block to include other values of M.

5.3.4 Functional description

The schematic representation of the system is presented in figure 5.6.

A complete description of the M-QAM transmitter and M-QAM receiver blocks can be found in the *Library* chapter of this document as well as a detailed description of the independent blocks that compose these blocks.

The M-QAM transmitter is a complex block that generates one or two optical signals by enconding a binary string using M-QAM modulation. It also outputs a binary signal that is used to perform a BER measurement.

The M-QAM receiver is a homodyne receiver. It is a complex block that accepts one input optical signal and outputs a binary signal. It performs the M-QAM demodulation of the input signal by combining the optical signal with a local oscillator.

The demodulated optical signal is compared to the one produced by the transmitter in order to estimate the Bit Error Rate (BER).

The files corresponding to each of the system's blocks are summarized in table **??**. Along with the library and corresponding source files these allow for the full operation of the M-QAM system described here.

Tabela 5.1: Main system files

System blocks	**cpp file**	**include file**
Main	m_qam_system_sdf.cpp	—
M-QAM transmitter	m_qam_transmitter.cpp	m_qam_transmitter.h
M-QAM receiver	homodyne_receiver.cpp	homodyne_receiver.h
Sink	sink.cpp	sink.h
BER estimator	bit_error_rate.cpp	bit_error_rate.h

5.3.5 Input Parameters

5.3.6 Output Parameters

As output this block

5.3.7 BER measurement

Tabela 5.2: Input parameters

Parameter	**Type**	**Description**
numberOfBitsGenerated	t_integer	Determines the number of bits to be generated by the binary source
samplesPerSymbol	t_integer	
prbsPatternLength	int	Determines the length of the pseudorandom sequence pattern (used only when the binary source is operated in *PseudoRandom* mode)
bitPeriod	t_real	Temporal interval occupied by one bit
rollOffFactor	t_real	
signalOutputPower_dBm	t_real	Determines the power of the output optical signal in dBm
numberOfBitsReceived	int	
iqAmplitudeValues	vector<t_iqValues>	Determines the constellation used to encode the signal in IQ space
symbolPeriod	double	Given by bitPeriod/samplesPerSymbol
localOscillatorPower_dBm	t_real	Power of the local oscillator
responsivity	t_real	Responsivity of the photodiodes (1 corresponds to having all optical power transformed into electrical current)
amplification	t_real	??
noiseAmplitude	t_real	??
samplesToSkip	t_integer	Number of samples to be skipped by the *sampler* block
confidence	t_real	Determines the confidence limits for the BER estimation
midReportSize	t_integer	
bufferLength	t_integer	Corresponds to the number of samples that can be processed in each run of the system

5.4 BB84 with Discrete Variables

Students Name	:	Mariana Ramos and Kevin Filipe
~~**Starting Date**~~	:	~~November 7, 2017~~
Goal	:	BB84 implementation with discrete variables.

BB84 is a key distribution protocol which involves three parties, Alice, Bob and Eve. Alice and Bob exchange information between each other by using a quantum channel and a classical channel. The main goal is continuously build keys only known by Alice and Bob, and guarantee that eavesdropper, Eve, does not gain any information about the keys.

5.4.1 Protocol Analysis

Students Name	:	Kevin Filipe (7/11/2017)
Goal	:	BB84 - Protocol Description

BB84 protocol was created by Charles Bennett and Gilles Brassard in 1984 [1]. It involves two parties, Alice and Bob, sharing keys through a quantum channel in which could be accessed by a eavesdropper, Eve. A basic model is depicted in figure 5.7.

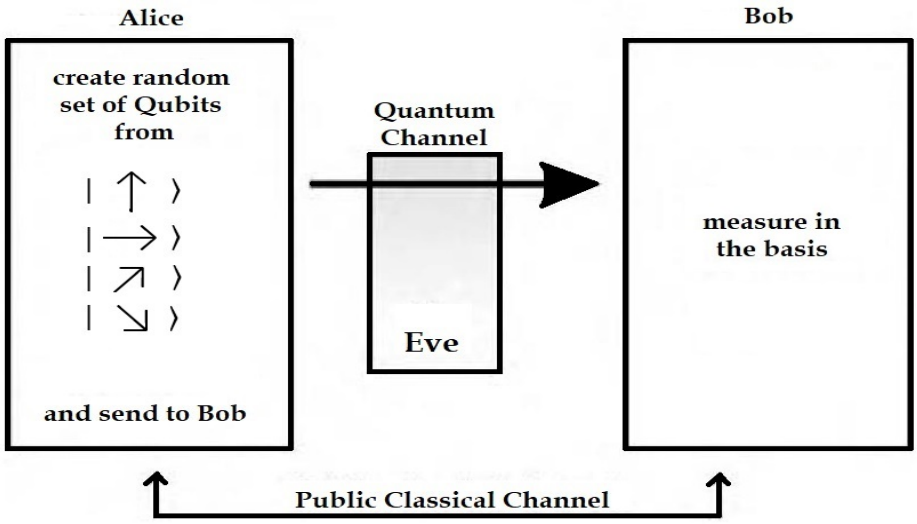

Figura 5.7: Basic QKD Model. Alice and Bob are connected by 2 communication channels, quantum and classical, with an eavesdropper, Eve, in the quantum communication channel (figure adapted from [3]).

We are going to analyse the BB84 protocol with bit encoding into photon state polarization. Two non-orthogonal basis are used to encode the information, the rectilinear and diagonal basis, + and x respectively. The following table shows this bit encoding.

Bit	***Rectilinear Basis,+***	***Diagonal Basis,×***
0	0º	-45º
1	90º	45º

The protocol is implemented with the following steps:

1. Alice generates two random bit strings. The random string , R_{A1}, corresponds to the data to be encoded into photon state polarization. R_{A2} is a random string in which 0 and 1 corresponds to the rectilinear, +, and diagonal, $\times$, respectively.

$$R_{A1} = \{0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1\}$$

$$R_{A2} = \{0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0\} \tag{5.17}$$
$$= \{+, +, \times, +, \times, \times, \times, +, \times, \times, \times, +, \times, +, +, +, \times, +, \times, +\} \tag{5.18}$$

2. Alice transmits a train of photons, S_{AB}, obtained by encoding the bits, R_{A1} with the respective photon polarization state R_{A2}.

$$S_{AB} = \{\rightarrow, \uparrow, \searrow, \rightarrow, \searrow, \nearrow, \nearrow, \uparrow, \searrow, \nearrow, \searrow, \uparrow, \searrow, \rightarrow, \rightarrow, \uparrow, \nearrow, \rightarrow, \nearrow, \uparrow\}.$$

3. Bob generates a random string, R_B, to receive the photon trains with the correspondent basis.

$$R_B = \{0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0\} \tag{5.19}$$
$$= \{+, \times, \times, \times, +, \times, +, +, \times, \times, +, +, \times, \times, +, +, \times, \times, +, +\}. \tag{5.20}$$

4. Bob performs the incoming photon states measurement, M_B, with its generated random basis, R_B. If the two photon detectors don't click, means the bit was lost during transference due to attenuation. If both photon detectors click, a false positive was detected. In the measurements, M_B, the no-click in both detectors is represented by a -1 and the false positives to -2. The measurements done in rectilinear or diagonal basis are represented by 0 and 1, respectively. This is represented 5.8

$$M_B = \{0, 1, 1, 1, -1, 1, 0, 0, -2, 1, 0, 0, -2, 1, 0, 0, 1, -1, 0, 0\}$$

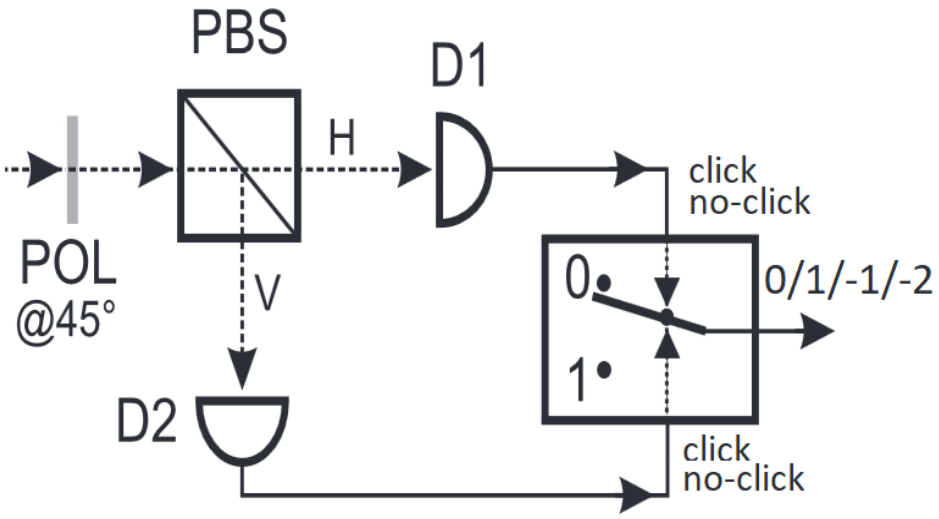

Figura 5.8: Photon Detection block with false-positives, -2, and attenuation, -1, detection depending on D1 and D2 output.

Esta parte estÃ¡ muito confusa, preciso que o professor me explique melhor The second phase, uses the classical communication channel:

1. After the measurement, Bob sends to Alice the values of M_B
2. Alice remove the bits from R_{A2} corresponding to -1 and -2, performs a negated XOR, generating the bit sequence B_B with correspondent key as K_A.

$$K_A = \{0, 1, 0, 1, 0, 1, 0, 1, 0, 1\}.$$

R_{A2}	0	0	1	0	1	1	0	1	1	0	0	0	0	1	1	0
R_B	0	1	1	1	1	0	0	1	0	0	1	0	0	1	0	0
B_A	1	0	1	0	1	0	1	1	0	1	0	1	1	1	0	1

3. Perform a scrambling over B_B using a known algorithm by Alice and Bob to accomplish distribute the error over all key to later and avoid error burst. Later, this will permit to have a good bit error rate estimation [4]. As a example, the even positions of the B_B will be shifted 2 positions.

$Positions$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B_A	0	1	1	1	1	0	0	1	0	0	1	0	0	1	0	0
P_A	0	1	1	1	0	0	0	1	1	0	0	0	0	1	0	0

4. For Bob to also have the same information, he will perform the same steps, 2 and 3 as Alice and the Key will be deduced. K_{AB}

$$K_{AB} = \{0, 1, 0, 1, 0, 1, 0, 1, 0, 1\}.$$

To determine the Quantum Bit Error Rate (QBER), it is necessary some input parameters such as:

1. qLB - QBER lower bound.
2. qUB - QBER upper bound.
3. qLim - acceptable QBER limit.

Then to verify if the channel is reliable or not, the flowchart presented in figure 5.9.

1. Bob will reveals a bit sequence from the deduced key, K_{AB} to Alice.
2. Alice then returns to Bob the estimated QBER value, mQBER, with a confidence interval, [qLB, qUB] using the using the equations in the Bit Error Rate section, but applied to this protocol
3. To check if the channel is compromised or not it is necessary to check if the QBER limit is higher than the QBER upper bound. If QBER limit is between the QBER lower and upper bound it is necessary to reveal more bits from the key. Otherwise the channel is compromised and the key determination process needs to restart.

