NetXPTO - LinkPlanner

31 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

PhotonStreamXY

A single photon is described by two amplitudes A_x and A_y and a phase difference between them, δ. This way, the signal PhotonStreamXY is a structure with two complex numbers, x and y.

PhotonStreamXY_MP

The multi-path signals are used to simulate the propagation of a quantum signal when the signal can follow multiple paths. The signal has information about all possible paths, and a measurement performed in one path immediately affects all other possible paths. From a Quantum approach, when a single photon with a certain polarization angle reaches a $50:50$ Polarizer, it has a certain probability of follow one path or another. In order to simulate this, we have to use a signal PhotonStreamXY_MP, which contains information about all the paths available. In this case, we have two possible paths: 0 related with horizontal and 1 related with vertical. This signal is the same in both outputs of the polarizer. The first decision is made by the detector placed on horizontal axis. Depending on that decision, the information about the other path 1 is changed according to the result of the path 0. This way, we guarantee the randomness of the process. So, signal PhotonStreamXY_MP is a structure of two PhotonStreamXY indexed by its path.

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.

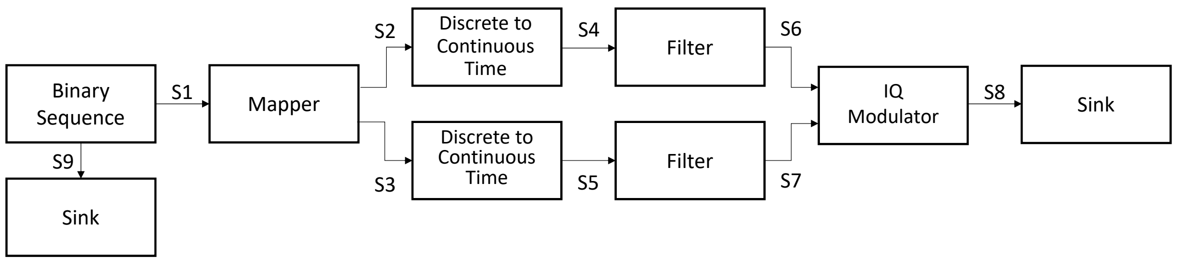


Figura 5.1: QPSK transmitter block diagram.

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).

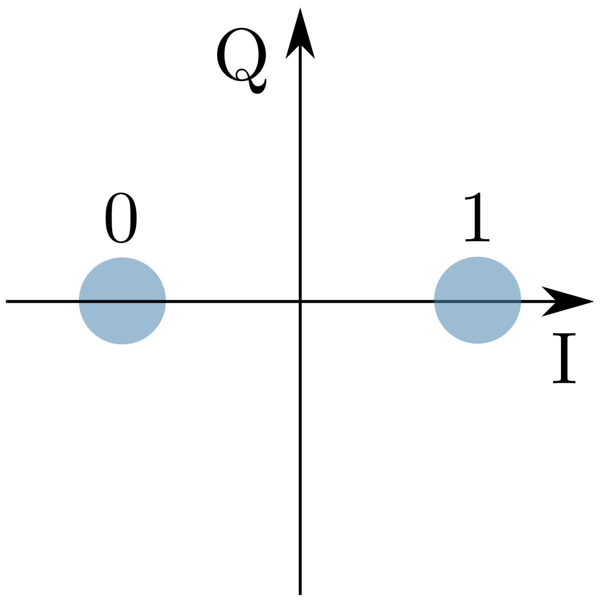


Figura 5.2: BPSK symbol constellation.

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

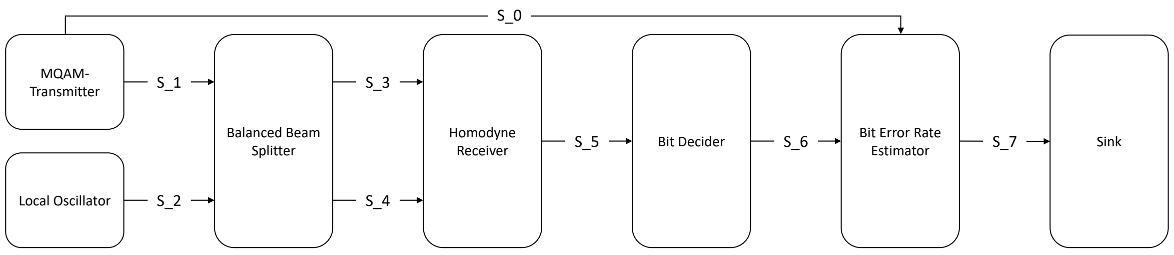


Figura 5.3: Overview of the BPSK system being simulated.

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].

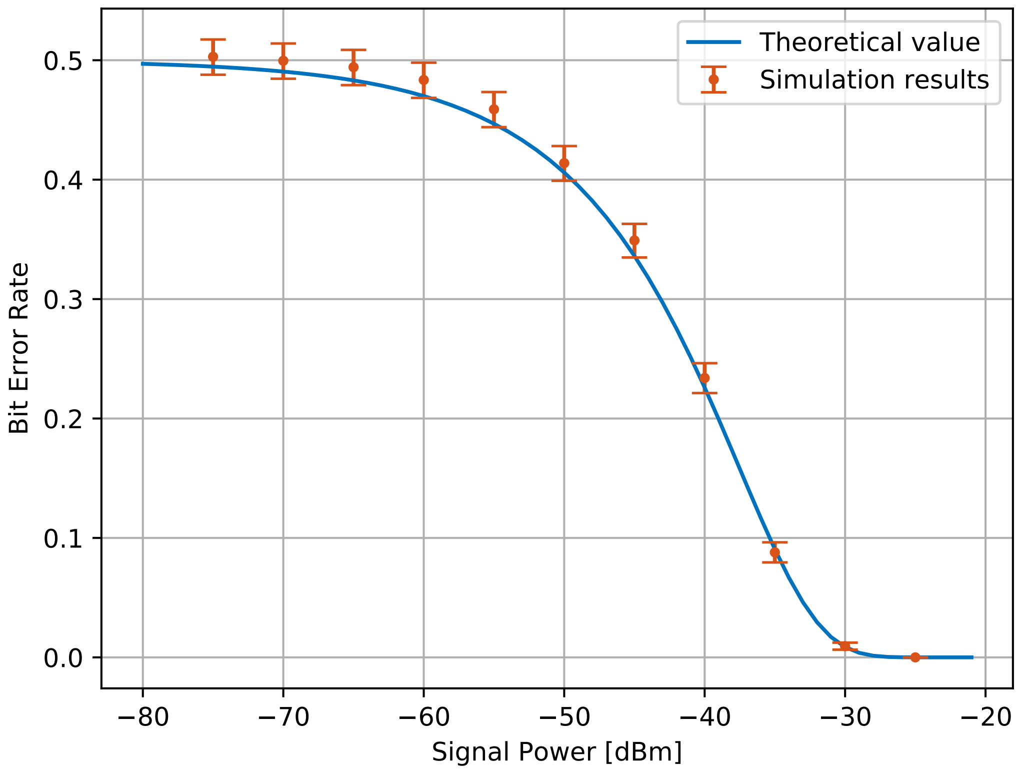


Figura 5.4: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with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

Bibliografia

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

[2] GE Katsoprinakis, M Polis, A Tavernarakis, AT Dellis, and IK Kominis. Quantum random number generator based on spin noise. *Physical Review A*, 77(5):054101, 2008.

[3] Thomas Jennewein, Ulrich Achleitner, Gregor Weihs, Harald Weinfurter, and Anton Zeilinger. A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71(4):1675–1680, 2000.

[4] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists, Fifth Edition*. Academic Press, 2014.

[5] 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:

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

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.

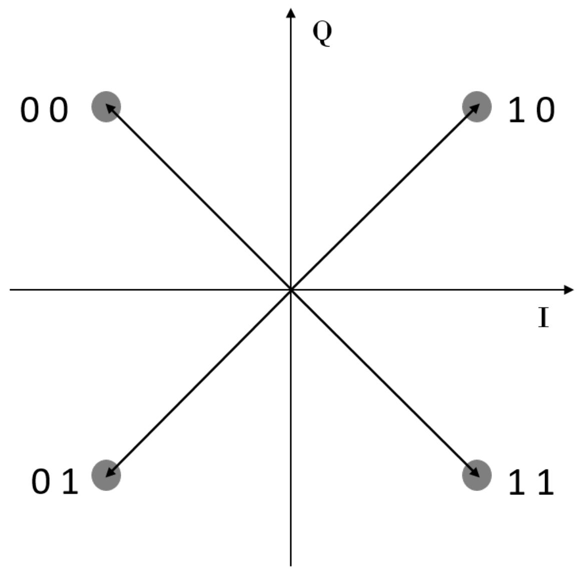


Figura 5.5: 4-QAM symbol constellation

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

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

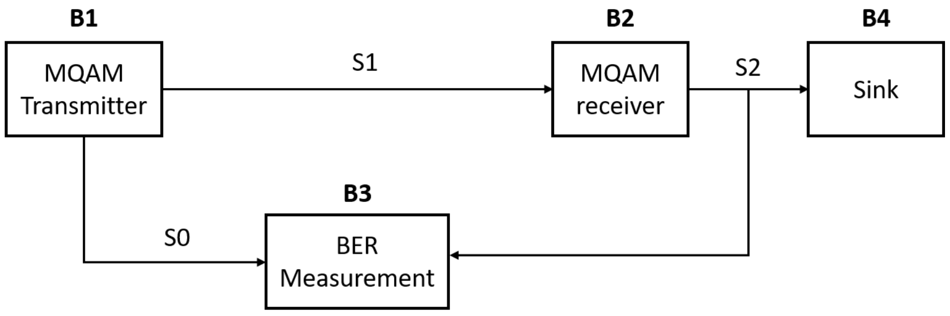


Figura 5.6: Schematic representation of the MQAM system.

