NetXPTO - LinkPlanner

8 de Março 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.

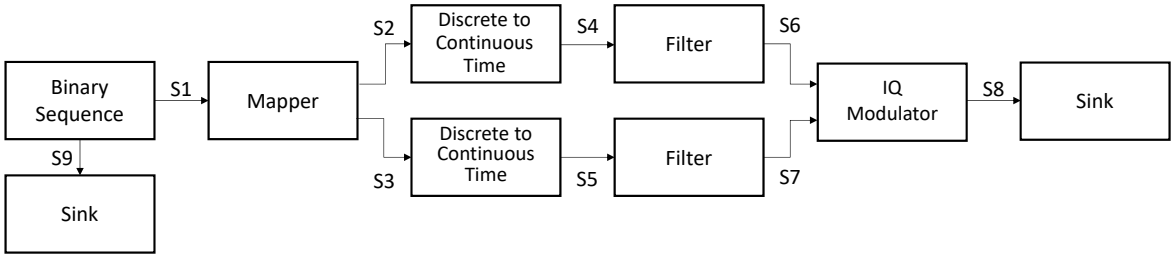


Figura 5.1: QPSK transmitter block diagram.

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 BPSK Transmission System

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

Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying (PSK), in which binary information is encoded into a two state constellation with the states being separated by a phase shift of π (see Figure 5.2).

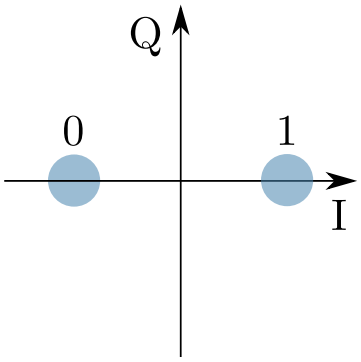


Figura 5.2: BPSK symbol constellation.

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 its spectral density equals its variance. For the purpose of this work, additive WGN is used to model thermal noise at the receivers.

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

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 trans-impedance amplifier in the coherent receiver and

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

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

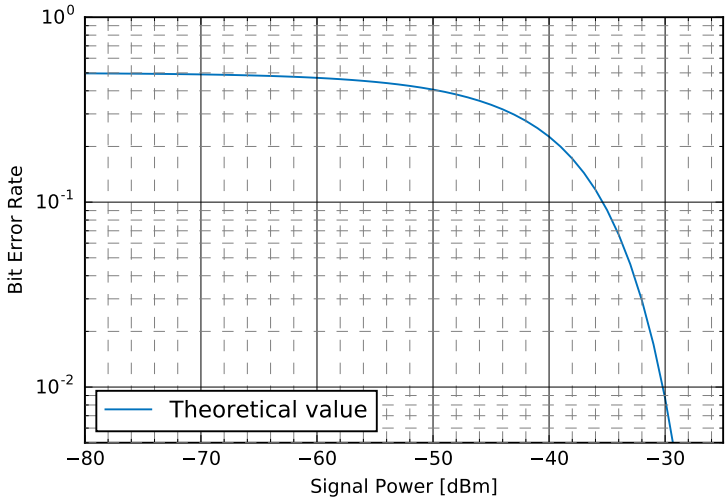


Figura 5.3: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

5.2.2 Simulation Analysis

A diagram of the system being simulated is presented in the Figure 5.4. 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.

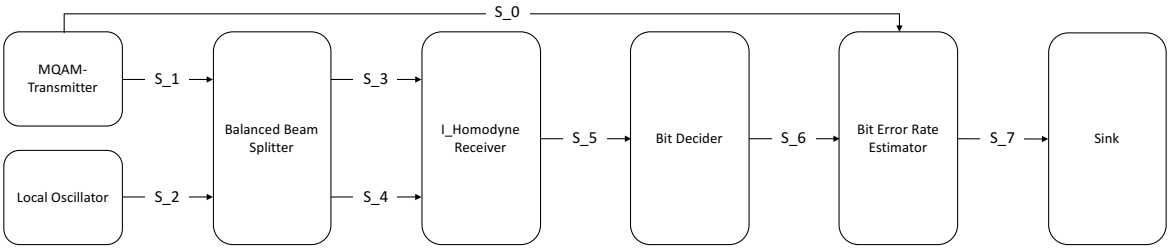


Figura 5.4: Overview of the BPSK system being simulated.

Required files

Header Files		
File	**Comments**	**Status**
add.h		✓
balanced_beam_splitter.h		✓
binary_source.h		✓
bit_decider.h		✓
bit_error_rate.h		✓
discrete_to_continuous_time.h		✓
netxpto.h		✓
m_qam_mapper.h		✓
m_qam_transmitter.h		✓
local_oscillator.h		✓
i_homodyne_reciever.h		✓
ideal_amplififer.h		✓
iq_modulator.h		✓
photodiode.h		✓
pulse_shaper.h		✓
sampler.h		✓
sink.h		✓
super_block_interface.h		✓
white_noise.h		✓

Source Files		
File	**Comments**	**Status**
add.cpp		✓
balanced_beam_splitter.cpp		✓
binary_source.cpp		✓
bit_decider.cpp		✓
bit_error_rate.cpp		✓
discrete_to_continuous_time.cpp		✓
netxpto.cpp		✓
m_qam_mapper.cpp		✓
m_qam_transmitter.cpp		✓
local_oscillator.cpp		✓
i_homodyne_reciever.cpp		✓
ideal_amplififer.cpp		✓
iq_modulator.cpp		✓
photodiode.cpp		✓
pulse_shaper.cpp		✓
sampler.cpp		✓
sink.cpp		✓
super_block_interface.cpp		✓
white_noise.cpp		✓

System Input Parameters

This system takes into account the following input parameters:

System Input Parameters		
Parameter	**Default Value**	**Comments**
numberOfBitsGenerated	40000	
bitPeriod	20×10^{-12}	
samplesPerSymbol	16	
pLength	5	
iqAmplitudesValues	$\{\{-1,\ 0\}\ ,\ \{1,\ 0\}\ \}$	
outOpticalPower_dBm	Variable	Value varied from -75 dBm to -25 dBm with intervals of 5 dBm
loOutOpticalPower_dBm	0	
localOscillatorPhase	0	
transferMatrix	$\{\{\frac{1}{\sqrt{2}},\ \frac{1}{\sqrt{2}},\ \frac{1}{\sqrt{2}},\ \frac{-1}{\sqrt{2}}\}\ \}$	
responsivity	1	
amplification	10^3	
noiseSpectralDensity	$5 \times 10^{-4}\sqrt{2}\ \mathrm{V}^2$	
confidence	0.95	
midReportSize	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)

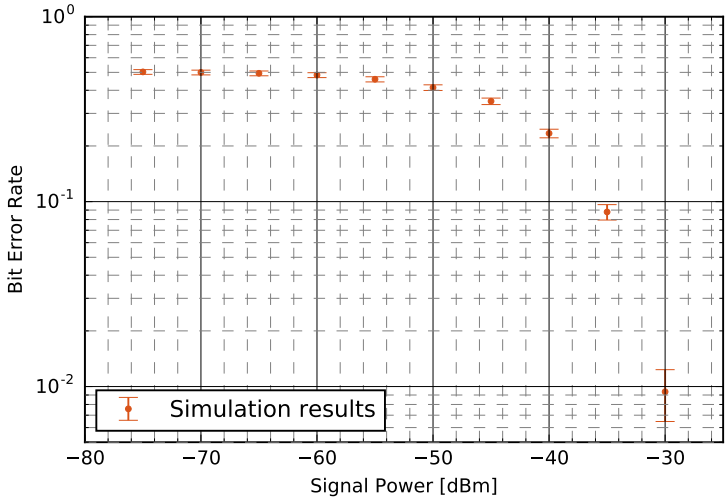


Figura 5.5: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

5.2.3 Comparative Analysis

The following results show the dependence of the error rate with the signal power assuming a constant Local Oscillator power of $0\ dBm$, the signal power was evaluated at levels between -70 and -25 dBm, in steps of 5 dBm between each. The simulation results are presented in orange with the computed lower and upper bounds, while the expected value, obtained from (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].

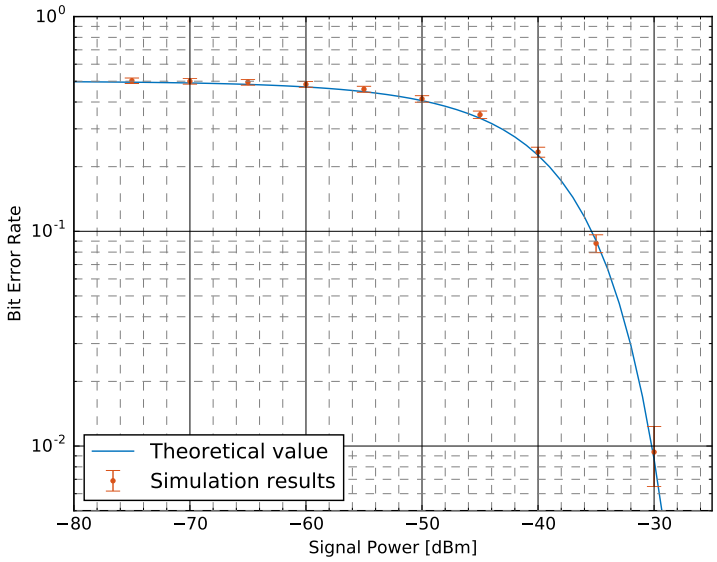


Figura 5.6: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 6.3

Bibliografia

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

[2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.

[3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

[4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.

[5] Yue-Meng Chi, Bing Qi, Wen Zhu, Li Qian, Hoi-Kwong Lo, Sun-Hyun Youn, AI Lvovsky, and Liang Tian. A balanced homodyne detector for high-rate gaussian-modulated coherent-state quantum key distribution. *New Journal of Physics*, 13(1):013003, 2011.

[6] 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		Andoni Santos (2018/01/03 -) Ana Luisa Carvalho (2017/04/01 - 2017/12/31)
Goal	:	M-QAM system implementation with BER measurement and comparison with theoretical and experimental values.
Directory	:	sdf/m_qam_system

The goal of this project is to simulate a Quadrature Amplitude Modulation transmission system with M points in the constellation diagram (M-QAM) and to perform a Bit Error Rate (BER) measurement that can be compared with theoretical and experimental values.

M-QAM systems can encode $\log_2 M$ bits per symbol which means they can transmit higher data rates keeping the same bandwidth when compared, for example, to PSK systems. However, because the states are closer together, these systems require a higher signal-to-noise ratio. The Bit Error Rate (BER) is a measurement of how a bit stream is altered by a transmission system due to noise (among other factors). To study this effect we introduced Additive White Gaussian Noise (AWGN) to model thermal noise at the receiver.

For $M = 4$ the M-QAM system can be reduced to a Quadrature Phase Shift Keying system (QPSK) system that uses four equispaced points in the constellation diagram (see figure 5.7).

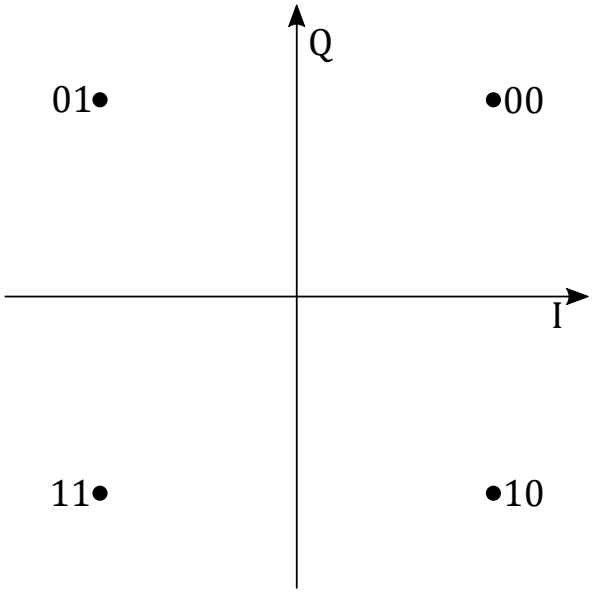


Figura 5.7: 4-QAM constellation points.

5.3.1 Theoretical Analysis

M-QAM is a modulation scheme that takes advantage of two sinusoidal carriers with a phase difference of $\pi/2$. The resultant output consists of a signal with both amplitude and phase variations. The two carriers, referred to as I (In-phase) and Q (Quadrature), can be represented as

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

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 cosine of a sum and the definitions of I and Q.

For the particular case of $M = 4$, considering there is no crosstalk or interference between the Q and I components, the signal can be treated as a pair of independent BPSK systems, one for the in-phase component and another for the quadrature component. Using Gray coding, adjacent symbols differ by only one bit. As such, an error in a single component leads to a single bit error. This means that the probability of an error in bit detection is independent among components, as there is no crosstalk or interference. That being the case, the bit error rate can be calculated for the BPSK case.

Let the $s(t)$ be the signal sampled at a given instant t for either the in-phase or quadrature component, $A(t)$ be the component corresponding to the transmitted signal with value equal to $\pm A$, depending on whether the transmitted signal was 1 or 0, and $n(t)$ the component associated with the Gaussian white noise with variance $n_0/2$, such that

$$s(t) = A(t) + n(t) \tag{5.10}$$

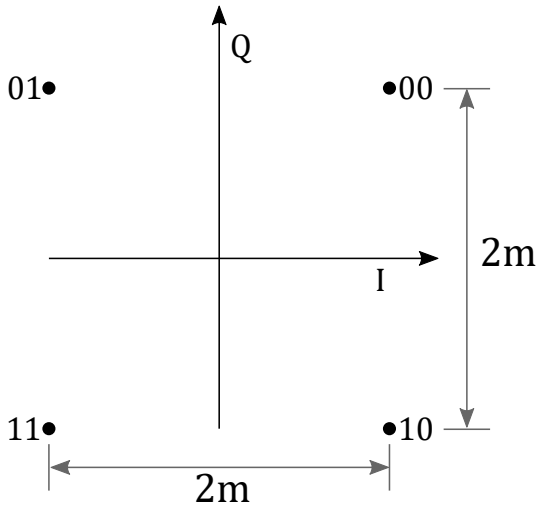


Figura 5.8: The relation between m and the distance between constellation points.

In this case, assuming the absence of inter-symbol interference, $s(t)$ will be a Gaussian random variable with average value of $\pm A$, depending on what signal was transmitted, and variance equal to $n_0/2$. In this case, using the constellation from Figure 5.8, with a decision boundary halfway between A and $-A$, an error occurs in two situations: when the a 0 is transmitted but a 1 is identified, or a 1 is transmitted and a 0 is identified. These mistakes happen when the perturbation in the signal due to noise is enough to push the value beyond the decision boundary. These situations are shown in Figures 5.9 and 5.10, by the colored area under the curve.

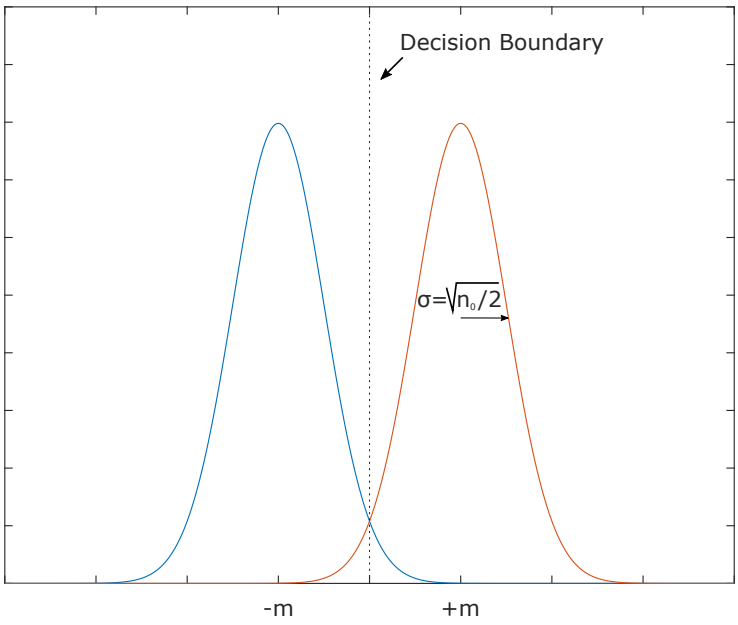


Figura 5.9: Probability density functions for $s(t) = m(t) + n(t)$, with $m(t) = \pm m$ and $n(t)$ as a Gaussian variable.

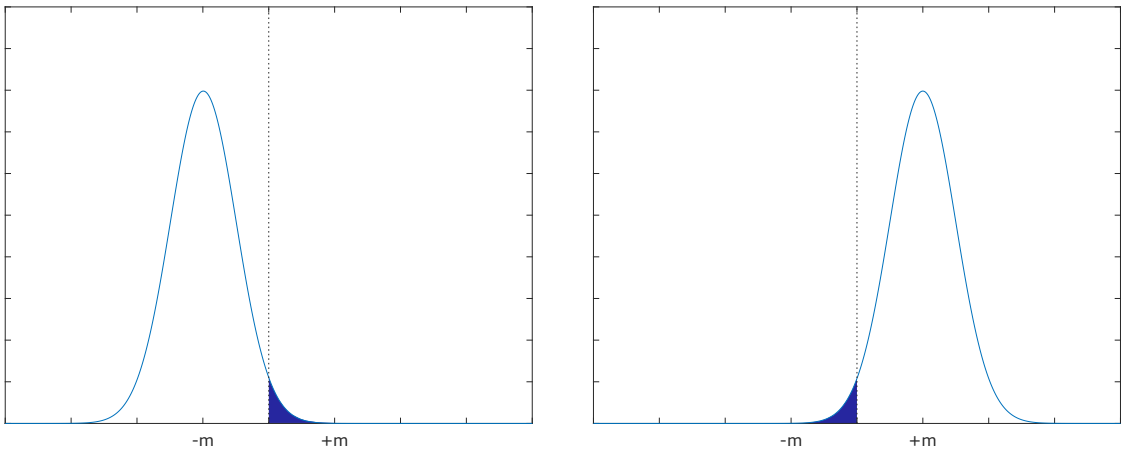


Figura 5.10: The area below the curves represents the probability of error for each transmitted bit.

The probability of bit error can be expressed as:

$$P_{be} = P_0 P_{e0} + P_1 P_{e1} \tag{5.11}$$

With equal probability for both bits, and considering the constellation's symmetry

$$P_{be} = Q\left(\frac{A}{N}\right) = \frac{1}{2}\text{erfc}\left(\frac{A}{\sqrt{2}N}\right) = \frac{1}{2}\text{erfc}\left(\frac{A}{\sqrt{n_0}}\right) \tag{5.12}$$

with

$$A = G_{ele}\sqrt{P_L P_S} \tag{5.13}$$

$$N = \sqrt{\frac{n_0}{2}} \tag{5.14}$$

where P_L is the local oscillator power, P_S is the average optical power of the laser source on which the signal is modulated, G_{ele} is the gain in the transimpedance amplifier and n_0 is the noise spectral density. Figure 5.8 shows the relation between the constellation points and A. N_0 will be the standard deviation of the sampled values.

The symbol error rate however is not the same, as it depends on both bits being correctly detected. The probability of both bits being correctly detected is:

$$P_C = (1 - P_{be})^2 \tag{5.15}$$

From this, the probability of symbol error is:

$$\begin{aligned} P_s &= 1 - P_C \\ &= 1 - \left(1 - Q\left(\frac{A}{n_0}\right)\right)^2 \\ &= 2Q\left(\frac{A}{n_0}\right)\left[1 - \frac{1}{2}Q\left(\frac{A}{n_0}\right)\right] \end{aligned} \tag{5.16}$$

It is possible to further decrease the error rate by using a matched filter before sampling the signal. The resulting signal will still have Gaussian noise, but the signal-to-noise ratio will be greatly improved. This can be achieved by using a root-raised cosine filter at the pulse shaper and another one before the sampler. Inter-symbol interference will still be null as it is equivalent to a raised cosine filter, where half the filtering is done on the transmitter side (while pulse-shaping) and the other half is done on the receiver side, before sampling.

Considering a signal similar to the one described by Equation 5.10, the output of the matched filter receiver will be [**?**]:

$$\begin{aligned} s_{mf}(t) &= \int_0^T s(t)\cos(\omega_0 t)dt \\ &= \int_0^T m(t)\cos(\omega_0 t)dt + \int_0^T n(t)\cos(\omega_0 t)dt \end{aligned} \tag{5.17}$$

In the case of QPSK with m=4, both the quadrature and in-phase components have $m(t) = \pm A$. The values that replace m and N_0 in Equation 5.12 become:

$$A_{mf} = \frac{AT}{2} = \frac{G_{ele}\sqrt{P_L P_S}T}{2} \tag{5.18}$$

$$N_{mf} = \sqrt{\sigma_o^2} = E(n_o^2) = \sqrt{\frac{n_0 T}{4}} \tag{5.19}$$

Here, P_L is the local oscillator power, P_S is the average optical power of the laser source on which the signal is modulated, G_{ele} is the gain in the transimpedance amplifier, T is the symbol period, A_mf is the average amplitude at the sampling points of the signal after amplification without noise, σ_o^2 is the variance of the noise component of the matched filter output, the and n_0 is the noise spectral density.

The optimal BER that can be obtained by using matched filtering is then given by:

$$\begin{aligned} P_{be} &= \frac{1}{2}Q\left(\frac{A_{mf}}{\sqrt{2}N_{mf}}\right) \\ &= \frac{1}{2}\text{erfc}\left(\frac{A_{mf}}{\sqrt{2}N_{mf}}\right) \\ &= \frac{1}{2}\text{erfc}\left(\sqrt{\frac{A^2T}{2n_0}}\right) \end{aligned} \tag{5.20}$$

(5.21)

with

$$A = G_{ele}\sqrt{P_L P_S} \tag{5.22}$$

where T is the symbol period, A is as defined in Equation 5.13, the average peak amplitude of the signal without noise prior to filtering.

Figure 5.11 show the curves for both these equations, in the case where $n_0 = 10^{-6}$, $P_L = 0\ dBm$ and $G_{ele} = 10^3$.

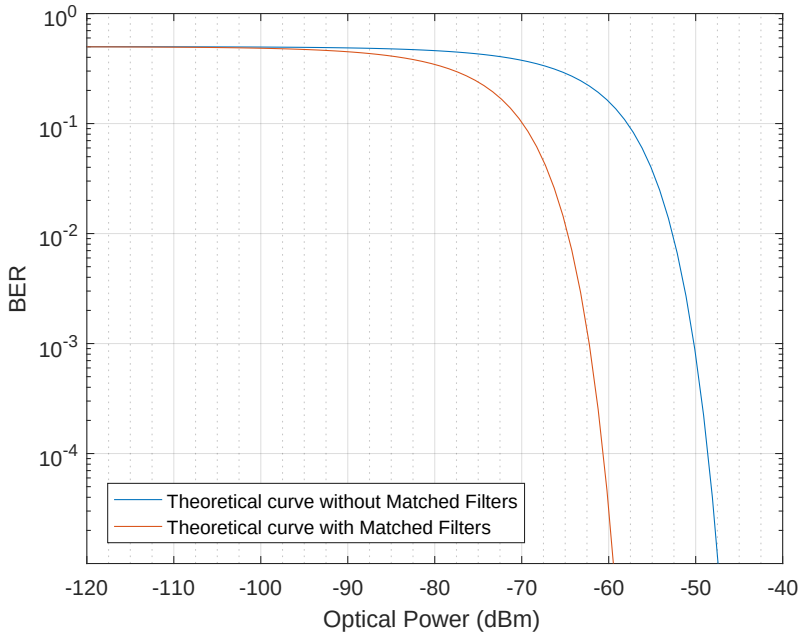


Figura 5.11: QPSK theoretical BER values as a function of the output optical power in dBm.

Figure 5.11 shows the two theoretical curves for QPSK. The blue one is for QPSK without matched filtering and the red one is using root-raised-cosine for matched filtering, which provides optimum transmission. As the use of root-raised-cosine for matched filtering maximizes the signal-to-noise ratio to its optimal value, it allows achieving the same BER with much lower optical power. On the blue curve, on the other hand, the sampling is affected by the full effect of the random noise, as there is no filtering at the receiver. Because of this, a higher optical power is required to achieve a similar BER.

It's worth noting that these equations are only valid for M=4, as in that case the system is similar to QPSK with a 4 point constellation. For $M > 4$ a different approach is required.

The use of matched filtering should allow transmission with a lower SNR to achieve the same results as a similar system, even when using a shape with no inter-symbol interference. This is due to the second filter reducing the noise impact before detection, while not causing inter-symbol interference or degradation of the signal data.

5.3.2 Simulation Analysis

The M-QAM 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. The schematic representation of the system is presented in figure 5.12.

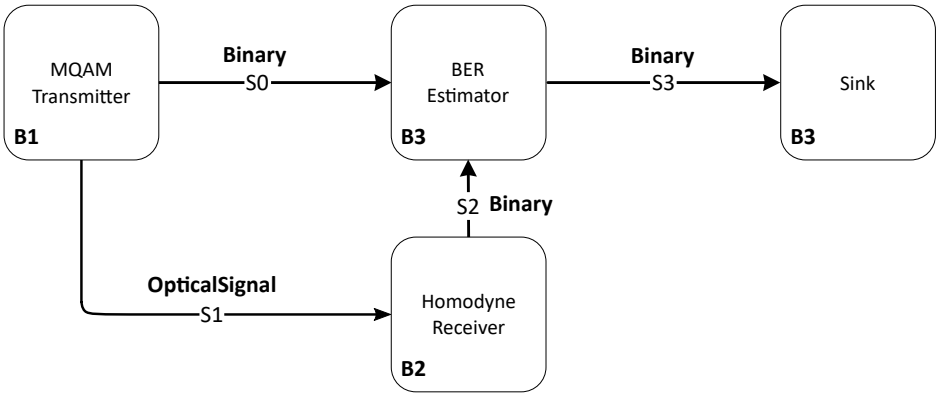


Figura 5.12: Schematic representation of the MQAM system.

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

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

Functional description

A complete description of the M-QAM transmitter and M-QAM homodyne 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 generates one or two optical signals by encoding a binary string using M-QAM modulation. It also outputs a binary signal that is used to perform the BER measurement.

The M-QAM homodyne receiver 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 binary one produced by the transmitter in order to estimate the Bit Error Rate (BER).

The files used are summarized in tables 5.1 and 5.2. These include all blocks and sub-blocks used and allow for the full operation of the M-QAM system.

Required files

Source Files		
File	**Comments**	**Status**
add_20171116.cpp		✓
binary_source_20180118.cpp		✓
bit_error_rate_20171810.cpp		✓
decoder_20180118.cpp		✓
discrete_to_continuous_time_20180118.cpp		✓
homodyne_receiver_20171203.cpp	[1]	✓
ideal_amplififer_20180118.cpp		✓
iq_modulator_20180118.cpp		✓
local_oscillator_20180118.cpp		✓
m_qam_mapper_20180118.cpp		✓
m_qam_system.cpp	[2]	✓
m_qam_transmitter_20180118.cpp		✓
netxpto_20180118.cpp	[2]	✓
optical_hybrid_20180118.cpp		✓
photodiode_old_20180118.cpp		✓
pulse_shaper_20180118.cpp		✓
sampler_20171119.cpp		✓
sink_20180118.cpp		✓
super_block_interface_20180118.cpp	[2]	✓
white_noise_20180118.cpp		✓

Tabela 5.1: [1] The library entry is under a different name, *m_qam_receiver*;
[2] No library entry as it is a main or general purpose file, not a specific block.

Header Files		
File	**Comments**	**Status**
add_20171116.h		✓
binary_source_20180118 .h		✓
bit_error_rate_20171810.h		✓
decoder_20180118.h		✓
discrete_to_continuous_time_20180118.h		✓
homodyne_receiver_20171203.h	[1]	✓
ideal_amplifier_20180118.h		✓
iq_modulator_20180118.h		✓
local_oscillator_20180118.h		✓
m_qam_mapper_20180118.h		✓
m_qam_transmitter_20180118.h		✓
netxpto_20180118.h	[2]	✓
optical_hybrid_20180118.h		✓
photodiode_old_20180118.h		✓
pulse_shaper_20180118.h		✓
sampler_20171119.h		✓
sink_20180118.h		✓
super_block_interface_20180118.h	[2]	✓
white_noise_20180118.h		✓

Tabela 5.2: [1] The library entry is under a different name, *m_qam_receiver*
[2] No library entry as it is a main or general purpose file, not a specific block.

Input Parameters

Tabela 5.3: Input parameters

Parameter	Type	Description
numberOfBitsGenerated	t_integer	Determines the number of bits to be generated by the binary source
samplesPerSymbol	t_integer	Number of samples per symbol
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_shp	t_real	Parameter of the pulse shaper filter
rollOffFactor_out	t_real	Parameter of the output filter
shaperFilter	enum	Type of filter used in Pulse Shaper
outputFilter	enum	Type of filter used in output filter
seedType	enum	Method of seeding noise generators
seedArray	array<int,2>	Seeds to initialize noise generators
signalOutputPower_dBm	t_real	Determines the power of the output optical signal in dBm
numberOfBitsReceived	int	Determines when the simulation should stop. If -1 then it only stops when there is no more bits to be sent
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	Amplification provided by the ideal amplifier
noiseAmplitude	t_real	Amplitude of the white noise
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

Simulation results

In this section we show results obtained through the simulations. The following sections show the eye diagrams of the signal obtained at three different points in the system: the optical signal S1, the signals after amplifying the electrical signal and adding the Gaussian white noise HMD12 and HMD13, and the signal after the filter in the receiver. These eye diagrams will be shown for a variety of system configurations, with varying noise and filters, displaying the advantages of using matched filtering with an optimal filter.

Inter-symbol Interference

These section presents the mentioned eye diagrams in configurations without any noise added to the signal. This allows studying the effects of inter-symbol interference caused only by the filters used at the pulse-shaper and at the receiver,

Figure 5.13 shows the eye diagrams for the S1 optical signals for two different values of the output optical power. These were obtained using a raised-cosine filter as a pulse shaper, with a roll-off factor of 0.9. It can be seen that no inter-symbol interference is present.

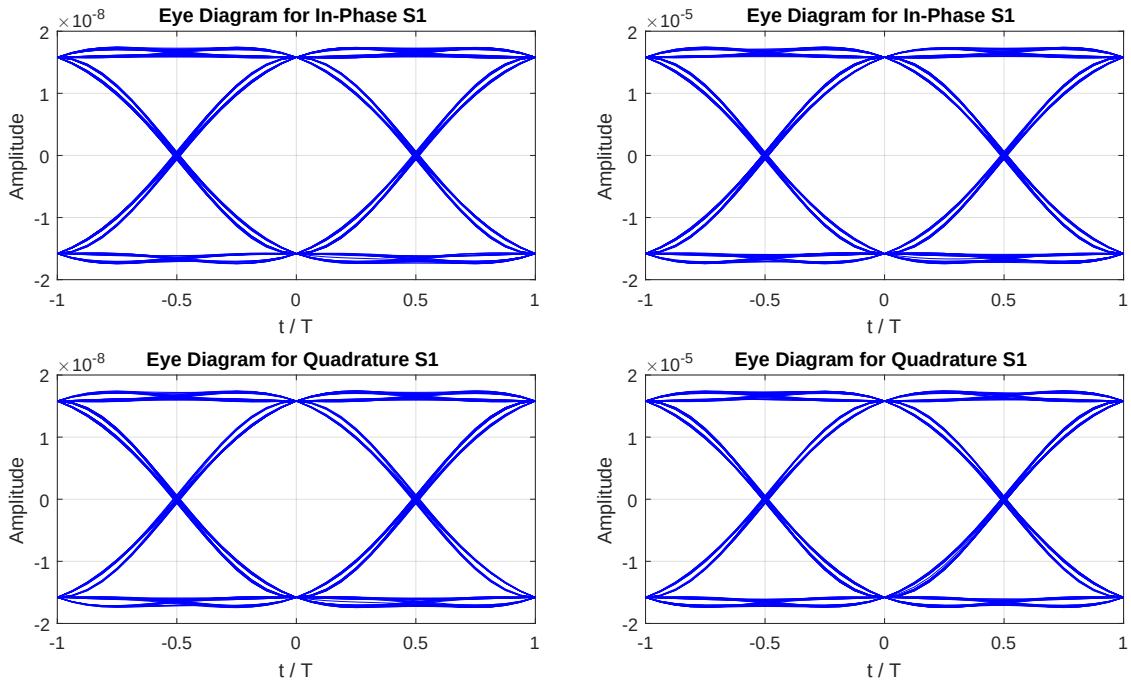


Figura 5.13: Eye diagrams for the S1 bandpass signal with an output optical power of -120dBm (left) and -60dBm (right) with no added noise.

As mentioned previously, using matched filters in highly beneficial, as it allows achieving much lower error rates with the same optical power.

Figure 5.14 shows the eye diagrams of the signal at the three mentioned points, for a system without any added white noise, while using matched filtering. The filter used in

this case is a raised cosine filter with a roll-off factor of 0.9. Although this is the ideal filter to use for pulse shaping, as it does not cause inter-symbol-interference, it can be seen that when used twice its results are less than ideal, as shown in the eye diagram captured after the second filter.