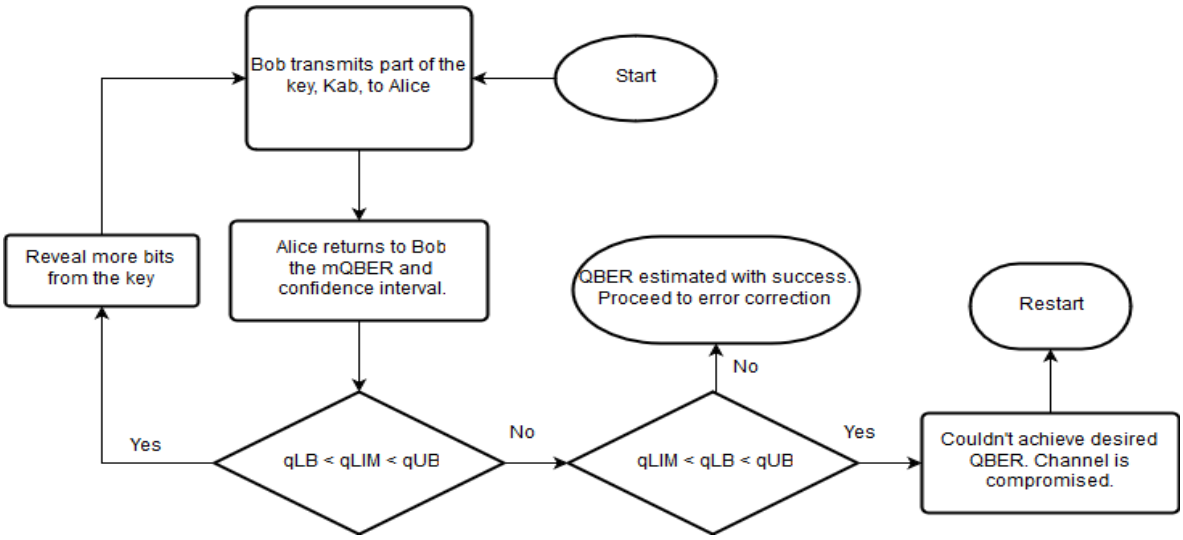

Figura 5.9: Flowchart to determine if the channel is reliable or not.

5.4.2 Simulation Analysis

Students Name	:	Mariana Ramos
Starting Date	:	November 7, 2017
Goal	:	Perform a simulation of the setup presented bellow in order to implement BB84 communication protocol.

In this sub section the simulation setup implementation will be described in order to implement the BB84 protocol. In figure 5.10 a top level diagram is presented. Then it will be presented the block diagram of the transmitter block (Alice) in figure 5.11, the receiver block (Bob) in figure 5.12 and finally the eavesdropper block (Eve) in figure 5.13.

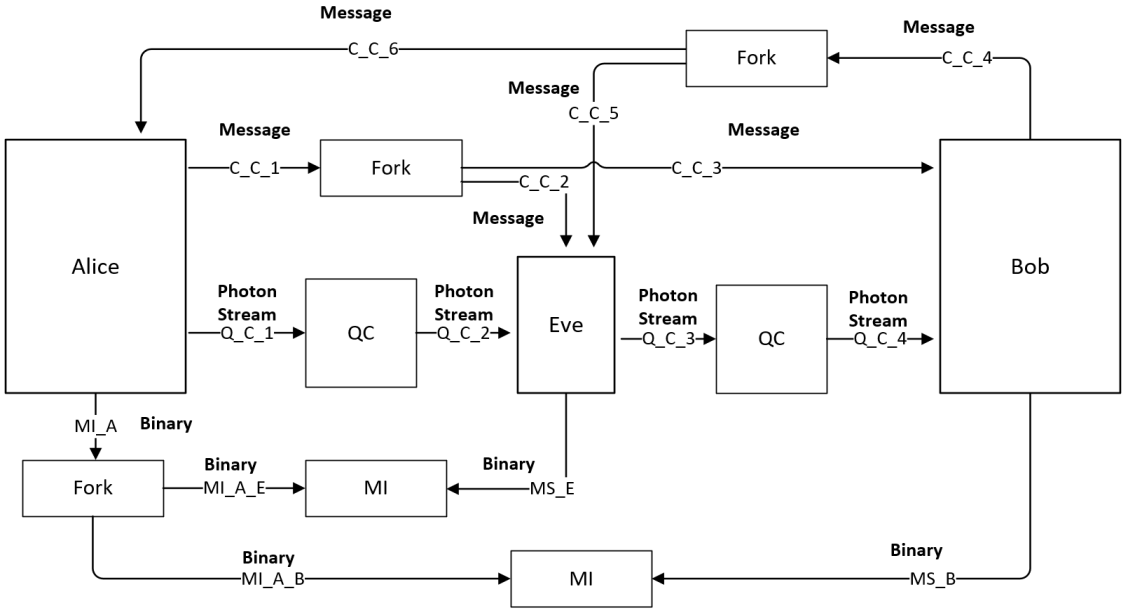

Figura 5.10: Simulation diagram at a top level

Figure 5.10 presents the top level diagram of our simulation. The setup contains three parties, Alice, Eve and Bob where the communication between them is done throughout two classical and one quantum channel. In the middle of the classical channel there is a Fork's diagram which has one input and two outputs. In the case of the classical channel **C_C_4** which has the information sent by Bob, the fork's block enables Alice and Eve have access to it. In the quantum communication, the information sent by Alice can be intercepted by Eve and changed by her, or can follow directly to Bob as we can see later in figure 5.13. Furthermore, for mutual information calculation there must be two blocks **MI**, one to calculate the mutual information between Alice and Eve, and other to calculate the mutual information between Alice and Bob.

In figure 5.11 one can observe a block diagram of the simulation at Alice's side. As it

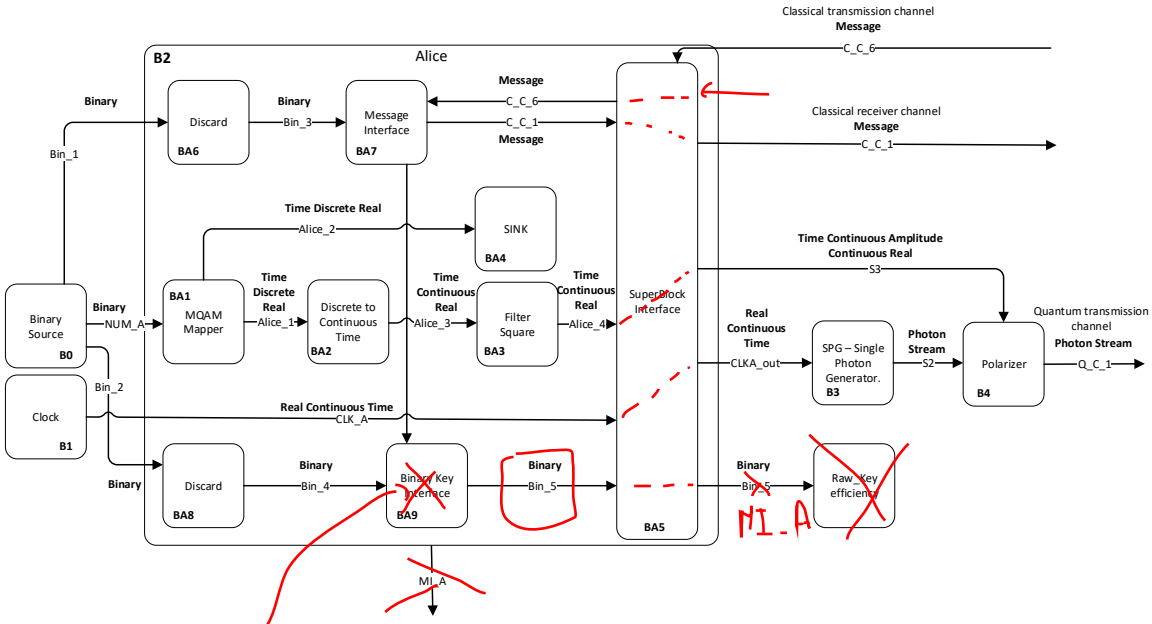

Figura 5.11: Simulation diagram at Alice's side

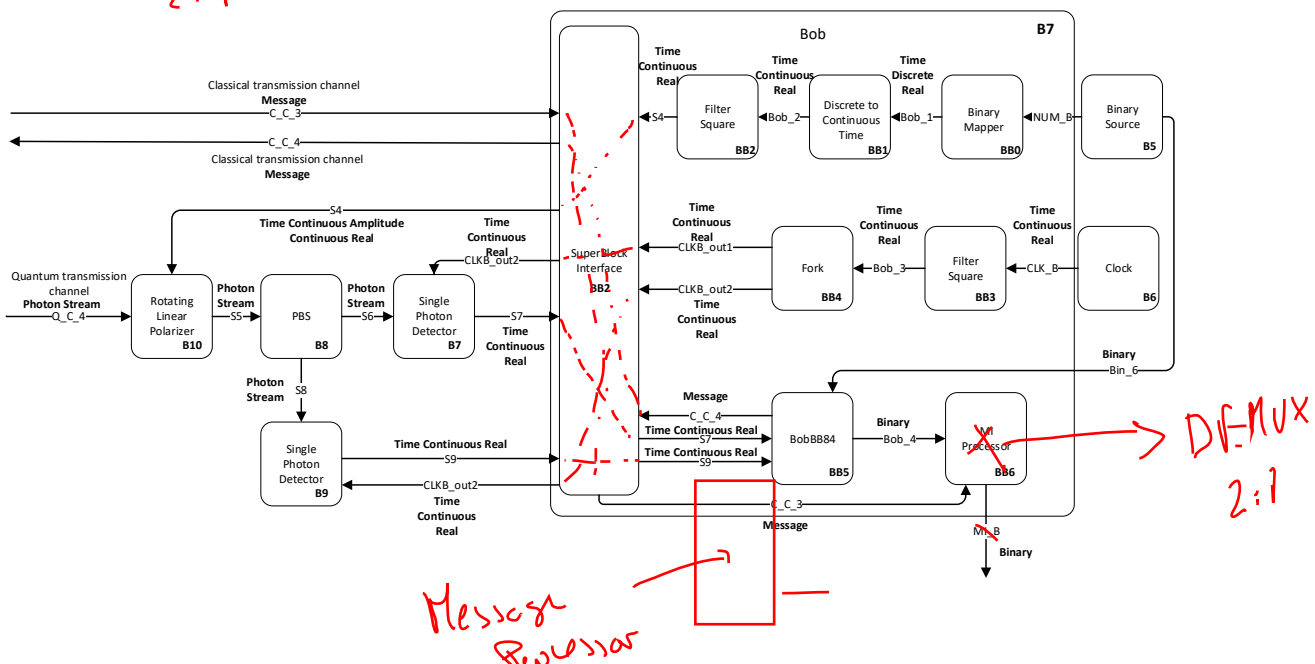

Figura 5.12: Simulation diagram at Bob's side

is shown in the figure, Alice must have one block for random number generation which is responsible for basis generation to polarize the photons, and for key random generation

in order to have a random state to encode each photon. Furthermore, she has a Processor block for all logical operations: array analysis, random number generation requests, and others. This block also receives the information from Bob after it has passed through a fork's block. In addition, it is responsible for set the initial length l of the first array of photons which will send to Bob. This block also must be responsible for send classical information to Bob. Finally, Processor block will also send a real continuous time signal to single photon generator, in order to generate photons according to this signal, and finally this block also sends to the polarizer a real discrete signal in order to inform the polarizer which basis it should use. Therefore, she has two more blocks for quantum tasks: the single photon generator and the polarizer block which is responsible to encode the photons generated from the previous block and send them throughout a quantum channel from Alice to Bob.