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 Quantum Random Number Generator

Students Name	:	Mariana Ramos (12/01/2018 -)
Goal	:	sdf/quantum_random_number_generator.

True random numbers are indispensable in the field of cryptography ~~to guarantee the security of the communication protocols~~ [2]. There are two approaches for random number generation: the pseudorandom generation which are based on an algorithm implemented on a computer, and the physical random generators which consist in measuring some physical observable with random behaviour. Since classical physics description is deterministic, all classical processes are in principle predictable. Therefore, a true random number generator must be based on a quantum process.

In this chapter, it is presented the theoretical ~~analysis~~ ~~and~~ simulation of a quantum random generator ~~by measuring the polarization of single photons with a polarizing beam splitter~~.

5.4.1 Theoretical Analysis

~~Nowadays, the only known way to generate truly random numbers is by building a physical source by using quantum mechanical decisions, since the occurrence of each individual result of such a quantum mechanical decision is truly random, ie it is inestimable or unknowable~~ [3]. One of the optical processes available as a source of randomness is the splitting of a polarized single photon beam.

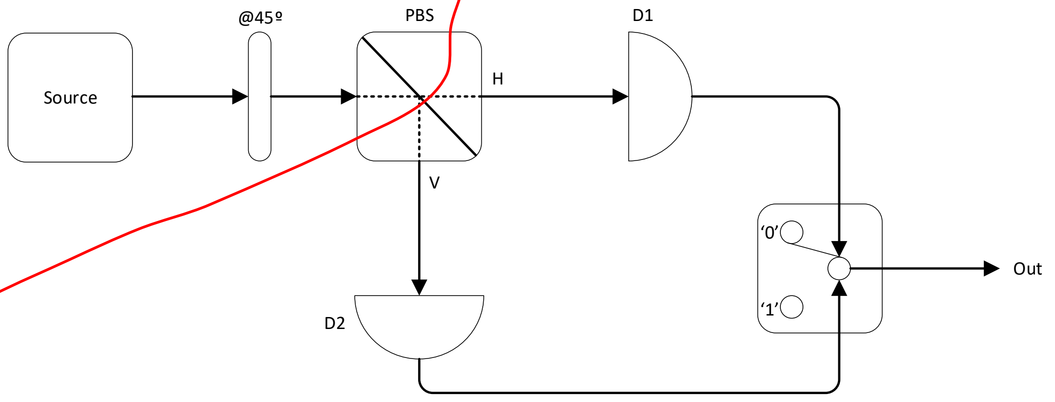


Figura 5.7: Source of randomness with a polarization beam splitter PBS where the incoming light is linearly polarized at 45° with respect to the PBS. Figure adapted from [3].

The principle of operation of the random generator is shown in figure 5.7. Each individual photon coming from the source is linearly polarized at 45° and has equal probability of found in the horizontal ~~polarization~~ (H) or in the vertical ~~polarization~~ (V) output of the PBS. Quantum theory estimates for both cases the individual choices are truly random ~~and~~ independent one from each other. ~~This way, the detection of the photons in each output of the polarization beam splitter is done with single photon detectors and combining the detection pulse in a switch, which has two possible states: "0" or "1".~~ When the detector **D1** fires, ~~the switch is flipped to state "0" and does not move until a detection event in detector **B2** occurs and it does not move until a detection occurs in detector **D1**. In the case of some detections occur in a row in the same detector, only the first detection clicks and the following detections leave the switch unaltered.~~

5.4.2 Simulation Analysis

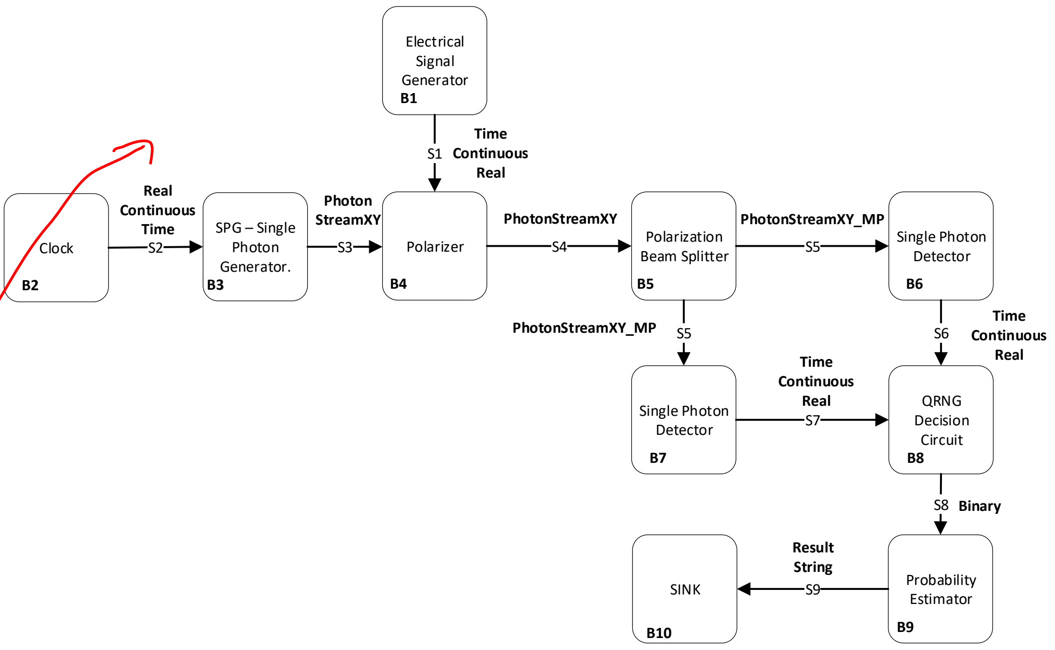


Figura 5.8: Block diagram of the simulation of a Quantum Random Generator.

The simulation diagram of the setup described in the previous section is presented in figure 5.8. The linear polarizer has an input control signal (S1) which allows to change the rotation angle. Nevertheless, the only purpose is to generate a time and amplitude continuous real signal with the value of the rotation angle in degrees.

In addition, the photons are generated by single photon source block at a rate defined by the clock rate.

At the end of the simulation there is a circuit decision block which will outputs a binary signal with value "0" if the detector at the end of the horizontal path clicks or "1" if the detector at the end of the vertical path clicks.

In table 5.3 are presented the input parameters of the system.

Tabela 5.3: System Input Parameters

Parameter	**Default Value**
RateOfPhotons	1e6
NumberOfSamplesPerSymbol	16
PolarizerAngle	45.0

In table 5.4 are presented the system signals to implement the simulation presented in figure 5.8.

Tabela 5.4: System Signals

Signal name	**Signal type**
S1	TimeContinuousAmplitudeContinuousReal
S2	TimeContinuousAmplitudeContinuousReal
S3	PhotonStreamXY
S4	PhotonStreamXY
S5	PhotonStreamXYMP
S6	TimeContinuousAmplitudeContinuousReal
S7	TimeContinuousAmplitudeContinuousReal
S8	Binary
S9	Binary

Table 5.5 presents the header files used to implement the simulation as well as the specific parameters that should be set in each block. Finally, table 5.6 presents the source files.

Tabela 5.5: Header Files

File name	**Description**	**Status**
netxpto.h		✓
electrical_signal_generator_20180124.h	setFunction(), setGain()	✓
clock_20171219.h	ClockPeriod(1 / RateOfPhotons)	✓
polarization_beam_splitter_20180109.h		✓
polarizer_20180113.h		✓
single_photon_detector_20180111.h	setPath(0), setPath(1)	✓
single_photon_source_20171218.h		✓
probability_estimator_20180124.h		✓
sink.h		✓
qrng_decision_circuit.h		✓

Tabela 5.6: Source Files

File name	**Description**	**Status**
netxpto.cpp		✓
electrical_signal_generator_20180124.cpp		✓
clock_20171219.cpp		✓
polarization_beam_splitter_20180109.cpp		✓
polarizer_20180113.cpp		✓
single_photon_detector_20180111.cpp		✓
single_photon_source_20171218.cpp		✓
probability_estimator_20180124.cpp		✓
sink.cpp		✓
qrng_decision_circuit.cpp		✓
qrng_sdf.cpp		✓

In theory, considering the results space Ω associated with a random experience and A an event such that $P(A) = p \in]0,1[$. Lets $X : \Omega \longrightarrow \mathbb{R}$ such that

$$\begin{aligned} X(\omega) = 1 \quad &\text{,if } \omega \in A \\ X(\omega) = 1 \quad &\text{,if } \omega \in \bar{A} \end{aligned} \tag{5.17}$$

This way,

$$\begin{aligned} P(X=1) = \quad & P(A) \quad = p \\ P(X=0) = \quad & P(\bar{A}) \quad = 1-p \end{aligned} \tag{5.18}$$

X follows the Bernoulli law with parameter **p**, $X \sim \mathbf{B}(p)$, being the expected value of the Bernoulli random value E(X) = p and the variance VAR(X) = $p(1-p)$ [4].