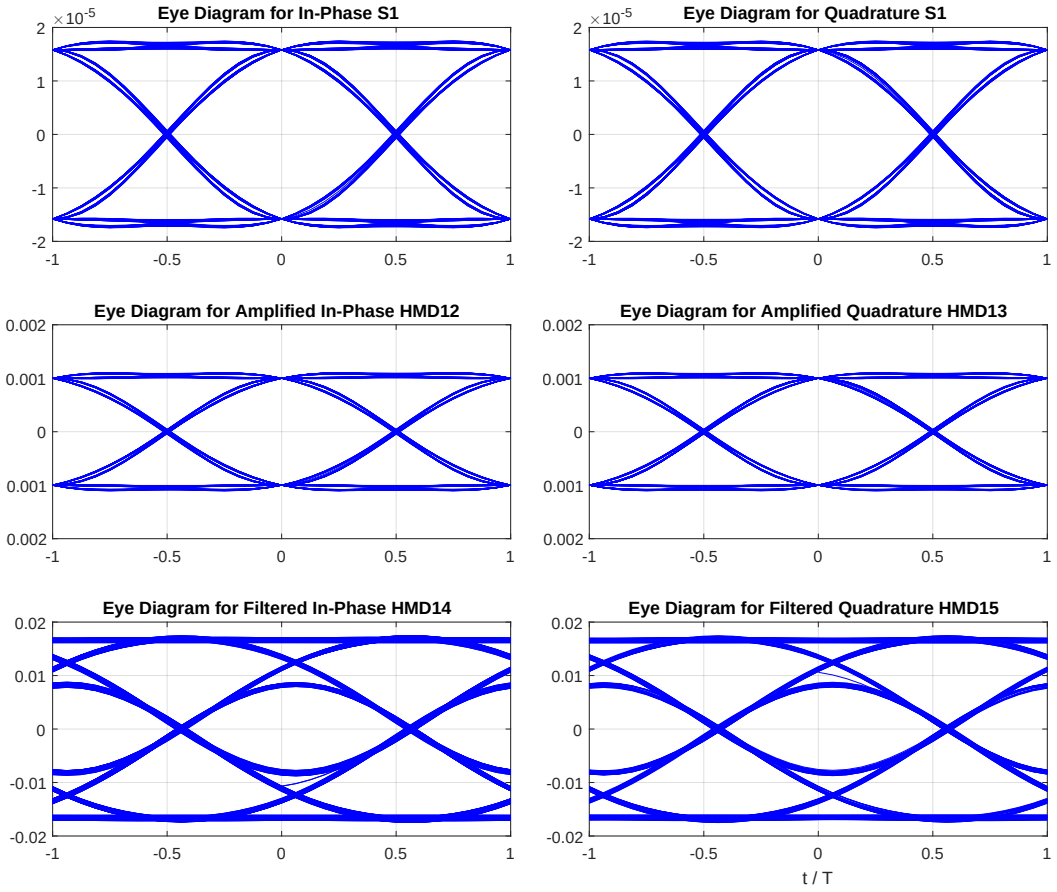


Figura 5.14: Eye diagrams using matched filtering with raised-cosine at three different points, without AWGN: the optical output signal S1 on the top; the amplified signal at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, and a rolloff factor of 0.9.

The optimum solution to achieving no inter-symbol interference while using matched filtering is to use a root-raised-cosine to do the pulse-shaping at the transmitter and the filtering at the receiver. The corresponding output of applying twice a root-raised-cosine is exactly the same as using a raised-cosine once. As such, the end result suffers from no inter-symbol interference while reaping the benefits of optimum matched filtering. Figure 5.15 shows the eye diagrams when using root-raised-cosine filter both in the transmitter's pulse-shaper and at the receiver's filter. The roll-off factor used in both was 0.9. It can be seen that

the shape of the eye diagram is equal to that of Figure 5.13, which uses a single raised cosine filter at the pulse-shaper. Thus, it shows no sign of inter-symbol interference.

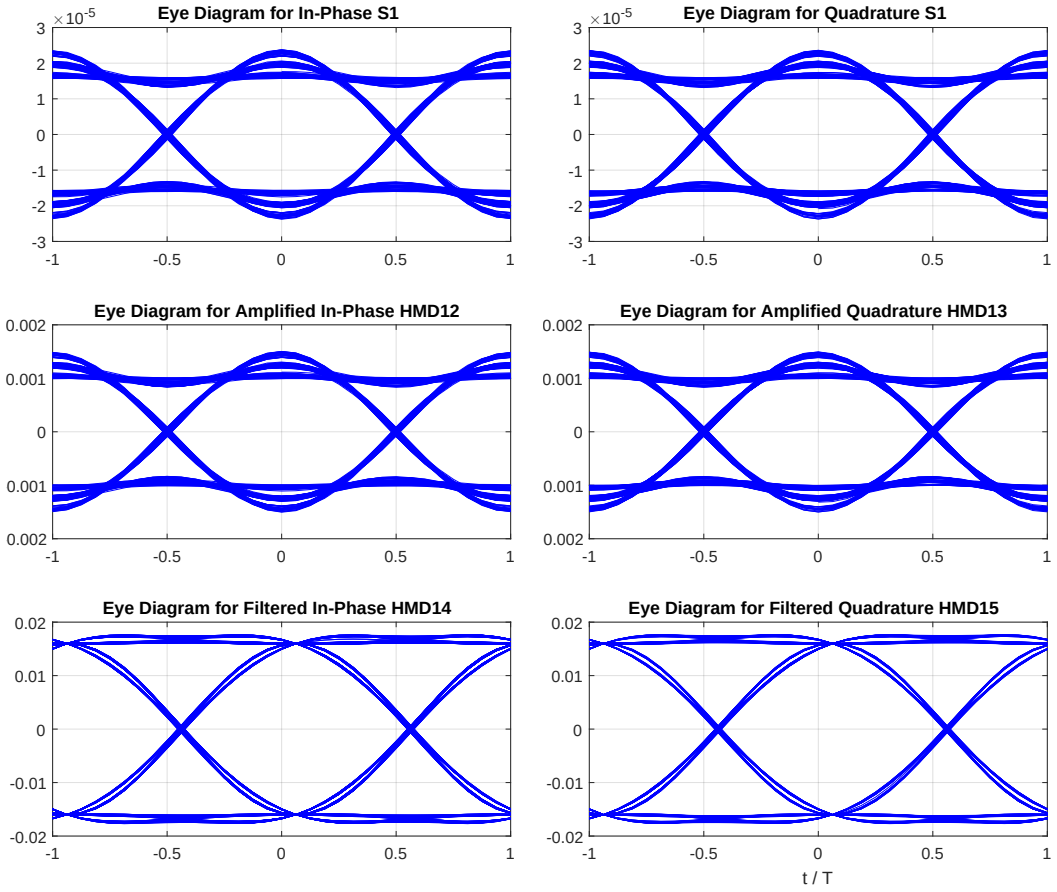


Figura 5.15: Eye diagrams using matched filtering with root-raised-cosine at three different points, without AWGN: the optical output signal S1 on the top; the amplified signal at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, and a rolloff factor of 0.9.

Figures 5.17 and 5.17 show a similar comparison between matched filtering using raised-cosine or root-raised-cosine filters, but with a roll-off factor of 0.3. Again, it can be seen that the final shape of the eye diagram when using the root-raised-cosine for matched filtering is the same as the shape of the optical signal S1 when using a raised-cosine-filter.

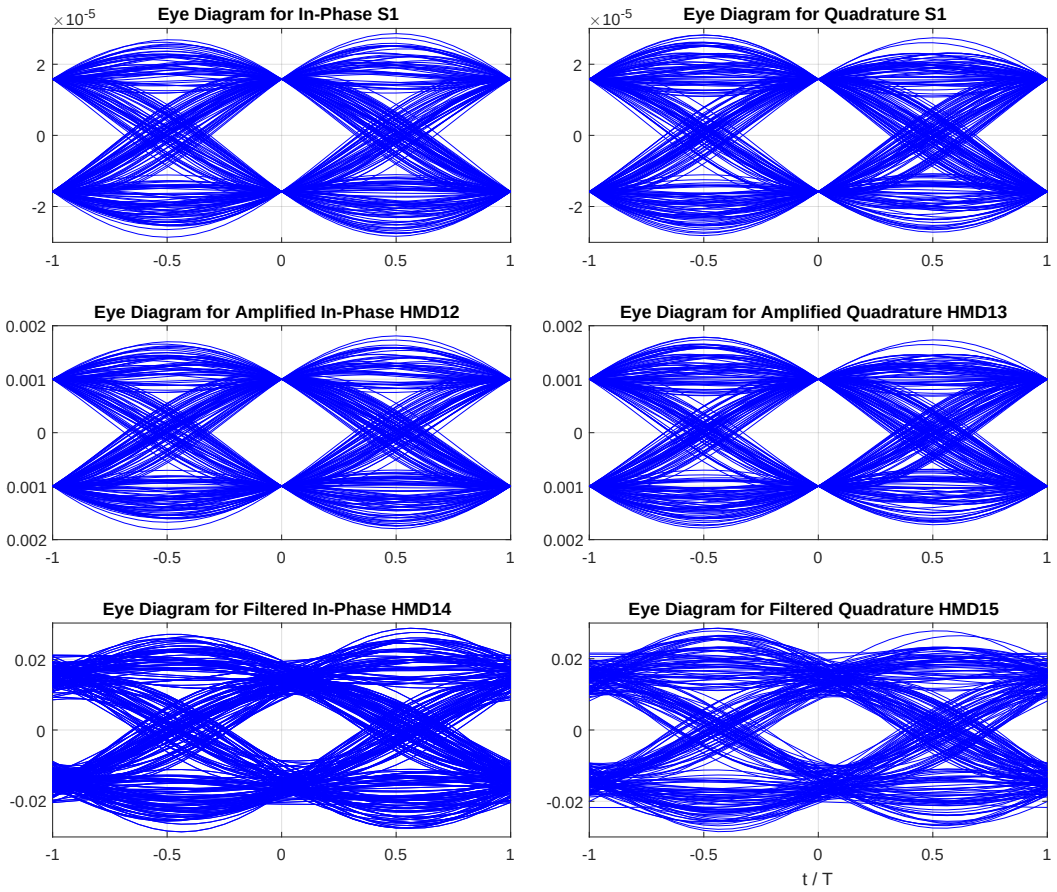


Figura 5.16: Eye diagrams using matched filtering with raised-cosine at three different points, without AWGN: the optical output signal S1 on the top; the amplified signal at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, and a rolloff factor of 0.3.

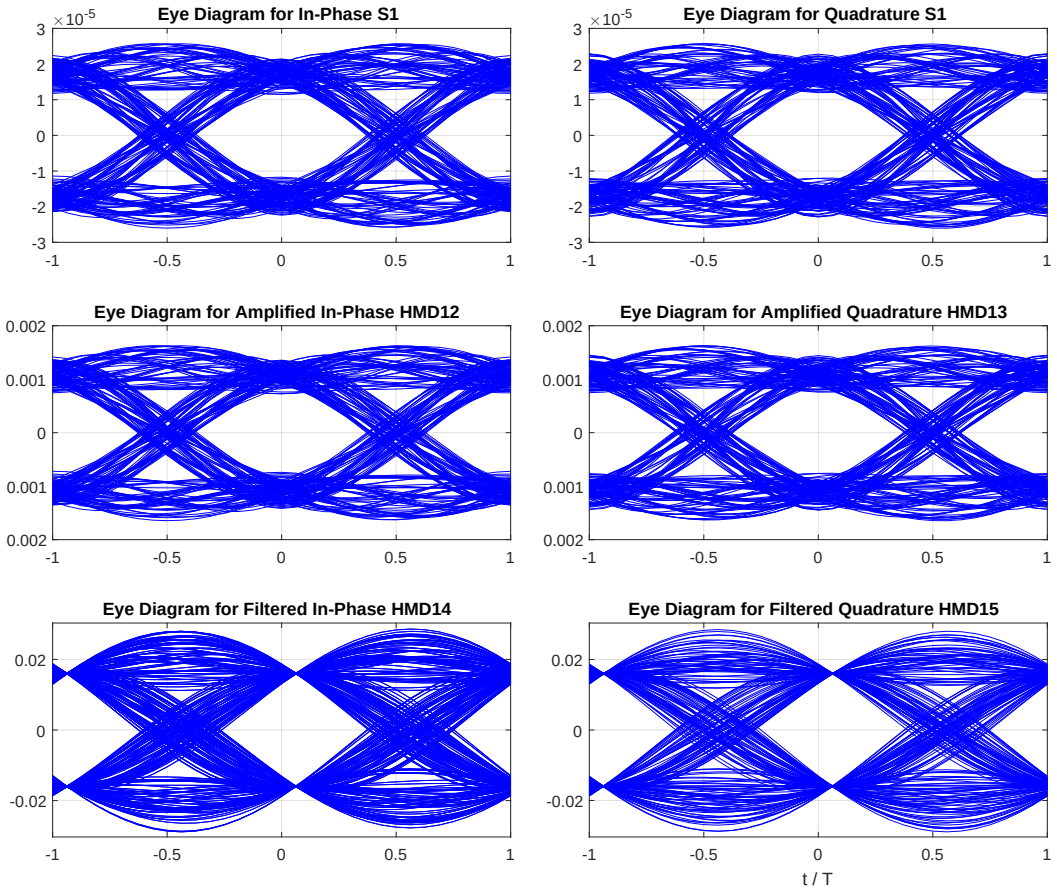


Figura 5.17: Eye diagrams using matched filtering with root-raised-cosine at three different points, without AWGN: the optical output signal S1 on the top; the amplified signal at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, and a rolloff factor of 0.3.

Thus, it can be concluded that, in order to avoid inter-symbol interference, the filters used should be raised-cosine or root-raised-cosine, if not using a filter at the receiver or if using matched filtering, respectively. As such, from now on only these configurations will be used.

Signals with AWGN and high SNR

In this section and the following one, a comparison will be presented between not using a filter on the receiver and using matched filtering. This comparison will be made for signals affected by added white gaussian noise, where the noise is added to the signal after the amplifier stage and before the signal passes through the filter on the receiver.

For the first case, where no filter is present at the receiver, a raised-cosine filter will be

used at the pulse shaper. For matched filtering, a root-raised-cosine filter will be used at the pulse-shaper and the receiver.

Figures 5.18-5.19 show the eye diagrams for both these cases. The optical power used was $-45dBm$, the noise spectral density was set at $10^6 W/Hz$, and the roll-off factor was set to 0.9 in both cases. In both cases, its is still possible to visibly see the approximate shape of the signal after noise is added, even without matched filtering. However, it can be seen that the output signal in the case with matched filtering is much less affected by noise, as the root-raised-cosine at the receiver is rather effective at filtering the noise.

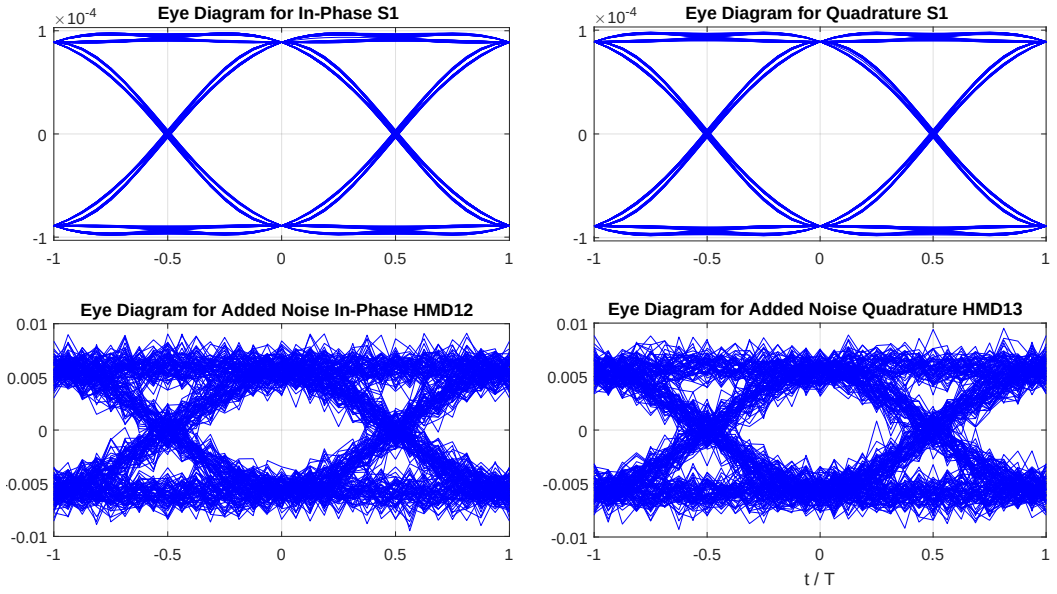


Figura 5.18: Eye diagrams without matched filtering using raised-cosine, at two different points: the optical output signal S1 on the top and the amplified signal with added noise, HMD12 and HMD13 for both components. Obtained through simulation with an optical power output of -45 dBm, 0 dBm at the local oscillator, gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.9.

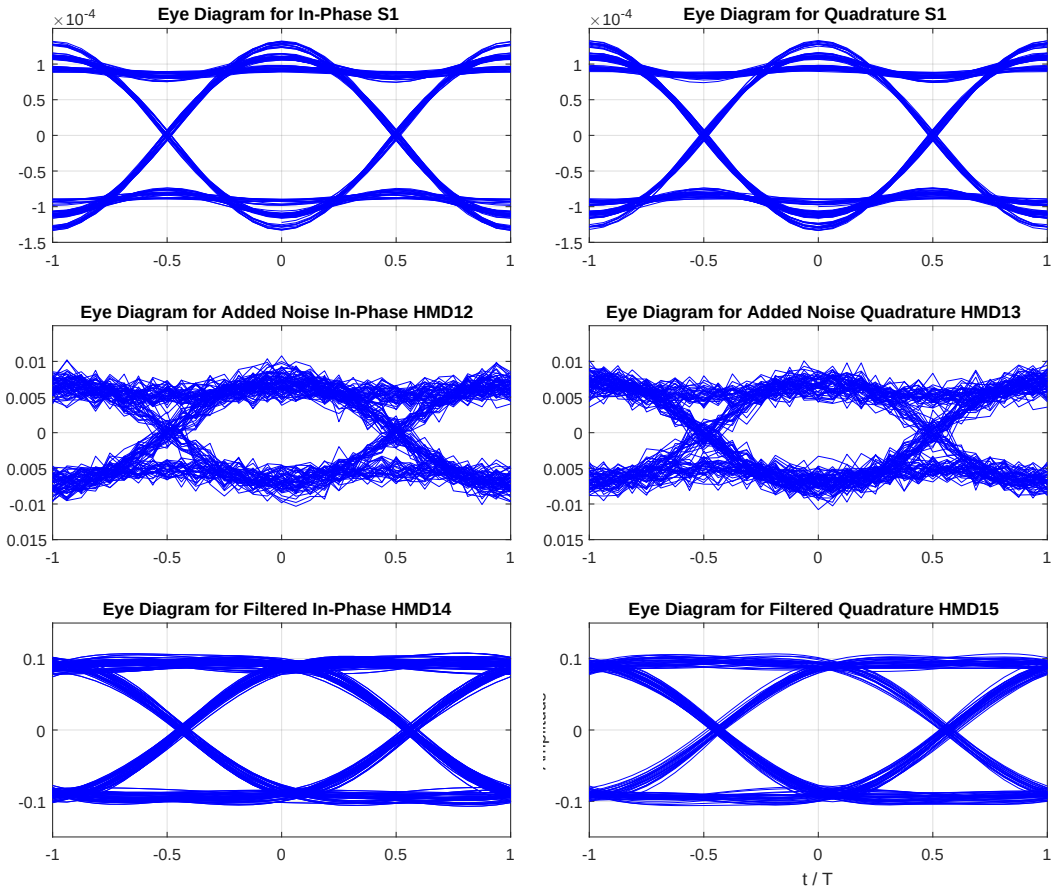


Figura 5.19: Eye diagrams using matched filtering with root-raised-cosine at three different points: the optical output signal S1 on the top; the amplified signal with added noise at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -45 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.9.

Figures and show the cases described above, but with a roll-off factor of 0.3. It can be seen that the case is in all aspects similar to the one presented above.

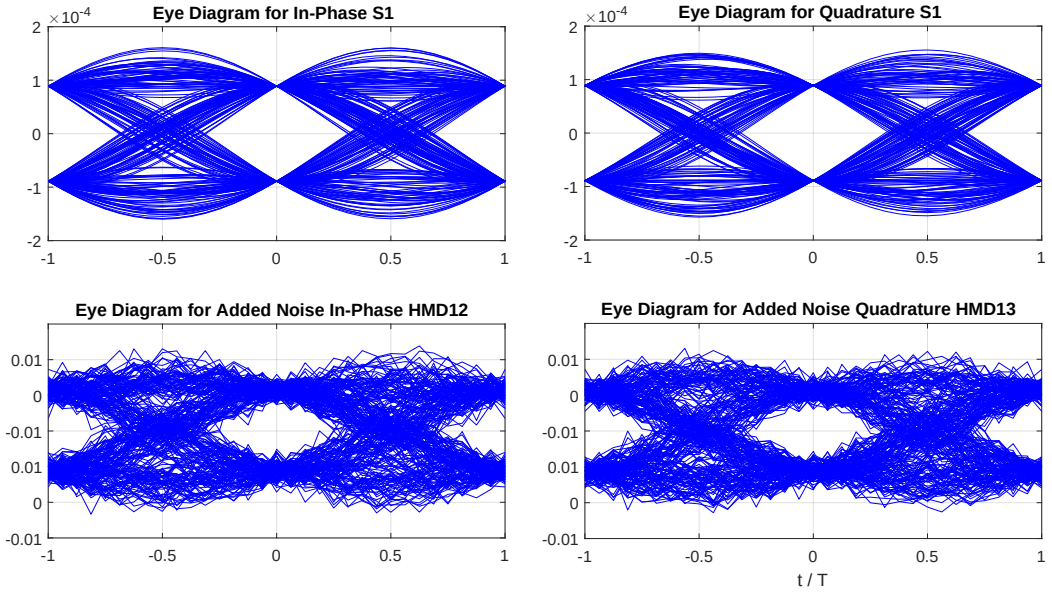


Figura 5.20: Eye diagrams without matched filtering with raised-cosine at two different points: the optical output signal S1 on the top and the amplified signal with added noise at the middle, HMD12 and HMD13, for both components. Obtained through simulation with an optical power output of -45 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.3.

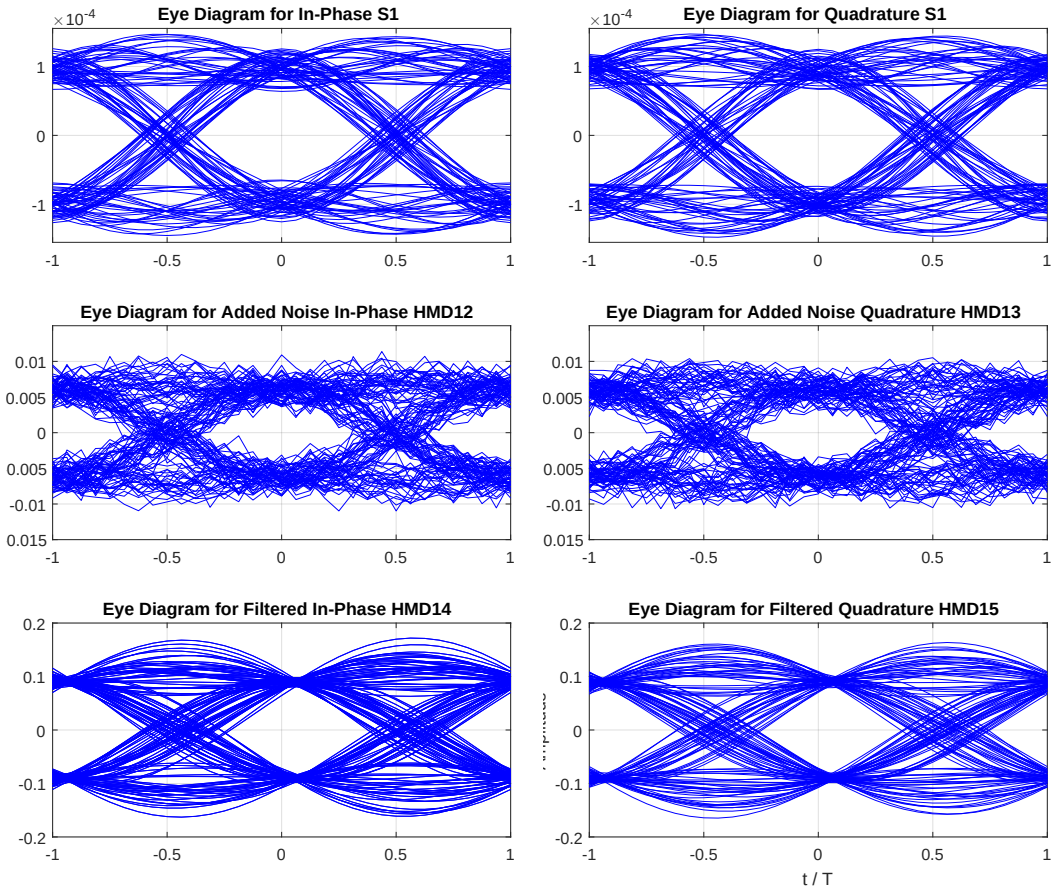


Figura 5.21: Eye diagrams using matched filtering with root-raised-cosine at three different points: the optical output signal S1 on the top; the amplified signal with added noise at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -45 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.3.

Signals with AWGN and low SNR

Figures 5.24-5.22 show eye diagrams similar to the previous section, but with a lower optical power ($-60\ dBm$), comparable to the spectral density of the noise (10^{-6}).

Figures 5.22 and 5.23 show the diagrams obtained without matched filtering and with matched filtering, respectively, both using a roll-off factor of 0.9.

In this example the effects of matched filtering is even more obvious, as without it the signal visually appears to be random noise.

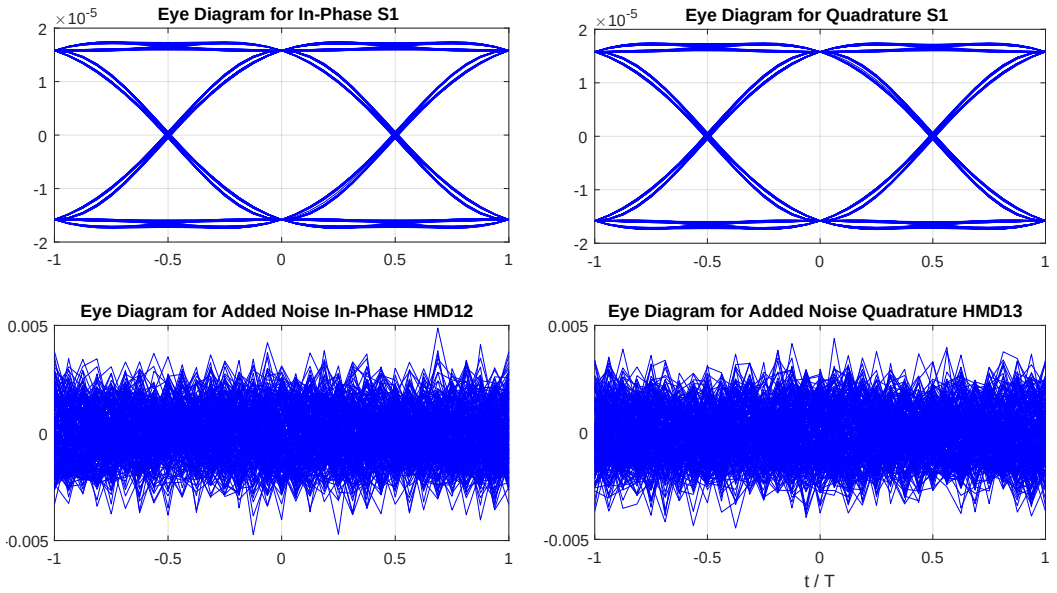


Figura 5.22: Eye diagrams without matched filtering with raised-cosine at two different points: the optical output signal S1 on the top and the amplified signal with added noise at the middle, HMD12 and HMD13 for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.9.

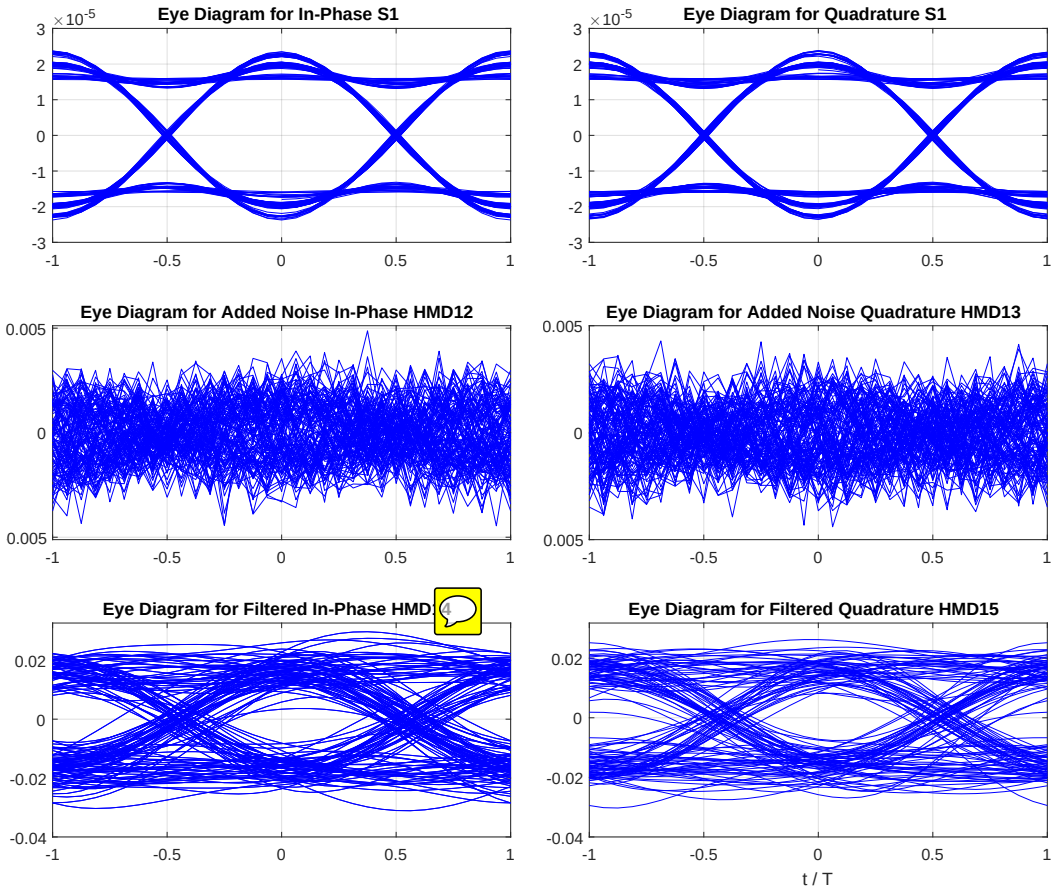


Figura 5.23: Eye diagrams using matched filtering with root-raised-cosine at three different points: the optical output signal S1 on the top; the amplified signal with added noise at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.9.

Figures 5.25 and 5.23 show the same case but using a roll-off factor of 0.3.

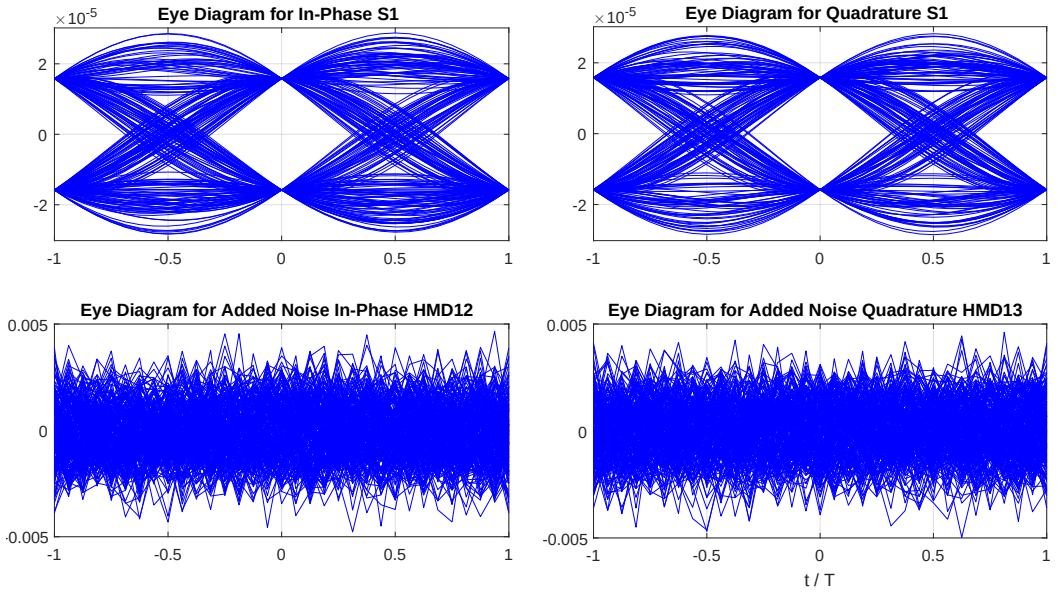


Figura 5.24: Eye diagrams without matched-filtering with raised-cosine at two different points: the optical output signal S1 on the top and the amplified signal with added noise at the middle, HMD12 and HMD13 for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.3.