Finally, Alice's processor has an output to Mutual Information top level block, Ms_A.

In figure 5.12 one can observe a block diagram of the simulation at Bob's side. From this side, Bob has one block for Random Number Generation which is responsible for randomly generate basis values which Bob will use to measure the photons sent by Alice throughout the quantum channel. Like Alice, Bob has a Processor block responsible for all logical tasks, analysing functions, requests for random number generator block, etc. Additionally, it receives information from Alice throughout a classical channel after passed through a fork's block and a quantum channel. However, Bob only sends information to Alice throughout a classical channel. Furthermore, Bob has one more block for single photon detection which receives from processor block a real discrete time signal, in order to obtain the basis it should use to measure the photons.

Finally, Bob's processor has an output to Mutual Information top level block, Ms_B.

Figure 5.13 presents the Eve's side diagram. Eve's processor has two receiver classical signals, one from Alice (**C_C_2**) and other from Bob (**C_C_5**). About quantum channel, Eve received a quantum message from Alice through the channel **Q_C_1** and depends on her decision the photon can follows directly to Bob or the photon's state can be changed by her. In this case, the photon is received by a block similar to Bob's diagram 5.12 and this block sends a message to Eve's processor in order to reveal the measurement result. After that, Eve's processor sends a message to Alice's diagram similar to figure 5.11 and this block is responsible for encode the photon in a new state. Now, the changed photon is sent to Bob.

In addition, Eve's diagram has one more output Ms_E which is a message sent to the mutual information block as an input parameter.

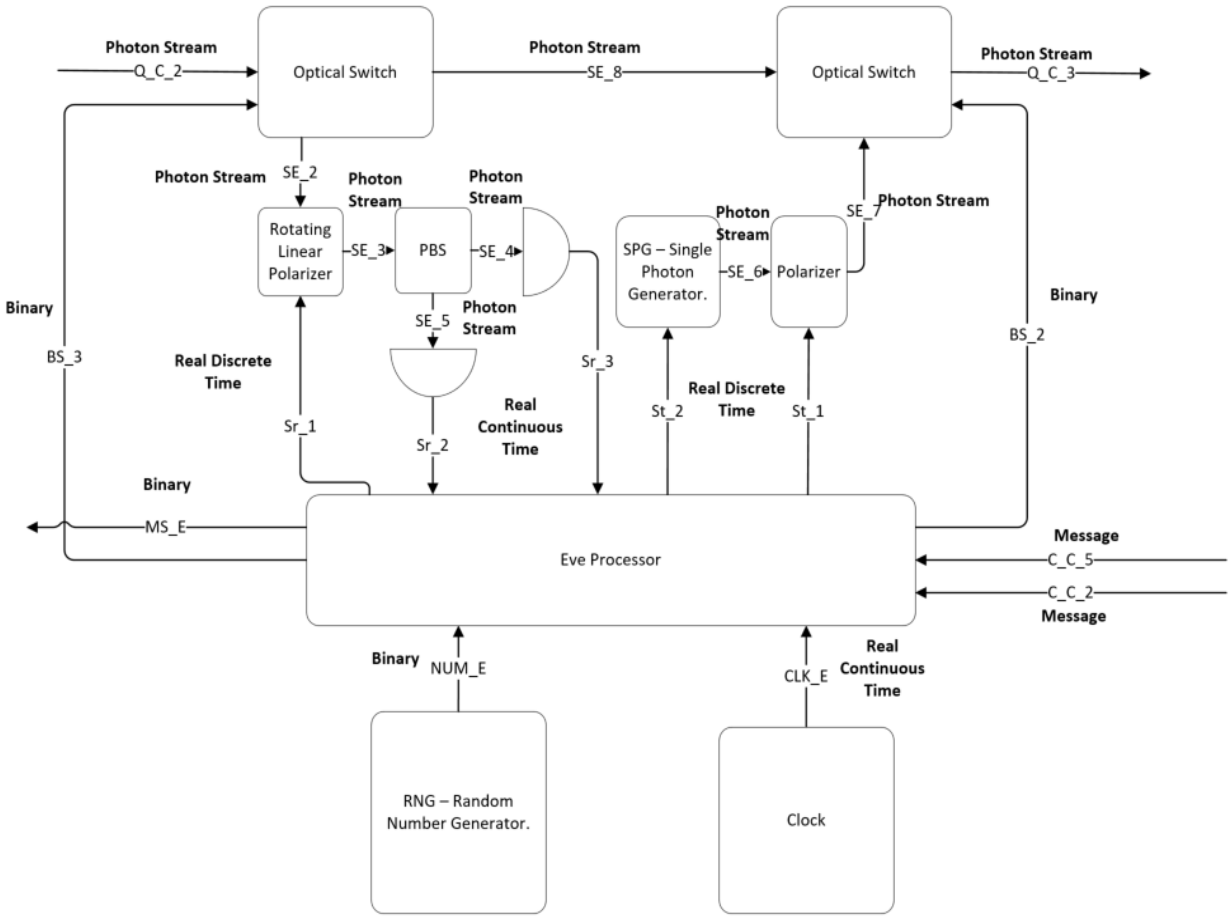

Figura 5.13: Simulation diagram at Eve's side

Tabela 5.3: System Signals

Signal name	Signal type	Status
NUM_A	Binary	
MI_A	Binary	
CLK_A	TimeContinuousAmplitudeContinuous	
CLK_A_1	TimeContinuousAmplitudeContinuous	
S2	PhotonStreamXY	
S3	TimeContinuousAmplitudeDiscreteReal	
C_C_1	Messages	
C_C_6	Messages	
Q_C_1	PhotonStreamXY	

Table 5.6 presents the system signals as well as them type.

Tabela 5.4: System Input Parameters

Parameter	**Default Value**	**Description**
RateOfPhotons	1K	
vector<t_iqValues> iqAmplitudeValues	{-45.0,0.0},{0.0,0.0},{45.0,0.0},{90.0,0.0}	

Tabela 5.5: Header Files

File name	**Description**	**Status**
netxpto.h		✓
alice_qkd.h		Working on
binary_source.h		✓
bob_qkd.h		Missing
clock_20171219.h		✓
discrete_to_continuous_time.h		✓
eve_qkd.h		Missing
m_qam_mapper.h		✓
optical_switch.h		Missing
polarization_beam_splitter.h		Working on
polarizer.h		Working on
pulse_shaper.h		✓
rotator_linear_polarizer.h		Working on
single_photon_detector.h		Missing
single_photon_source_20171218.h		✓
sink.h		✓
super_block_interface.h		✓

Tabela 5.6: Source Files

File name	Description	Status
netxpto.cpp		✓
bb84_with_discrete_variables.cpp		Working on
alice_qkd.cpp		Working on
binary_source.cpp		✓
bob_qkd.cpp		Missing
clock_20171219.cpp		✓
discrete_to_continuous_time.cpp		✓
eve_qkd.cpp		Missing
m_qam_mapper.cpp		✓
optical_switch.cpp		Missing
polarization_beam_splitter.cpp		Working on
polarizer.cpp		Working on
pulse_shaper.cpp		✓
rotator_linear_polarizer.cpp		Working on
single_photon_detector.cpp		Missing
single_photon_source_20171218.cpp		✓
sink.cpp		✓
super_block_interface.cpp		✓

5.4.3 Open Issues

There still are some open issues in simulation code.

One of them was detected in block **single_photon_source_20171218.cpp**. This block should assume each sample with 4 real values, since it writes two complex values each time the block runs, i.e each *bufferput()* should write an array of two complex values in outputSignal, outputSignals[0]->bufferPut(valueXY), where *t_complex_xy valueXY = {valueX, valueY}* and *t_complex valueX = (realValue_1,realValue_2)*. This way, independently of the number of samples these four values should always be written. However, if we chose a number of samples which is not divisible by 4, the four numbers are not written in the last "sample"and the array data for X's values and Y's values have different sizes which is wrong. For example, if we chose 10 samples to acquire, the last values correspond to X's values instead of Y's values and the first array data is longer than the other.

Bibliografia

[1] Bennett, C. H. and Brassard, G. Quantum Cryptography: Public key distribution and coin tossing. International Conference on Computers, Systems and Signal Processing, Bangalore, India, 10-12 December 1984, pp. 175-179.

[2] Mart Haitjema, A Survey of the Prominent Quantum Key Distribution Protocols

[3] Christopher Gerry, Peter Knight, "Introductory Quantum Optics"Cambridge University Press, 2005

[4] Varadarajan, S., Ngo, H. Q., & Srivastava, J. (n.d.). An Adaptive , Perception-Driven Error Spreading Scheme in Continuous Media Streaming.

5.5 Radio Over Fiber Transmission System

Student Name	:	Celestino Martins
Starting Date	:	September 25, 2017
Goal	:	Simulation of Radio over fiber Transmission considering the uplink of base station cooperation systems.

Radio over fiber (RoF) technology comprises the transmission over fiber technology, where radio signal is modulated onto optical carrier and transmitted over an optical fiber link to provide a simple antenna front ends with increased capacity and broadband wireless services. In this network a central processing units (CPU) is connected to numerous base stations (BSs) via optic fibers. That means, RoF networks use optic fiber links to distribute radio frequency (RF) signals between the CPU and BSs. The downlink RF signals are distributed from a CPU to many BSs through the fibres, while the uplink signals received at BSs are sent back to the CPU for any signal processing. Figure 5.14 shows a general RoF architecture, where the wireless signals are transported over the optical fiber between a CPU and a set of base stations before being radiated through the air. RoF transmission systems are usually classified into two main categories, depending on the frequency range of the radio signal to be transported: i) RF-over-Fiber; ii) intermediate frequency (IF)-over-Fiber.

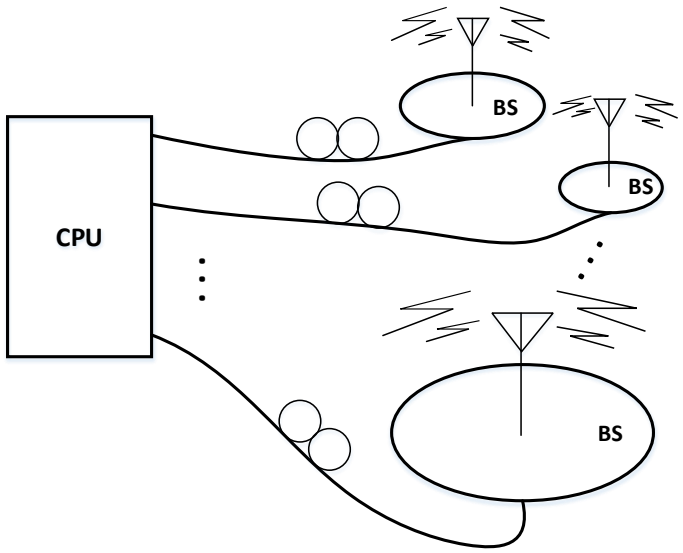

Figura 5.14: Schematic showing the concept of a centralized CPU architecture for future integrated optical wireless networks based on RoF.