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 99%.

We will use *z-score* (in this case 2.576, since a confidence interval of $99\%$was chosen) to calculate the expected error margin,

$$E = z_{\alpha/2}\frac{\sigma}{\sqrt{N}},$$

where, E 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. Lets assume, for an angle of $45°$, a number of samples$N = 1 \times 10^6$ and the expected probability of reach each detector of $\hat{p} = 0.5$. We have an error margin of $E = 1.288 \times 10^{-3}$, which is acceptable. This way, the simulation will be performed for $N = 1 \times 10^6$ samples for

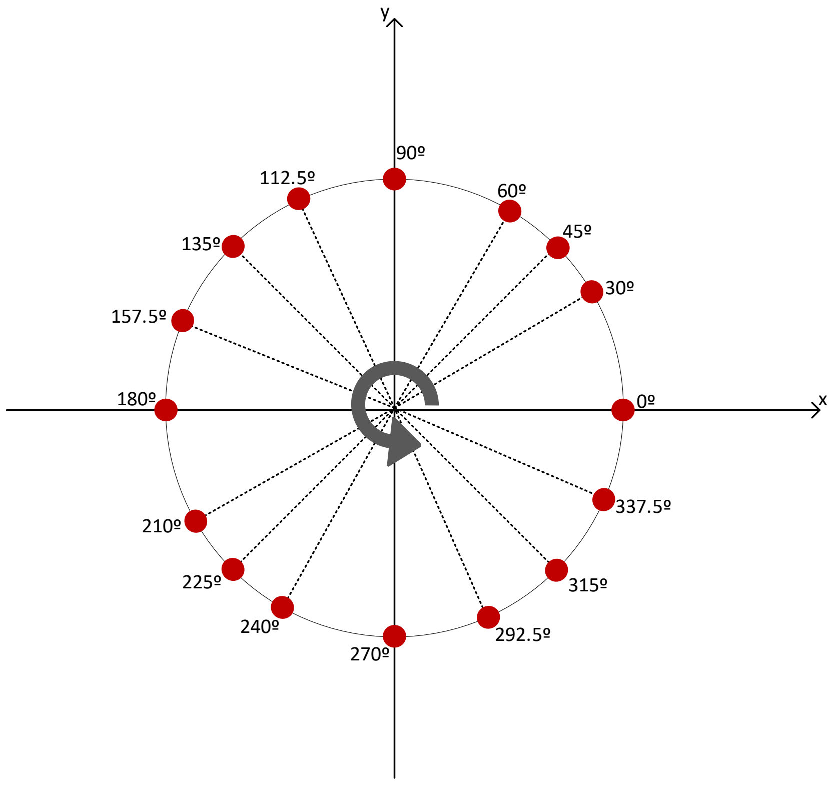


Figura 5.9: Angles used to perform the qrng simulation for $N = 1 \times 10^{\overset{6}{\cancel{8}}}$ samples.

different angles of polarization shown in figure 5.9 with different error margin's values since the expected probability changes depending on the polarization angle.

For a quantum random number generator with equal probability of obtain a "0" or "1" the polarizer must be set at $45°$. This way, we have 50% possibilities to obtain a "0" and 50% of possibilities to obtain a "1" . This theoretical value meets the value obtained from the simulation when it is performed for the number of samples mentioned above.

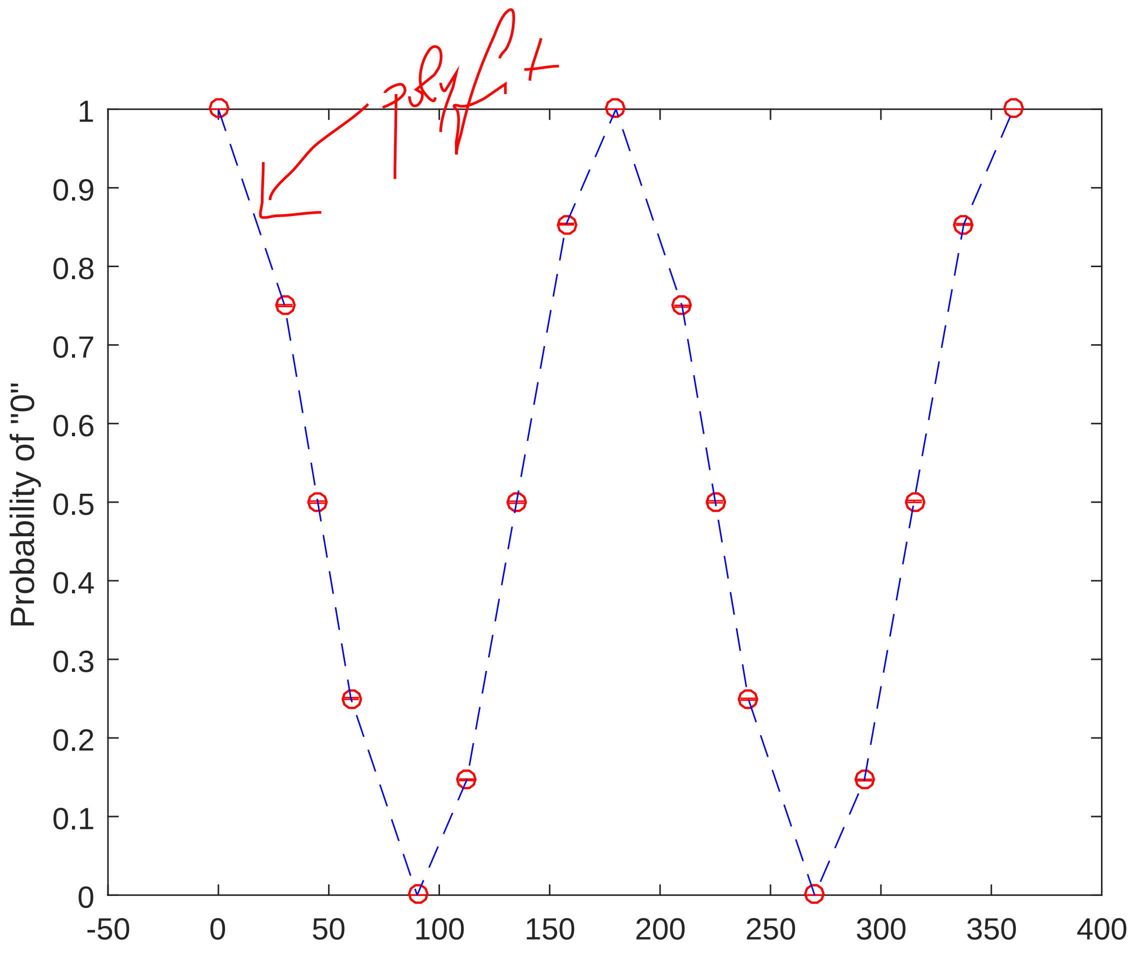


Figura 5.10: Probability of outputs a number "0" depending on the polarization angle.

Figure 5.10 shows the probability of a single photons reaches the detector placed on Horizontal axis depending on the polarization angle of the photon, and this way the output number is "0". On the other hand, figure 5.11 shows the probability of a single photon reaches the detector placed on Vertical component of the polarization beam splitter, and this way the output number is "1". As we can see in the figures the two detectors have complementary probabilities, i.e the summation of both values must be equals to 1.

One can see that "Probability of 1" behaves almost like a sine function and "Probability of 0" behaves almost like a cosine function with a variable angle.