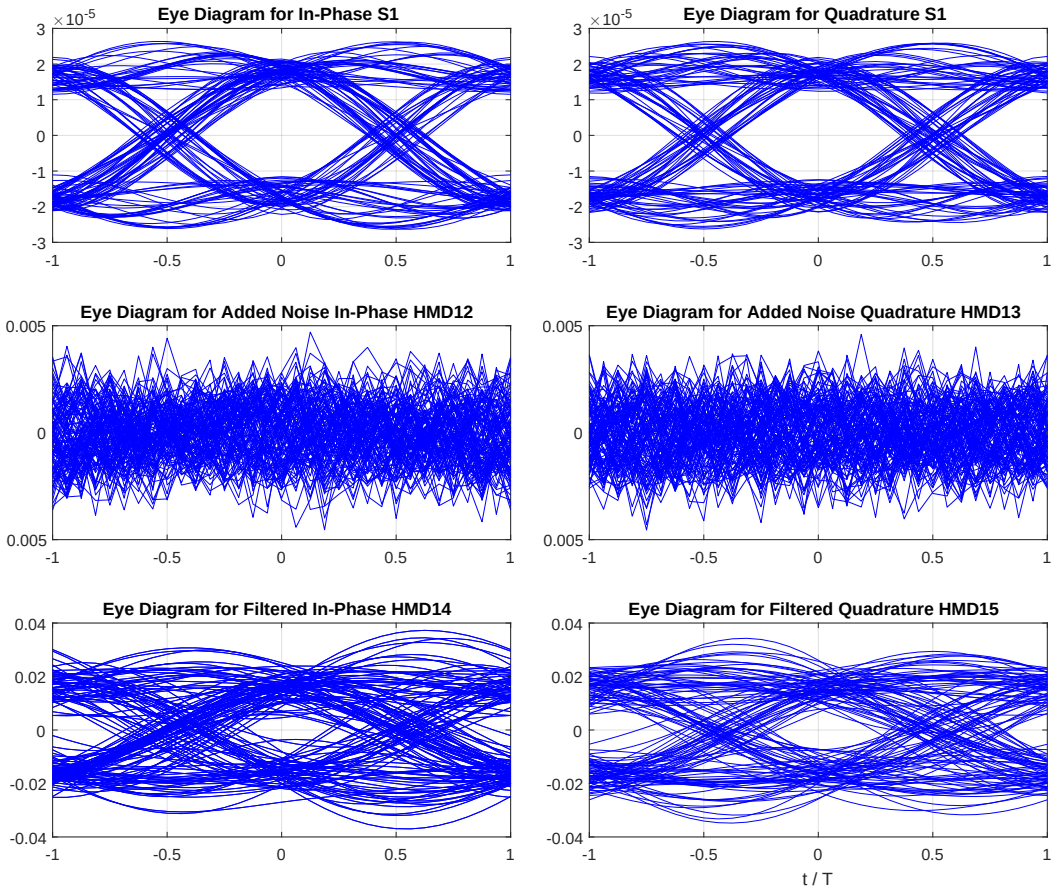


Figura 5.25: Eye diagrams using matched filtering with root-raised-cosine at three different points: the optical output signal S1 on the top; the amplified signal with added noise at the middle, HMD12 and HMD13 for both components; and after passing through the last root-raised-cosine filter, HMD14 and HMD15, for both components. Obtained through simulation with an optical power output of -60 dBm, 0 dBm at the local oscillator, a gain of 10^3 at the amplifier, a noise spectral density of 10^{-6} and a rolloff factor of 0.3.

5.3.3 BER Curves

The simulated results show agreement with the theoretical curves, as can be seen in Figure 5.26.

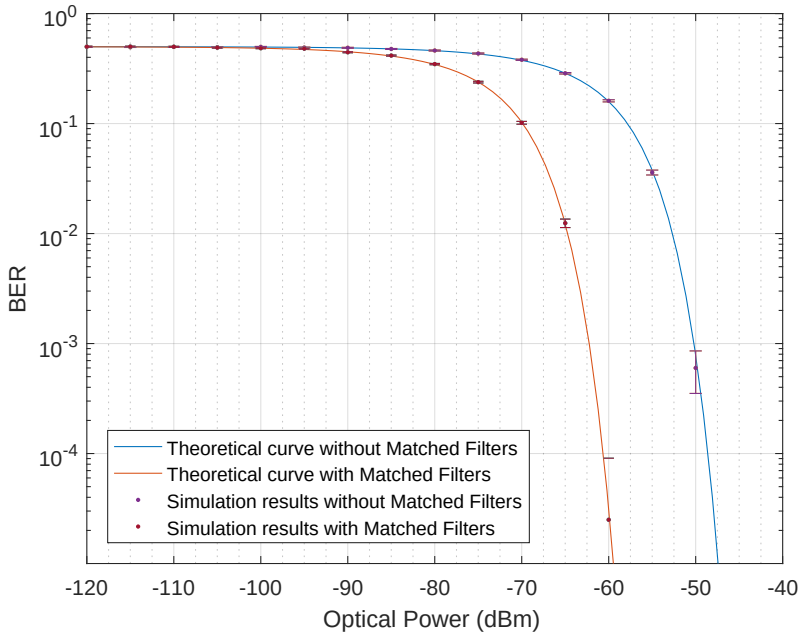


Figura 5.26: Simulation results for a random binary sequence with 40000 bits, a noise power of 10^{-6} and an amplification of 10^3. The simulated values which used a matched filter were obtained by shaping the pulse with a root-raised-cosine FIR filter and filtering the signal before the sample with the same filter. The results without matched filtering were obtained by using shaping the pulses with a raised-cosine filter. The margins shown were obtained for a 95% confidence level.

Conclusions

The use of a root-raised-cosine filters for shaping and filtering the signal provides the best results, due to reducing noise while creating no inter-symbol interference. The experimental BER curves agree with the theoretical values and show the advantages of matched filtering.

Experimental Analysis

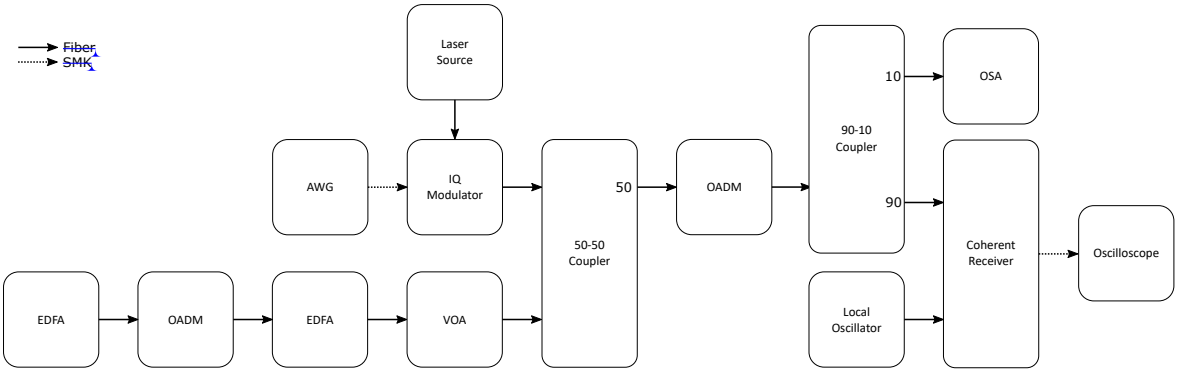


Figura 5.27: Experimental setup

5.3.4 Comparative Analysis

In this section we show the simulation results and compared them with the theoretical predictions for an M-QAM system with $M = 4$. Figure 5.26 shows the variation of the BER with the optical power of the signal, using 40000 bits produced by a random number generator. The noise power was set at 10^{-6}, the local oscillator at $0\ dBm$ and the amplification at the transimpedance amplifier was set at 10^3. The blue line represents the theoretical curve, while the orange points represent the simulated values with the respective confidence margins. The simulation agrees closely with the theoretical values.

5.3.5 Open Issues

References

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

[2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.

[3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

[4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.

[5] Yue-Meng Chi, Bing Qi, Wen Zhu, Li Qian, Hoi-Kwong Lo, Sun-Hyun Youn, AI Lvovsky, and Liang Tian. A balanced homodyne detector for high-rate gaussian-modulated coherent-state quantum key distribution. *New Journal of Physics*, 13(1):013003, 2011.

[6] 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.

5.4 Optical Detection

Contributors	:	Nelson Muga, (2017-12-21 - ...)
	:	Diamantino Silva, (2017-08-18 - ...)
	:	Armando Pinto (2017-08-15 - ...)
Goal	:	Analise of various optical detection schemes.

The detection of light is a fundamental stage in every optical communication system, bridging the optical domain into the electrical domain. This section will review various theoretical and practical aspects of light detection, as well as a series of implementations and schemes. The objective of this work is to develop numerical models for the various optical detections schemes and to validate such numerical models with experimental results.

5.4.1 Theoretical Analysis

Contributors	:	Nelson Muga (2017-12-20 -)
	:	Diamantino Silva (2017-08-18 - ...)
	:	Armando Pinto (2017-08-15 - ...)
Goal	:	Theoretical description of various optical detection schemes.

In this subsection, we are going to calculate the signal-to-noise ratio at the input of the decision circuit for the various detection schemes under analysis. For each detection scheme a classical and a quantum description is going to be developed and a comparative analysis is going to performed.

Classical Description

Contributors	:	Nelson Muga (2017-12-20 -)
	:	Diamantino Silva (2017-08-18 - ...)
	:	Armando Pinto (2017-08-15 - ...)
Goal	:	Develop a classical description of various optical detection schemes.

Direct Detection

One of the most simple detection methods is the direct detection of light with a single detector and the analysis of the resulting photocurrent.

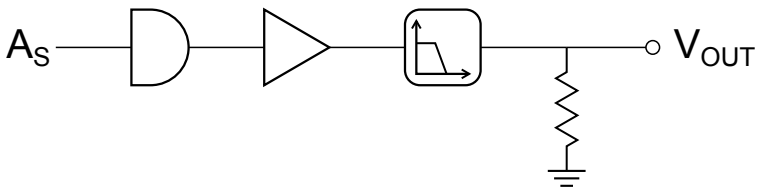


Figura 5.28: Direct detection.

Given an electric field generated by an ideal monochromatic single mode laser, it can be modulated in amplitude and phase by a signal $A(t)$ resulting in the function

$$E_R(t) = \sqrt{2}|A(t)|cos\left(-\omega t + \theta\right) \quad \sqrt{W} \tag{5.23}$$

with $A(t) = |A(t)|e^{-i\theta(t)}$, and where ω_0 is the carrier frequency and θ is the phase.

We will consider that the detector has a bandwidth B, greater that the signal $A(t)$, but much smaller than $2\omega_0$. The calculation of power incident in the photodiode is given by the expected value of the square of the amplitude during a time interval $\Delta t = 2\pi/\omega$

Measurable optical power, assuming that the detector bandwidth, B, is greater than the signal, $A(t)$, bandwidth but much small than $2\omega_0$

$$\begin{aligned} P(t) &= \overline{E_R^2(t)} \\ &= \overline{|A(t)|^2} + \overline{|A(t)|^2 cos\left(-2\omega t + 2\theta(t)\right)} \\ &= |A(t)|^2 \quad W \end{aligned} \tag{5.24}$$

To simplify calculations, the electric field can be expressed the complex notation

$$E(t) = A(t)e^{-i\omega_0 t} \tag{5.25}$$

The physically measurable quantities are obtained by taking the real part of the complex wave. Using this notation, the beam power, $P(t)$, is obtained by multiplying the electric field's conjugate by itself

$$\begin{aligned} P(t) &= E^*(t)E(t) \\ &= |A(t)|^2 \end{aligned} \tag{5.26}$$

$$i(t) = \eta q \frac{P(t)}{\hbar\omega_0} \tag{5.27}$$

in which η is the photodiode's responsivity, q is the unit charge and $P(t)/\hbar\omega_0$ is the number of removed electrons.

The signal is ...

$$E(t) = A(t)e^{-i\omega_0} \tag{5.28}$$

Using the definition of electric power of a complex electric field representation, we will get

$$P(t) = |A(t)|^2 \tag{5.29}$$

recovering the result of the real representation. The photocurrent can be rewriten as a function of the signal $A(t)$

$$i(t) = \eta q \frac{|A(t)|^2}{\hbar\omega_0} \tag{5.30}$$

which will use to express the second moment of the photocurrent as

$$\langle i^2(t)\rangle = \eta^2 q^2 \frac{\langle |A(t)|^4\rangle}{\hbar^2\omega_0^2} \tag{5.31}$$

Assuming a fase modulation, in which the amplitude is constant, the signal is simplified to

$$A(t) = |A|e^{i\theta} \tag{5.32}$$

Therefore, the current becomes constant

$$i(t) = I_0 = \eta q \frac{A_s^2}{\hbar\omega_0} \tag{5.33}$$

and it's second moment becomes simply

$$\langle i^2(t)\rangle = I_0^2 \tag{5.34}$$

Shot noise in photodiodes

$$\langle i_n^2(t)\rangle = 2qBI_0 \tag{5.35}$$

The signal to noise ratio is obtained by the relation between the second moment of the sinal to the second moment of the noise

$$\begin{aligned} \frac{S}{N} &= \frac{\langle i^2(t)\rangle}{\langle i_n^2(t)\rangle} \\ &= \frac{I_0}{2qB} \\ &= \eta\frac{|A|^2}{\hbar\omega_0 B} \end{aligned} \tag{5.36}$$

Homodyne Detection

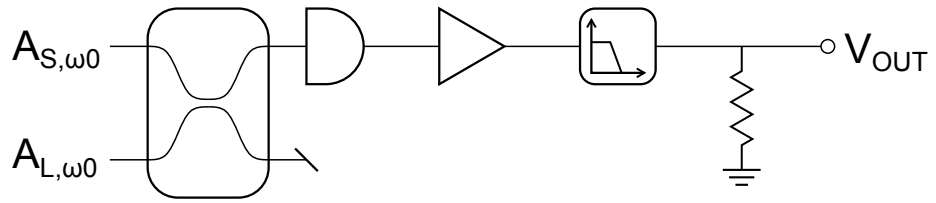


Figura 5.29: Homodyne detection.

The homodyne detection scheme uses an auxiliary local oscillator, which is combined in a beamsplitter with the signal beam. After this step, it is similar to the direct detection. As we will see this has some implications in the phase detection???

Given a splitter with intensity transmission ϵ, the resulting field incident to the photodetector is [**?**]

$$E(t) = \sqrt{\epsilon}E_S(t) + \sqrt{1-\epsilon}E_{LO}(t) \tag{5.37}$$

in which $E_{LO} = e^{i\omega_0 t}$. Given a local oscillator with a much larger power that the signal, then, the incident power in the photodiode is

$$P(t) = \eta\left[(1-\epsilon)P_{LO}(t) + 2\sqrt{\epsilon(1-\epsilon)}\text{Re}\left[E_S(t)E^*_{LO}(t)\right]\right] \tag{5.38}$$

$$= \eta\left[(1-\epsilon)P_{LO}(t) + 2\sqrt{\epsilon(1-\epsilon)}|E_S(t)||E_{LO}(t)|\cos(\phi)\right] \tag{5.39}$$

$$\tag{5.40}$$

Balanced Homodyne Detection

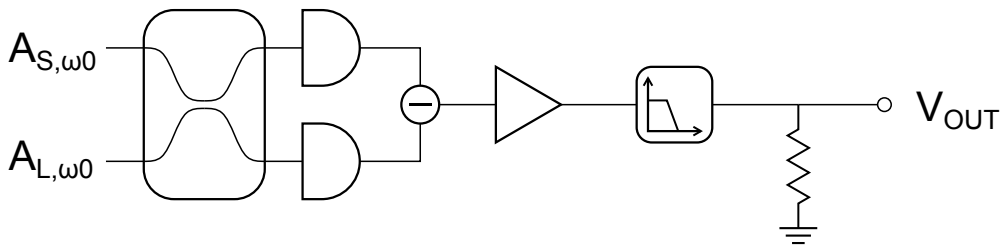


Figura 5.30: Balanced homodyne detection.

"In a balanced homodyne detector (BHD), the signal to be measured is mixed with a local oscillator (LO) at a beam splitter. The interference signals from the two output ports of the beam splitter are sent to two photodiodes followed by a subtraction operation, and then, amplification may be applied. The output of a BHD can be made to be proportional to either the amplitude quadrature or the phase quadrature of the input signal depending on the relative phase between the signal and the LO".

IQ Homodyne Balanced Detection

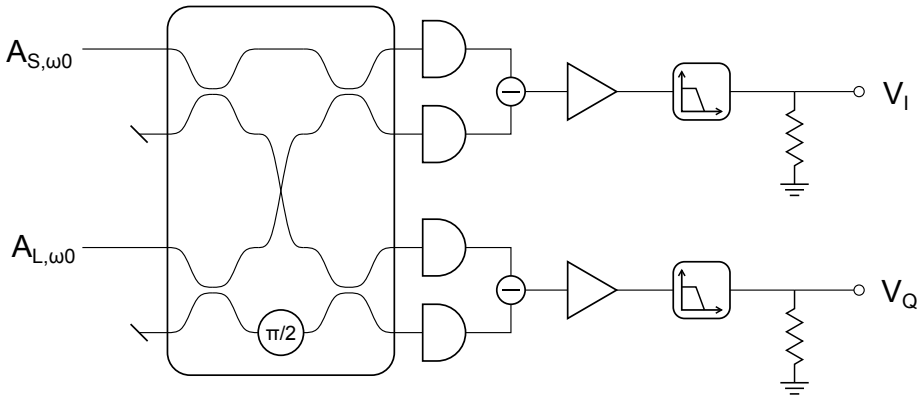


Figura 5.31: IQ balanced homodyne detection.

Semiclassical model

Quantum model

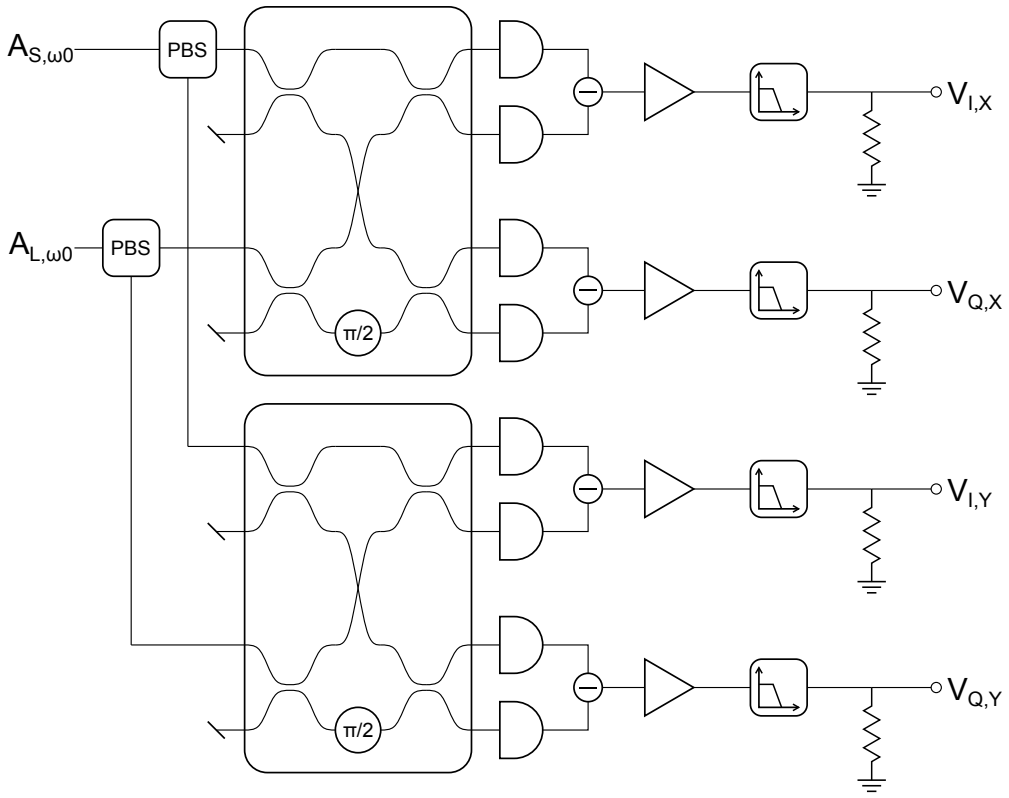


Figura 5.32: Double polarized IQ balanced homodyne detection.

Heterodyne Detection

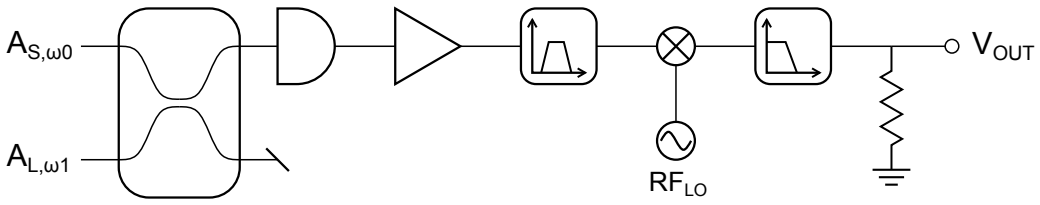


Figura 5.33: Heterodyne detection.

In contrast with the homodyne detection, in which the frequency of the signal carrier is equal to the frequency of the local oscillator, in the heterodyne detection, these frequencies are different.

Because of this, the inference will result in a new signal with an intermediate frequency at...

Balanced Heterodyne Detection

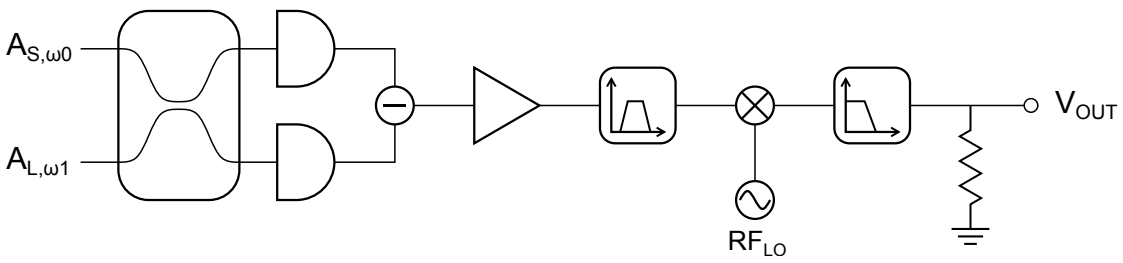


Figura 5.34: Balanced heterodyne detection.

Semiclassical model

Quantum model

IQ Heterodyne Balanced Detection

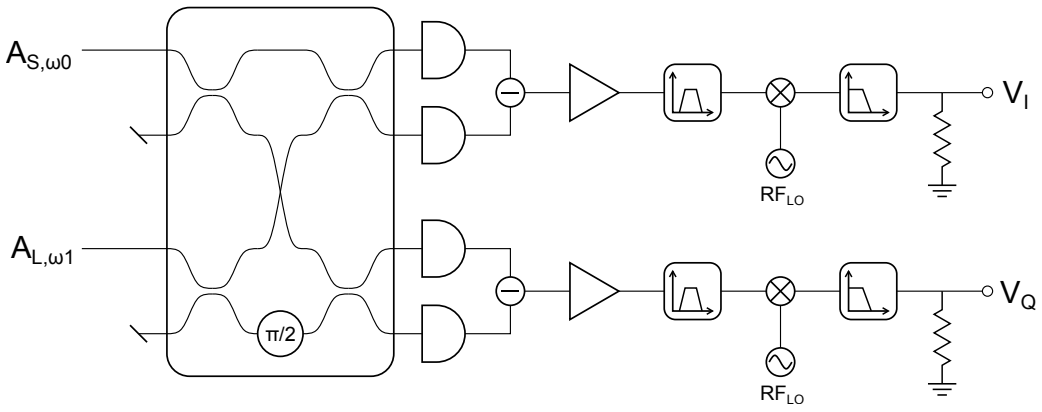


Figura 5.35: IQ balanced heterodyne detection.

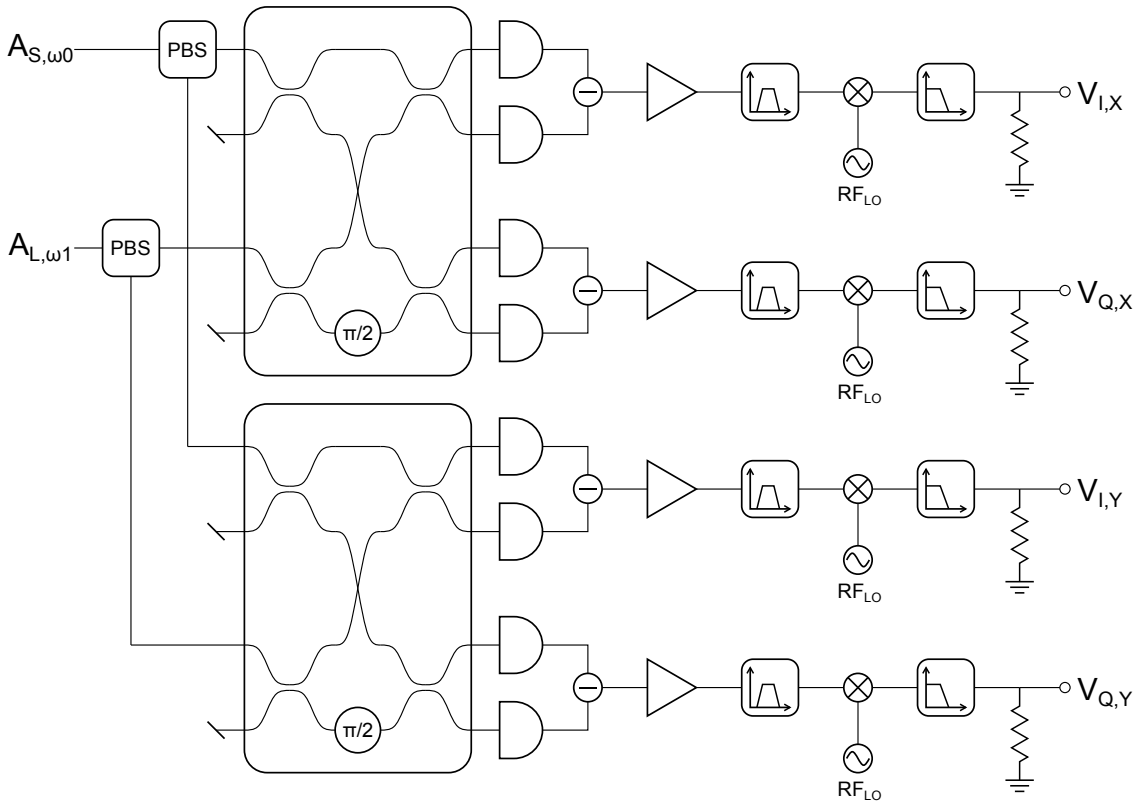


Figura 5.36: Double polarized IQ balanced heterodyne detection.

Thermal noise

Thermal noise is generated by electrons in response to temperature. It's contribution to the resulting current can be described by the following equation [2]

$$\langle (\Delta i_T)^2 \rangle = 4K_B T_0 B / R_L \tag{5.41}$$

in which K_B it's Boltzmann's constant, T_0 is the absolute temperature, B is the bandwidth and R_L is the receiver load impedance. The B value is imposed by default or chosen when the measurements are made, but the R_L value is dependent in the internal setup of the various components of the detection system. Nevertheless, for simulation purposes, we can just introduce an experimental value.

Quantum Description

Contributors	:	Diamantino Silva (2017-08-18 - ...)
	:	Armando Pinto (2017-08-15 - ...)
Goal	:	Develop a quantum description of various optical detection schemes, and compare with the classical description.

We start by defining number states $|n\rangle$ (or Fock states), which correspond to states with perfectly fixed number of photons [3]. Associated to those states are two operators, the creation $\hat{a}^\dagger$ and annihilation $\hat{a}$ operators, which in a simple way, remove or add one photon from a given number state [2]. Their action is defined as

$$\hat{a}\,|n\rangle = \sqrt{n}\,|n-1\rangle \quad (5.42)\,, \qquad \hat{a}^\dagger\,|n\rangle = \sqrt{n+1}\,|n+1\rangle \quad (5.43)\,, \qquad \hat{n}\,|n\rangle = n\,|n\rangle \quad (5.44)$$

in which $\hat{n} = \hat{a}^\dagger\hat{a}$ is the number operator. Therefore, number states are eigenvectors of the number operator.

Coherent states have properties that closely resemble classical electromagnetic waves, and are generated by single-mode lasers well above the threshold. [3] We can defined them, using number states in the following manner

$$|\alpha\rangle = e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}}\,|n\rangle \tag{5.45}$$

in which the complex number α is the sole parameter that characterizes it. In fact, if we calculate the expected number of photons with $\langle\alpha|\,\hat{n}\,|\alpha\rangle$ we will obtain $|\alpha|^2$. The coherent state is an eigenstate of the annihilation operator, $\hat{a}\,|\alpha\rangle = \alpha\,|\alpha\rangle$.

Using the creation and annihilation operators, we can define two quadrature operators [3]

$$\hat{X} = \frac{1}{2}\left(\hat{a}^\dagger + \hat{a}\right) \quad (5.46)\,, \qquad \hat{Y} = \frac{i}{2}\left(\hat{a}^\dagger - \hat{a}\right) \quad (5.47)$$

The expected value of these two operators, using a coherent state $|\alpha\rangle$ are

$$\langle\hat{X}\rangle = \mathrm{Re}(\alpha) \quad (5.48)\,, \qquad \langle\hat{Y}\rangle = \mathrm{Im}(\alpha) \quad (5.49)$$

We see that the expected value of these operators give us the real and imaginary part of α. Now, we can obtain the uncertainty of these operators, using:

$$\mathrm{Var}(\hat{X}) = \langle\hat{X}^2\rangle - \langle\hat{X}\rangle^2 \tag{5.50}$$

For each of these quadrature operators the variance will be

$$\mathrm{Var}(\hat{X}) = \mathrm{Var}(\hat{Y}) = \frac{1}{4} \tag{5.51}$$

This result show us that for both quadratures, the variance of measurement is the same and independent of the value of α.

Homodyne detection

The measurent of a quadrature of an input signal (S) is made by using the balanced homodyne detection technique, which measures the phase difference between the input signal and a local oscillator (LO). The measurement of quadrature are made relative to a reference phase of the LO, such that if the measurement is made in-phase with this reference, the value will be proportional to the $\hat{X}$ quadrature of the signal. If the phase of the LO is has an offset of $\pi/2$ relative to the reference, the output will be proportional to the $\hat{Y}$ quadrature of the signal.

Experimentally, the balanced homodyne detection requires a local oscillator with the same frequency as the input signal, but with a much larger amplitude. These two signals are combined using a 50:50 beam splitter, from were two beams emerge, which are then converted to currents using photodides. Finally, the two currents are subtracted, resulting in an output current proportional to a quadrature of the input signal [2].
A phase of the local oscillator can be defined as the reference phase. A phase offset equal to 0 or $\pi/2$ will give an output proportional to the signal's in-phase component or to the quadrature component, respectively. Therefore, the $\hat{X}$ operator will correspond to the in-phase component and $\hat{Y}$ operator correspond to quadrature component

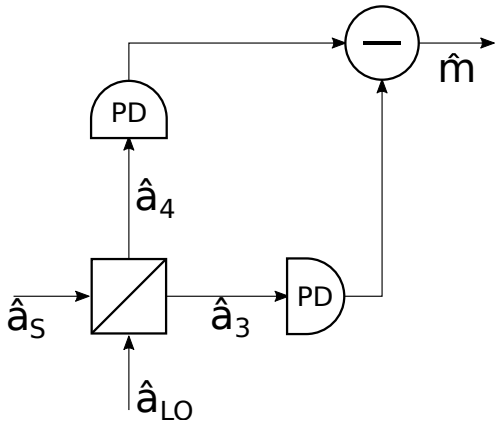


Figura 5.37: Balanced homodyne detection.

In the lab and in our simulations, a more complex system is used, the double balanced homodyne detection, which allows the simultaneous measurement of the $\hat{X}$ and $\hat{Y}$ components. The signal is divided in two beam with half the power of the original. One of the beams is used in a balanced homodyne detection with a local oscillator. The other beam is used in another balanced homodyne detection, but using a local oscillator with a phase difference $\pi/2$ relative to the first one.

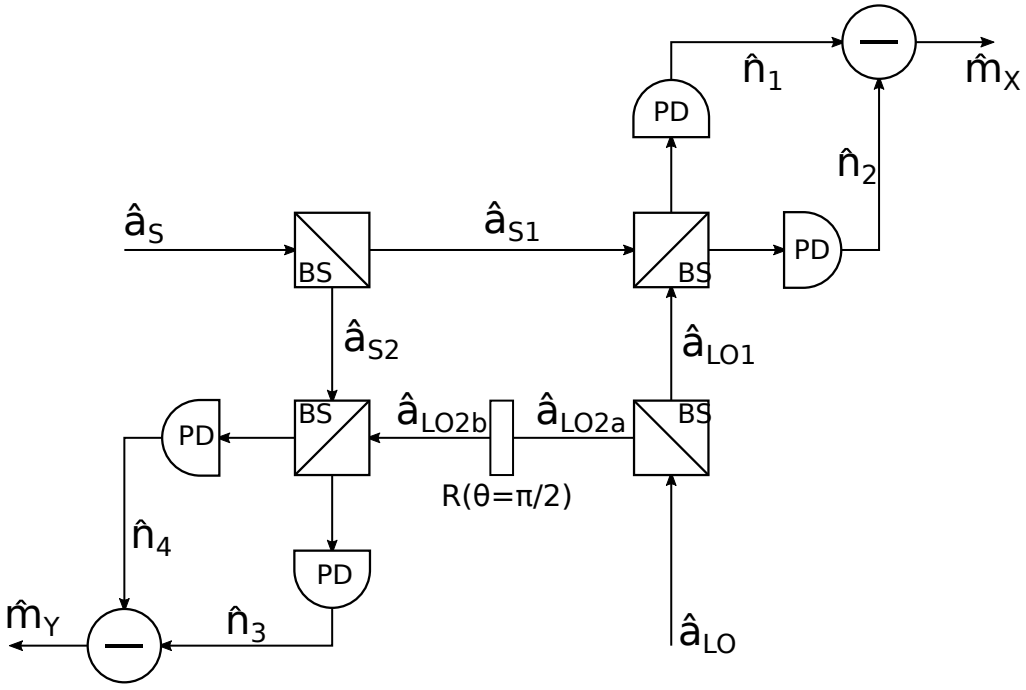


Figura 5.38: Balanced double homodyne detection.