(i) In RF-over-Fiber architecture, a data-carrying RF (Radio Frequency) signal with a high frequency (usually greater than 10 GHz) is imposed on a lightwave signal before being transported over the optical link. Therefore, wireless signals are optically distributed to base stations directly at high frequencies and converted from the optical to electrical

domain at the base stations before being amplified and radiated by an antenna. As a result, no frequency up/down conversion is required at the various base stations, thereby resulting in simple and rather cost-effective implementation is enabled at the base stations.

(ii) In IF-over-Fiber architecture, an IF (Intermediate Frequency) radio signal with a lower frequency (less than 10 GHz) is used for modulating light before being transported over the optical link. Therefore, before radiation through the air, the signal must be up-converted to RF at the base station.

In addition, the RoF technology can be implemented as analog RoF or digital RoF:

(i) In analog RoF technology, the analog signal is transmitted over the optical fiber, being either RF signal, IF signal or baseband BB signal. In the optical transmitter, the RF/IF/BB signal is modulated onto the optical carrier by either using direct or external modulation of the laser. In this case, the signal distribution through RoF has the advantage of simplified BS design, however it is susceptible to fiber chromatic dispersion and nonlinearity generated by optical devices.

(ii) In the digitized RoF the wireless carrier RF signal is first digitized prior to transport over the optical link. The digitalization of an RF signal produces a sampled digital signal in a serial form that can be directly modulated on an optical carrier, transmitted over the fiber optic link, and then detected like any other digital information. Modulation of the digital signal onto an optical carrier minimizes the nonlinear effects originating from the optical-to-electrical conversion function presented on analog RoF. In order to use not so high sample rates at the ADC/DAC components generally the bandpass sampling technique i applied to the RF signal.

5.5.1 Simulation

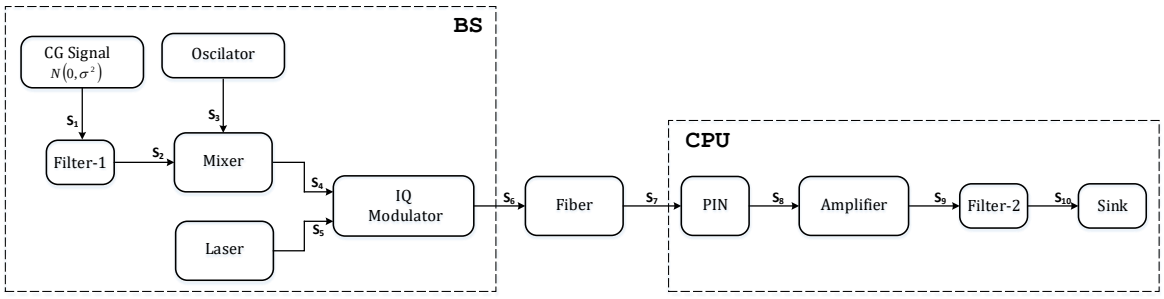

Figura 5.15: Simulation setup for the uplink of RoF transmission system. Complex Gaussian (CG);

Figure 5.15 depicts the simulation setup for an uplink of RoF, providing the connection between BS to CPU. At BS, we model the RF signal received from a mobile terminal, as zero

mean complex gaussian (CG) signal with a given bandwidth imposed by the low pass filter, Filter-1. The generated baseband signal is then up-converted to RF carrier frequency by utilizing an oscillator and a mixer. In this simulation we consider the RF carrier frequency between 2 to 5 GHz, according to the 5G technologies specifications. The generated RF passband signal modulates optical carrier by utilizing a laser and a mach-zehnder modulator (MZM). The optical signal is then transmitted to CPU using optical fiber. In CPU, the optical signal is detected by a PIN, amplified and followed with an electrical filter. After these operations, digital signal processing techniques can be applied to recover the transmitted signal.

Tabela 5.7: System Input Parameters.

Parameter	Default Value	Description
sourceMode		
symbolPeriod		
samplePeriod		
numberOfSamplesPerSymbol		
filterType1		
rollOffFactor		
filterType2		
outputOpticalPower		
outputOpticalWavelength		
rfCenterFrequency		
fiberAttenuation		

Tabela 5.8: Header Files for RoF Transmission System.

File Name	Description	Status
complex_gaussian_signal.h		
pulse_shaper.h		✓
local_oscillator.h		✓
mixer.h		
cw_laser.h		
iq_modulator.h		✓
fiber.h		
pin.h		
amplifier.h		
filter_rx.h		
sink.h		✓
netxpto.h		✓

5.5.2 Experimental

Tabela 5.9: Source Files for RoF Transmission System.

File Name	Description	Status
complex_gaussian_signal.cpp		
pulse_shaper.cpp		✓
local_oscillator.cpp		✓
mixer.cpp		
cw_laser.cpp		
iq_modulator.cpp		✓
fiber.cpp		
pin.cpp		
amplifier.cpp		
filter_rx.cpp		
sink.cpp		✓
netxpto.cpp		✓

Capítulo 6

Library

6.1 Add

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)

6.2 Bit Error Rate

Header File	:	bit_error_rate.h
Source File	:	bit_error_rate.cpp
Version	:	20171810 (Responsible: Daniel Pereira)

Input Parameters

Parameter: setConfidence **Parameter:** setMidReportSize

Functional Description

This block accepts two binary strings and outputs a binary string, outputting a 1 if the two input samples are equal to each other and 0 if not. This block also outputs *.txt* files with a report of the estimated Bit Error Rate (BER), $\widehat{\text{BER}}$ as well as the estimated confidence bounds for a given probability α.

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

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \tag{6.1}$$

The upper and lower bounds, BER_{UB} and BER_{LB} respectively, are calculated using the Clopper-Pearson confidence interval, which returns the following simplified expression for $N_T > 40$ [2]:

$$\text{BER}_{\text{UB}} = \widehat{\text{BER}} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 + (2 - \widehat{\text{BER}})\right] \tag{6.2}$$

$$\text{BER}_{\text{LB}} = \widehat{\text{BER}} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 - (1 + \widehat{\text{BER}})\right], \tag{6.3}$$

where $z_{\alpha/2}$ is the $100\left(1 - \frac{\alpha}{2}\right)$th percentile of a standard normal distribution.

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.

Input Signals

Number: 2

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1

Type: Binary (DiscreteTimeDiscreteAmplitude)

Bibliografia

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

[2] Álvaro J Almeida, Nelson J Muga, Nuno A Silva, João M Prata, Paulo S André, and Armando N Pinto. Continuous control of random polarization rotations for quantum communications. *Journal of Lightwave Technology*, 34(16):3914–3922, 2016.

6.3 Binary source

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

1. Random
2. PseudoRandom
3. DeterministicCyclic
4. DeterministicAppendZeros

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

Input Parameters

Parameter: mode{PseudoRandom}
(Random, PseudoRandom, DeterministicCyclic, DeterministicAppendZeros)

Parameter: probabilityOfZero{0.5}
(real $\in$ [0,1])

Parameter: patternLength{7}
(integer $\in$ [1,32])

Parameter: bitStream{"0100011101010101"}
(string of 0's and 1's)

Parameter: numberOfBits{-1}
(long int)

Parameter: bitPeriod{1.0/100e9}
(double)

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-*probabilityOfZero*.

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

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)

Examples

Random Mode

PseudoRandom Mode As an example 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 6.1 numbered in this order). Some of these require wrap.

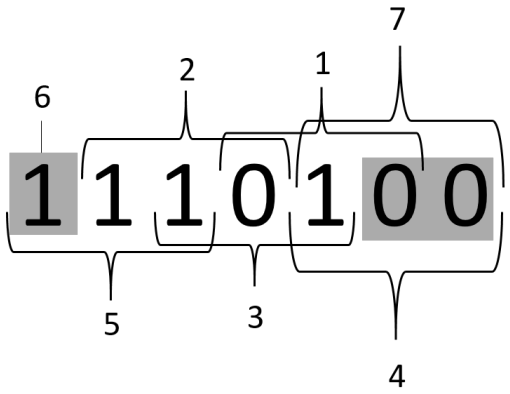

Figura 6.1: Example of a pseudorandom sequence with a pattern length equal to 3.

DeterministicCyclic Mode As an example take the *bit stream* '0100011101010101'. The generated binary signal is displayed in.

DeterministicAppendZeros Mode Take as an example the *bit stream* '0100011101010101'. The generated binary signal is displayed in 6.2.

Sugestions for future improvement

Implement an input signal that can work as trigger.

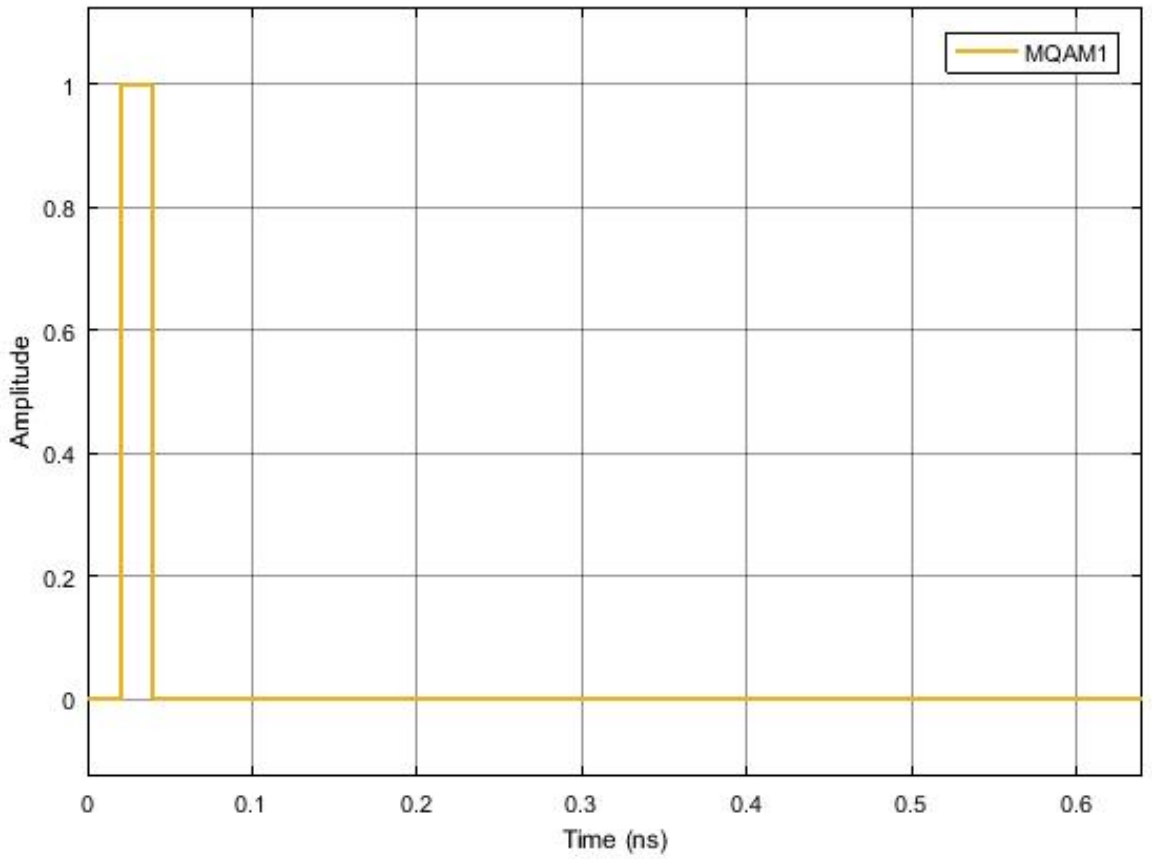

Figura 6.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

6.4 Bit Decider

Input Parameters

Parameter: setPosReferenceValue **Parameter:** setNegReferenceValue

Functional Description

This block accepts one real discrete signal and outputs a binary string, outputting a 1 if the input sample is above the predetermined reference level and 0 if it is below another reference value The reference values are defined by the values of *PosReferenceValue* and *NegReferenceValue*.