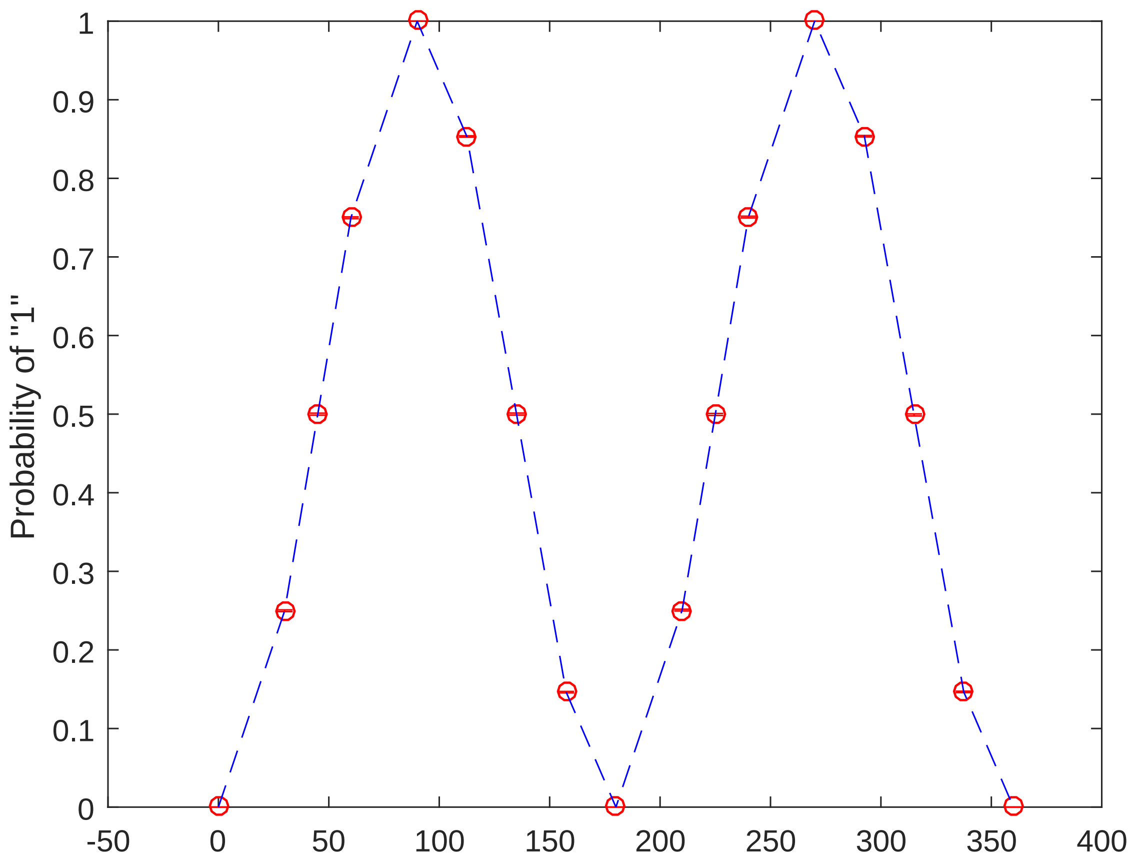


Figura 5.11: Probability of outputs a number "1" depending on the polarization angle.

5.4.3 Experimental Analysis

In order to have a real experimental quantum random number generator, a setup shown in figure 5.12 was built in the lab. To simulate a single photon source we have a CW-Pump laser with 1550 nm wavelength followed by an interferometer Mach-Zenhder in order to have a pulsed beam. The interferometer has an input signal given by a Pulse Pattern Generator. This device also gives a clock signal for the Single Photon Detector (APD-Avalanche Photodiode) which sets the time during which the window of the detector is open. After the MZM there is a Variable Optical Attenuator (VOA) which reduces the amplitude of each pulse until the probability of one photon per pulse is achieved. Next, there is a polarizer controller followed by a Linear Polarized, which is set at $45°$, then a Polarization Beam Splitter (PBS) and finally, one detector at the end of each output of the PBS. The output signals from the detector will be received by a Processing Unit. Regarding to acquired the output of the detectors, there is an oscilloscope capable of record 1×10^6 samples.

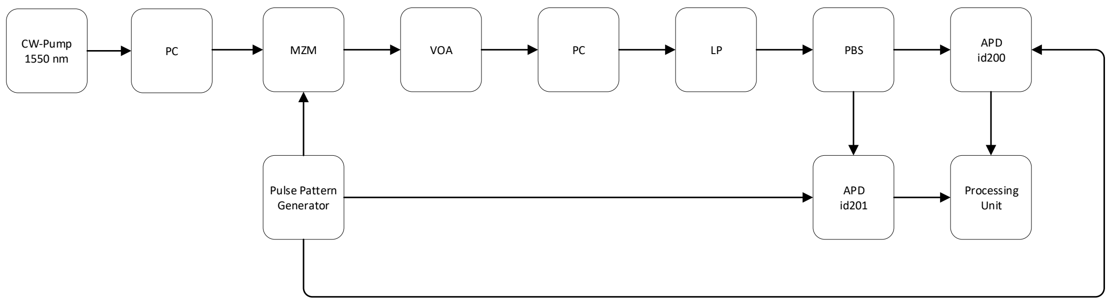


Figura 5.12: Experimental setup to implement a quantum random number generator.

5.4.4 Open Issues

References

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

[2] GE Katsoprinakis, M Polis, A Tavernarakis, AT Dellis, and IK Kominis. Quantum random number generator based on spin noise. *Physical Review A*, 77(5):054101, 2008.

[3] Thomas Jennewein, Ulrich Achleitner, Gregor Weihs, Harald Weinfurter, and Anton Zeilinger. A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71(4):1675–1680, 2000.

[4] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists, Fifth Edition*. Academic Press, 2014.

[5] 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.

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 $N_T > 40$ [5]:

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

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

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)

References

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

[2] GE Katsoprinakis, M Polis, A Tavernarakis, AT Dellis, and IK Kominis. Quantum random number generator based on spin noise. *Physical Review A*, 77(5):054101, 2008.

[3] Thomas Jennewein, Ulrich Achleitner, Gregor Weihs, Harald Weinfurter, and Anton Zeilinger. A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71(4):1675–1680, 2000.

[4] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists, Fifth Edition*. Academic Press, 2014.

[5] 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 ∈ [0,1])

Parameter: patternLength{7}
(integer ∈ [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.

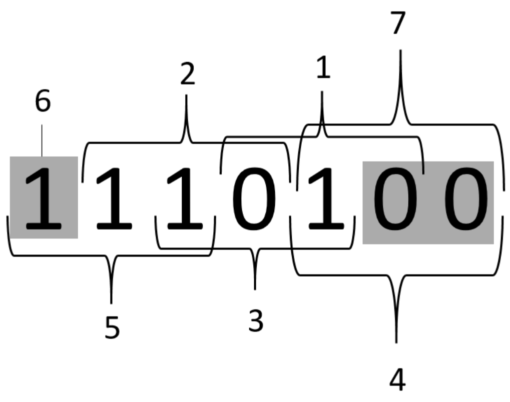


Figura 6.1: Example of a pseudorandom sequence with a pattern length equal to 3.

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.