Noise sources in homodyne detection

The detection of light using photodiodes is subjected to various sources of noise. One of these sources is the electrical field itself. The interaction of the signal with the vaccuum field adds quantum noise to the detection. Another source of noise comes from the detection system, such as photodiodes and other electrical circuits, originating various kinds of noise, such as thermal noise, dark noise and amplifier noise [4]. In the following sections, we will focus on two noise sources, quantum noise and thermal noise.

Quantum Noise

In order to grasp this effect, the quantum mechanical description of balanced homodyne detection will be used, employing quantum operators to describe the effect of each component in the system (fig. **??**). We start with the operators $\hat{a}_S$ and $\hat{a}_{LO}$ corresponding to the annihilation operator for the signal and local oscillator, which are the inputs in a beam divisor. The outputs will be $\hat{a}_3$ and $\hat{a}_4$. Using a balanced beam splitter, we can write the output as

$$\hat{a}_3 = \frac{1}{\sqrt{2}}\left(\hat{a}_S + \hat{a}_{LO}\right) \quad (5.52)\,, \qquad \hat{a}_4 = \frac{1}{\sqrt{2}}\left(\hat{a}_S - \hat{a}_{LO}\right) \quad (5.53)$$

The final output of a homodyne measurement will be proportional to the difference between the photocurrents in arm 3 and 4. Then

$$I_{34} = I_3 - I_4 \sim \langle \hat{n}_3 - \hat{n}_4 \rangle \tag{5.54}$$

We can define an operator that describes the difference of number of photons in arm 3 and arm 4:

$$\hat{m} = \hat{a}_3^\dagger \hat{a}_3 - \hat{a}_4^\dagger \hat{a}_4 \tag{5.55}$$

If we assume that the local oscillator produces the the coherent state $|\beta\rangle$, then the expected value of this measurement will be

$$\langle m \rangle = 2|\alpha||\beta| \cos(\theta_\alpha - \theta_\beta) \quad (5.56)\,, \qquad \text{Var}(m) = |\alpha|^2 + |\beta|^2 \quad (5.57)$$

The local oscillator normally has a greater power than the signal , then $|\alpha| \ll |\beta|$. If we use as unit, $2|\beta|$, then these two quantities can be simplified to

$$\langle m \rangle = |\alpha| \cos(\theta_\alpha - \theta_\beta) \quad (5.58)\,, \qquad \text{Var}(m) \approx \frac{1}{4} \quad (5.59)$$

[4]
Has we have seen previously, in order to measure two quadratures simultaneously, we can use double balanced homodyne detection. For each quadrature, the input signal now has half the power, so $|\alpha| \to |\alpha/\sqrt{2}|$. If we use a local oscillator that produces states $|\beta\rangle$, then we can divide it in two beams in state $|\beta/\sqrt{2}\rangle$ and $|i\beta/\sqrt{2}\rangle$ which will be used in each homodyne detection. In this setting, the expected values for each quadrature, X and Y, (in normalized values of $\sqrt{2}|\beta|$) are

$$\langle m_X \rangle = \left| \frac{\alpha}{\sqrt{2}} \right| \cos(\theta_\alpha - \theta_\beta) \quad (5.60)\,, \qquad \text{Var}(m_X) \approx \frac{1}{4} \quad (5.61)$$

$$\langle m_Y \rangle = \left| \frac{\alpha}{\sqrt{2}} \right| \sin(\theta_\alpha - \theta_\beta) \quad (5.62)\,, \qquad \text{Var}(m_Y) \approx \frac{1}{4} \quad (5.63)$$

Therefore the measurement of each quadrature will have half the amplitude, but the same variance.

5.4.2 Simulation Analysis

Contributors	:	Diamantino Silva, (2017-08-18 - ...)
Goal	:	Simulation of various optical detection schemes.
Directory	:	sdf/optical_detection

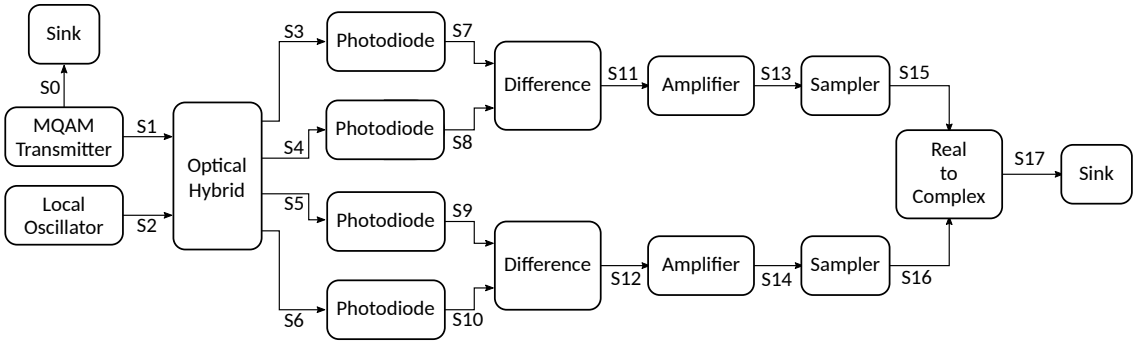


Figura 5.39: Overview of the simulated optical system.

List of signals used in the simulation:

Signal name	Signal type	Status
S0	Binary	check
S1	OpticalSignal	check
S2	OpticalSignal	check
S3	OpticalSignal	check
S4	OpticalSignal	check
S5	OpticalSignal	check
S6	OpticalSignal	check
S7	TimeContinuousAmplitudeContinuousReal	check
S8	TimeContinuousAmplitudeContinuousReal	check
S9	TimeContinuousAmplitudeContinuousReal	check
S10	TimeContinuousAmplitudeContinuousReal	check
S11	TimeContinuousAmplitudeContinuousReal	check
S12	TimeContinuousAmplitudeContinuousReal	check
S13	TimeContinuousAmplitudeContinuousReal	check
S14	TimeContinuousAmplitudeContinuousReal	check
S15	TimeDiscreteAmplitudeContinuousReal	check
S16	TimeDiscreteAmplitudeContinuousReal	check
S17	OpticalSignal	check

This system takes into account the following input parameters:

System Parameters	Default value	Description
localOscillatorPower1	2.0505×10^{-8}W	Sets the optical power, in units of W, of the local oscillator inside the MQAM
localOscillatorPower2	2.0505×10^{-8}W	Sets the optical power, in units of W, of the local oscillator used for Bob's measurements
localOscillatorPhase	0 rad	Sets the initial phase of the local oscillator used in the detection
responsivity	1 A/W	Sets the responsivity of the photodiodes used in the homodyne detectors
iqAmplitudeValues	$\{\{1,1\},\{-1,1\},$ $\{-1,-1\},\{1,-1\}\}$	Sets the amplitude of the states used in the MQAM

The simulation setup is represented in figure 5.39. The starting point is the MQAM, which generates random states from the constelation given by the variable `iqAmplitudeValues`. The output from the generator is received in the Optical Hybrid where it is mixed with a local oscillator, outputing two optical signal pairs. Each pair is converted to currents by two photodiodes, and the same currents are subtracted from each other, originating another current proportional to one of the quadratures of the input state. The other pair suffers the same process, but the resulting subtraction current will be proportional to another quadrature, dephased by $\pi/2$ relative to the other quadrature.

Required files

Header Files

File	Description	Status
netxpto.h	Generic purpose simulator definitions.	check
m_qam_transmitter.h	Outputs a QPSK modulated optical signal.	check
local_oscillator.h	Generates continuous coherent signal.	check
optical_hybrid.h	Mixes the two input signals into four outputs.	check
photodiode.h	Converts an optical signal to a current.	check
difference.h	Ouputs the difference between two input signals.	check
ideal_amplifier.h	Performs a perfect amplification of the input sinal	check
sampler.h	Samples the input signal.	check
real_to_complex.h	Combines two real input signals into a complex signal	check
sink.h	Closes any unused signals.	check

Source Files

File	**Description**	**Status**
netxpto.cpp	Generic purpose simulator definitions.	check
m_qam_transmitter.cpp	Outputs a QPSK modulated optical signal.	check
local_oscillator.cpp	Generates continuous coherent signal.	check
optical_hybrid.cpp	Mixes the two input signals into four outputs.	check
photodiode.h	Converts an optical signal to a current.	check
difference.h	Ouputs the difference between two input signals.	check
ideal_amplifier.h	Performs a perfect amplification of the input sinal	check
sampler.cpp	Samples the input signal.	check
real_to_complex.cpp	Combines two real input signals into a complex signal	check
sink.cpp	Empties the signal buffer.	check

Simulation Results

To test the simulated implementation, a series of states $\{|\phi_i\rangle\}$ were generated and detected, resulting in a series of measurements $\{(x_i, y_i)\}$. The simulation result is presented in figure 5.40:

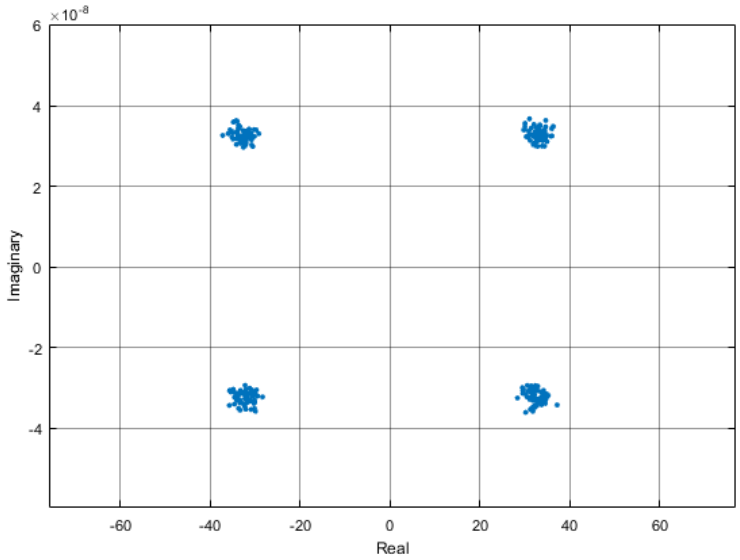


Figura 5.40: Simulation of a constelation of 4 states (n = 100)

We see that the measurements made groups in certain regions. Each of this groups is centered in the expected value $(\langle \mathrm{X} \rangle, \langle \mathrm{Y} \rangle)$ of one the generated states. Also, they show some variance, which was tested for various expected number of photons, $\langle n \rangle$, resulting in figure 5.41:

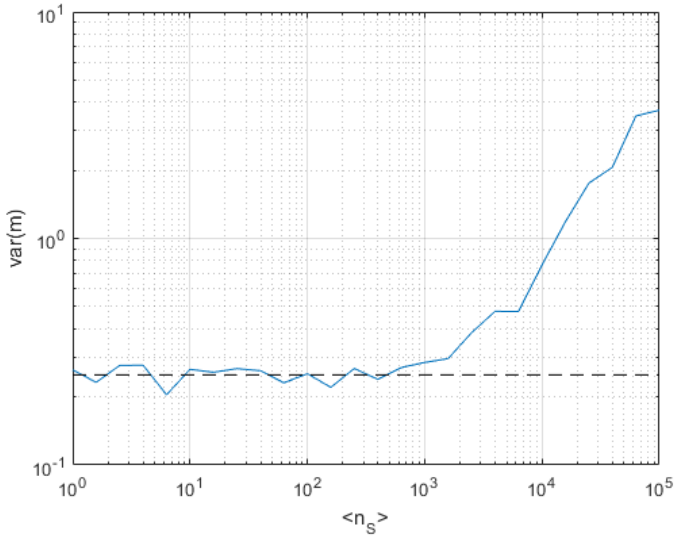


Figura 5.41: Simulation of the variance of m.
Local oscillator expected number of photons: 10^4

It was expected that the variance should independent of the input's signal number of photons. Plot 5.41 shows that for low values of n_S, the simulation is in accordance with the theoretical prevision, with Var(X) = Var(Y) = $\frac{1}{4}$. For large values of n_S, when the number of photons is about the same has the local oscillator, the quantum noise variance starts to grow proportionally to n_S, in accordance with the non approximated calculation of quantum noise (eq. **??**).

Noise Variance with LO power Simulation
The following plot shows the behavior of current noise variance $\langle(\Delta i)^2\rangle$ with local oscilator power, P_{LO}:

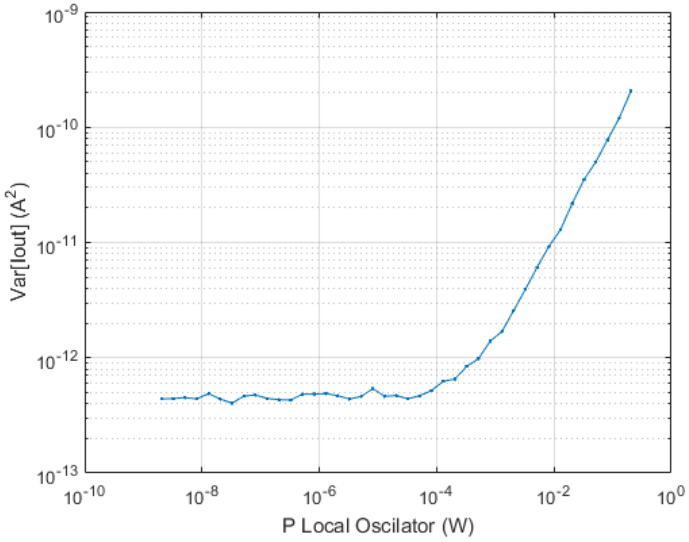


Figura 5.42: Output current variance in function of LO power.

We see that for low LO power, the dominant noise is the thermal contribution, but for higher power, quantum noise dominates, growing proportionally to P_{LO}. This in accordance with equation 5.58

5.4.3 Experimental Analysis

In this section, we will test experimentally the setups discussed in section 5.4.1. This comparison between the theoretical and experimental results will require a high degree of precision from the devices used in these setups. Therefore, the correct characterization of these devices must be the starting point of this experimental phase. One of the most fundamental components is the photodetector, which performs the signal's conversion from the optical domain into the electrical domain.

Thorlalabs detector

The detector used in the laboratory is the Thorlabs PDB 450C. This detector consists of two well-matched photodiodes and a transimpedance amplifier that generates an output voltage (RF OUTPUT) proportional to the difference between the photocurrents of the photodiodes. Additionally, the unit has two monitor outputs (MONITOR+ and MONITOR-) to observe the optical input power level on each photodiode separately. [1]

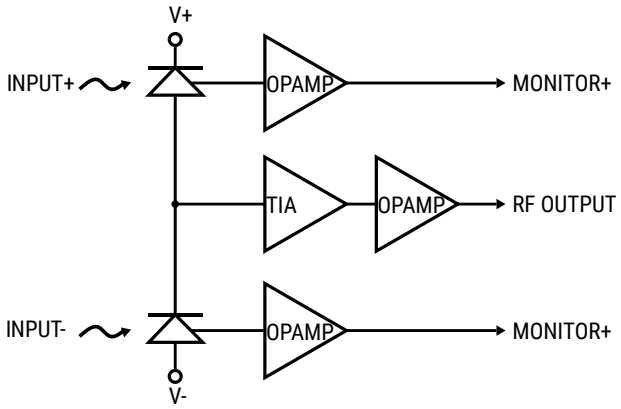


Figura 5.43: Functional block diagram of the PDB 450C Thorlabs detector [1]

Figure 5.43 shows a functional block diagram of the photodetector, in which the TIA (transimpedance amplifier) is an opamp-based current to voltage conversion amplifier. In contrast to the photodiode's non-linear voltage response to incident light, it's current response is linear. To take advantage of this linear response, the TIA is used to perform the conversion of the difference of currents between the two photodiodes into a voltage proportional to that same difference.

In section 5.4.1, the relation between the input optical power and output current was established by a very simplified model of the photodetector. To make sense of it's output, the photodetector model must be improved by characterizing it's parameters such as responsivity (at various power levels and frequencies), bandwidth, amplification and noise levels. Various of those parameters can be readly extracted from the device's manual, which are presented in the following tables and plots.

Parameter	**Value**
Max Responsivity	1.0 A/W

Tabela 5.4: Thorlabs PDB450C PIN parameters

Parameter	**Value**
Bandwidth	1 MHz
Voltage Gain	10 V/mW @ peak responsivity
Voltage Noise (RMS)	<180 µV (RMS)

Tabela 5.5: Thorlabs PDB450C MONITOR +/- output parameters

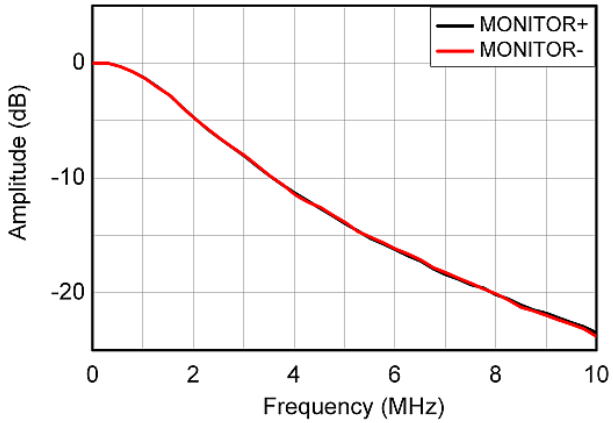


Figura 5.44: MONITOR +/- output response with frequency. [1]

The parameters of the RF OUTPUT have 5 values each, corresponding to the 5 gain settings of the TIA.

Parameter	**Values**					
Bandwidth(-3dB)	150	45	4	0.3	0.1	MHz
Transimpedance Gain	10^3	10^4	10^5	10^6	10^7	V/A
Conversion Gain	10^3	10^4	10^5	10^6	10^7	V/W
Overall Output Voltage Noise (RMS)	0.50	0.80	1.0	1.1	2.0	mV

Tabela 5.6: Thorlabs PDB450C RF OUTPUT parameters

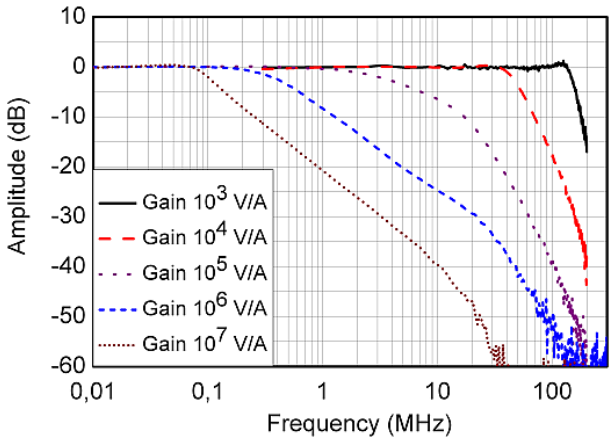


Figura 5.45: RF output response with frequency, for various gain values. [1]

Data analysis

For each configuration of amplitude and frequency of the input signal, the photodetector output voltage is collected by the Digital Oscilloscope during a time interval and saved in a

data file. The data consists on a sequence of pulses and it's analysis will be focused on the samples with equal phase. The steps will be the following

1. For each phase value, the average and variance of the samples with the same phase is calculated;

2. The representative amplitude and variance for the present configuration will be the obtained from the middle of the maximum plateau.

To confirm the theoretical results obtained in section 5.4.1, two experimental setups will be created. In the first experiement, we will study quantum noise in the single homodyne detection. The experimental setup will be based on the paper [5]. In the second experiment, we will study quantum noise in the double homodyne detection setup, which will be basically an extension of the single homodyne detection setup.

Single homodyne detection

To keep the experiment simple and avoid extra sources of noise, we will avoid using black boxes wich have complicated inner workings, having a preference in using simple components, as shown in fig. 5.46:

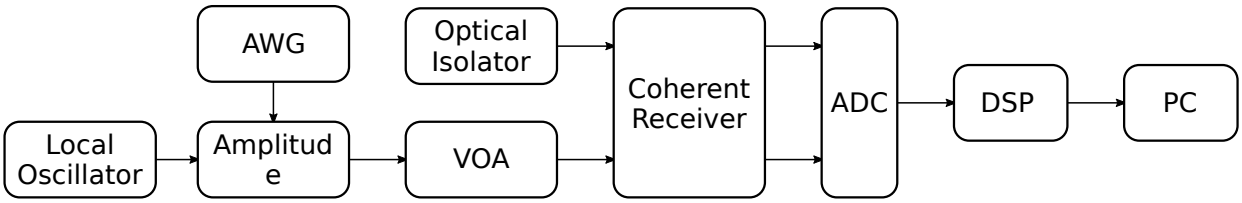


Figura 5.46: Experimental setup

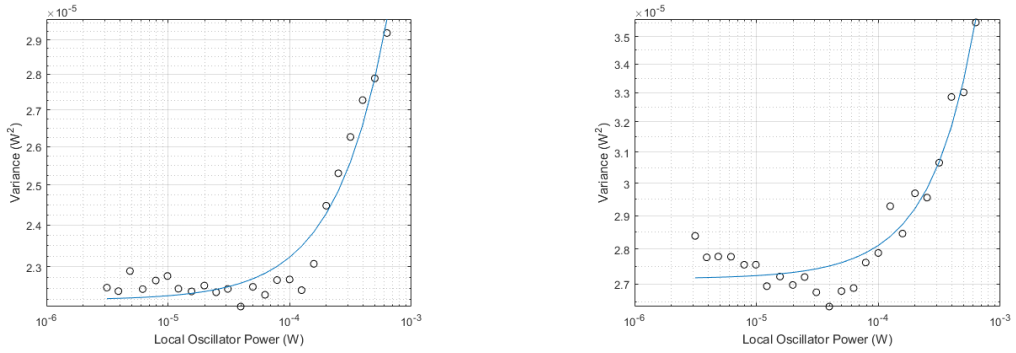
Material list

Device	**Description**
Local Oscillator	Yenista OSICS Band C/AG
BS	Beam Splitter
Pulse Generator	HP 8116A Pulse Generator
Amplitude Modulator	Mach Zehnder SDL OC 48
VOA	Eigenlicht Power Meter 420
VOA	Thorlabs VOA 45-APC
PIN	Thorlabs PDB 450C
ADC	Picoscope 6403D

A single laser is splitted and used as the source for the signal (S) and the local oscillator (LO). The signal beam is pulsed and highly attenuated. The local oscillator is also attenuaded, but not pulsed. The signal and local oscillator interfere in a Beam Splitter originating two beams which are then converted to voltages in the PIN. These voltages are read in the Digital Oscilator (OSC) and collect in the computer. In the post processing phase, the quantum noise is measured by applying a difference between the two beams and measuring it's variance.
The second stage of the experiment will be very similar to the first one, in which the signal and local oscillator branches will be divided. One of the new branches of the local oscilator will suffer a phase delay of $\pi/2$, in order to measure the quadrature component of the incoming signal.

5.4.4 Comparative analysis

Given the theoretical, simulated and experimental frameworks, we will now compare the results obtained by each of them.



(a) X quadrature (b) Y quadrature

Figura 5.47: Noise variance dependency with local oscilator power for two different quadratures. Experimental vs fitted data.

Figures $5.47a$ and $5.47b$ show measurements of total noise for two different quadratures. For low power of LO, the noise variance flutuates around a constant value. For high power of LO, $(P_{LO} > 10^{-4}W)$, the variance of noise shows an increasing trend roughly proportional to P_{LO}^2. The polynomial fittings confirm this trend, showing a degree 2 coefficient much larger than the degree 1 coefficient

$$\text{Var}_X = 2.22{\times}10^{-5} + 9.6{\times}10^{-3}P_{LO} + 3.40P_{LO}^2 \tag{5.64}$$

$$\text{Var}_Y = 2.71{\times}10^{-5} + 8.9{\times}10^{-3}P_{LO} + 7.25P_{LO}^2 \tag{5.65}$$

The expected growth should be proportional to P_{LO}, but the RIN noise, originated by the electric apparatus, which grows quadratically with the power, is dominating the noise

amplitude for large P_{LO}.
We see that both the simulation and experimental data display a similar behaviour, but the quadratic growth of noise for large P_{LO} was not predicted in the simulations.

5.4.5 Known problems

5.5 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 [**?**]. 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.

5.5.1 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 [**?**]. One of the optical processes available as a source of randomness is the splitting of a polarized single photon beam.

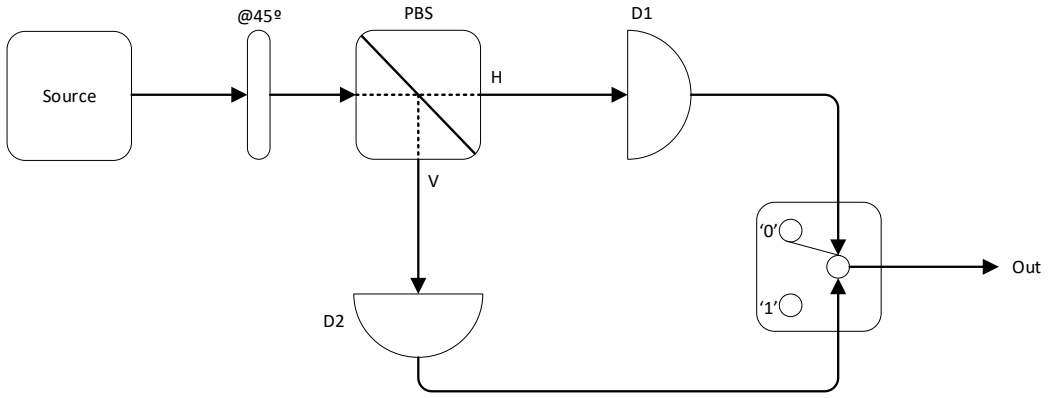


Figura 5.48: Source of randomness with a polarization beam splitter PBS where the incoming light is linearly polarized at $45°$ with respect to the PBS. Figure adapted from [**?**].

The principle of operation of the random generator is shown in figure 5.48. 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.

5.5.2 Simulation Analysis

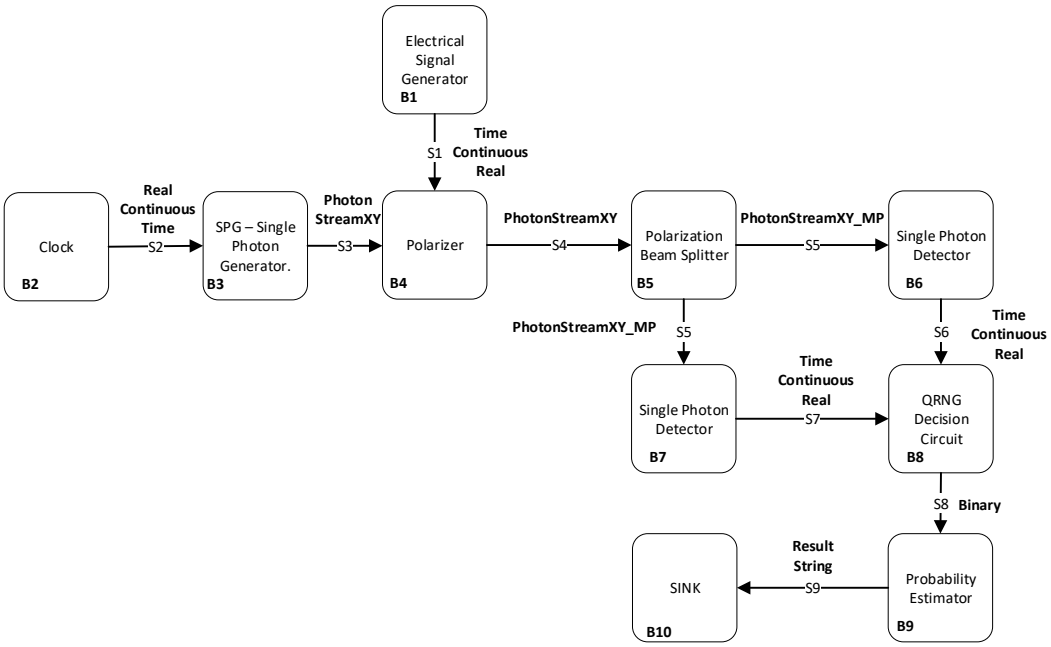


Figura 5.49: Block diagram of the simulation of a Quantum Random Generator.

The simulation diagram of the setup described in the previous section is presented in figure 5.49. 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 5.7 are presented the input parameters of the system.

Tabela 5.7: System Input Parameters

Parameter	Default Value
RateOfPhotons	1e6
NumberOfSamplesPerSymbol	16
PolarizerAngle	45.0

In table 5.8 are presented the system signals to implement the simulation presented in figure 5.49.

Tabela 5.8: 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 5.9 presents the header files used to implement the simulation as well as the specific parameters that should be set in each block. Finally, table 5.10 presents the source files.

Tabela 5.9: 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 5.10: 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

$$\begin{aligned} X(\omega) &= 1 \quad \text{,if } \omega \in A \\ X(\omega) &= 1 \quad \text{,if } \omega \in \bar{A} \end{aligned} \tag{5.66}$$

This way,

$$\begin{aligned} P(X = 1) = &\quad P(A) \quad = p \\ P(X = 0) = &\quad P(\bar{A}) \quad = 1 - p \end{aligned} \tag{5.67}$$

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

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 different

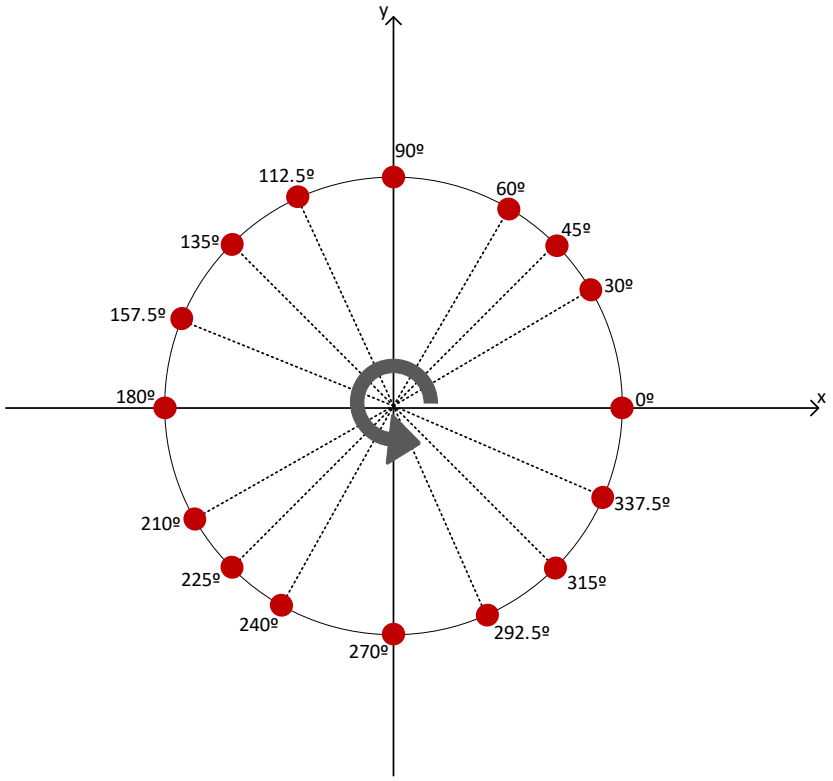


Figura 5.50: Angles used to perform the qrng simulation for $N = 1 \times 10^8$ samples.

angles of polarization shown in figure 5.50 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.