Input Signals

Number: 1
Type: Real signal (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

6.5 Clock

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: period{ 0.0 };

Parameter: samplingPeriod{ 0.0 };

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

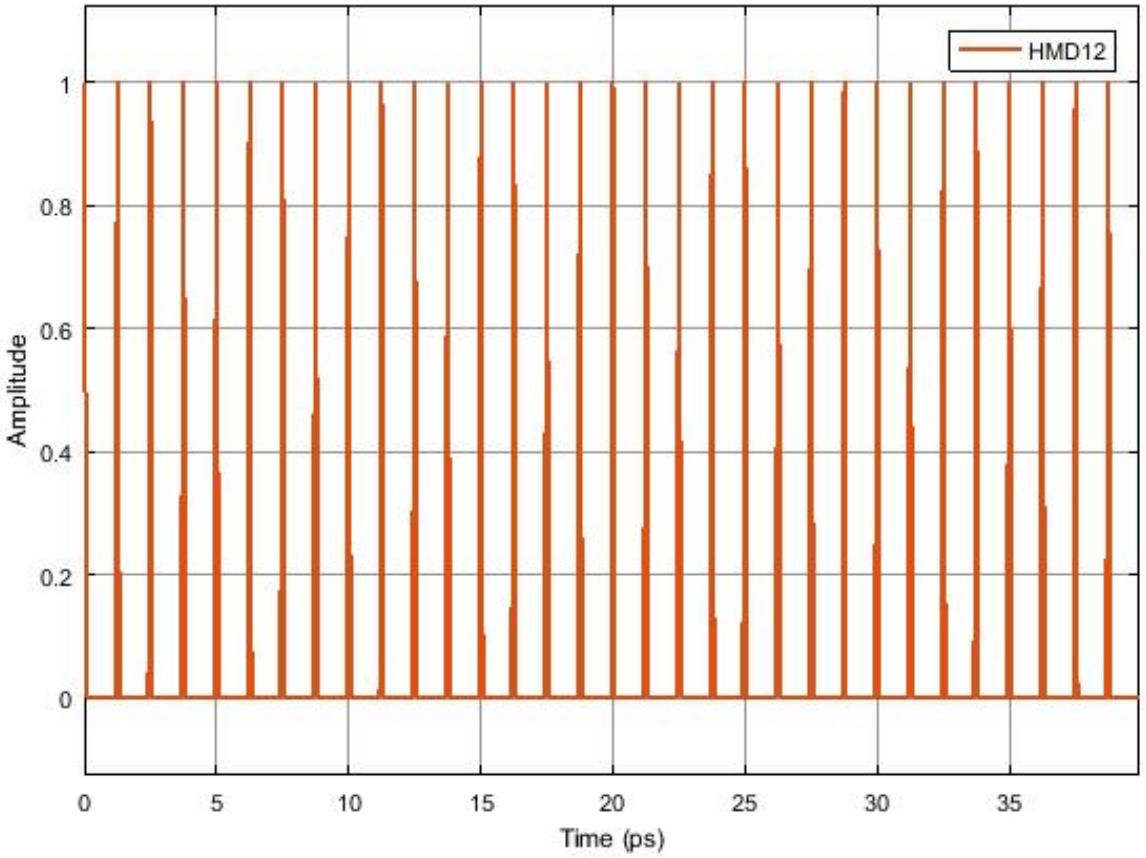

Figura 6.3: Example of the output signal of the clock

Sugestions for future improvement

6.6 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

Parameter: period{ 0.0 };

Parameter: samplingPeriod{ 0.0 };

Parameter: 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

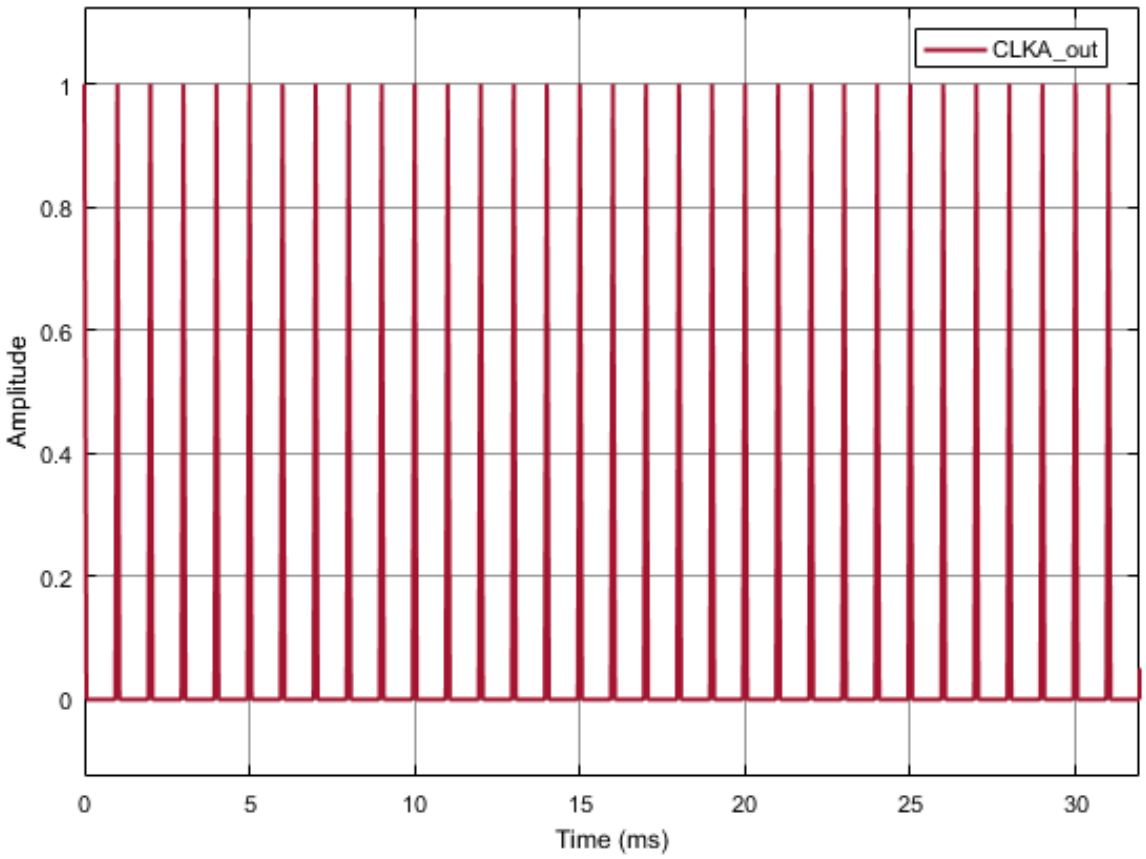

Figura 6.4: Example of the output signal of the clock without phase shift.

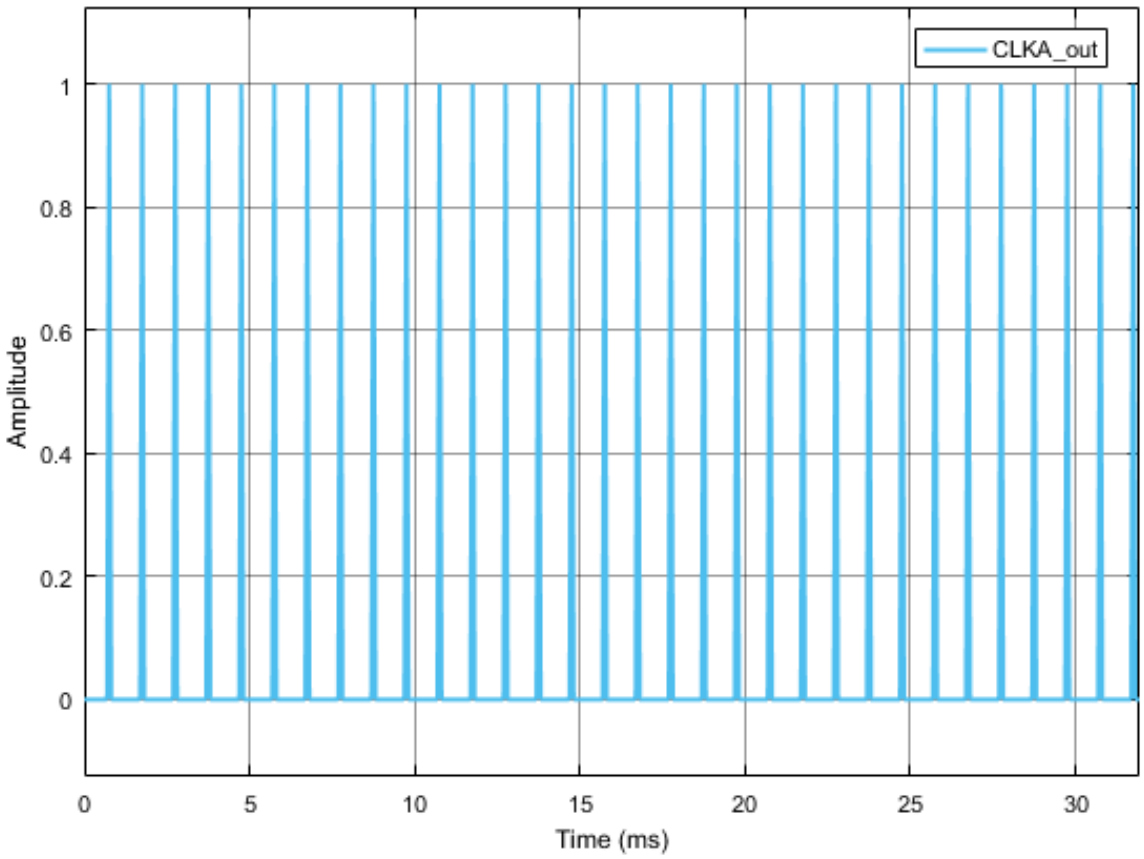

Figura 6.5: Example of the output signal of the clock with phase shift.

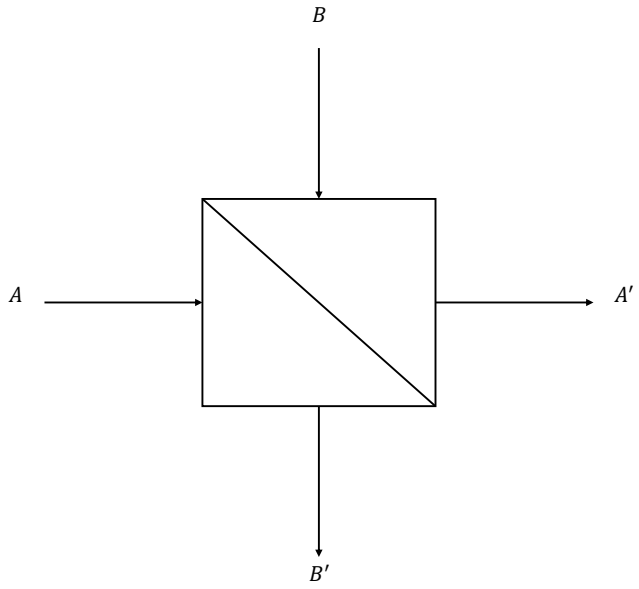

Figura 6.6: 2x2 coupler

6.7 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} . \begin{bmatrix} A \\ B \end{bmatrix} \tag{6.4}$$

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,

$$T = \sqrt{1 - \eta_R} \tag{6.5}$$

$$R = \sqrt{\eta_R} \tag{6.6}$$

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.