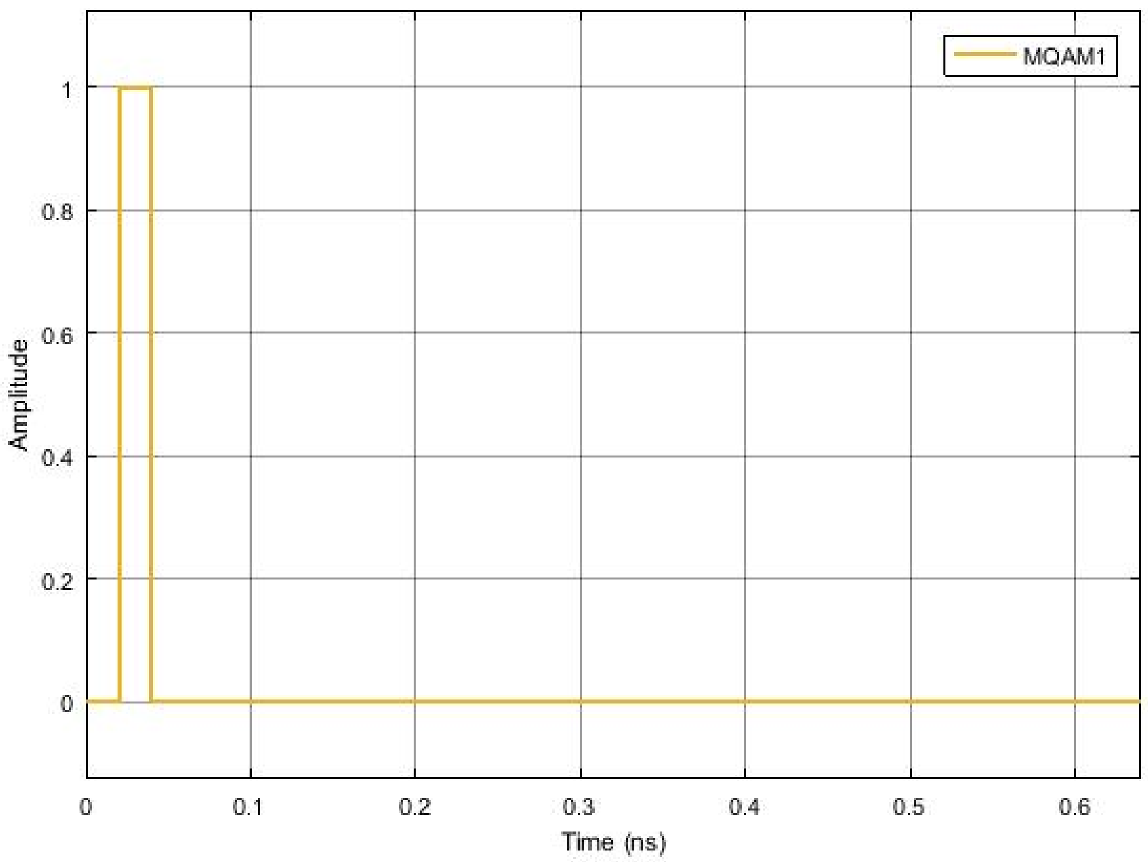


Figura 6.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

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

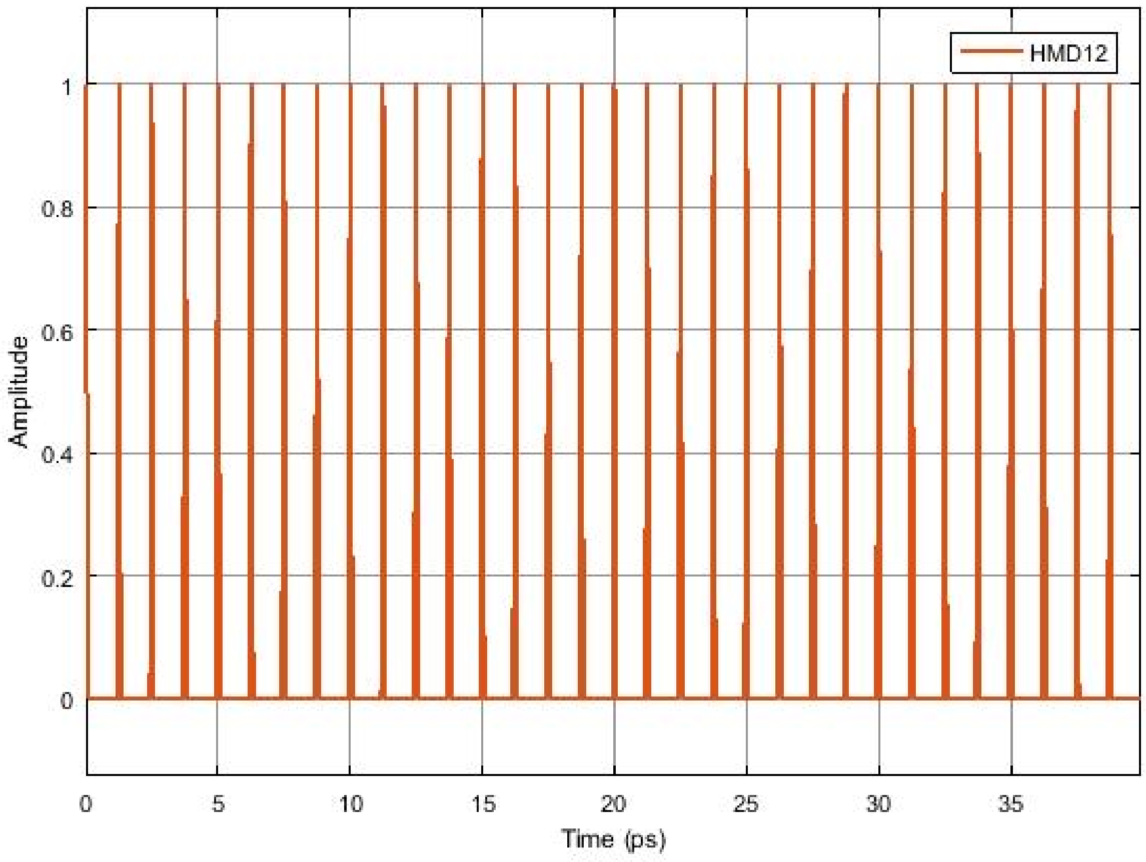


Figura 6.3: Example of the output signal of the clock

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

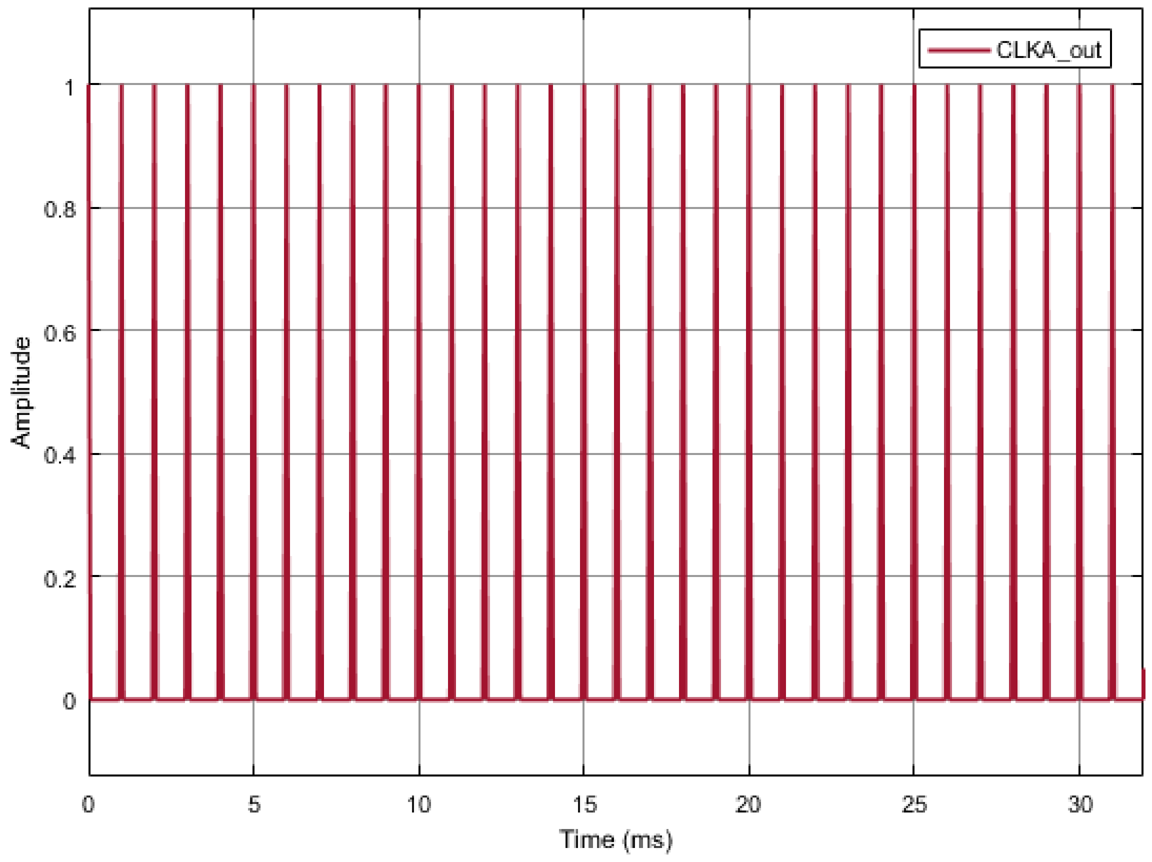


Figura 6.4: Example of the output signal of the clock without phase shift.

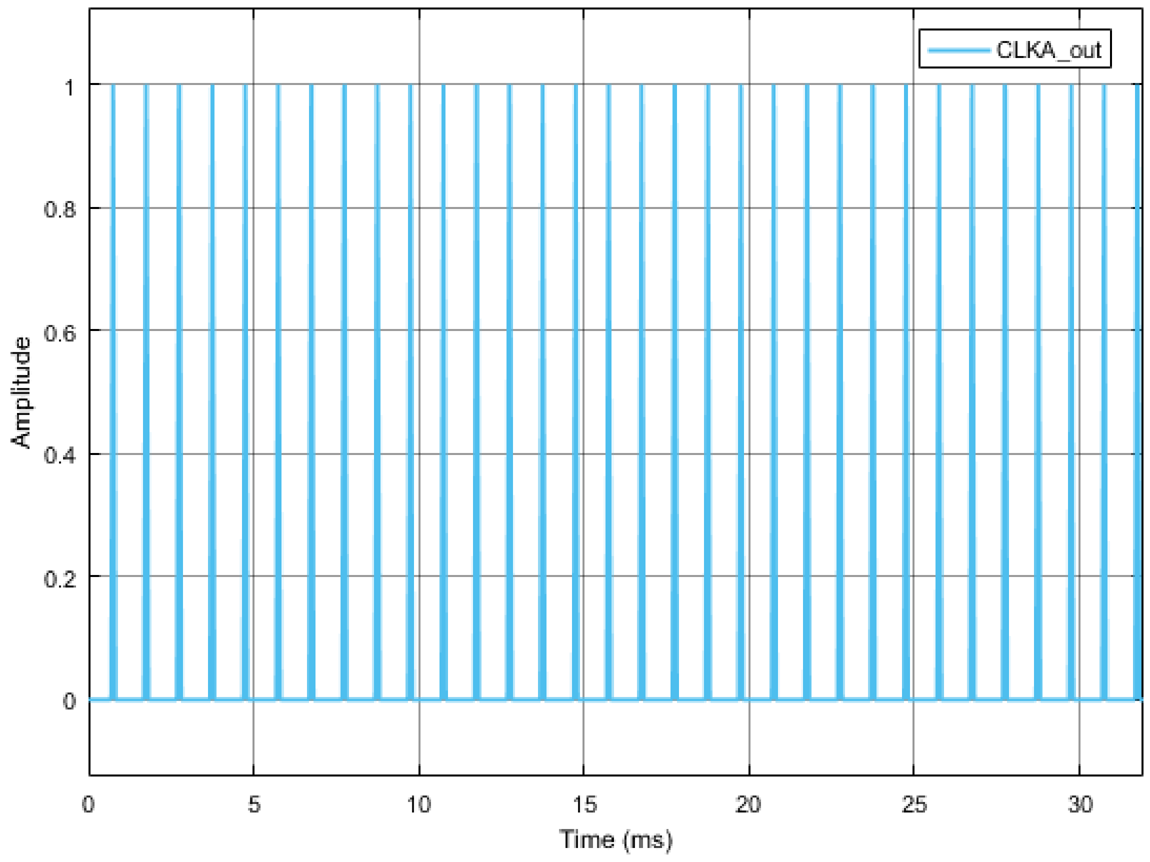


Figura 6.5: Example of the output signal of the clock with phase shift.