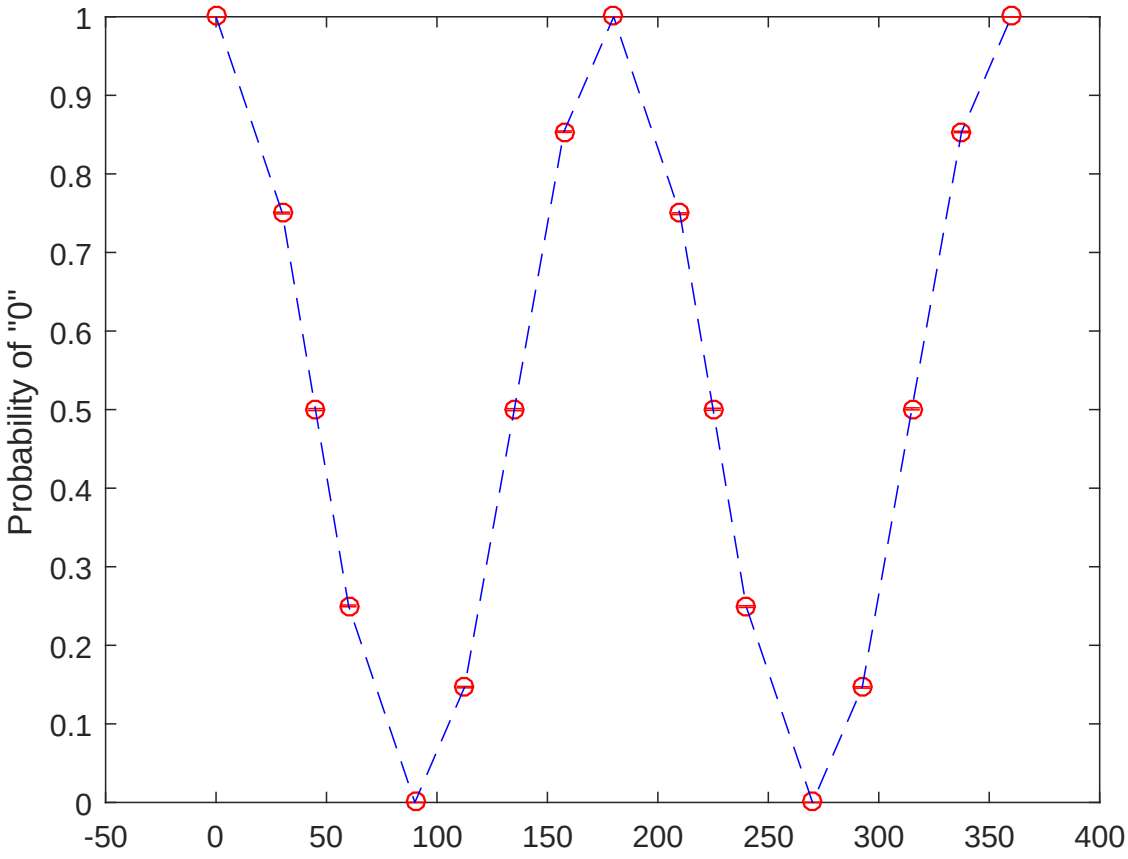


Figura 5.51: Probability of outputs a number "0" depending on the polarization angle.

Figure 5.51 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 5.52 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.

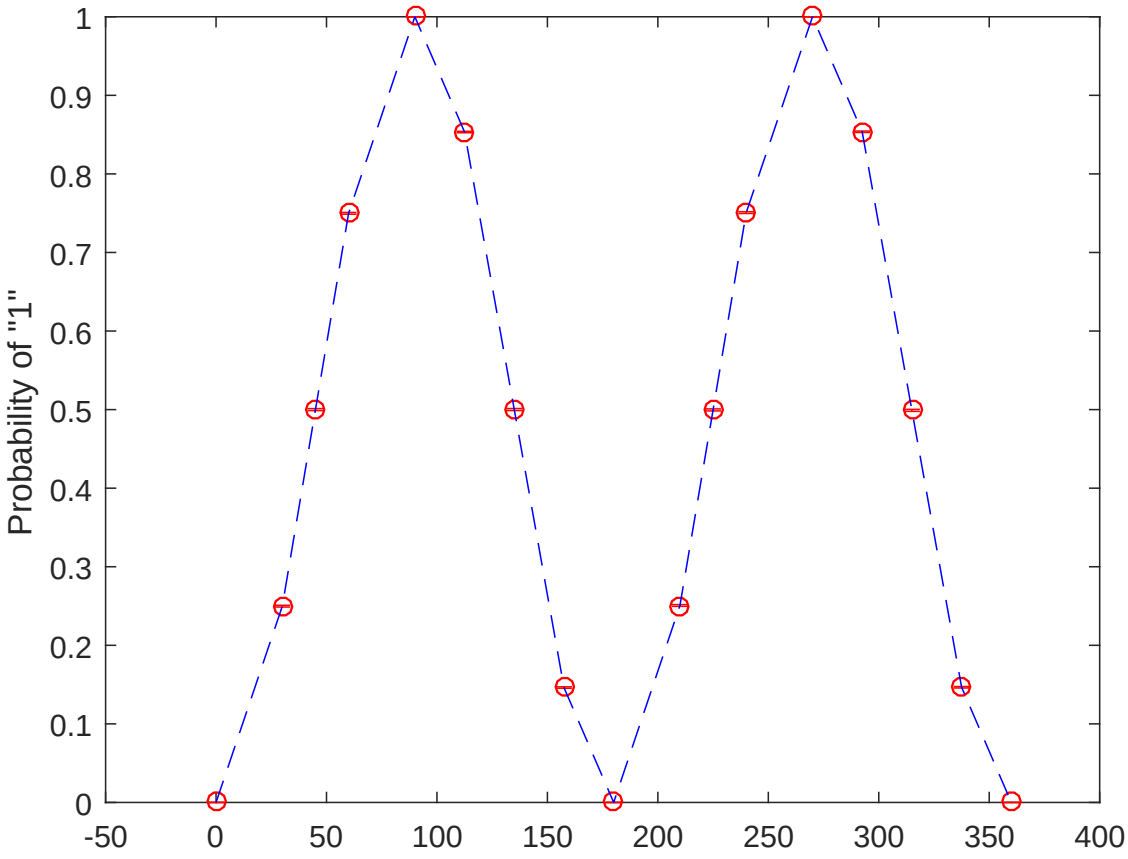


Figura 5.52: Probability of outputs a number "1" depending on the polarization angle.

5.5.3 Experimental Analysis

In order to have a real experimental quantum random number generator, a setup shown in figure 5.53 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.

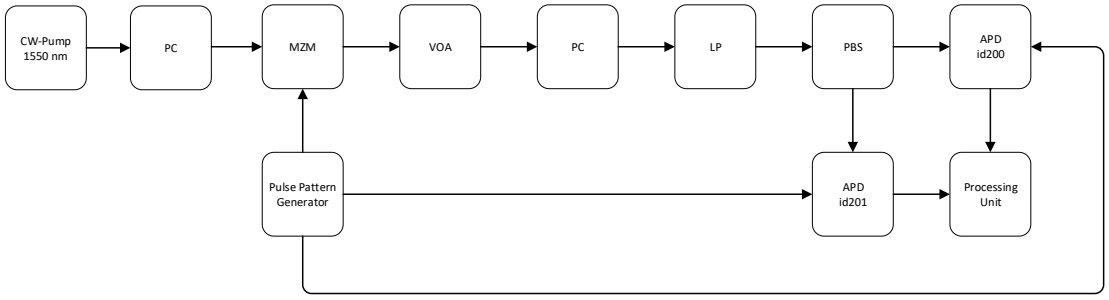


Figura 5.53: Experimental setup to implement a quantum random number generator.

5.5.4 Open Issues

References

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

[2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.

[3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

[4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.

[5] Yue-Meng Chi, Bing Qi, Wen Zhu, Li Qian, Hoi-Kwong Lo, Sun-Hyun Youn, AI Lvovsky, and Liang Tian. A balanced homodyne detector for high-rate gaussian-modulated coherent-state quantum key distribution. *New Journal of Physics*, 13(1):013003, 2011.

[6] 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.

5.6 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.

5.6.1 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 5.54.

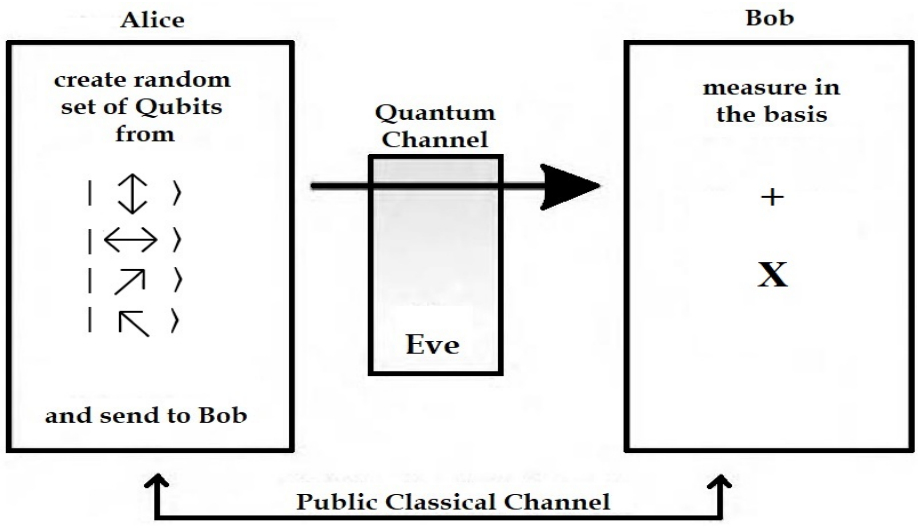


Figura 5.54: Basic QKD Model. Alice and Bob are connected by 2 communication channels, quantum and classical, with an eavesdropper, Eve, in the quantum communication channel (figure adapted from [3]).

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

	Rectilinear Basis,+	*Diagonal Basis,x*
0	0	-45
1	90	45

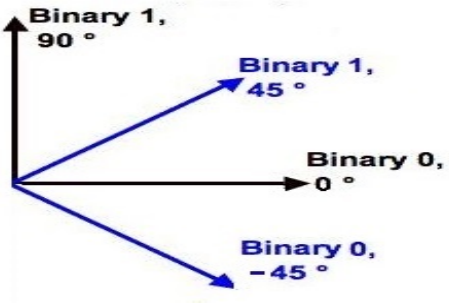


Figura 5.55: Simple representation of the bit encoding using the corresponding bases. [2]

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, $\times$, basis of B_A, respectively.

$$R_{A1} = \{0,1,1,0,1,0,0,1,1,0,1,1,1,0,0,1,0,0,0,1\}$$
$$R_{A2} = \{0,0,1,0,1,1,1,0,1,1,1,0,1,0,0,0,1,0,1,0\}$$
$$=$$
$$\{+,+,\times,+,\times,\times,\times,+,\times,\times,\times,+,\times,+,+,+,\times,+,\times,+\}$$

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.

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

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 and so there is attenuation. If both photon detectors click, a false positive was detected. In the measurements, M_B, the attenuation 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.

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

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 and mixes the bits using a known algorithm by Alice and Bob.

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
BAB	1	0	1	0	1	0	1	1	0	1	0	1	1	1	0	1

3. Using also the negated XOR, Bob also mixes the bits with the same algorithm used by Alice and obtains the same key, K_{AB}

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

To determine the Quantum Bit Error Rate (QBER), Bob will reveal a bit sequence from the deduced key to Alice. Alice then returns to Bob the estimated QBER value, mQBER, with a confidence interval, [qLB, qLB]. To verify if the channel is reliable or not the flowchart presented in figure 5.56 must be followed and a acceptable QBER limit, qLim, must be imposed. Alice performs the QBER and confidence interval calculation by using the equations in the Bit Error Rate section, but applied to this protocol. By following the presented flowchart if the qLimit is inside the confidence interval, Bob should use more bits from key until qLim is bigger than qLB and qUB. If this condition is achieved, the QBER is successfully estimated and a reliable communication can be achieved. Otherwise the link is declared as unreliable and no further communication is made.

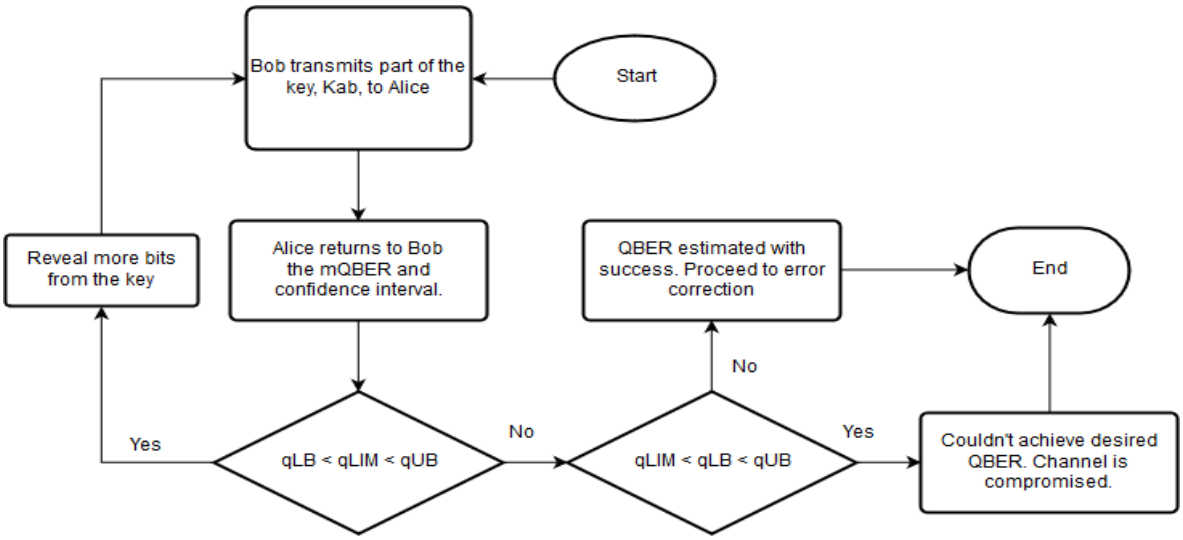


Figura 5.56: Flowchart to determine if the channel is reliable or not.

The presence of a eavesdropper will carry the risk of changing the bits. This will produce disagreement between Bob and Alice in the bits they should agree. When Eve measures and retransmits a photon she can deduce correctly with a probability of 50%. So by learning the correct polarization of half of the photons, the induced error is 25%. Alice and Bob can detect Eve presence by sacrifice the secrecy of some bits in order to test.

References

[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

Basis	
0	$+$
1	$\times$

	Basis "+"
0	$\rightarrow (0°)$
1	$\uparrow (90°)$

	Basis "$\times$"
0	$\searrow (-45°)$
1	$\nearrow (45°)$

1. Alice randomly generate a bit sequence with length ks being, in this case, $k = 2$ and $s = 4$ as it was defined at the beginning. Therefore, she must define two sets randomly: S_{A1} which contains the basis values; and S_{A2}, which contains the key values.

   In that case, lets assume she gets the following sets S_{A1} and S_{A2}:

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

   $$S_{A2} = \{1, 1, 0, 0, 0, 1, 0, 0\}.$$

2. Next, Alice sends to Bob throughout a quantum channel ks photons encrypted using the basis defined in S_{A1} and according to the keys defined in S_{A2}.

   In the current example, Alice sends the photons, throughout a quantum channel, according to the following,

   $$S_{AB} = \{\uparrow, \nearrow, \searrow, \rightarrow, \rightarrow, \nearrow, \rightarrow, \searrow\}.$$
   $$S_{AB} = \{90°, 45°, -45°, 0°, 0°, 45°, 0°, -45°\}.$$

3. Bob also randomly generates ks bits, which are going to define his measurement basis, S_{B1}. He will measure the photons sent by Alice. Lets assume:

   $$S_{B1} = \{0, 1, 0, 1, 0, 1, 1, 1\}.$$

   When Bob receives photons from Alice, he measures them using the basis defined in S_{B1}. In the current example, S_{B1} corresponds to the following set:

   $$\{+, \times, +, \times, +, \times, \times, \times\}.$$

   Bob will get ks results:

   $$S_{B1'} = \{1, 1, 0, 1, 0, 1, 1, 0\}.$$

4. Bob will send a *Hash Function* result HASH1 to Alice. This value will do Bob's commitment with the measurements done. In this case, this *Hash Function* is calculated from *SHA-256* algorithm for each pair (Basis from S_{B1} and measured value from $S_{B1'}$), i.e Bob sends to Alice sk pairs as his commitment. In this case, Bob sends eight pairs encoded using a *Hash Function* which is also send to Alice. From that moment on Bob cannot change his commitment neither the basis which he uses to measure the photons sent by Alice.

5. Once Alice has received the confirmation of measurement from Bob, she sends throughout a classical channel the basis which she has used to codify the photons, which in this case we assumed $S_{A1} = \{0, 1, 1, 0, 0, 1, 0, 1\}$.

6. In order to know which photons were measured correctly, Bob does the operation $S_{B2} = S_{B1} \oplus S_{A1}$. In the current example the operation will be:

S_{B1}	0	1	0	1	0	1	1	1
S_{A1}	0	1	1	0	0	1	0	1
$\oplus$	1	1	0	0	1	1	0	1

   In this way, Bob gets

$$S_{B2} = \{1, 1, 0, 0, 1, 1, 0, 1\}.$$

   When Bob uses the right basis he gets the values correctly, when he uses the wrong basis he just guess the value. The values "1" correspond to the values he measured correctly and "0" to the values he just guessed.

   Next, Bob sends to Alice, through a classical channel, information about the minimum number between "ones" and "zeros", i.e

$$n = min(\#0, \#1) = 3,$$

   where $\#0$ represents the number of zeros in S_{B2} and $\#1$ the number of ones in S_{B2}. At this time, Alice must be able to know if Bob is being honest or not. Therefore, she will open Bob's commitment from *step 4* and she verify if the number n sent by Bob is according with the commitment values sent by him. In other words, she opens a number of pairs committed by Bob which is known from the beginning.

7. If $n < s$, being s the message's size, Alice and Bob will repeat the steps from 1 to 7. In this case, $n = 3$ which is smaller than $s = 4$. Therefore, Alice and Bob repeat the steps from 1 to 7 in order to enlarge Bob's sets S_{B1} and S_{B2} as well as Alice's sets S_{A1} and S_{A2}.

8. Lets assume :

$$S_{B1} = \{1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1\}.$$

At Alice's side the new sets S_{A1}, which contains the basis values, and S_{A2}, which contains the key values, will be the following:

$$S_{A1} = \{0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0\},$$
$$S_{A2} = \{1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1\}.$$

Finally, for $S_{B2} = S_{B1} \oplus S_{A1}$ Bob gets the following sequence:

$$S_{B2} = \{1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1\}.$$

Note that the sets were enlarge in the second iteration.

9. At this time, Bob sends again to Alice, through a classical channel, the minimum number between "ones" and "zeros", $n = min(\#0, \#1)$. In this case, n is equal to 7 which is the number of zeros.

10. Alice checks if $n > s$ and acknowledge to Bob that she already knows that $n > s$. In this case, $n = 7$ and $s = 4$ being $n > s$ a valid condition.

11. Next, Bob defines two new sub-sets, I_0 and I_1. I_0 is a set of values with photons array positions which Bob just guessed the measurement since he did not measure them with the same basis as Alice, I_1 is a set of values with photons array positions which Bob measured correctly since he used the same basis as Alice used to encoded them.

    In this example, Bob defines two sub-sets with size $s = 4$:

    $$I_0 = \{3, 4, 7, 11\},$$

    and

    $$I_1 = \{2, 5, 6, 13\},$$

    where I_0 is the sequence of positions in which Bob was wrong about basis measurement and I_1 is the sequence of positions in which Bob was right about basis measurement. Bob sends to Alice the set S_b

    Thus, if Bob wants to know m_0 he must send to Alice throughout a classical channel the set $S_0 = \{I_1, I_0\}$, otherwise if he wants to know m_1 he must send to Alice throughout a classical channel the set $S_1 = \{I_0, I_1\}$.

12. With both the received set S_b and the hash function value HASH1, Alice must be able to prove that Bob has being honest.

13. Lets assume Bob sent $S_0 = \{I_1, I_0\}$. Alice defines two encryption keys K_0 and K_1 using the values in positions defined by Bob in the set sent by him. In this example, lets assume:

    $$K_0 = \{1, 0, 1, 0\}$$
    $$K_1 = \{0, 0, 0, 1\}.$$

Alice does the following operations:

$$m = \{m_0 \oplus K_0, m_1 \oplus K_1\}.$$

m_0	0	0	1	1
K_0	1	0	1	0
$\oplus$	1	0	0	1

m_1	0	0	0	1
K_1	0	0	0	1
$\oplus$	0	0	0	0

Adding the two results, m will be:

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

After that, Alice sends to Bob the encrypted message m through a classical channel.

14. When Bob receives the message m, in the same way as Alice, Bob uses S_{B1}, values of positions given by I_1 and I_0 and does the decrypted operation. In this case, he does following operation:

m	1	0	0	1	0	0	0	0
	1	0	1	0	0	1	1	0
$\oplus$	0	0	1	1	0	1	1	0

The first four bits corresponds to message 1 and he received $\{0, 0, 1, 1\}$, which is the right message m_0 and $\{0, 1, 1, 0\}$ which is a wrong message for m_1.

5.6.2 Simulation Setup

5.6.3 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 5.57 a top level diagram is presented. Then it will be presented the block diagram of the transmitter block (Alice) in figure 5.58, the receiver block (Bob) in figure 5.59 and finally the eavesdropper block (Eve) in figure 5.60.

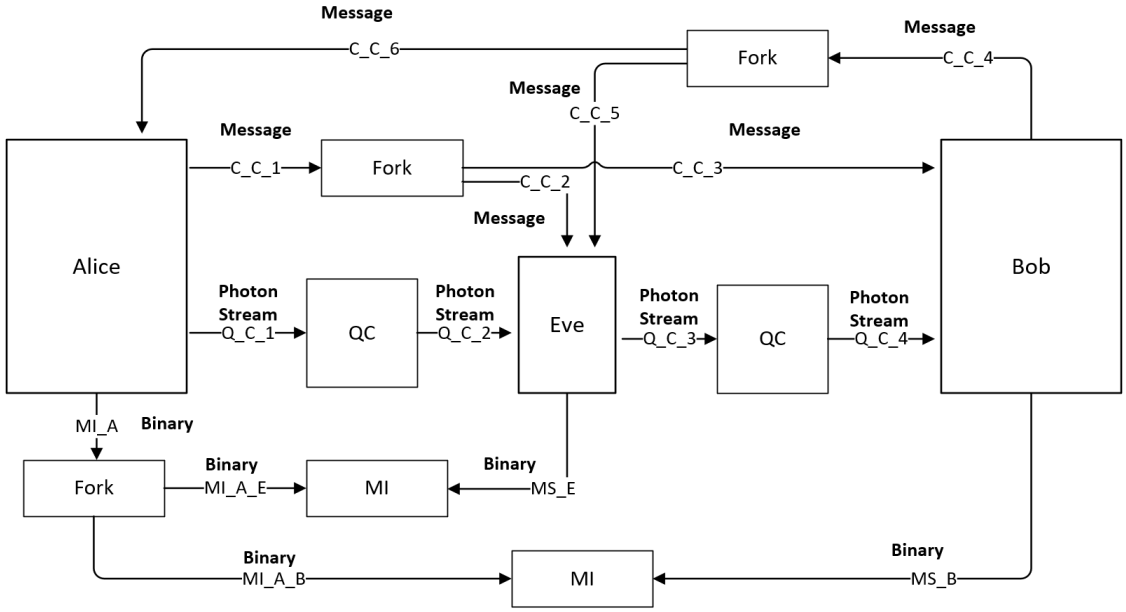


Figura 5.57: Simulation diagram at a top level

Figure 5.57 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 5.60. 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 5.58 one can observe a block diagram of the simulation at Alice's side. As it 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

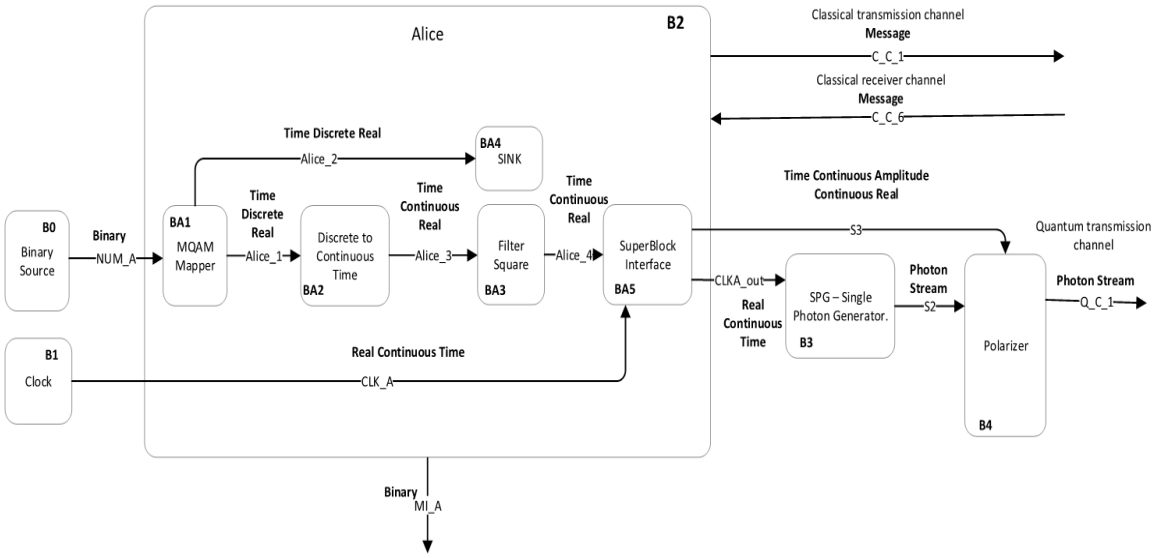


Figura 5.58: Simulation diagram at Alice's side

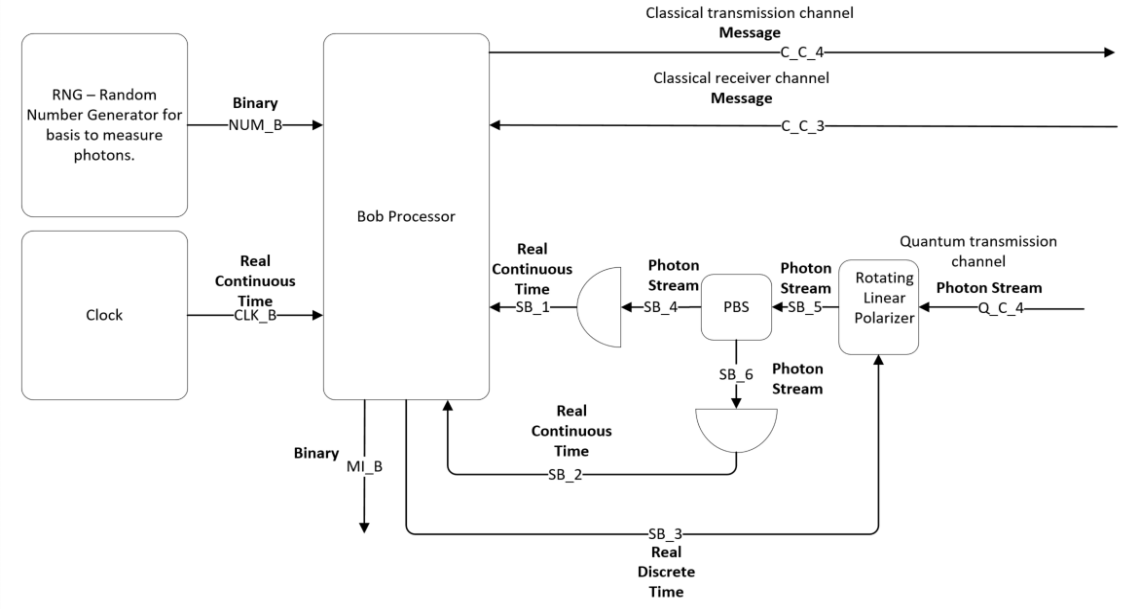


Figura 5.59: Simulation diagram at Bob's side

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 5.59 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.

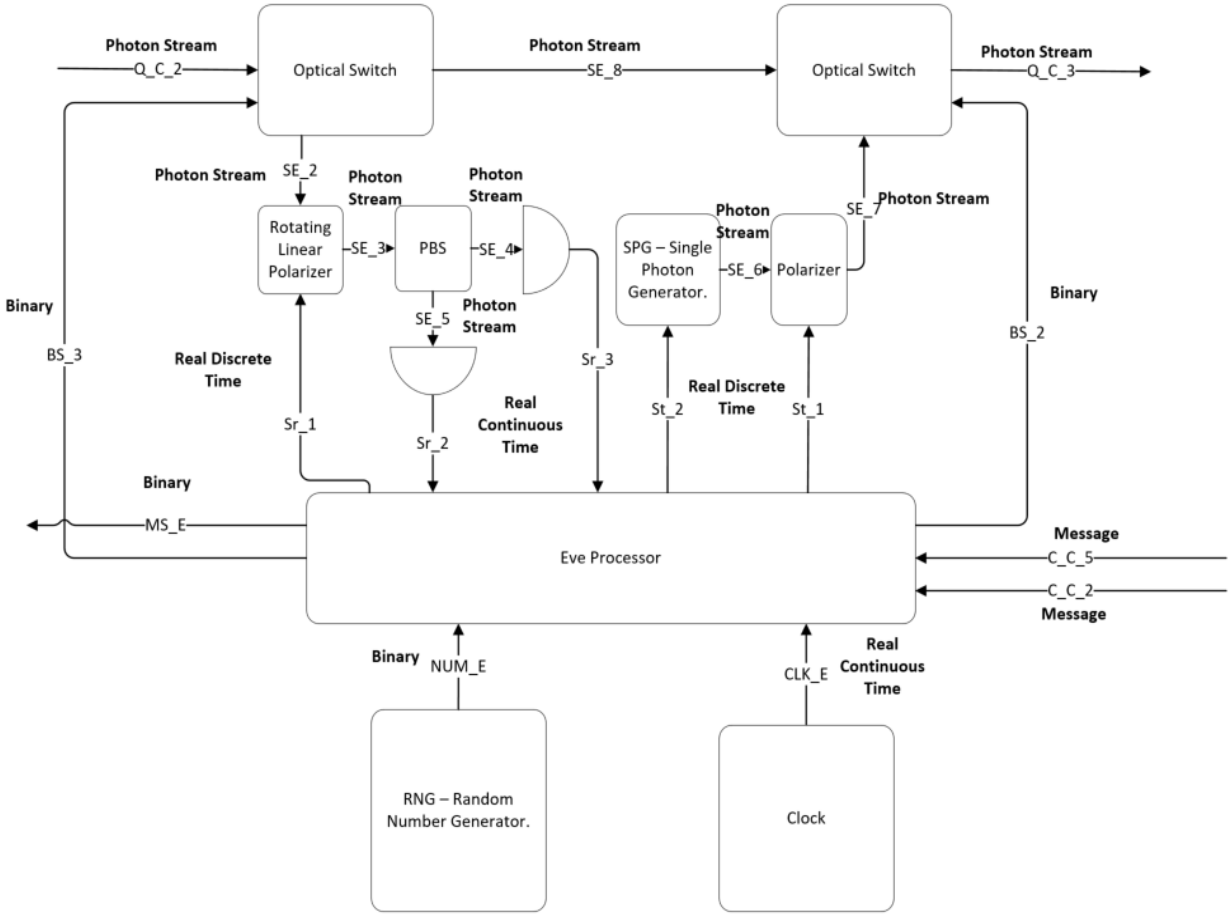


Figura 5.60: Simulation diagram at Eve's side

Figure 5.60 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 5.59 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 5.58 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.

Tabela 5.11: System Signals

Signal name	**Signal type**	**Status**
C_C_1 ... C_C_6	Message	
Q_C_1 .. Q_C_4	Photon Stream	
Ms_A, Ms_B, Ms_E	Binary	
NUM_A , NUM_B, NUM_E	Binary	
CLK_A, CLK_B, CLK_E	Real continuous time	
SB_1, SB_2, Sr_1, Sr_2	Real continuous time	
SA_1, SA_2, St_1, St_2	Real discrete time	
SA_3	Photon Stream	
S_2, S_3, S_4, S_5	Photon Stream	
BS_1, BS_2	Binary	

Table 5.14 presents the system signals as well as them type.

Tabela 5.12: System Input Parameters

Parameter	**Default Value**	**Description**
SymbolRate	100K	
NumberOfBits	Number of photons that Alice sends to Bob	

Tabela 5.13: Header Files

File name	Description	Status
binary_source.h		
single_photon_source.h		
single_photon_detector.h		
optical_switch.h		Missing
polarization_beam_splitter.h		
mutual_information.h		Missing
bit_error_rate.h		
clock.h		
fiber.h		
qrng_decision_circuit.h		
message_to_send.h		Missing
message_to_receive.h		Missing
netxpto.h		

Tabela 5.14: Source Files

File name	Description	Status
binary_source.cpp		
single_photon_sourcer.cpp		
single_photon_detector.cpp		
optical_switch.cpp		Missing
polarization_beam_splitter.cpp		
mutual_information.cpp		Missing
bit_error_rate.cpp		
clock.cpp		
fiber.cpp		
qrng_decision_circuit.cpp		
message_to_send.cpp		Missing
message_to_receive.cpp		Missing
netxpto.cpp		
bb84_sdf.cpp		

Capítulo 6

Library

6.1 Add

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

Input Parameters

This block takes no parameters.

Functional Description

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

Input Signals

Number: 2
Type: Real, Complex or Complex_XY signal (ContinuousTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Real, Complex or Complex_XY signal (ContinuousTimeContinuousAmplitude)

6.2 Balanced Beam Splitter

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

Input Parameters

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

Functional Description

The structure of the beam splitter can be controlled with the parameter mode.
When **Mode = 0** the beam splitter will have one input port and two output ports - **1x2 Beam Splitter**. If Mode has a value different than 0, the splitter will have two input ports and two output ports - **2x2 Beam Splitter**.
Considering the first case, the matrix representing a 2x2 Beam Splitter can be summarized in the following way,

$$M_{BS} = \frac{\sqrt{2}}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6.1}$$

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \tag{6.2}$$

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

Input Signals

Number: 1 or 2
Type: Complex

Output Signals

Number: 2
Type: Complex

References

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

[2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.

[3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

[4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.

[5] Yue-Meng Chi, Bing Qi, Wen Zhu, Li Qian, Hoi-Kwong Lo, Sun-Hyun Youn, AI Lvovsky, and Liang Tian. A balanced homodyne detector for high-rate gaussian-modulated coherent-state quantum key distribution. *New Journal of Physics*, 13(1):013003, 2011.