6.8 Decoder

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: `t_integer` m{ 4 }

Parameter: vector<`t_complex`> iqAmplitudes{ { 1.0, 1.0 },{ -1.0, 1.0 },{ -1.0, -1.0 },{ 1.0, -1.0 } };

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.

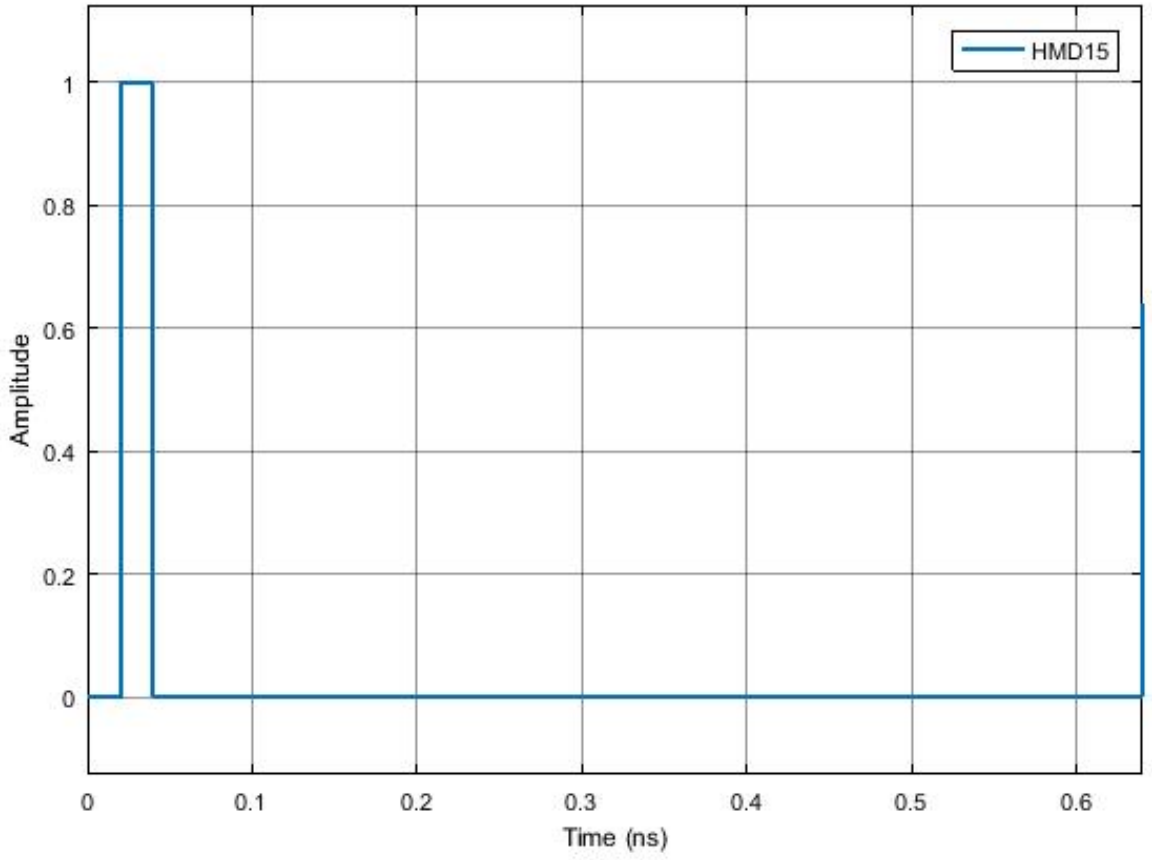

Figura 6.7: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

6.9 Discrete to continuous time

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: numberOfSamplesPerSymbol{8}
(int)

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

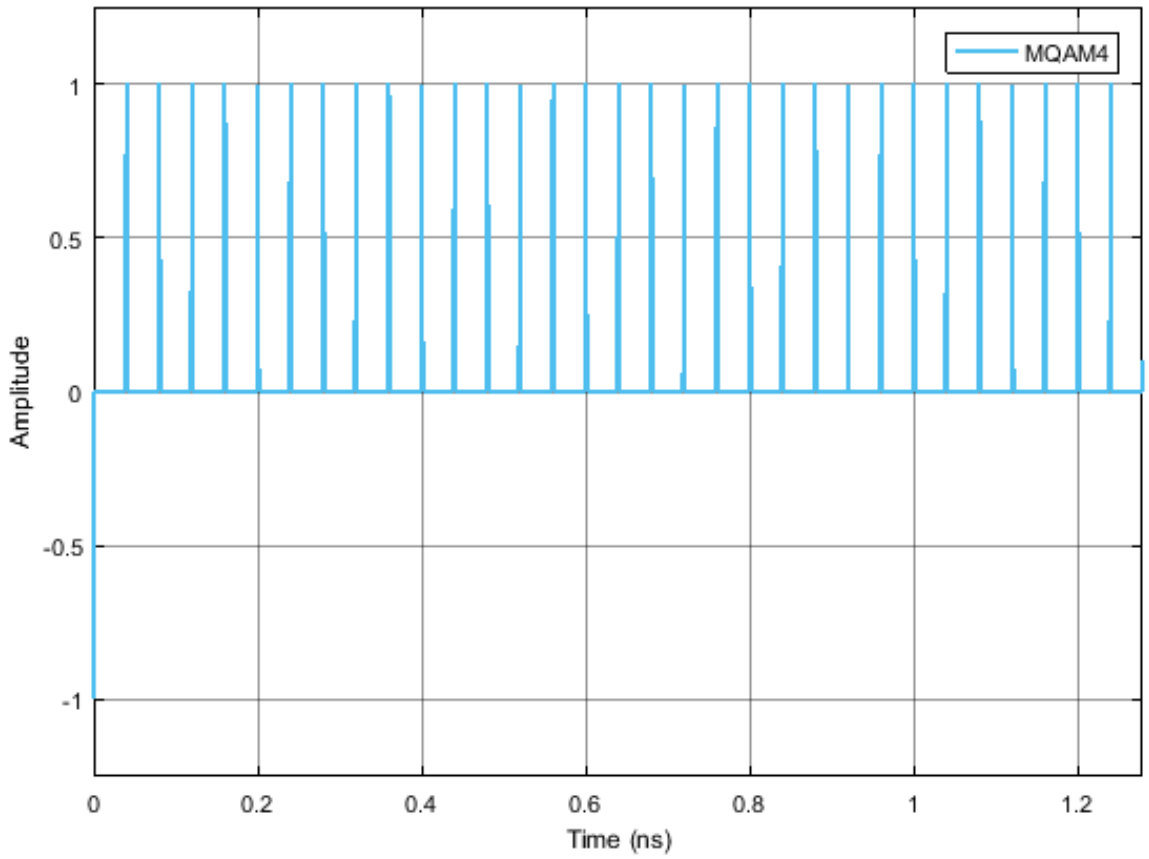

Figura 6.8: Example of the type of signal generated by this block for a binary sequence 0100...

6.10 Homodyne receiver

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 6.9.

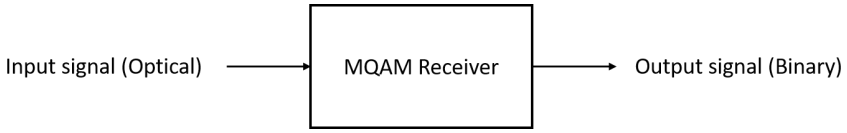

Figura 6.9: Basic configuration of the MQAM receiver

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 6.10) of code made up of several simpler blocks whose description can be found in the *lib* repository.

In can also be seen from figure 6.10 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*.

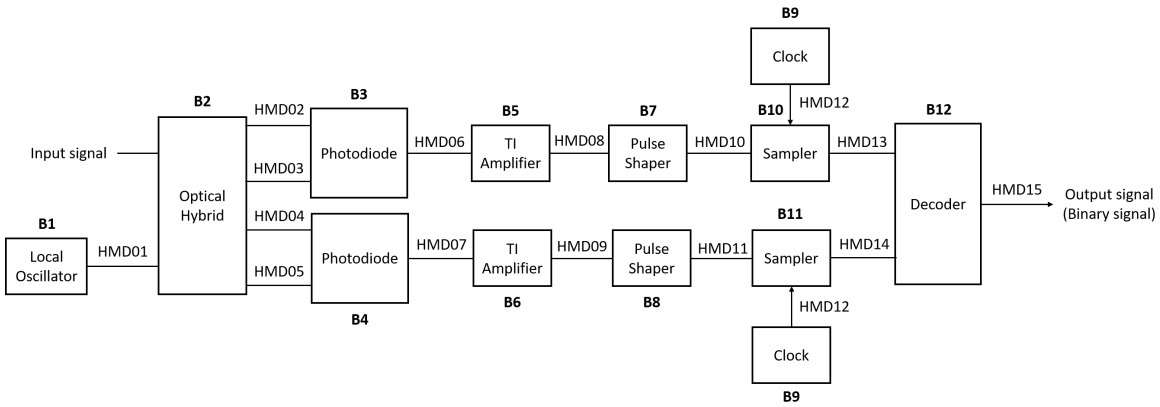

Figura 6.10: Schematic representation of the block homodyne receiver.

Input parameters

This block has some input parameters that can be manipulated by the user in order oto change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 6.2) 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 skipe	setSamplesToSkip	int(t_integer)	
Save internal signals	setSaveInternalSignals	bool	True or False
Sampling period	setSamplingPeriod	double	Givem by *symbolPeriod/samplesPerSy*

Tabela 6.1: 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

6.11 IQ modulator

This blocks accepts one inupt 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: outputOpticalPower{1e-3}
(double)

Parameter: outputOpticalWavelength{1550e-9}
(double)

Parameter: outputOpticalFrequency{speed_of_light/outputOpticalWavelength}
(double)

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 $\frac{1}{2}\sqrt{outputOpticalPower}$ 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 transmited to the receptor.

The binary signal is sent to the Bit Error Rate (BER) meaurement block.