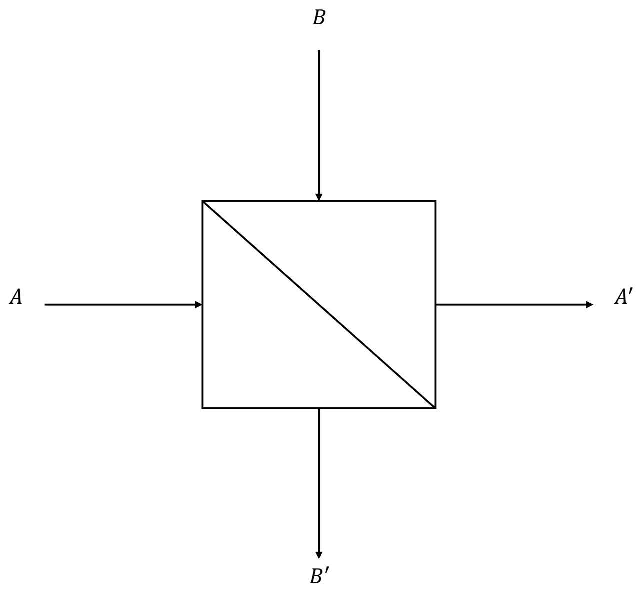


Figura 6.6: 2x2 coupler

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.

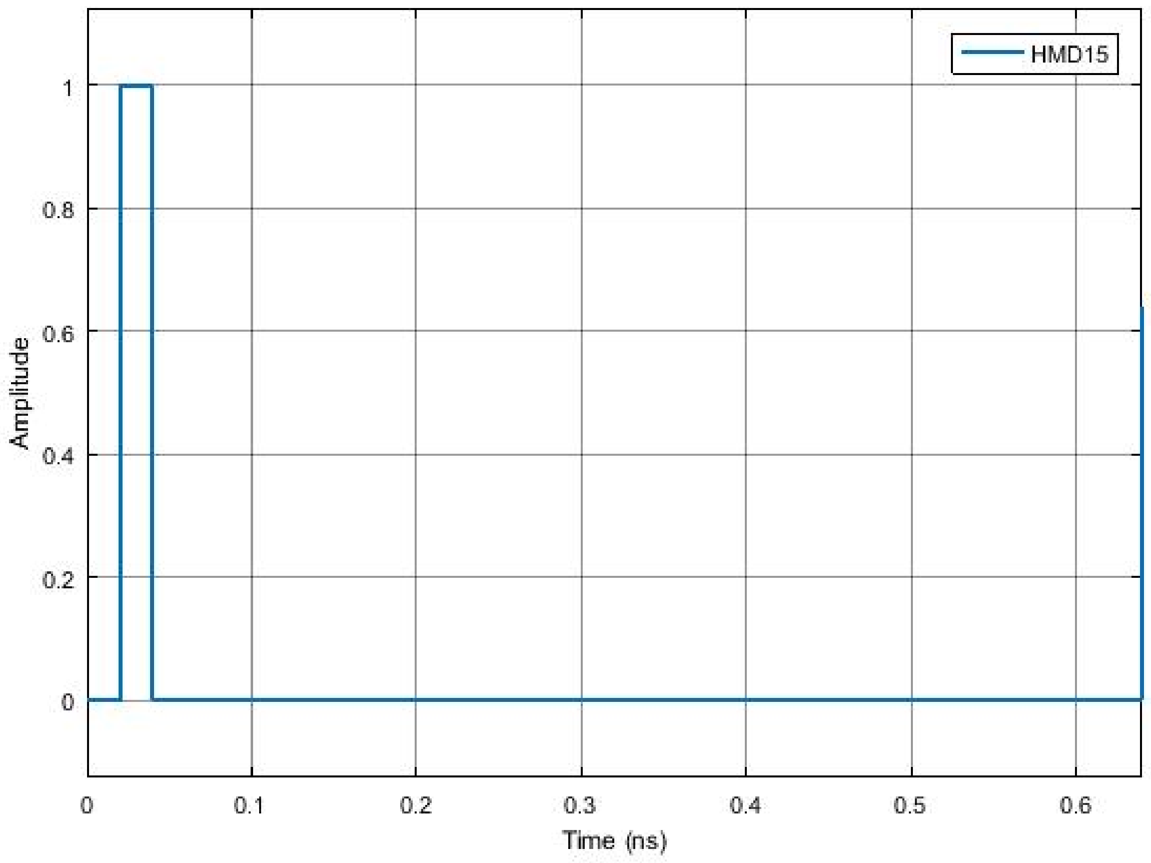


Figura 6.7: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

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

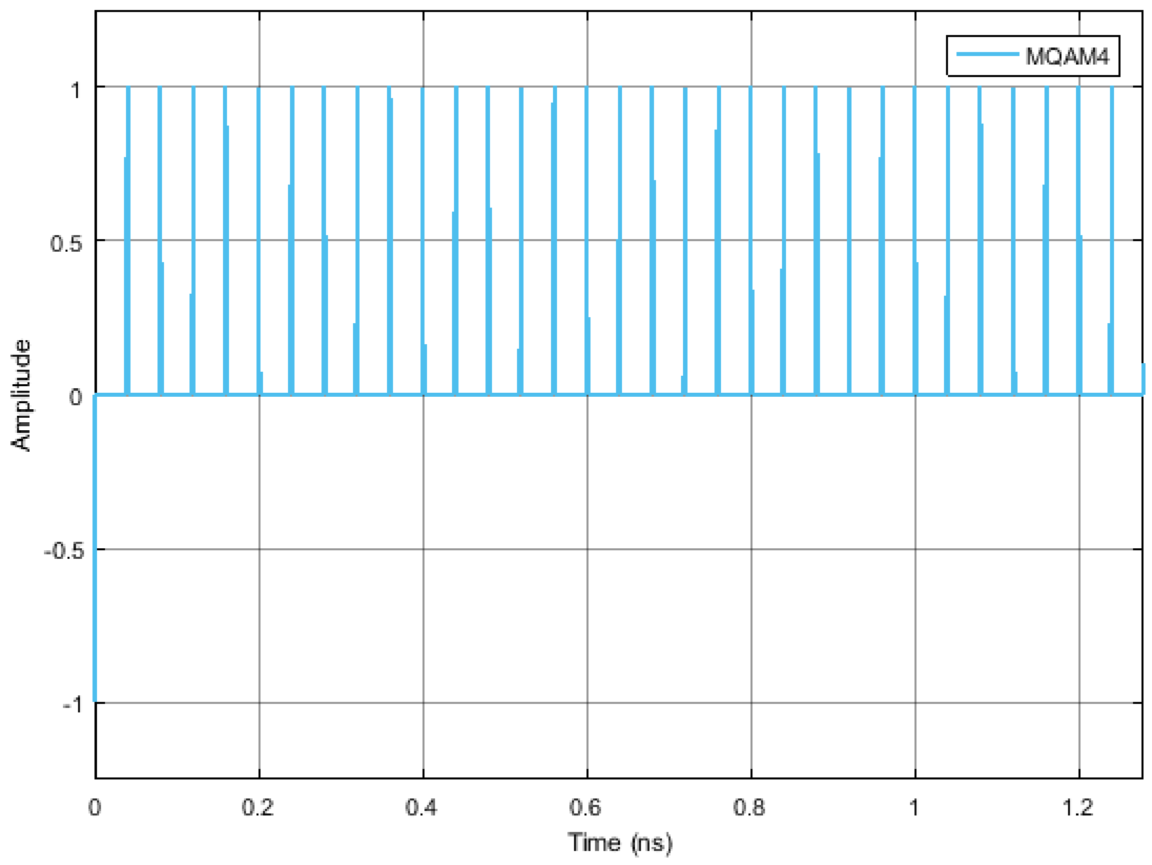


Figura 6.8: Example of the type of signal generated by this block for a binary sequence 0100...

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.

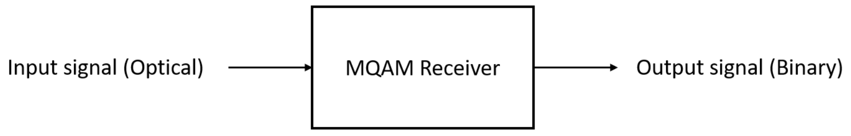


Figura 6.9: Basic configuration of the MQAM receiver

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*.

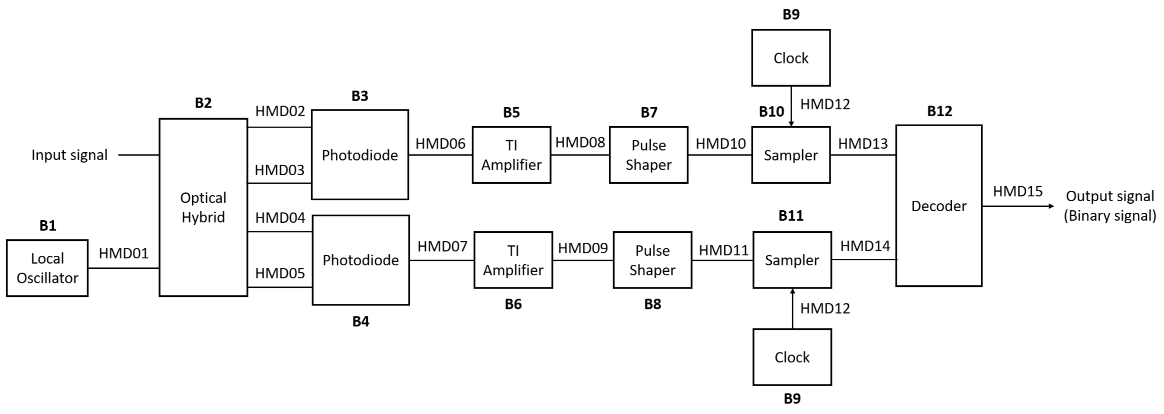


Figura 6.10: Schematic representation of the block homodyne receiver.