[6] 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 Bit Error Rate

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

Input Parameters

Name	Type	Default Value
Confidence	double	0.95
MidReportSize	integer	0
LowestMinorant	double	1×10^{-10}

Input Signals

Number: 2
Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional Description

This block accepts two binary strings and outputs a binary string, outputting a 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 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.

Theoretical Description

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:

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \tag{6.3}$$

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

$N_T > 40$ [6]:

$$\text{BER}_{\text{UB}} = \widehat{\text{BER}} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 + (2 - \widehat{\text{BER}})\right] \quad (6.4)$$

$$\text{BER}_{\text{LB}} = \widehat{\text{BER}} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 - (1 + \widehat{\text{BER}})\right], \quad (6.5)$$

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

References

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

[2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.

[3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

[4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.

[5] Yue-Meng Chi, Bing Qi, Wen Zhu, Li Qian, Hoi-Kwong Lo, Sun-Hyun Youn, AI Lvovsky, and Liang Tian. A balanced homodyne detector for high-rate gaussian-modulated coherent-state quantum key distribution. *New Journal of Physics*, 13(1):013003, 2011.

[6] 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.

6.4 Binary Source

Header File	:	binary_source.h
Source File	:	binary_source.cpp

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

Tabela 6.1: Binary source input parameters

Methods

BinarySource(vector⟨Signal *⟩ &InputSig, vector⟨Signal *⟩ &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setMode(BinarySourceMode m) BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

Functional description

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

Random Mode Generates a 0 with probability *probabilityOfZero* and a 1 with probability 1-*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.

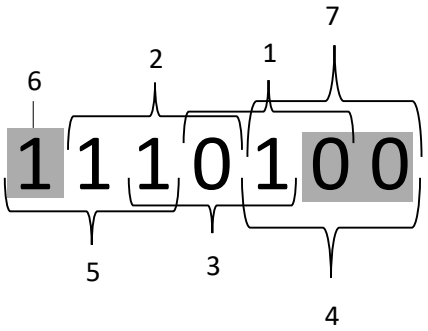


Figura 6.1: Example of a pseudorandom sequence with a pattern length equal to 3.

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.

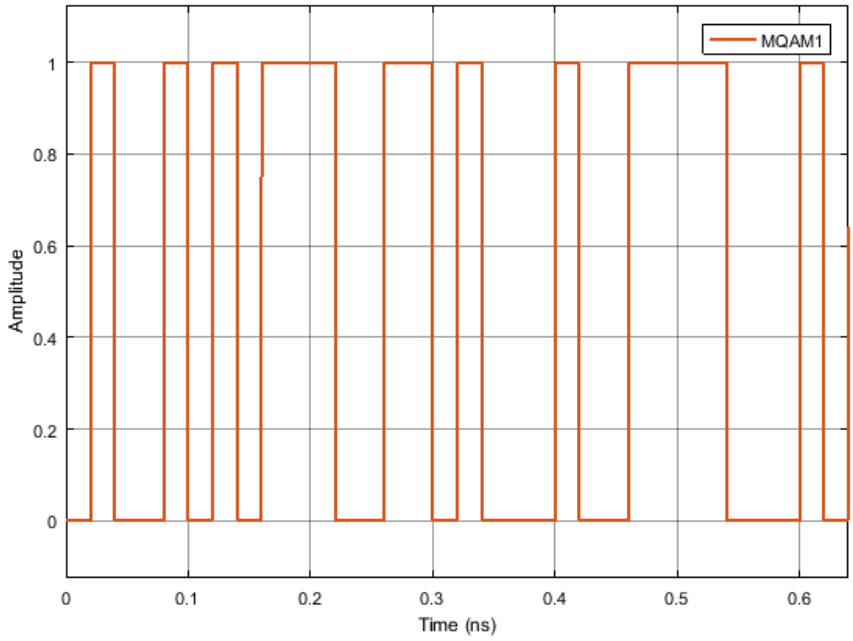


Figura 6.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

6.5 Bit Decider

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

Input Parameters

Name	Type	Default Value
decisionLevel	double	0.0

Functional Description

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

Input Signals

Number: 1
Type: Real signal (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

6.6 Clock

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

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

Input Parameters

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

Tabela 6.2: Binary source input parameters

Methods

Clock()

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

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per)

void setSamplingPeriod(double sPeriod)

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

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

Examples

Sugestions for future improvement

6.7 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

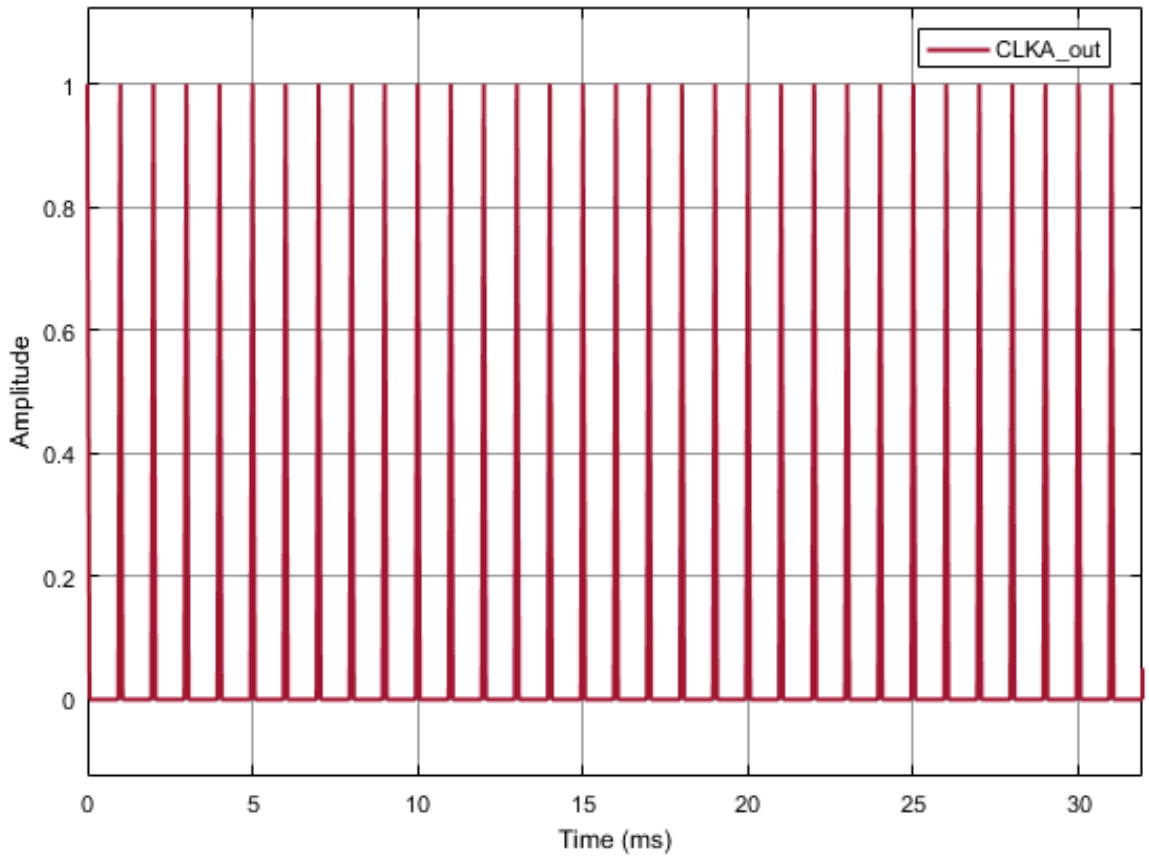


Figura 6.3: Example of the output signal of the clock without phase shift.

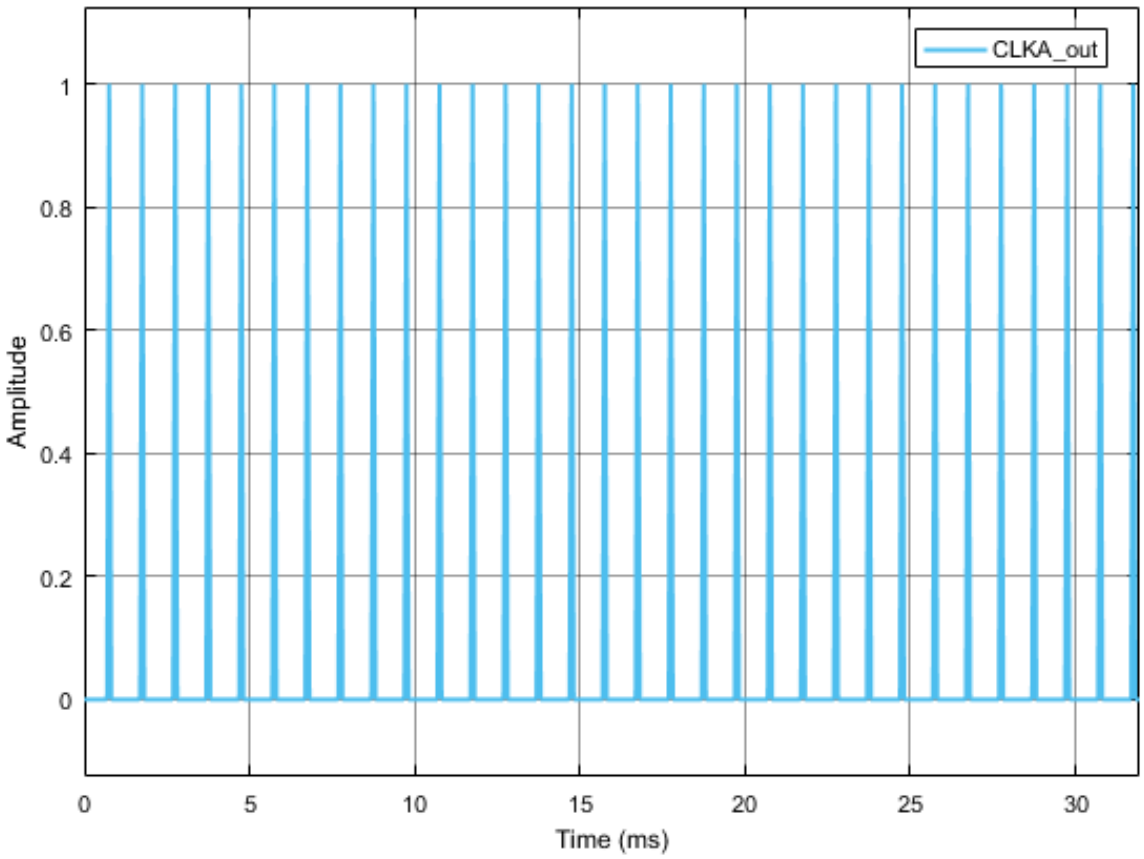


Figura 6.4: Example of the output signal of the clock with phase shift.

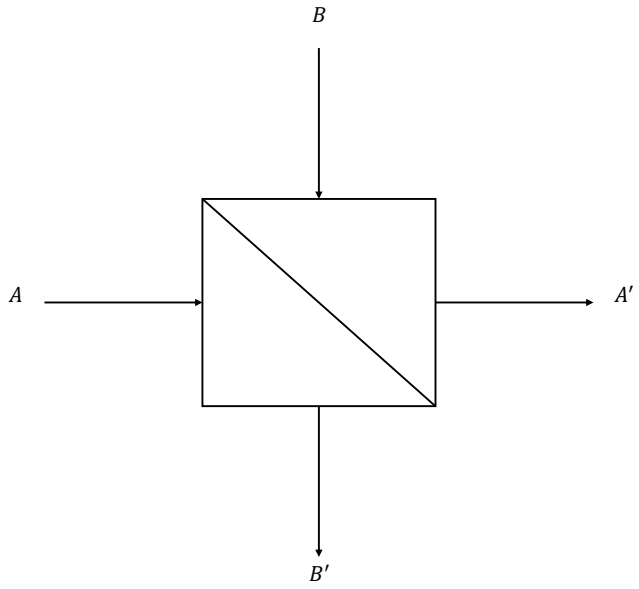


Figura 6.5: 2x2 coupler

6.8 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} . \begin{bmatrix} A \\ B \end{bmatrix} \tag{6.6}$$

Where, A and B represent inputs to the 2x2 coupler and A′ and B′ represent output of the 2x2 coupler. Parameters T and R represent transmitted and reflected part respectively which can be quantified in the following form,

$$T = \sqrt{1 - \eta_R} \tag{6.7}$$

$$R = \sqrt{\eta_R} \tag{6.8}$$

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

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

6.9 Decoder

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

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

Input Parameters

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

Tabela 6.3: Binary source input parameters

Methods

Decoder()

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

void initialize(void)

bool runBlock(void)

void setM(int mValue)

void getM()

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues>getIqAmplitudes()

Functional description

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

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

Input Signals

Number: 1

Type: Electrical complex (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Binary

Examples

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

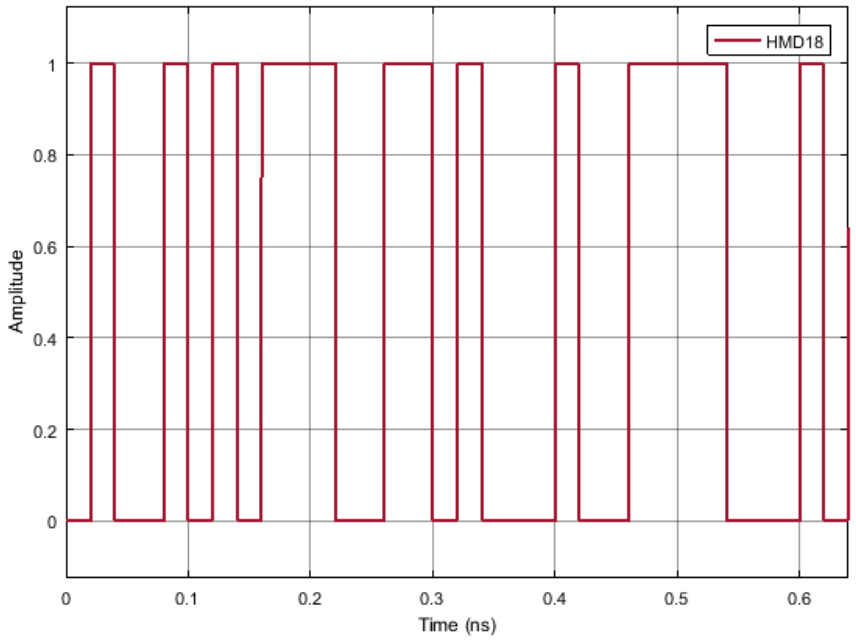


Figura 6.6: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

6.10 Discrete To Continuous Time

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

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

Input Parameters

Parameter	Type	Values	Default
numberOfSamplesPerSymbol	int	any	8

Tabela 6.4: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)

int const getNumberOfSamplesPerSymbol(void)

Functional Description

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

Input Signals

Number : 1

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

Output Signals

Number : 1

Type : Sequence of Dirac delta functions (ContinuousTimeDiscreteAmplitude)

Example

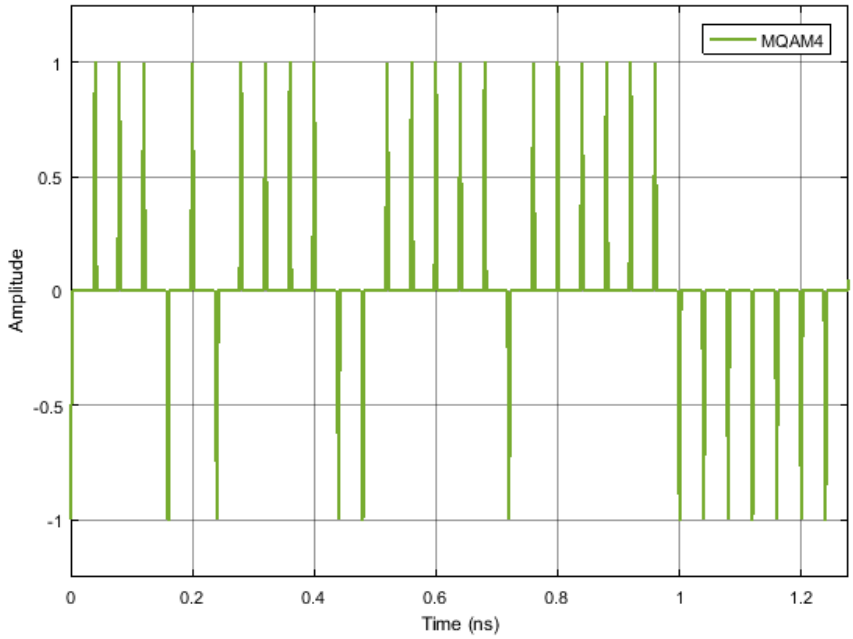


Figura 6.7: Example of the type of signal generated by this block for a binary sequence 0100...

6.11 Electrical Signal Generator

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

6.11.1 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 signalFunction
(ContinuousWave)

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.

6.12 Fork

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

Input Parameters

— NA —

Input Signals

Number: 1
Type: Any type (BinaryValue, IntegerValue, RealValue, ComplexValue, ComplexValueXY, PhotonValue, PhotonValueMP, Message)

Output Signals

Number: 2
Type: Same as applied to the input.

Number: 3
Type: Same as applied to the input.

Functional Description

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

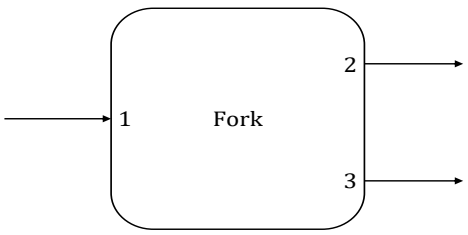


Figura 6.8: Fork

6.13 Gaussian Source

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

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

Input Parameters

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

Tabela 6.5: Gaussian source input parameters

Methods

GaussianSource()

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

void initialize(void);

bool runBlock(void);

void setAverage(double Average) ;

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Continuous signal (TimeDiscreteAmplitudeContinuousReal)

Examples

Sugestions for future improvement

6.14 MQAM Receiver

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

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

This block of code simulates the reception and demodulation of an optical signal (which is the input signal of the system) outputing a binary signal. A simplified schematic representation of this block is shown in figure 6.9.

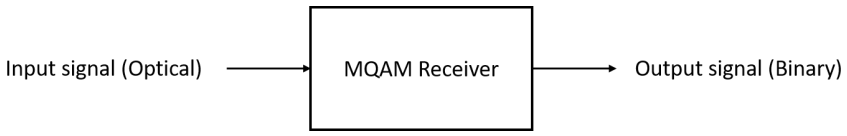


Figura 6.9: Basic configuration of the MQAM receiver

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*.

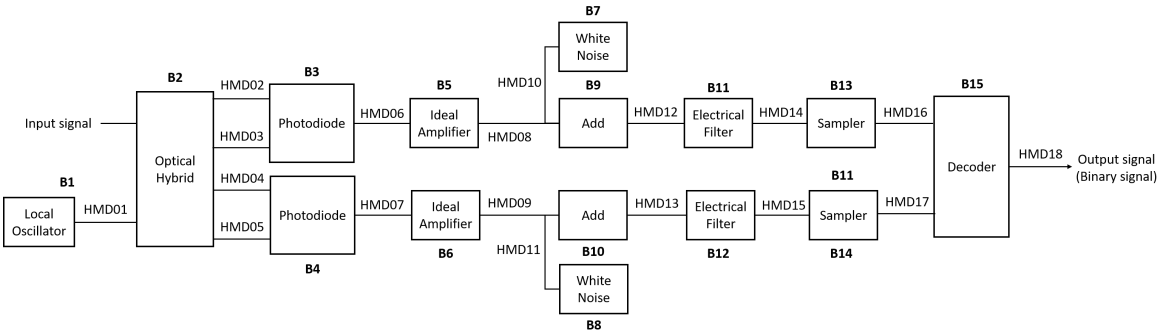


Figura 6.10: Schematic representation of the block homodyne receiver.

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 6.11) 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	Given by *symbolPeriod/samplesPerSym*

Tabela 6.6: 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

6.15 IQ Modulator

Header File	:	iq_modulator.h
Source File	:	iq_modulator.cpp

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

Tabela 6.7: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setOutputOpticalPower(double outOpticalPower)

void setOutputOpticalPower_dBm(double outOpticalPower_dBm)

void setOutputOpticalWavelength(double outOpticalWavelength)

void setOutputOpticalFrequency(double outOpticalFrequency)

Functional Description

This block takes the two parts of the signal: in phase and in amplitude and it combines them to produce a complex signal that contains information about the amplitude and the phase.

This complex signal is multiplied by $\frac{1}{2}\sqrt{outputOpticalPower}$ 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

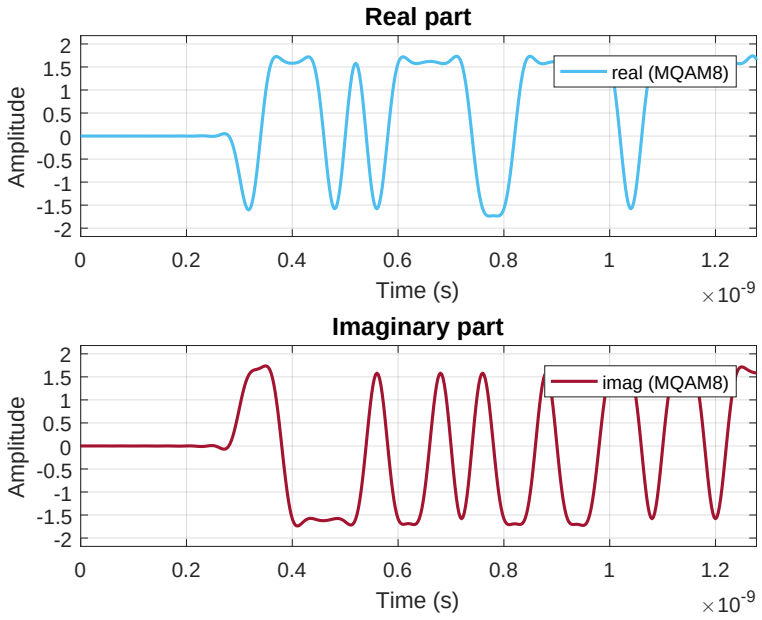


Figura 6.11: Example of a signal generated by this block for the initial binary signal 0100...

6.16 Local Oscillator

Header File	:	local_oscillator.h
Source File	:	local_oscillator.cpp
Version	:	20180130

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

Input Parameters

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

Tabela 6.8: Binary source input parameters

Methods

LocalOscillator()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setSymbolPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setFrequency(double freq);

void setPhase(double lOscillatorPhase);

void setSignaltoNoiseRatio(double sNoiseRatio);

void setLaserLinewidth(double laserLinewidth);

void setLaserRIN(double laserRIN);

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1
Type: Optical signal

6.17 Local Oscillator

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

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

Input Parameters

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

Tabela 6.9: Local oscillator input parameters

Methods

LocalOscillator()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setPhase(double lOscillatorPhase);

void setLaserLinewidth(double laserLinewidth);

double getLaserLinewidth();

void setLaserRIN(double LOlaserRIN);

double getLaserRIN();

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal

Examples

Sugestions for future improvement

6.18 MQAM Mapper

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

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

Input Parameters

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

Tabela 6.10: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setM(int mValue);

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues);

Functional Description

In the case of m=4 this block 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)

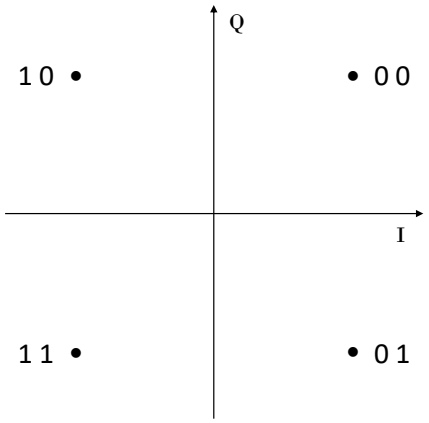


Figura 6.12: Constellation used to map the signal for m=4

Output Signals

Number : 2

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

Example

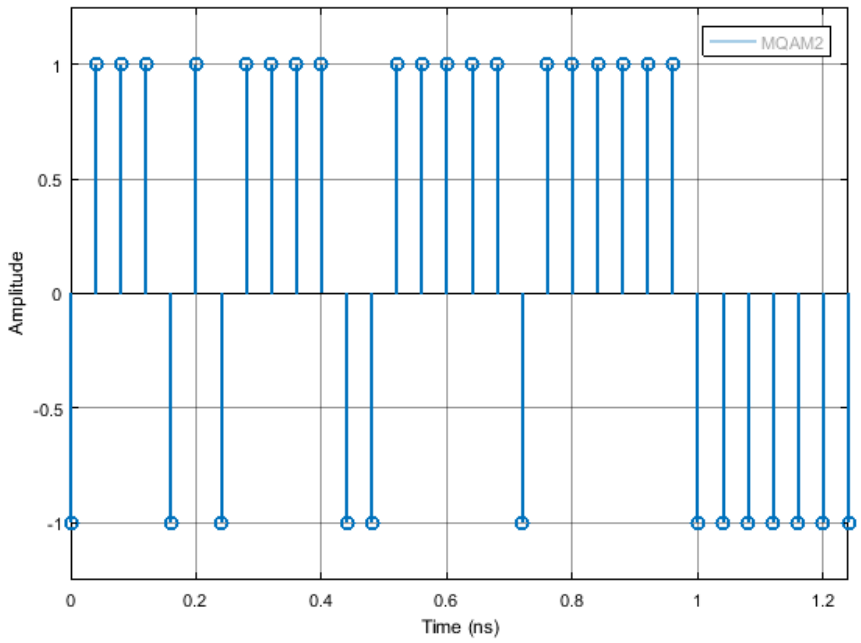


Figura 6.13: Example of the type of signal generated by this block for the initial binary signal 0100...

6.19 MQAM Transmitter

Header File	:	m_qam_transmitter.h
Source File	:	m_qam_transmitter.cpp

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.

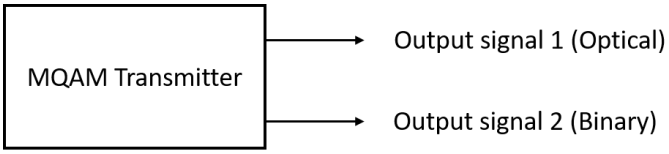


Figura 6.14: Basic configuration of the MQAM transmitter

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).

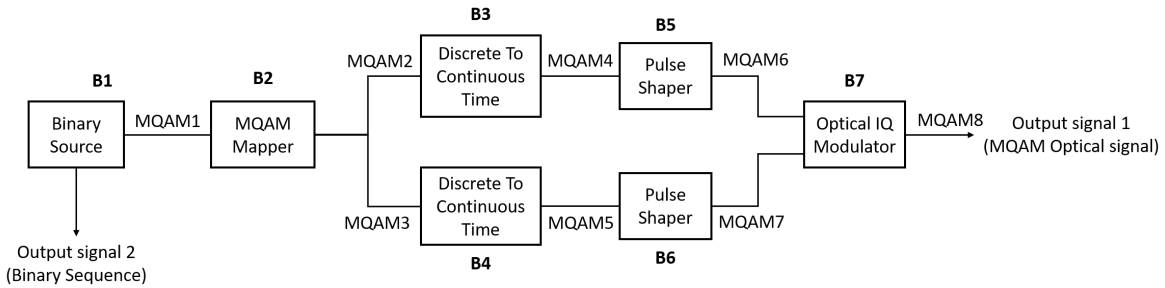


Figura 6.15: Schematic representation of the block MQAM transmitter.

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 6.11.

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 6.11: 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

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.

6.20 Netxpto

Header File	:	netxpto.h
	:	netxpto_20180118.h
Source File	:	netxpto.cpp
	:	netxpto_20180118.cpp

This block can work in two configurations: with an external clock or without it. In the latter it accepts two input signals one being the clock and the other the signal to be demodulated. In the other configuration there's only one input signal which is the signal.

The output signal is obtained by sampling the input signal with a predetermined samplig rate provided either internally or by the clock.

Input Parameters

Parameter	Type	Values	Default
samplesToSkip	int	any (smaller than the number of samples generated)	0

Tabela 6.12: Sampler input parameters

Methods

Sampler()

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

void initialize(void)

bool runBlock(void)

void setSamplesToSkip(`t_integer` sToSkip)

Functional description

This block can work with an external clock or without it.

In the case of having an external clock it accepts two input signals. The signal to be demodulate which is complex and a clock signal that is a sequence of Dirac delta functions with a predetermined period that corresponds to the sampling period. The signal and the clock signal are scanned and when the clock has the value of 1.0 the correspondent complex value of the signal is placed in the buffer corresponding to the output signal.

There's a detail worth noting. The electrical filter has an impulse response time length of 16 (in units of symbol period). This means that when modulating a bit the spike in the signal corresponding to that bit will appear 8 units of symbol period later. For this reason there's the need to skip the earlier samples of the signal when demodulating it. That's the purpose of the *samplesToSkip* parameter.

Between the binary source and the current block the signal is filtered twice which means that this effect has to be taken into account twice. Therefore the parameter *samplesToSkip* is given by $2 * 8 * samplesPerSymbol$.

Input Signals

Number: 1

Type: Electrical real (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical real (TimeDiscreteAmplitudeContinuousReal)

Examples

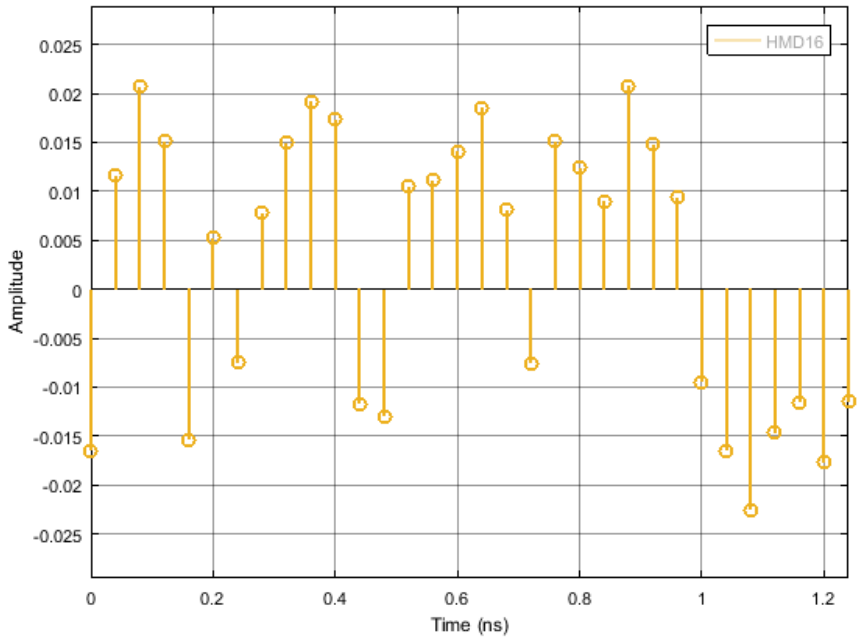


Figura 6.16: Example of the output signal of the sampler

6.20.1 Version 20180118

Adds the type t_photon_mp_xy, to support multi-path photon signals with polarization information.

Changes the signal data type to make private its data structure, only allowing its access through appropriate methods.

Sugestions for future improvement

6.21 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

6.22 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

6.23 Probability Estimator

This blocks accepts an input binary signal and it calculates the probability of having a value "1" or "0" according to the number of samples acquired and according to the z-score value set depending on the confidence interval. It produces an output binary signal equals to the input. Nevertheless, this block has an additional output which is a txt file with information related with probability values, number of samples acquired and margin error values for each probability value.

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

$$\begin{aligned} X(\omega) &= 1 \quad \text{,if } \omega \in A \\ X(\omega) &= 0 \quad \text{,if } \omega \in \bar{A} \end{aligned} \tag{6.9}$$

This way, there only are two possible results: success when the outcome is 1 or failure when the outcome is 0. The probability of success is $P(X = 1)$ and the probability of failure is $P(X = 0)$,

$$\begin{aligned} P(X = 1) = &\quad P(A) &&= p \\ P(X = 0) = &\quad P(\bar{A}) &&= 1 - p \end{aligned} \tag{6.10}$$

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) $= p(1 - p)$ **[?]**.

Assuming that *N* independent trials are performed, in which a success occurs with probability *p* and a failure occurs with probability *1-p*. If *X* is the number of successes that occur in the *N* trials, *X* is a binomial random variable with parameters *(n,p)*. Since *N* is large enough, *X* can be approximately normally distributed with mean np and variance $np(1 - p)$.

$$\frac{X - np}{\sqrt{np(1 - p)}} \backsim N(0, 1). \tag{6.11}$$

In order to obtain a confidence interval for *p*, lets assume the estimator $\hat{p} = \frac{X}{N}$ the fraction of samples equals to 1 with regard to the total number of samples acquired. Since $\hat{p}$ is the estimator of *p*, it should be approximately equal to *p*. As a result, for any $\alpha \in 0, 1$ we have that:

$$\frac{X - np}{\sqrt{n\hat{p}(1 - \hat{p})}} \backsim N(0, 1) \tag{6.12}$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{n\hat{p}(1 - \hat{p})}} < z_{\alpha/2}\} &\approx 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{n\hat{p}(1 - \hat{p})} < np - X < z_{\alpha/2}\sqrt{n\hat{p}(1 - \hat{p})}\} &\approx 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1 - \hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1 - \hat{p})/n}\} &\approx 1 - \alpha \end{aligned} \tag{6.13}$$

This way, a confidence interval for p is approximately $100(1-\alpha)$ percent.