Input Signals

Number : 2

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

Output Signals

Number : 1 or 2

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

Example

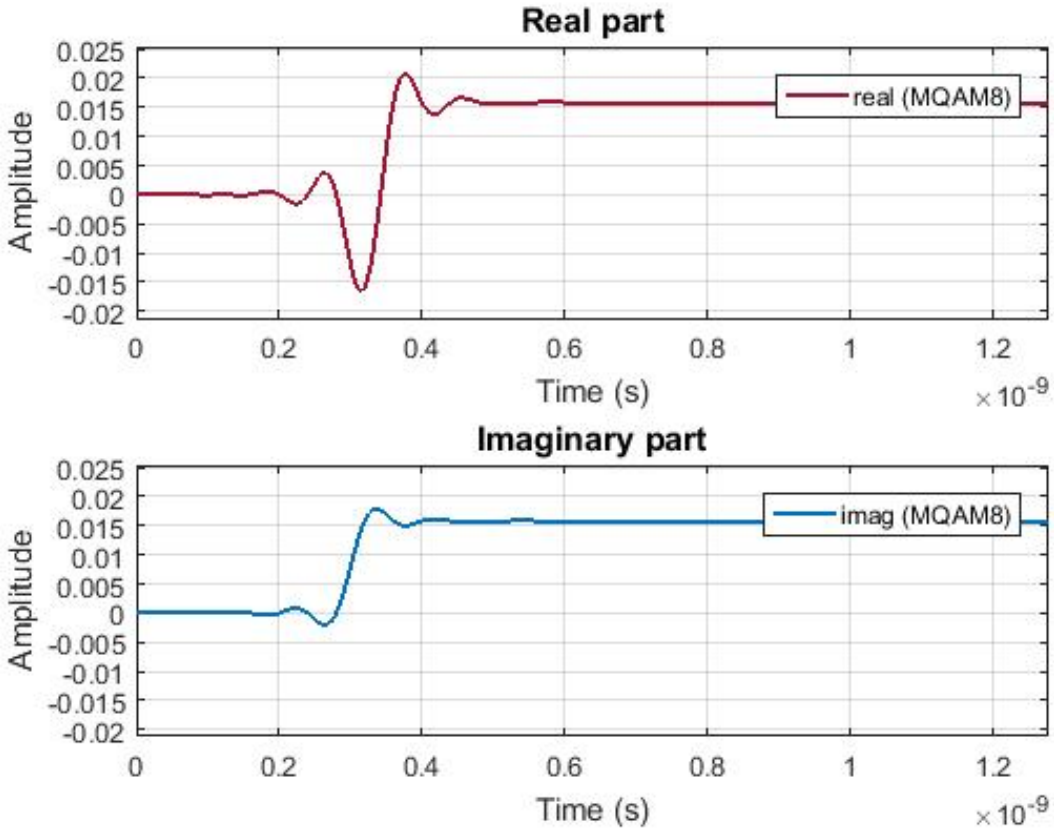

Figura 6.11: Example of a signal generated by this block for the initial binary signal 0100...

6.12 Local Oscillator

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: opticalPower{ 1e-3 }

Parameter: wavelength{ 1550e-9 }

Parameter: frequency{ SPEED_OF_LIGHT / wavelength }

Parameter: phase{ 0 }

Parameter: samplingPeriod{ 0.0 }

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);

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

Examples

Sugestions for future improvement

6.13 MQAM mapper

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: m{4}
(m should be of the form 2^n with n integer)

Parameter: iqAmplitudes{{ 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 }}

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 atributes 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 ilustrated in figure 6.12.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number : 2

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

Example

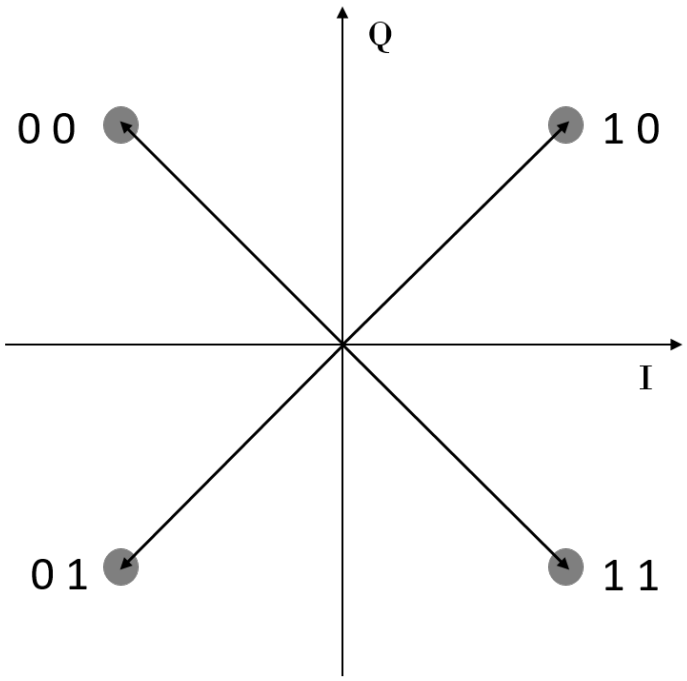

Figura 6.12: Constellation used to map the signal for m=4

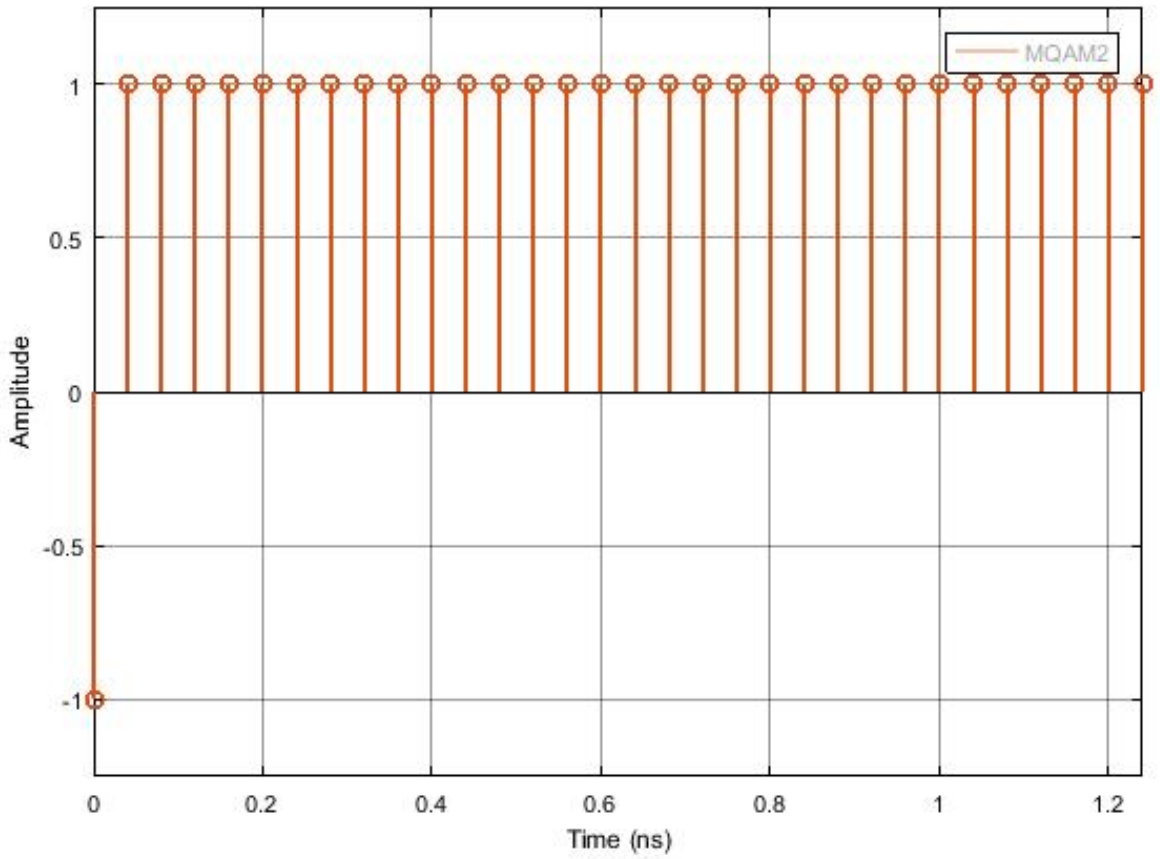

Figura 6.13: Example of the type of signal generated by this block for the initial binary signal 0100...

6.14 MQAM transmitter

This block generates a MQAM optical signal. It can also output the binary sequence. A schematic representation of this block is shown in figure 6.14.

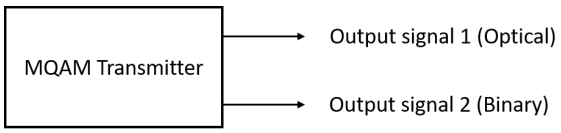

Figura 6.14: Basic configuration of the MQAM transmitter

Functional description

This block generates an optical signal (output signal 1 in figure 6.15). The binary signal generated in the internal block Binary Source (block B1 in figure 6.15) 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 6.15).

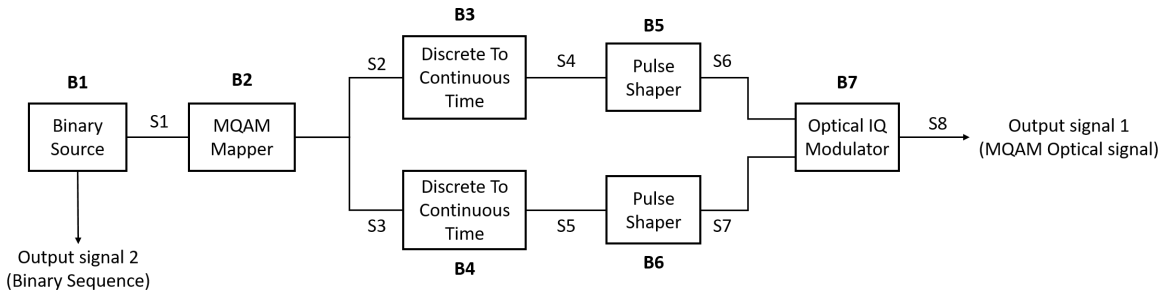

Figura 6.15: Schematic representation of the block MQAM transmitter.

Input parameters

This block has a special set of functions that allow the user to change the basic configuration of the transmitter. The list of input parameters, functions used to change them and the values that each one can take are summarized in table 6.2.

Input parameters	**Function**	Type	**Accepted values**
Mode	setMode()	string	PseudoRandom Random DeterministicAppendZeros DeterministicCyclic
Number of bits generated	setNumberOfBits()	int	Any integer
Pattern length	setPatternLength()	int	Real number greater than zero
Number of bits	setNumberOfBits()	long	Integer number greater than zero
Number of samples per symbol	setNumberOfSamplesPerSymbol()	int	Integer number of the type 2^n with n also integer
Roll of factor	setRollOfFactor()	double	$\in$ [0,1]
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 } }
Output optical power	setOutputOpticalPower()	int	Real number greater than zero
Save internal signals	setSaveInternalSignals()	bool	True or False

Tabela 6.2: List of input parameters of the block MQAM transmitter

Methods

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)

Output Signals

Number: 1 optical and 1 binary (optional)

Type: Optical signal

Example

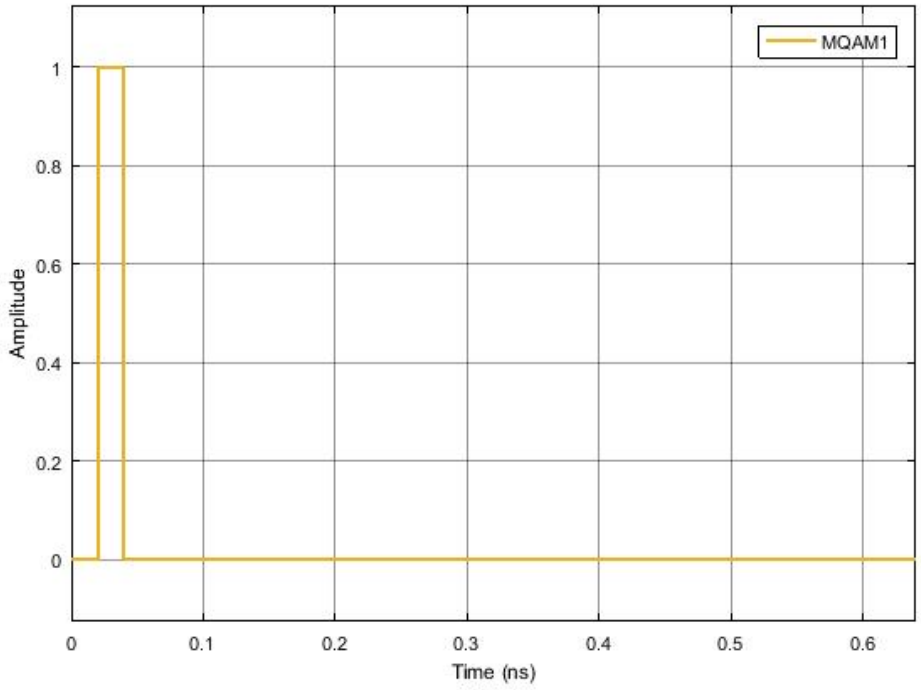

Figura 6.16: Example of the binary sequence generated by this block for a sequence 0100...