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)	∈ [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/samplesPerSym*

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

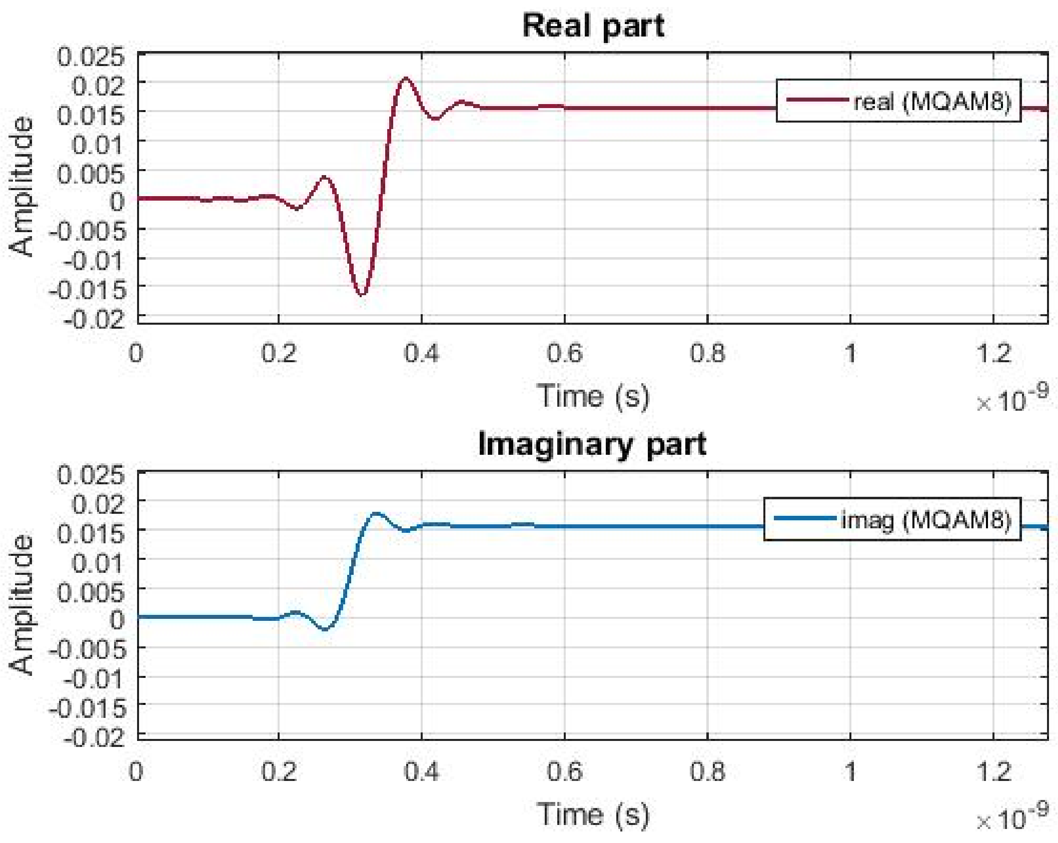


Figura 6.11: Example of a signal generated by this block for the initial binary signal 0100...

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

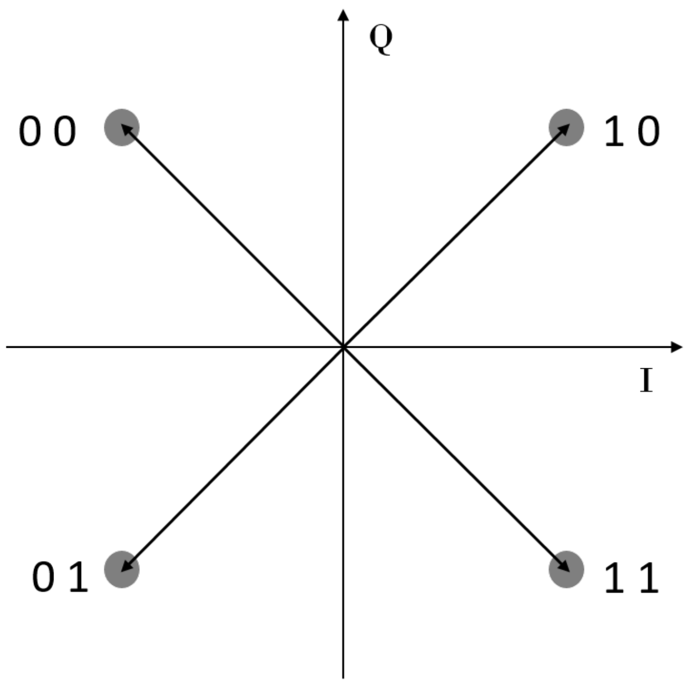


Figura 6.12: Constellation used to map the signal for m=4

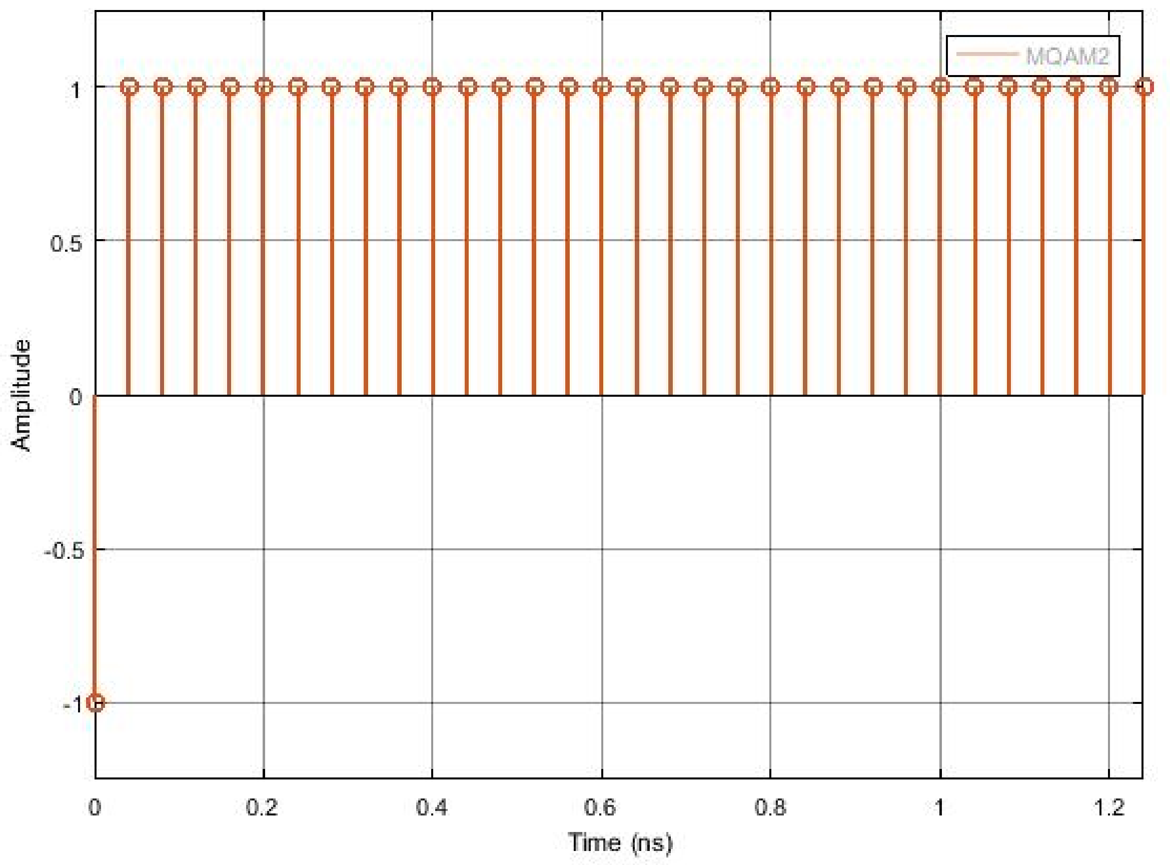


Figura 6.13: Example of the type of signal generated by this block for the initial binary signal 0100...

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.

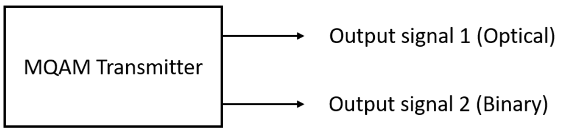
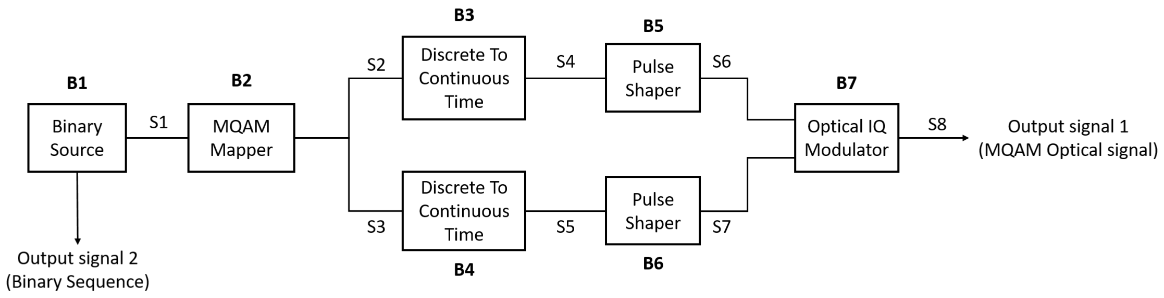


Figura 6.14: Basic configuration of the MQAM transmitter

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).