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:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \tag{6.14}$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \tag{6.15}$$

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

$$\text{ME} = \text{z}_{\text{score}} \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \tag{6.16}$$

being $\hat{p}$ the expected probability calculated using the formulas above and N the total number of samples.

This block outputs a txt file with information regarding with the total number of received bits, the probability of 1, the probability of 0 and the respective errors.

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 2

Type: Binary

Type: txt file

Examples

Lets calculate the margin error for N of samples in order to obtain X inside a specific confidence interval, which in this case we assume a confidence interval of 99%.

We will use *z-score* from a table about standard normal distribution, which in this case is 2.576, since a confidence interval of $99\%$was chosen, to calculate the expected error margin,

$$\begin{aligned} \text{ME} &= \pm z_{\alpha/2}\frac{\sigma}{\sqrt{N}} \\ \text{ME} &= \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{N}}, \end{aligned} \tag{6.17}$$

where, ME is the error margin, $z_{\alpha/2}$ is the *z-score* for a specific confidence interval, $\sigma = \sqrt{\text{VAR(X)}} = \sqrt{\hat{p}(1-\hat{p})}$ is the standard deviation and N the number of samples.

This way, with a 99% confidence interval, between $(\hat{p} - \text{ME}) \times 100$ and $(\hat{p} + \text{ME}) \times 100$ percent of the samples meet the standards.

Sugestions for future improvement

6.24 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

6.25 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

6.26 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

6.27 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

6.28 Optical Hybrid

Header File	:	optical_hybrid.h
Source File	:	optical_hybrid.cpp

This block simulates an optical hybrid. It accepts two input signals corresponding to the signal and to the local oscillator. It generates four output complex signals separated by $90°$ in the complex plane. Figure 6.17 shows a schematic representation of this block.

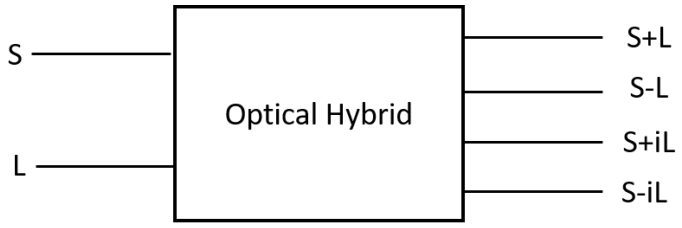


Figura 6.17: Schematic representation of an optical hybrid.

Input Parameters

Parameter	Type	Values	Default
outputOpticalPower	double	any	$1e-3$
outputOpticalWavelength	double	any	$1550e-9$
outputOpticalFrequency	double	any	SPEED_OF_LIGHT / outputOpticalWavelength
powerFactor	double	≤ 1	0.5

Tabela 6.13: Optical hybrid input parameters

Methods

OpticalHybrid()

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

void initialize(void)

bool runBlock(void)

void setOutputOpticalPower(double outOpticalPower)

void setOutputOpticalPower_dBm(double outOpticalPower_dBm)

void setOutputOpticalWavelength(double outOpticalWavelength)

void setOutputOpticalFrequency(double outOpticalFrequency)

void setPowerFactor(double pFactor)

Functional description

This block accepts two input signals corresponding to the signal to be demodulated (S) and to the local oscillator (L). It generates four output optical signals given by *powerFactor* $\times (S + L)$, *powerFactor* $\times (S - L)$,*powerFactor* $\times (S + iL)$, *powerFactor* $\times (S - iL)$. The input parameter *powerFactor* assures the conservation of optical power.

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 4

Type: Optical (OpticalSignal)

Examples

Sugestions for future improvement

6.29 Photodiode pair

Header File	:	photodiode_old.h
Source File	:	photodiode_old.cpp

This block simulates a block of two photodiodes assembled like in figure 6.18. It accepts two optical input signals and outputs one electrical signal. Each photodiode converts an optical signal to an electrical signal. The two electrical signals are then subtracted and the resulting signals corresponds to the output signal of the block.

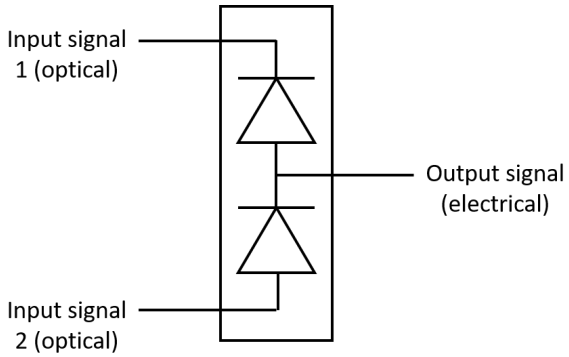


Figura 6.18: Schematic representation of the physical equivalent of the photodiode code block.

Input Parameters

Parameter: responsivity{1}

Parameter: outputOpticalWavelength{ 1550e-9 }

Parameter: outputOpticalFrequency{ SPEED_OF_LIGHT / wavelength }

Methods

Photodiode()

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

void initialize(void)

bool runBlock(void)

void setResponsivity(`t_real` Responsivity)

Functional description

This block accepts two input optical signals. It computes the optical power of the signal (given by the absolute value squared of the input signal) and multiplies it by the *responsivity* of the photodiode. This product corresponds to the current generated by the photodiode. This is done for each of the input signals. The two currents are then subtracted producing a single output current, that corresponds to the output electrical signal of the block.

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

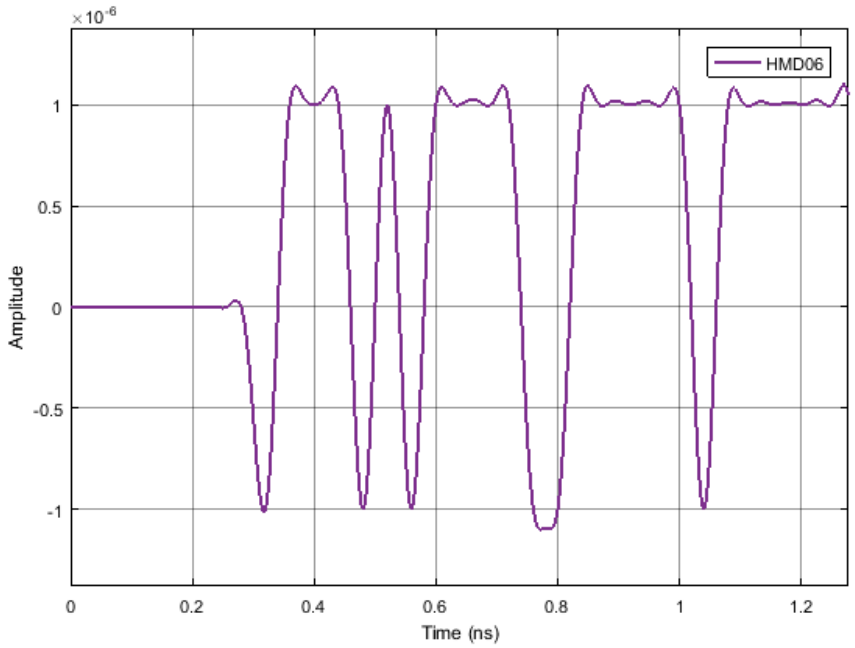


Figura 6.19: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

6.30 Pulse Shaper

Header File	:	pulse_shaper.h
Source File	:	pulse_shaper.cpp

This block applies an electrical filter to the signal. It accepts one input signal that is a sequence of Dirac delta functions and it produces one output signal continuous in time and in amplitude.

Input Parameters

Parameter	Type	Values	Default
filterType	string	RaisedCosine, Gaussian	RaisedCosine
impulseResponseTimeLength	int	any	16
rollOfFactor	real	$\in [0, 1]$	0.9

Tabela 6.14: Pulse shaper input parameters

Methods

PulseShaper(vector<Signal *> &InputSig, vector<Signal *> OutputSig) :FIR_Filter(InputSig, OutputSig){};

void initialize(void);

void setImpulseResponseTimeLength(int impResponseTimeLength)

int const getImpulseResponseTimeLength(void)

void setFilterType(PulseShaperFilter fType)

PulseShaperFilter const getFilterType(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor()

Functional Description

The type of filter applied to the signal can be selected trough the input parameter *filterType*. Currently the only available filter is a raised cosine.

The filter's transfer function is defined by the vector *impulseResponse*. The parameter *rollOfFactor* is a characteristic of the filter and is used to define its transfer function.

Input Signals

Number : 1

Type : Sequence of Dirac Delta functions (ContinuousTimeDiscreteAmplitude)

Output Signals

Number : 1

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

Example

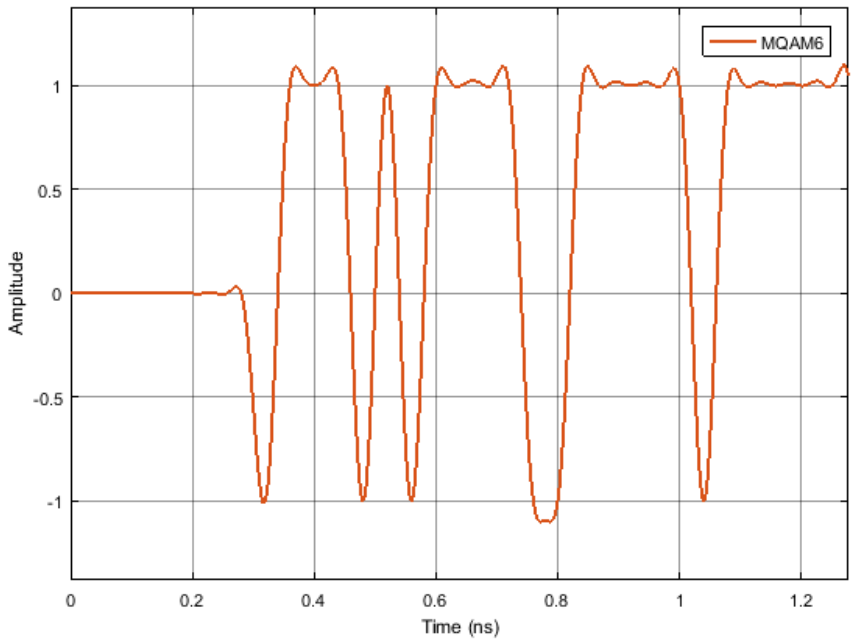


Figura 6.20: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

6.31 Sampler

Header File	:	sampler.h
Source File	:	sampler_20171119.cpp

This block can work in two configurations: with an external clock or without it. In the latter it accepts two input signals one being the clock and the other the signal to be demodulated. In the other configuration there's only one input signal which is the signal.

The output signal is obtained by sampling the input signal with a predetermined sampling rate provided either internally or by the clock.

Input Parameters

Parameter	Type	Values	Default
samplesToSkip	int	any (smaller than the number of samples generated)	0

Tabela 6.15: Sampler input parameters

Methods

Sampler()

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

void initialize(void)

bool runBlock(void)

void setSamplesToSkip(`t_integer` sToSkip)

Functional description

This block can work with an external clock or without it.

In the case of having an external clock it accepts two input signals. The signal to be demodulate which is complex and a clock signal that is a sequence of Dirac delta functions with a predetermined period that corresponds to the sampling period. The signal and the clock signal are scanned and when the clock has the value of 1.0 the correspondent complex value of the signal is placed in the buffer corresponding to the output signal.

There's a detail worth noting. The electrical filter has an impulse response time length of 16 (in units of symbol period). This means that when modulating a bit the spike in the signal corresponding to that bit will appear 8 units of symbol period later. For this reason there's

the need to skip the earlier samples of the signal when demodulating it. That's the purpose of the *samplesToSkip* parameter.

Between the binary source and the current block the signal is filtered twice which means that this effect has to be taken into account twice. Therefore the parameter *samplesToSkip* is given by $2 * 8 * samplesPerSymbol$.

Input Signals

Number: 1

Type: Electrical real (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical real (TimeDiscreteAmplitudeContinuousReal)

Examples

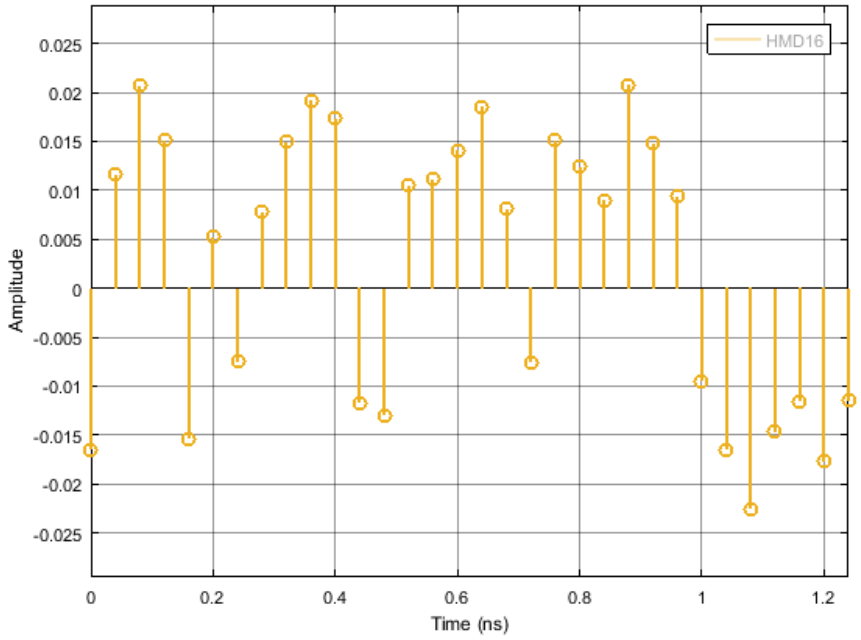


Figura 6.21: Example of the output signal of the sampler

Sugestions for future improvement

6.32 Sink

Header File	:	sink.h
Source File	:	sink.cpp

This block accepts one input signal and it does not produce output signals. It takes samples out of the buffer until the buffer is empty. It has the option of displaying the number of samples still available.

Input Parameters

Parameter	Type	Values	Default
numberOfSamples	long int	any	-1

Tabela 6.16: Sampler input parameters

Methods

Sink(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)

bool runBlock(void)

void setNumberOfSamples(long int nOfSamples)

void setDisplayNumberOfSamples(bool opt)

Functional Description

6.33 White Noise

Header File	:	white_noise_20180118.h
Source File	:	white_noise_20180118.cpp

This block generates a gaussian pseudo-random noise signal with a given spectral density. It can be initialized with three different seeding methods to allow control over correlation and reproducibility:

1. DefaultDeterministic
2. RandomDevice
3. Selected

This block does not accept any input signal. It produces can produce a real or complex output, depending on the used output signal.

Input Parameters

Parameter	Type	Values	Default
seedType	enum	DefaultDeterministic, RandomDevice, Selected	RandomDevice
spectralDensity	real	> 0	10^{-4}
seed	int	$\in [1, 2^{32} - 1]$	1

Tabela 6.17: White noise input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNoiseSpectralDensity(double SpectralDensity) spectralDensity = SpectralDensity;

double const getNoiseSpectralDensity(void) return spectralDensity;

void setSeedType(SeedType sType) seedType = sType; ;

SeedType const getSeedType(void) return seedType; ;

void setSeed(int newSeed) seed = newSeed;

int getSeed(void) return seed;

Functional description

The *seedType* parameter allows the user to select between one of the three seeding methods to initialize the pseudo-random number generators (PRNGs) responsible for generating the noise signal.

DefaultDeterministic: Uses default seeds to initialize the PRNGs. These are different for all generators used within the same block, but remain the same for sequential runs or different *white_noise* blocks. Therefore, if more than one *white_noise* block is used, another seeding method should be chosen to avoid producing the exact same noise signal in all sources.

RandomDevice: Uses randomly chosen seeds using *std::random_device* to initialize the PRNGs.

SingleSelected: Uses one user selected seed to initialize the PRNGs. The selected seed is passed through the variable *seed*. If more than one generator is used, additional seeds are created by choosing the next sequential integers. For instance, if the user selected seed is 10, and all the four PRNGs are used, the used seeds will be $[10, 11, 12, 13]$.

The noise is then obtained from a gaussian distribution with variance equal to the spectral density. If the signal is complex, the noise is calculated independently for the real and imaginary parts, with half the spectral density in each.

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: RealValue, ComplexValue or ComplexValueXY

Examples

Random Mode

Suggestions for future improvement

6.34 Ideal Amplifier

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

Input Parameters

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

Tabela 6.18: Ideal Amplifier input parameters

Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;

Functional description

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

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

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_20171121

Structure of a function

[dataDecimate, data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate, samplingRateDecimate] = sgnToWfm_20171121 (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:

dataDecimate: A vector which contains decimated signal data by an appropriate decimation factor to make it compatible with the AWG.

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.

samplingRateDecimate: Reduced sampling rate which is compatible with AWG. (i.e. less than 16 GSa/s).

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:

[dataDecimate, data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate, samplingRateDecimate] = 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:

[dataDecimate, data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate, samplingRateDecimate] = 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:

[dataDecimate, data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate, samplingRateDecimate] = 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).

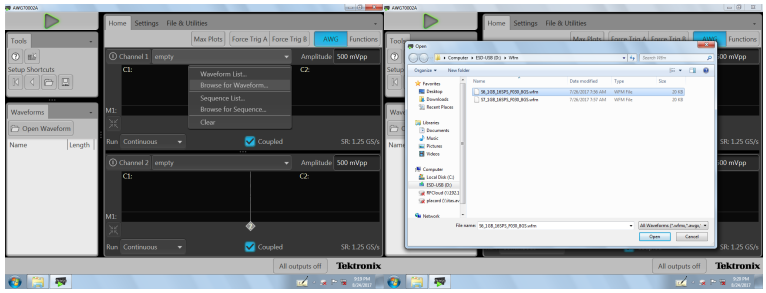


Figura 7.1: Selecting your waveform in the AWG

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

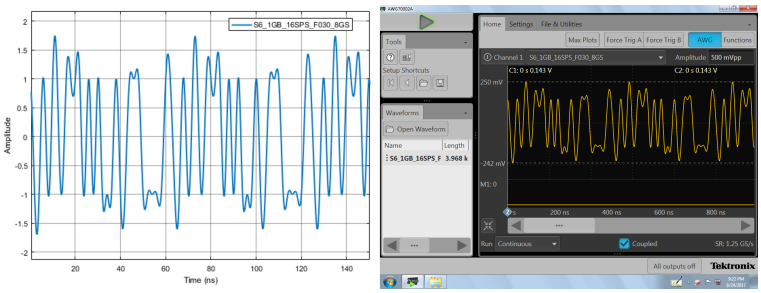


Figura 7.2: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

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

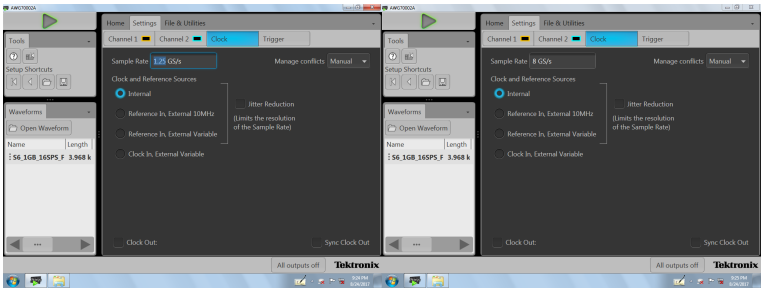


Figura 7.3: Configuring the right sampling rate

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

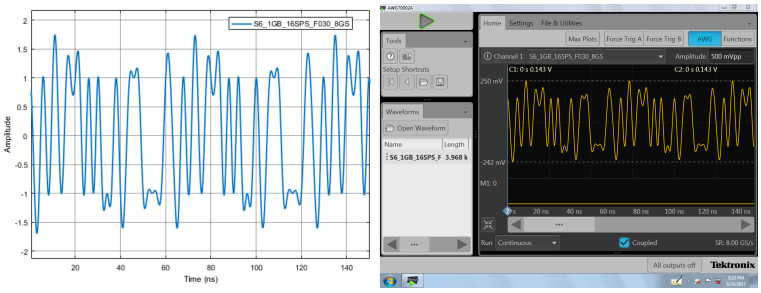


Figura 7.4: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

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

Capítulo 8

Algorithms

8.1 Fast Fourier Transform

Header File	:	fft_*.h
Source File	:	fft_*.cpp
Version	:	20180201 (Romil Patel)

Algorithm

The algorithm for the FFT will follow the following formula,

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \ e^{-i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{8.1}$$

Similarly, for IFFT,

$$x_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X_k \ e^{i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{8.2}$$

From equations 8.1 and 8.2, we can write only one script for the implementations of the direct and inverse Discrete Fourier Transfer and manipulate its functionality as a FFT or IFFT by applying an appropriate input arguments. The generalized form for the algorithm can be given as,

$$y = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x \ e^{mi2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{8.3}$$

where, x is an input complex signal, y is the output complex signal and m equals -1 or 1 for FFT and IFFT respectively.

Function description

To perform FFT operation, the fft_*.h header file must be included and the input argument to the function can be given as follows,

$$y = fft(x, -1)$$

or

$$y = fft(x)$$

where x and y are of the C++ type vector<complex>. In a similar way, IFFT can be manipulated as,

$$x = fft(y, 1)$$

or

$$x = ifft(y)$$

Flowchart

The figure 8.1 displays top level architecture of the FFT algorithm. If the length of the input signal is 2^N, it'll execute Radix-2 algorithm otherwise it'll execute Bluestein algorithm [**?**]. The computational complexity of Radix-2 and Bluestein algorithm is $O(Nlog_2N)$, however, the computation of Bluestein algorithm involves the circular convolution which increases the number of computations. Therefore, to reduce the computational time it is advisable to work with the vectors of length 2^N [**?**].

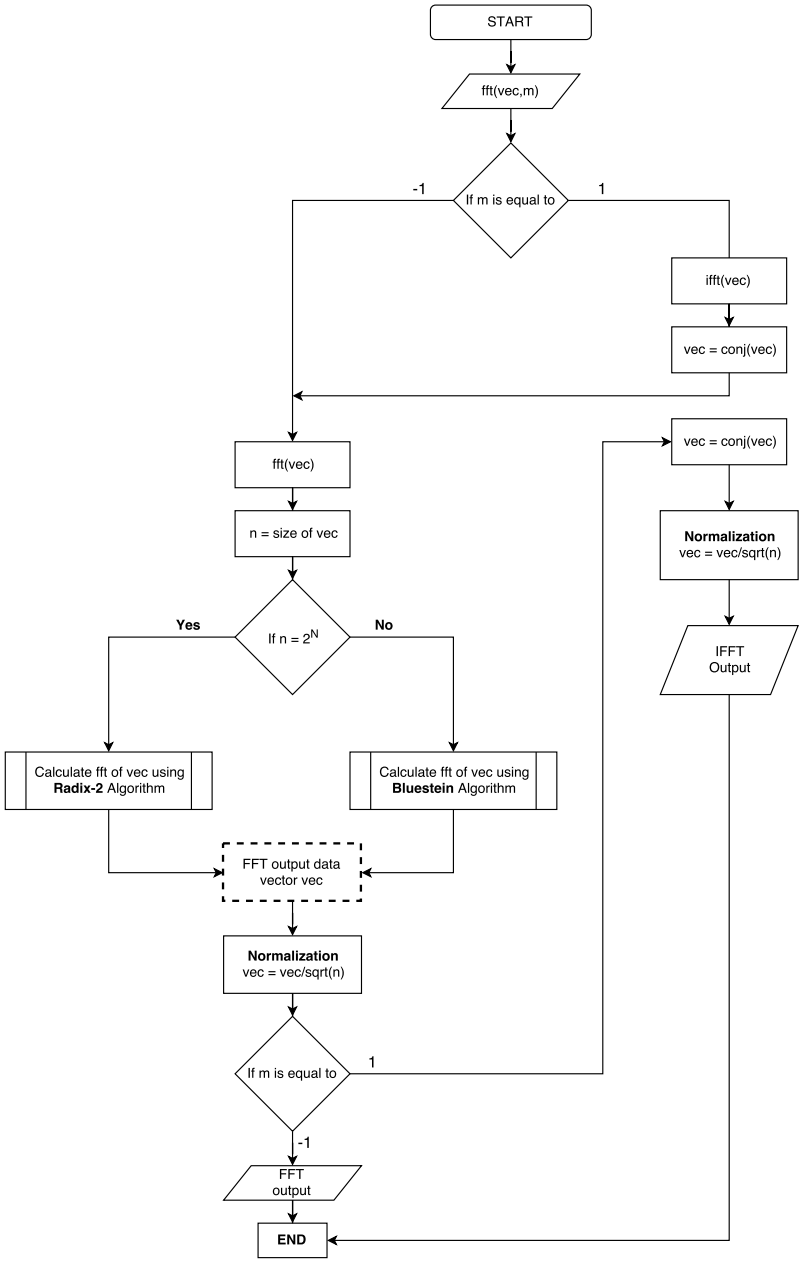


Figura 8.1: Top level architecture of FFT algorithm

Radix-2 algorithm

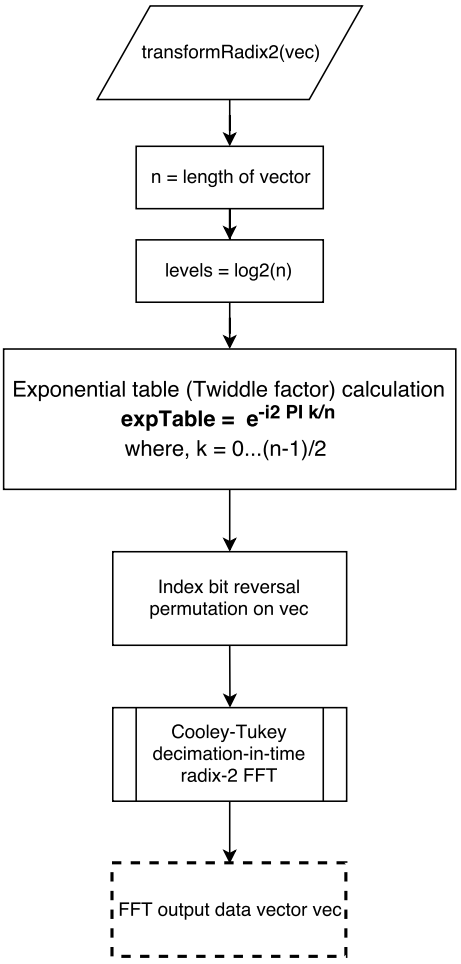


Figura 8.2: Radix-2 algorithm

Cooley-Tukey algorithm

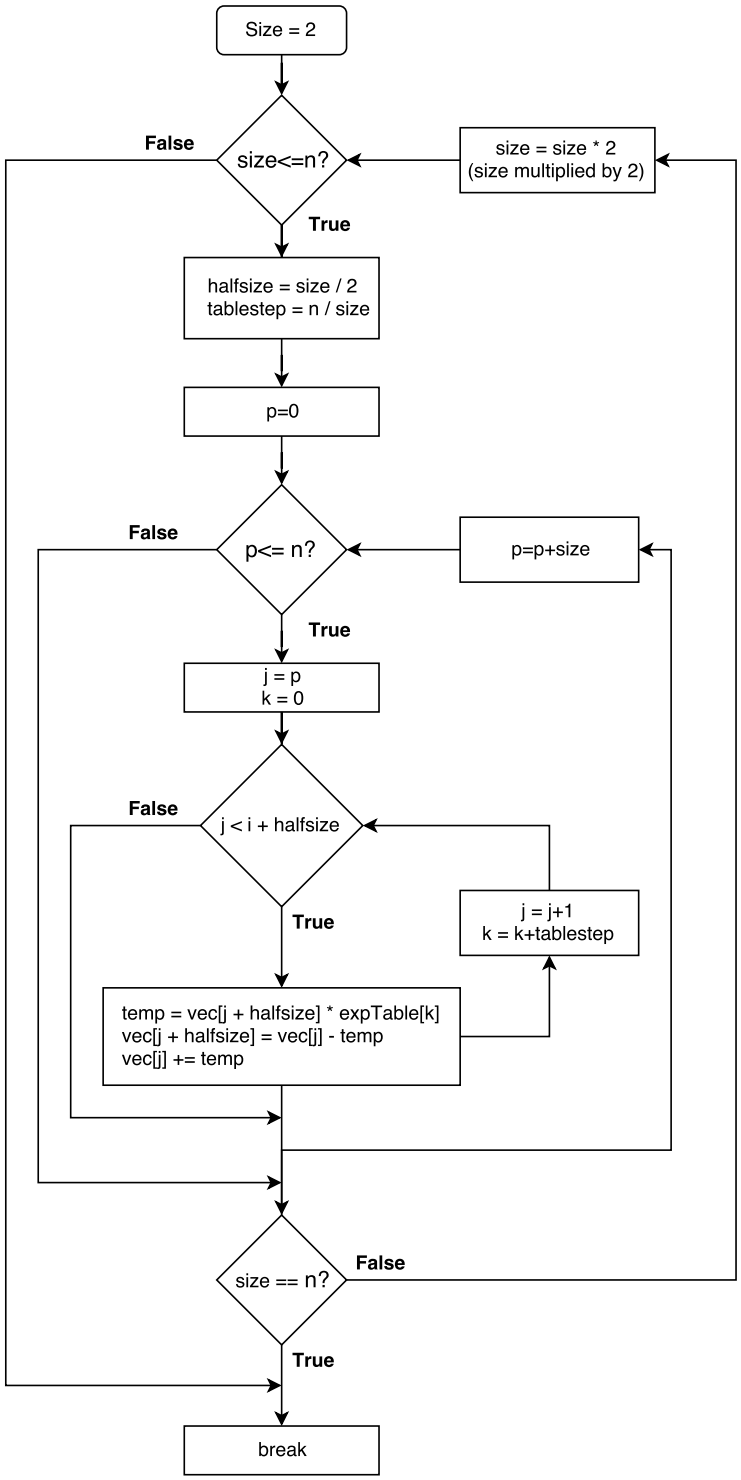


Figura 8.3: Cooley-Tukey algorithm

Bluestein algorithm

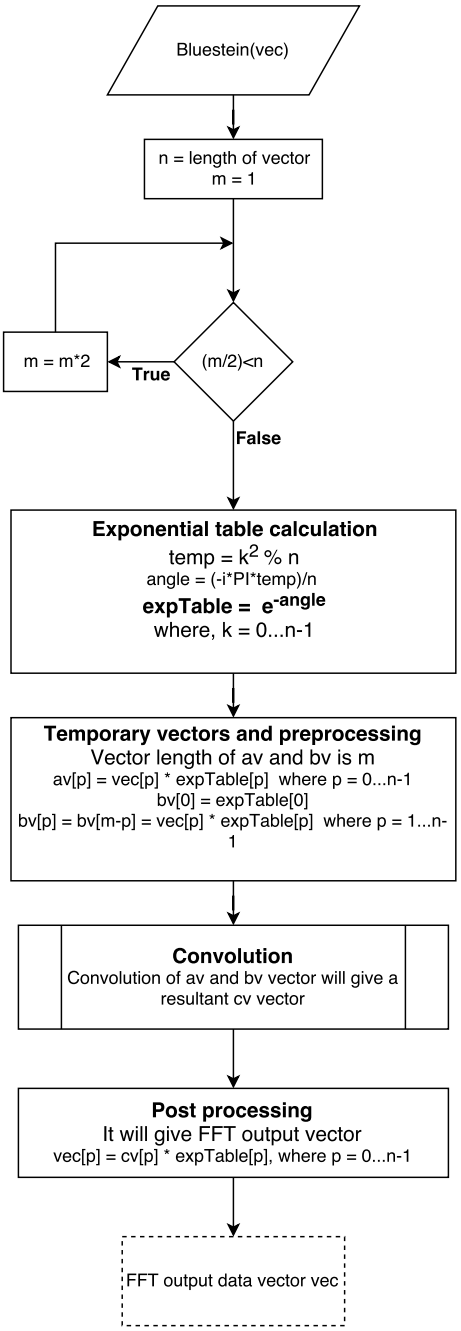


Figura 8.4: Bluestein algorithm

Convolution algorithm

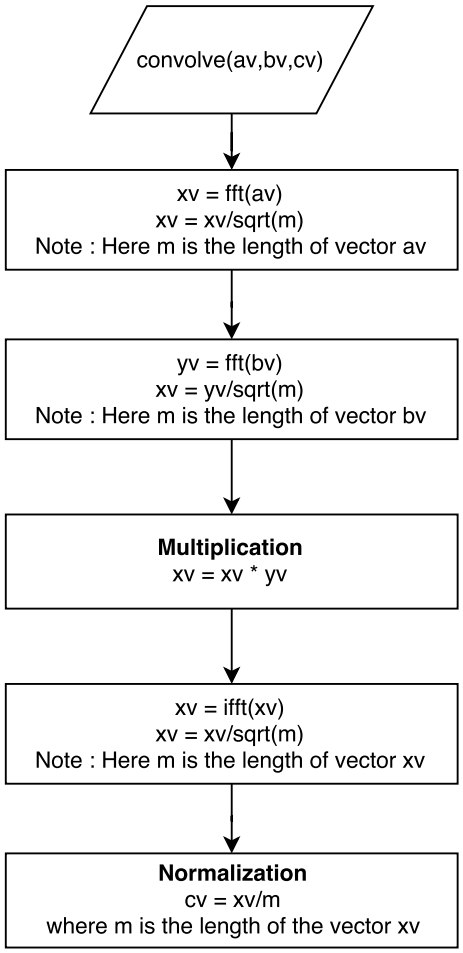


Figura 8.5: Circular convolution algorithm

Test example

This sections explains the steps to compare our C++ FFT program with the MATLAB FFT program.