Sugestions for future improvement

Add to the system another block similar to this one in order to generate two optical signals with perpendicular polarizations. This would allow to combine the two optical signals and generate an optical signal with any type of polarization.

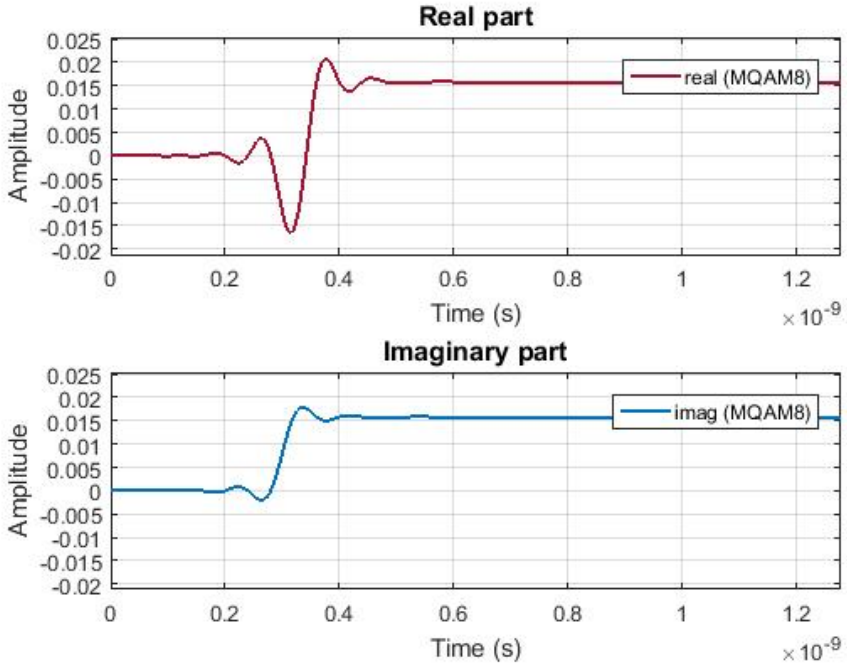

Figura 6.17: Example of the output optical signal generated by this block for a sequence 0100...

6.15 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

Parameter: double RateOfPhotons{1e3}

Parameter: 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

6.16 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

Parameter: m{4}

Parameter: 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 atributes for each pair of bits a point in space. In this case, it is be considered four possible polarization states: $0°$, $45°$, $90°$ and $^{1}35°$.

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

6.17 Bob QKD

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

1.

2.

Input Parameters

Parameter:

Parameter:

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

6.18 Eve QKD

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

1.

2.

Input Parameters

Parameter:

Parameter:

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

6.19 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

Parameter: m{2}

Parameter: 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

6.20 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

Capítulo 7

Mathlab Tools

7.1 Generation of AWG Compatible Signals

Student Name	:	Francisco Marques dos Santos
Starting Date	:	September 1, 2017
Goal	:	Convert simulation signals into waveform files compatible with the laboratory's Arbitrary Waveform Generator
Directory	:	mtools

This section shows how to convert a simulation signal into an AWG compatible waveform file through the use of a matlab function called sgnToWfm. This allows the application of simulated signals into real world systems.

7.1.1 sgnToWfm

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm(fname_sgn, nReadr, fname_wfm);

Inputs

fname_sgn: Input filename of the signal (*.sgn) you want to convert. It must be a real signal (Type: TimeContinuousAmplitudeContinuousReal).

nReadr: Number of symbols you want to extract from the signal.

fname_wfm: Name that will be given to the waveform file.

Outputs

A waveform file will be created in the Matlab current folder. It will also return six variables in the workspace which are:

data: A vector with the signal data.

symbolPeriod: Equal to the symbol period of the corresponding signal.

samplingPeriod: Sampling period of the signal.

type: A string with the name of the signal type.

numberOfSymbols: Number of symbols retrieved from the signal.

samplingRate: Sampling rate of the signal.

Functional Description

This matlab function generates a *.wfm file given an input signal file (*.sgn). The waveform file is compatible with the laboratory's Arbitrary Waveform Generator (Tekatronix AWG70002A). In order to recreate it appropriately, the signal must be real, not exceed $8 * 10^9$ samples and have a sampling rate equal or bellow 16 GS/s.

This function can be called with one, two or three arguments:

Using one argument:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm('S6.sgn');

This creates a waveform file with the same name as the *.sgn file and uses all of the samples it contains.

Using two arguments:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm('S6.sgn',256);

This creates a waveform file with the same name as the signal file name and the number of samples used equals nReadr x samplesPerSymbol. The samplesPerSymbol constant is defined in the *.sgn file.

Using three arguments:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm('S6.sgn',256,'myWaveform.wfm');

This creates a waveform file with the name "myWaveform"and the number of samples used equals nReadr x samplesPerSymbol. The samplesPerSymbol constant is defined in the *.sgn file.

7.1.2 Loading a signal to the Tekatronix AWG70002A

The AWG we will be using is the Tekatronix AWG70002A which has the following key specifications:

Sampling rate up to 16 GS/s: This is the most important characteristic because it determines the maximum sampling rate that your signal can have. It must not be over 16 GS/s or else the AWG will not be able to recreate it appropriately.

8 GSample waveform memory: This determines how many data points your signal can have.

After making sure this specifications are respected you can create your waveform using the function. When you load your waveform, the AWG will output it and repeat it constantly until you stop playing it.

1. Using the function sgnToWfm: Start up Matlab and change your current folder to mtools and add the signals folder that you want to convert to the Matlab search path. Use the function accordingly, putting as the input parameter the signal file name you want to convert.

2. AWG sampling rate: After calling the function there should be waveform file in the mtools folder, as well as a variable called samplingRate in the Matlab workspace. Make sure this is equal or bellow the maximum sampling frequency of the AWG (16 GS/s), or else the waveform can not be equal to the original signal. If it is higher you have to adjust the parameters in the simulation in order to decrease the sampling frequency of the signal(i.e. decreasing the bit period or reducing the samples per symbol).

3. Loading the waveform file to the AWG: Copy the waveform file to your pen drive and connect it to the AWG. With the software of the awg open,go to browse for waveform on the channel you want to use, and select the waveform file you created (Figure 7.1).

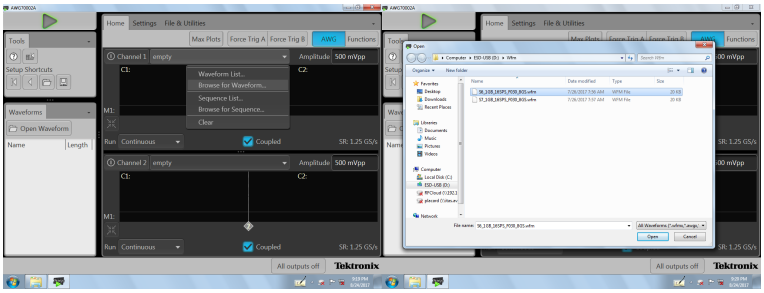

Figura 7.1: Selecting your waveform in the AWG

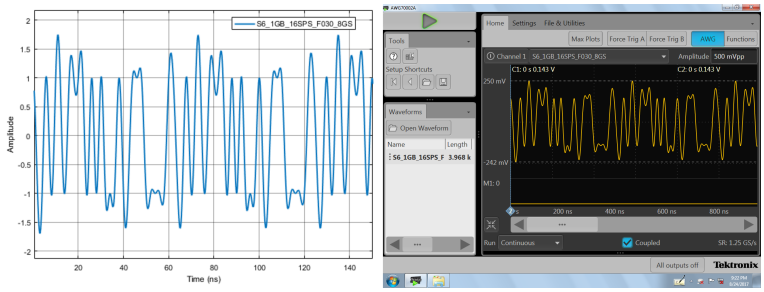

Figura 7.2: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

Now you should have the waveform displayed on the screen. Although it has the same shape, the waveform might not match the signal timing wise due to an incorrect sampling rate configured in the AWG. In this example (Figure 7.2), the original signal has a sample

rate of 8 GS/s and the AWG is configured to 1.25 GS/s. Therefore it must be changed to the correct value. To do this go to the settings tab, clock settings, and change the sampling rate to be equal to the one of the original signal, 8 GS/s (Figure 7.3). Compare the waveform in

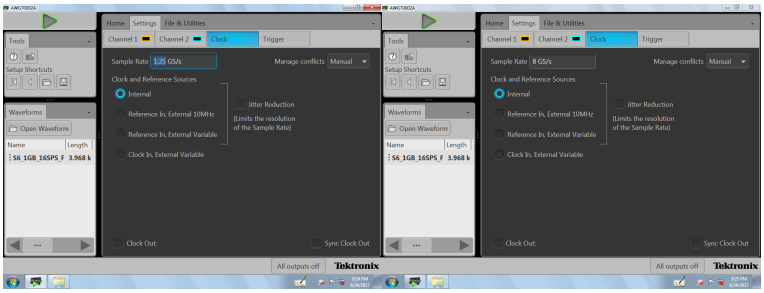

Figura 7.3: Configuring the right sampling rate

the AWG with the original signal, they should be identical (Figure 7.4).

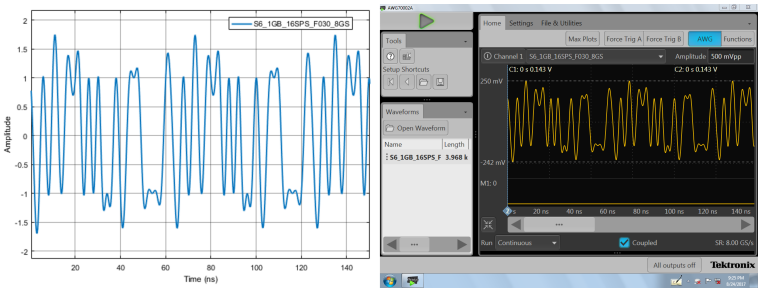

Figura 7.4: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

4. Generate the signal: Output the wave by enabling the channel you want and clicking on the play button.