B1
Binary Source
S1
B2
MQAM Mapper
S2
S3
B3
Discrete To Continuous Time
B4
Discrete To Continuous Time
S4
S5
B5
Pulse Shaper
B6
Pulse Shaper
S6
S7
B7
Optical IQ Modulator
S8
Output signal 1 (MQAM Optical signal)
Output signal 2 (Binary Sequence)

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

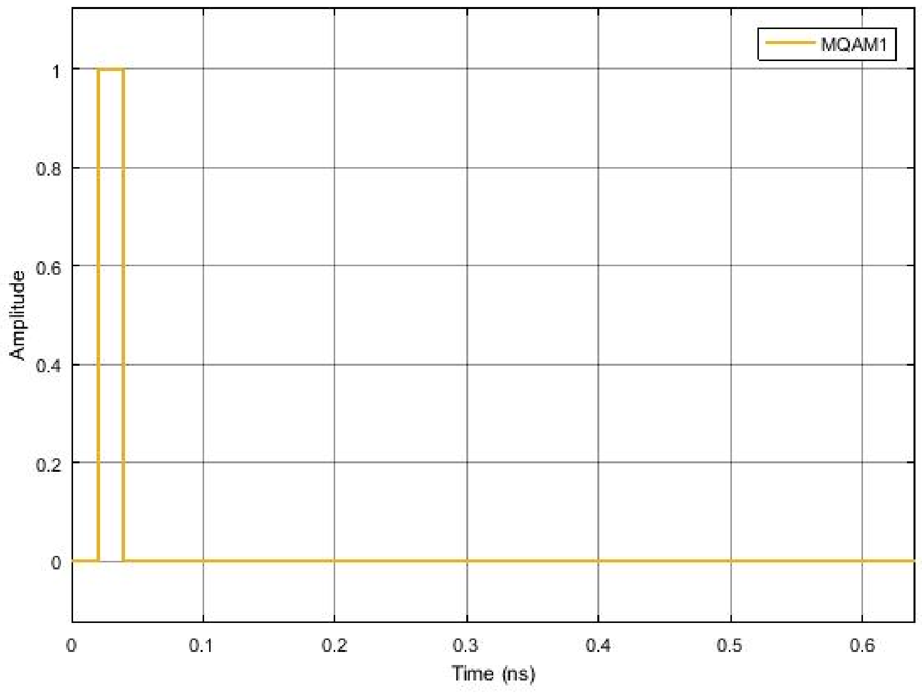


Figura 6.16: Example of the binary sequence generated by this block for a sequence 0100...

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.

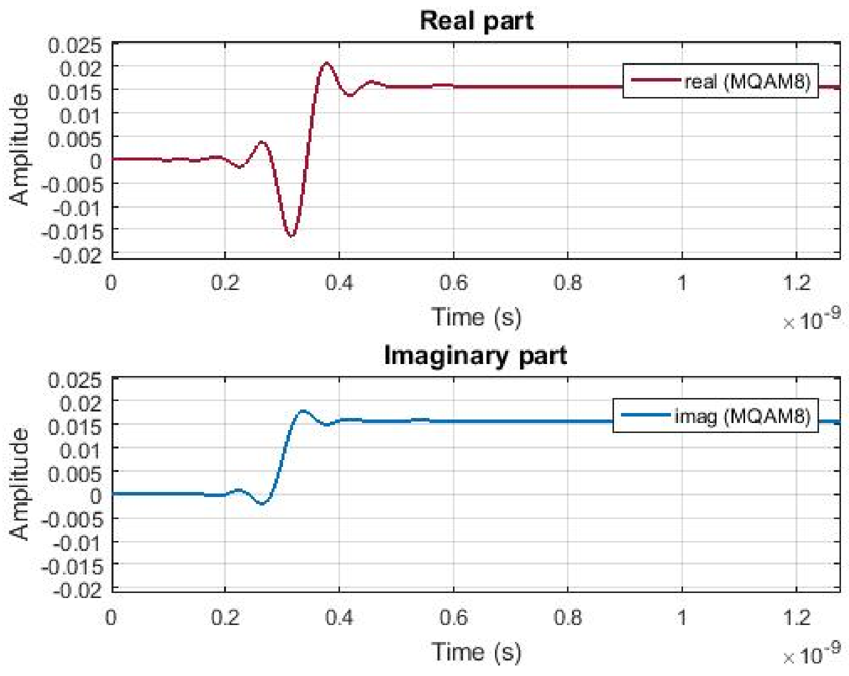


Figura 6.17: Example of the output optical signal generated by this block for a sequence 0100...

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_20170113

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 $^1 35^\circ$.

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 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

6.20 Electrical Signal Generator

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

6.20.1 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 $1 \times$ gain.

Input Parameters

Parameter: ElectricalSignalFunction (ContinuousWave) signalFunction

Parameter: samplingPeriod{} (double)

Parameter: 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.

6.21 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.

Input Parameters

Parameter: zscore
(double)

Parameter: 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:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \tag{6.7}$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \tag{6.8}$$

The error margin is calculated based on the z-score set which specifies the confidence interval using the following formula:

$$\mathrm{ME} = \mathrm{z}_{\mathrm{score}} \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \tag{6.9}$$

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

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).

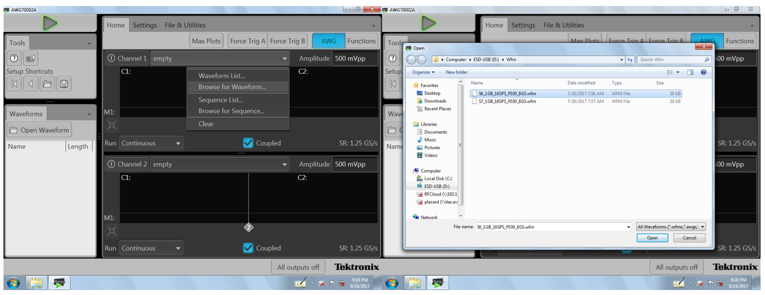


Figura 7.1: Selecting your waveform in the AWG

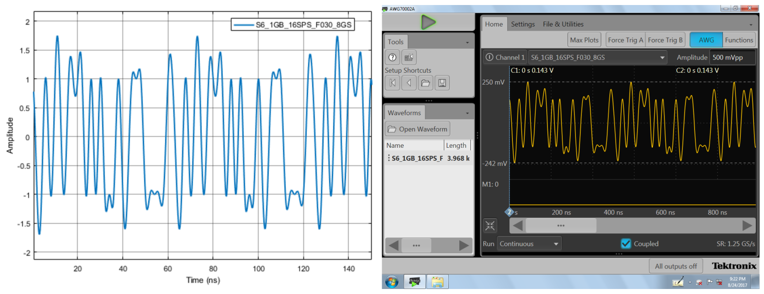
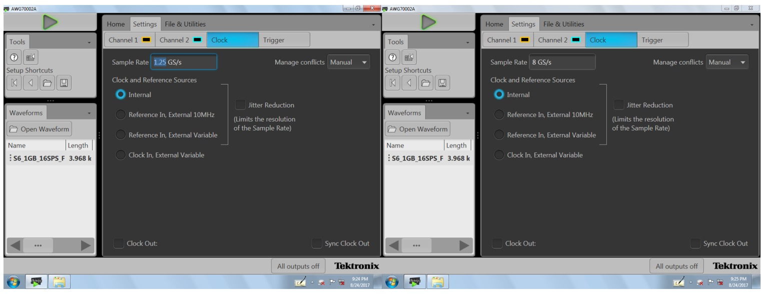


Figura 7.2: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

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



Sample Rate 1.25 GS/s — Manage conflicts Manual

Clock and Reference Sources: Internal; Reference In, External 10MHz; Reference In, External Variable; Clock In, External Variable

Jitter Reduction (Limits the resolution of the Sample Rate)

Sample Rate 8 GS/s — Manage conflicts Manual

Figura 7.3: Configuring the right sampling rate

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

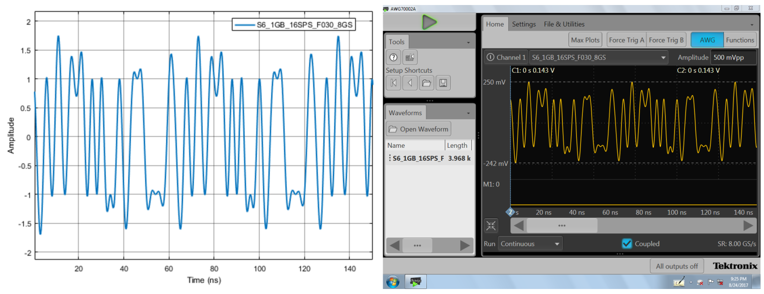


Figura 7.4: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

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