Step 1 : Open the **fft_test** folder by following the path "/algorithms/fft/fft_test".

Step 2 : Find the **fft_test.m** file and open it.
This fft_test.m consists of two sections; section 1 generates the time domain signal and save it in the form of the text file with the name *time_function.txt* in the same folder. Section 2 reads the fft complex data generated by C++ program.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
clear all
close all

Fs = 1e5;                  % Sampling frequency
T = 1/Fs;                  % Sampling period
L = 2^3;                   % Length of signal
t = (0:L-1)*(5*T);         % Time vector
f = linspace(-Fs/2,Fs/2,L);

%Choose for sig a value between [1, 7]
sig = 7;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
        X = S + 2*randn(size(t));
    case 2
        signal_title = 'Sinusoids with Random Noise';
        S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
        X = S + 2*randn(size(t));
    case 3
        signal_title = 'Single sinusoids';
        X = sin(2*pi*t);
    case 4
        signal_title = 'Summation of two sinusoids';
        X = sin(2*pi*t) + cos(2*pi*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';
        X = sin(2*pi*10*t).*exp(-abs(t));
    case 6
        signal_title = 'Mixed signal 1';
        X = sin(2*pi*10*t).*exp(-t)+sin(2*pi*t)+7*sin(2*pi*+5*t)+7*cos(2*pi*+20*t
    )+5*sin(2*pi*+50*t);
    case 7
```

```
        signal_title = 'Mixed signal 2';
        X = 2*sin(2*pi*100*t).*exp(-t)+2.5*sin(2*pi*+250*t)+sin(2*pi*+50*t).*cos(2*pi*+20*t)+1.5*sin(2*pi*+50*t).*sin(2*pi*+150*t);
end

plot(t(1:end),X(1:end))
title(signal_title)
axis([min(t) max(t) 1.1*min(X) 1.1*max(X)]);
xlabel('t (Seconds)')
ylabel('X(t)')

% dlmwrite will generate text file which represents the time domain signal.
% dlmwrite('time_function.txt', X, 'delimiter','\t');
fid=fopen('time_function.txt','w');
b=fprintf(fid,'%0.15f\n',X); % 15-Digit accuracy
fclose(fid);

tic
fy =fft(X);
toc
fy = fftshift(fy);
figure(2);
subplot(2,1,1)
plot(f,abs(fy));
axis([-Fs/(2*5) Fs/(2*5) 0 1.1*max(abs(fy))]);
xlabel('f');
ylabel('|Y(f)|');
title('MATLAB program Calculation : Magnitude');

subplot(2,1,2)
plot(f,phase(fy));
xlim([-Fs/(2*5) Fs/(2*5)]);
xlabel('f');
ylabel('phase(Y(f))');
title('MATLAB program Calculation : Phase');

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read C++ transformed data file
fullData = load('frequency_function.txt');
A=1;
B=A+1;
l=1;
Z=zeros(length(fullData)/2,1);
while (l<=length(Z))
Z(l) = fullData(A)+fullData(B)*1i;
A = A+2;
B = B+2;
l=l+1;
end
```

```
89
% Comparsion of the MATLAB and C++ fft calculation.
91 figure;
subplot(2,1,1)
93 plot(f,abs(fftshift(fft(X))))
hold on
95 %Multiplied by sqrt(n) to verify our C++ code with MATLAB implemenrtation.
plot(f,(sqrt(length(Z))*abs(fftshift(Z))), '--o')
97 axis([-Fs/(2*5) Fs/(2*5) 0 1.1*max(abs(fy))]);
xlabel('f');
99 title('Main reference for Magnitude')
legend('MATLAB','C++')
101
subplot(2,1,2)
103 plot(f,phase(fftshift(fft(X))))
hold on
105 plot(f,phase(fftshift(Z)),'--o')
xlim([-Fs/(2*5) Fs/(2*5)])
107 title('Main reference for Phase')
xlabel('f');
109 legend('MATLAB','C++')

111 % IFFT test comparision Plot
% figure; plot(X); hold on; plot(real(Z),'--o');
```

Listing 8.1: fft_test.m code

Step 3 : Choose for sig a value between [1, 7] and run the first section namely **section 1** by pressing "ctrl+Enter".
This will generate a *time_function.txt* file in the same folder which contains the time domain signal data.

Step 4 : Now, find the **fft_test.vcxproj** file in the same folder and open it.
In this project file, find *fft_test.cpp* and click on it. This file is an example of FFT calculation using C++ program. Basically this *fft_test.cpp* file consists of four sections:
Section 1. Read the input text file (import "time_function.txt"data file)
Section 2. It calculates FFT.
Section 3. Save FFT calculated data (export *frequency_function.txt* data file).
Section 4. Displays in the screen the FFT calculated data and length of the data.

```
# include "fft_20180208.h"
2
# include <complex>
4 # include <fstream>
# include <iostream>
6 # include <math.h>
# include <stdio.h>
8 # include <string>
# include <strstream>
10 # include <algorithm>
```

```
# include <vector>
#include <iomanip>

using namespace std;

int main()
{
  ////////////////////////////////// Section 1 ////////////////////////////////////
  /////////////////////////////////////////////////////////////////////////////////
  /////////// Read the input text file (import "time_function.txt" ) ///////////
  /////////////////////////////////////////////////////////////////////////////////
  ifstream inFile;
  inFile.precision(15);
  double ch;
  vector <double> inTimeDomain;
  inFile.open("time_function.txt");

  // First data (at 0th position) appled to the ch it is similar to the "cin".
  inFile >> ch;

  // It'll count the length of the vector to verify with the MATLAB
  int count=0;

  while (!inFile.eof())
  {
    // push data one by one into the vector
    inTimeDomain.push_back(ch);

    // it'll increase the position of the data vector by 1 and read full vector.
    inFile >> ch;

    count++;
  }

  inFile.close(); // It is mandatory to close the file at the end.


  ////////////////////////////////// Section 2 ////////////////////////////////////
  /////////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////// Calculate FFT ////////////////////////////////
  /////////////////////////////////////////////////////////////////////////////////

  vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
  vector <complex<double>> fourierTransformed;
  vector <double> re(inTimeDomain.size());
  vector <double> im(inTimeDomain.size());

  for (unsigned int i = 0; i < inTimeDomain.size(); i++)
  {
    // Real data of the signal
    re[i] = inTimeDomain[i];

```

```
    // Imaginary data of the signal
64    im[i] = 0;
  }
66
  // Next, Real and Imaginary vector to complex vector conversion
68  inTimeDomainComplex = reImVect2ComplexVector(re, im);

70  // calculate FFT
  fourierTransformed = fft(inTimeDomainComplex);
72  fourierTransformed = ifft(inTimeDomainComplex);

74  ////////////////////////////////// Section 3 ////////////////////////////////////
  ////////////////////////////////////////////////////////////////////////////////
76  //////// Save FFT calculated data (export "frequency_function.txt" ) /////////
  ////////////////////////////////////////////////////////////////////////////////
78  ofstream outFile;
  complex<double> outFileData;
80  outFile.open("frequency_function.txt");
  outFile.precision(15);
82  for (unsigned int i = 0; i <fourierTransformed.size(); i++)
  {
84    outFile <<  fourierTransformed[i].real() << endl;
    outFile << fourierTransformed[i].imag() << endl;
86  }


88
  ////////////////////////////////// Section 4 ////////////////////////////////////
90  ////////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////// Display Section /////////////////////////////
92  ////////////////////////////////////////////////////////////////////////////////
  for (unsigned int i = 0; i < fourierTransformed.size(); i++)
94  {
    // Display all FFT calculated data
96    cout << fourierTransformed[i] << endl;
  }
98
  // Display length of data
100 cout <<"\nTotal length of of data :"<< count << endl;

102 getchar();
  return 0;
104 }
```

Listing 8.2: fft_test.cpp code

Step 5 : Run the *fft_test.cpp* file.
This will generate a *frequency_function.txt* file in the same folder which contains the Fourier transformed data.

Step 6 : Now, go to the **fft_test.m** and run section 2 in the code by pressing "ctrl+Enter".
The section 2 reads *frequency_function.txt* and compares both C++ and MATLAB calculation

of Fourier transformed data.

Resultant analysis of various test signals

The following section will display the comparative analysis of MATLAB and C++ FFT program to calculate several type of signals.

1. Signal with two sinusoids and random noise

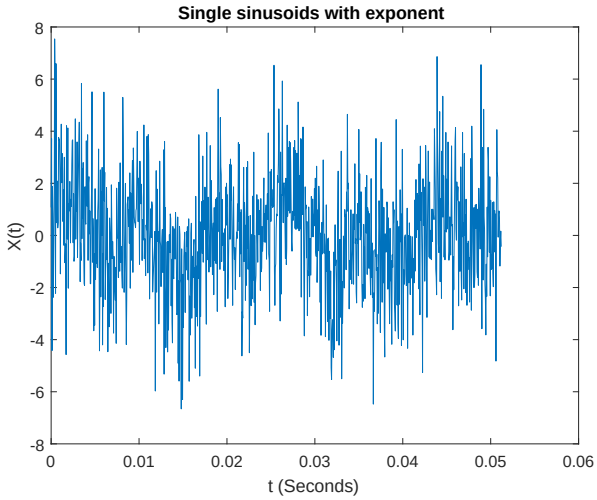


Figura 8.6: Random noise and two sinusoids

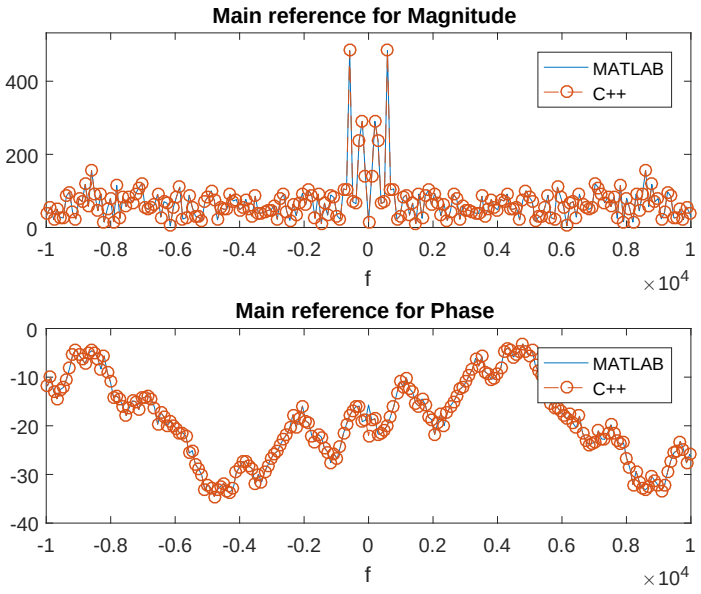


Figura 8.7: MATLAB and C++ comparison

2. Sinusoid with an exponent

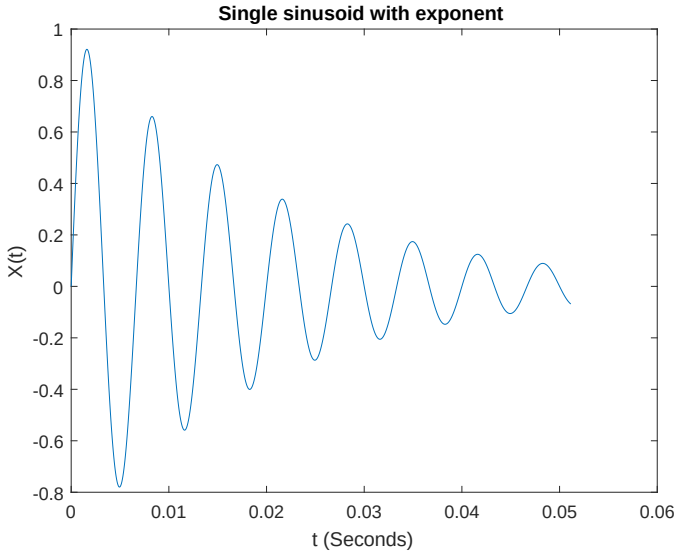


Figura 8.8: Sinusoids with exponent

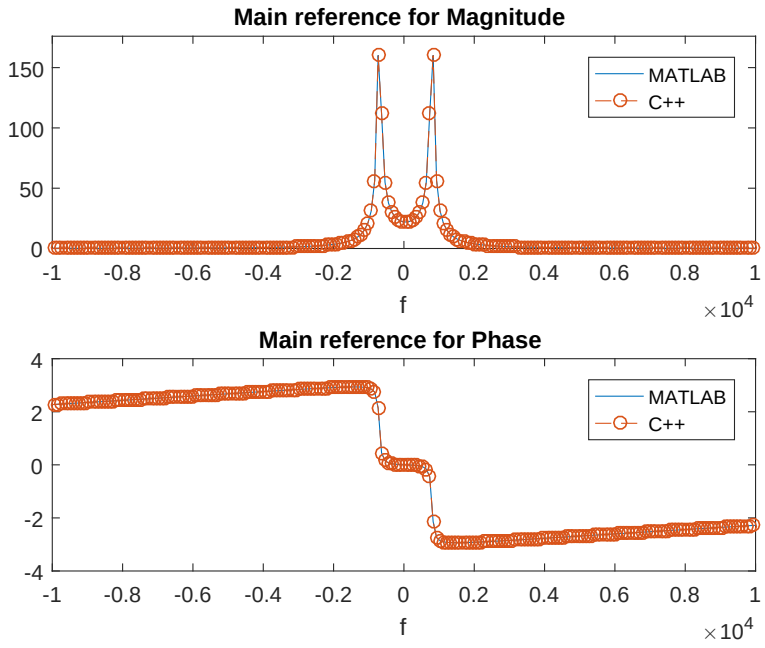


Figura 8.9: MATLAB and C++ comparison

3. Mixed signal

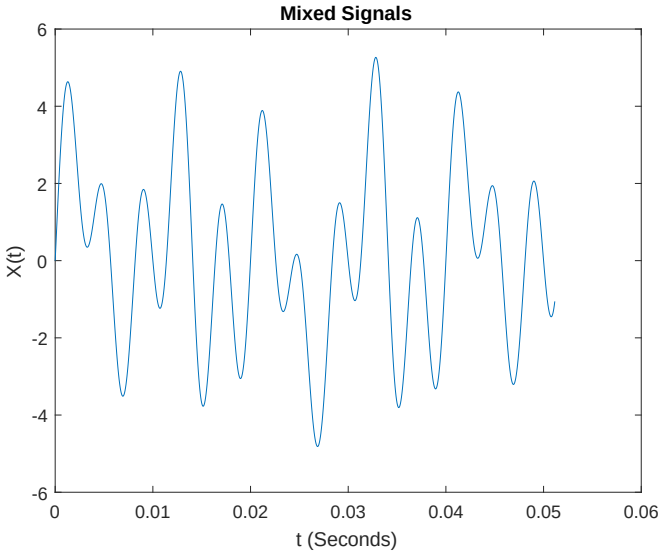


Figura 8.10: mixed signal

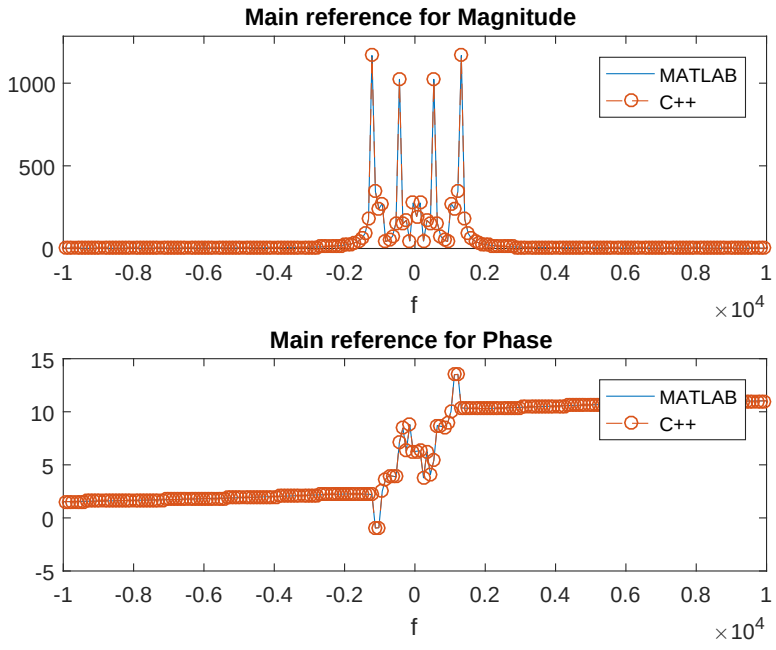


Figura 8.11: MATLAB and C++ comparison

8.1.1 Remarks

To write the data from the MATLAB in the form of text file, **fprintf** MATLAB function was used with the accuracy of the 15 digits. Similarly; to write the fft calculated data from the C++ in the form of text file, C++ **double** data type with precision of 15 digits applied to the object of **ofstream** class.

8.2 Overlap-Save Method

Header File	:	overlapSave_*.h
Source File	:	overlapSave_*.cpp
Version	:	20180201 (Romil Patel)

Overlap-save is an efficient way to evaluate the discrete convolution between a very long signal and a finite impulse response (FIR) filter. The overlap-save procedure cuts the signal into equal length segments with some overlap and then it performs convolution of each segment with the FIR filter. The overlap-save method can be computed in the following steps:
Step 1 : Determine the length M of impulse response, $h(n)$.

Step 2 : Define the size of FFT and IFFT operation, N. The value of N must greater than M and it should in the form $N = 2^k$ for the efficient implementation.

Step 3 : Determine the length L to section the input sequence $x(n)$, considering that $N = L + M - 1$.

Step 4 : Pad $L - 1$ zeros at the end of the impulse response $h(n)$ to obtain the length N.

Step 5 : Make the segments of the input sequences of length L, $x_i(n)$, where index i correspond to the i^{th} block. Overlap $M - 1$ samples of the previous block at the beginning of the segmented block to obtain a block of length N. In the first block, it is added $M - 1$ null samples.

Step 6 : Compute the circular convolution of segmented input sequence $x_i(n)$ and $h(n)$ described as,

$$y_i(n) = x_i(n) \circledast h(n). \tag{8.4}$$

This is obtained in the following steps:
1. Compute the FFT of x_i and h both with length N.
2. Compute the multiplication of $X_i(f)$ and the transfer function $H(f)$.
3. Compute the IFFT of the multiplication result to obtain the time-domain block signal, y_i.

Step 7 : Discarded $M - 1$ initial samples from the y_i, and save only the error-free $N - M - 1$ samples in the output record.

In the Figure 8.12 it is illustrated an example of overlap-save method.

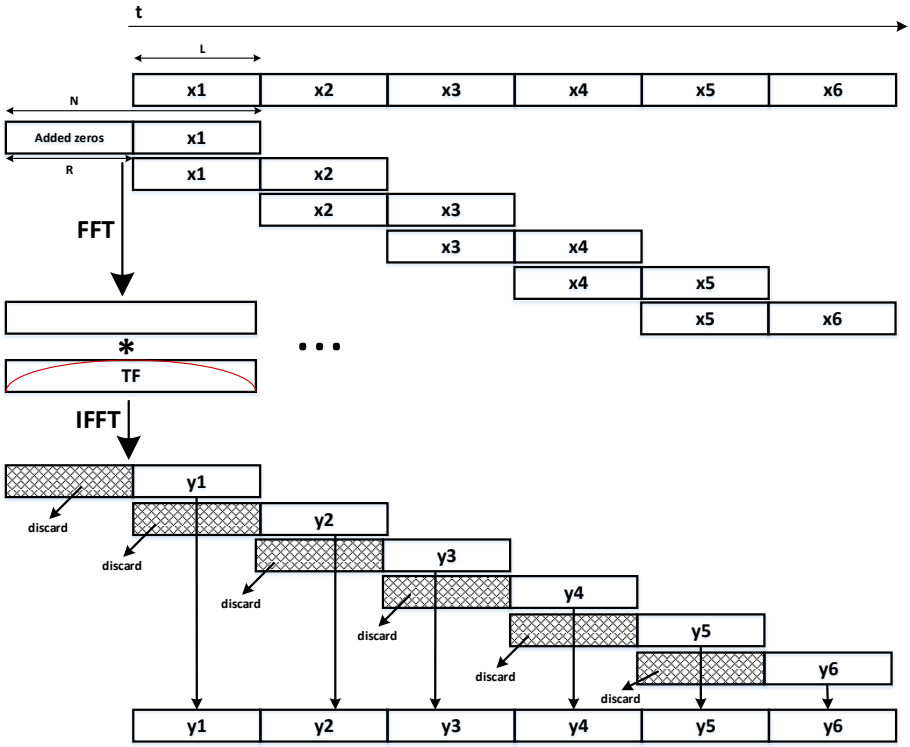


Figura 8.12: Illustration of Overlap-save method.

Function description

To perform convolution between the time domain signal $x(n)$ with the filter $h(n)$, include the header file overlapSave_*.h and then supply input argument to the function as follows,

$$y(n) = overlapSave(x(n), h(n))$$

Where, $x(n)$, $h(n)$ and $y(n)$ are of the C++ type vector< complex<double> > and the length of the signal $x(n)$ and filter $h(n)$ could be arbitrary.

Linear and circular convolution

In the circular convolution, if we determine the length of the singal $x(n)$ is $N_1 = 8$ and length of the filter is $h(n)$ is $N_2 = 5$; then the length of the output signal is determined by $N = max(N_1, N_2) = 8$. Next, the circular convolution can be performed after padding 0 in the filter $h(n)$ to make it's length equals N.

In the linear convolution, if we determine the length of the singal $x(n)$ is $N_1 = 8$ and length of the filter is $h(n)$ is $N_2 = 5$; then the length of the output signal is determined by $N = N_1 + N_2 - 1 = 12$. Next, the linear convolution using circular convolution can be performed after padding 0 in the signal $x(n)$ filter $h(n)$ to make it's length equals N.

Flowchart of overlap-save method

The following three flowcharts describe the logical flow of the overlap-save method.

1. Decide length of FFT, data block and filter

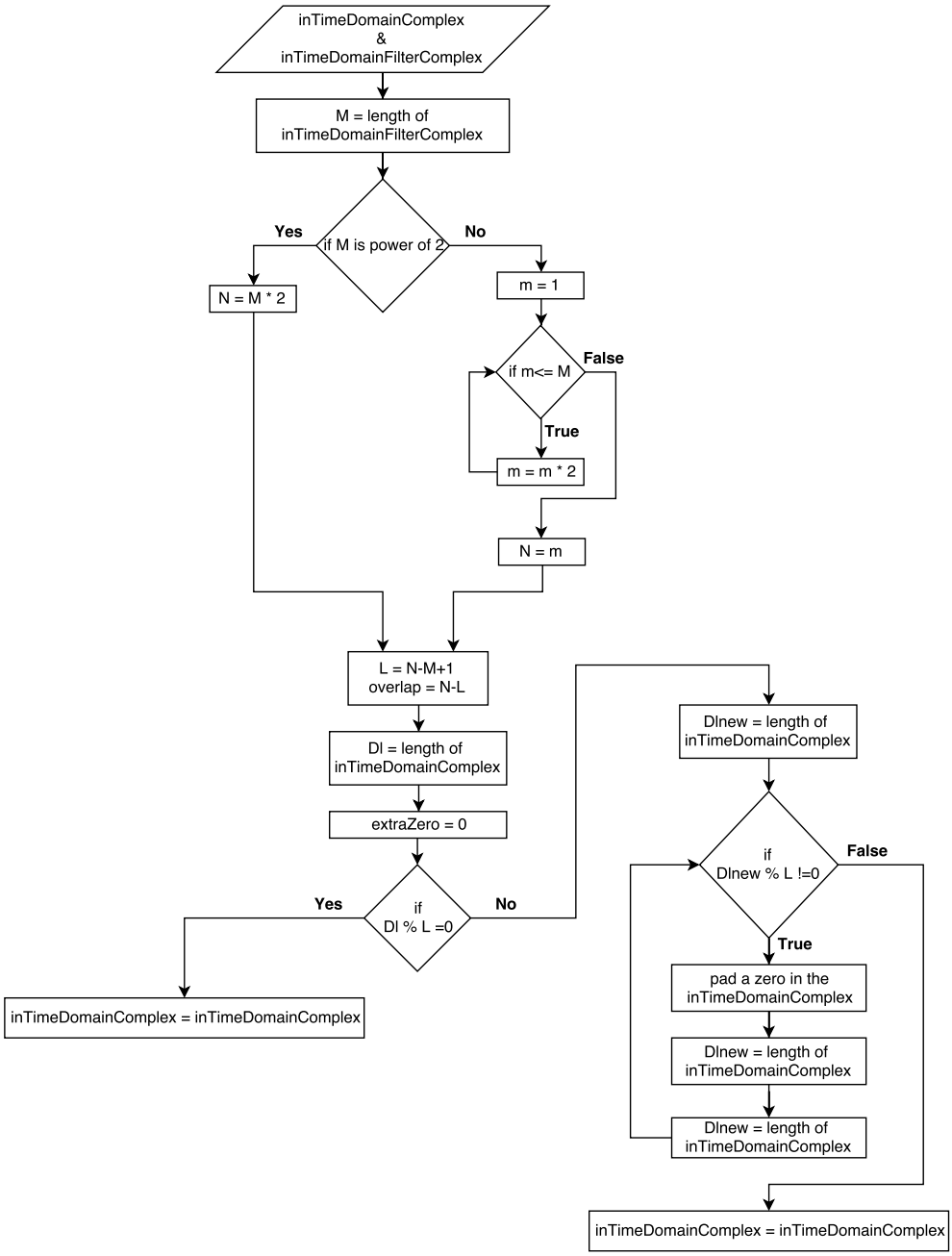


Figura 8.13: Flowchart for calculating length of FFT, data block and filter

2. Create matrix with overlap

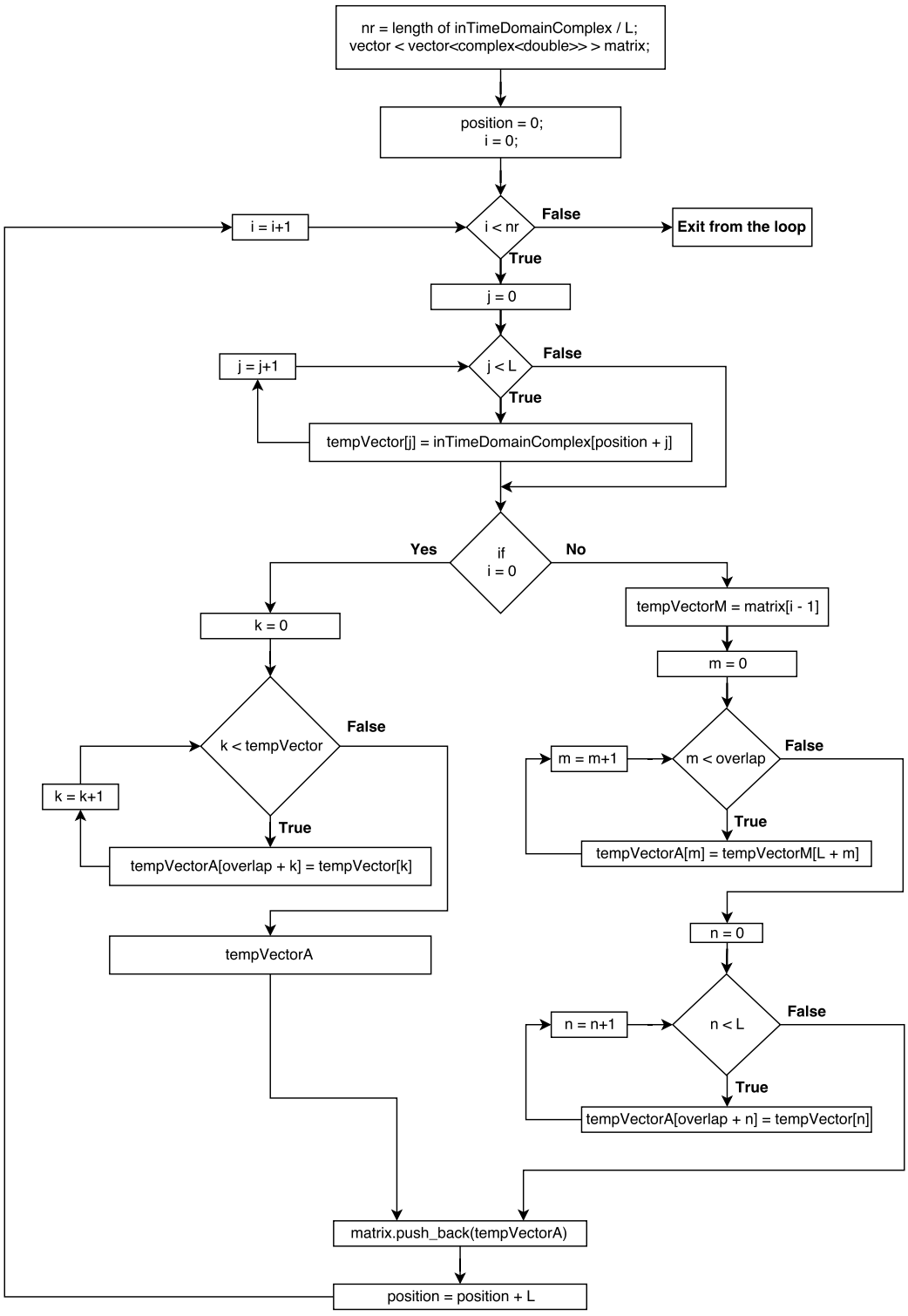


Figura 8.14: Flowchart of creating matrix with overlap

3. Convolution between filter and data blocks

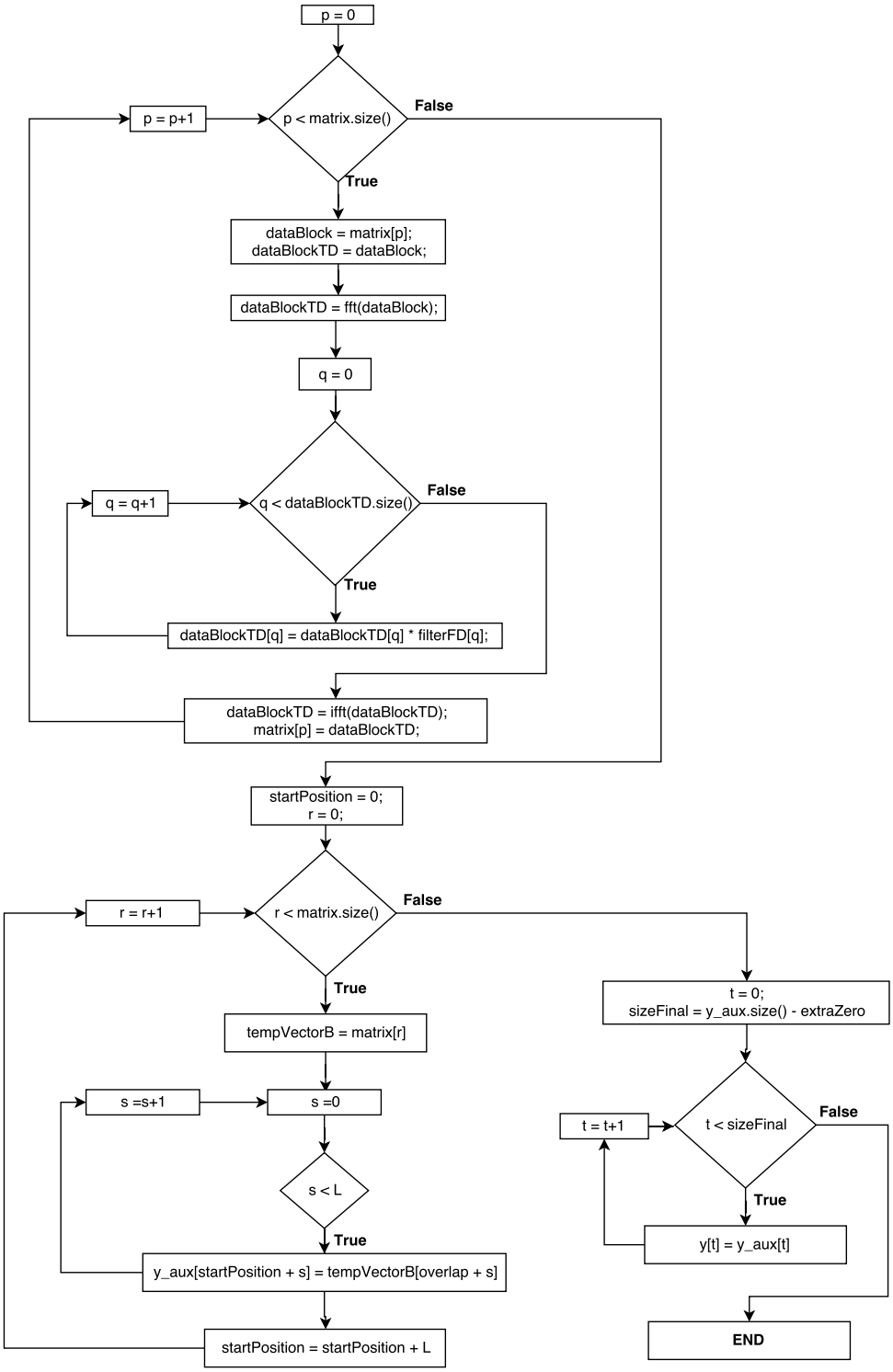


Figura 8.15: Flowchart of the convolution

Test example

This sections explains the steps of comparing our C++ overlap-save program with the MATLAB overlap-save program.

Step 1 : Open the folder namely **overlapSave_test** by following the path "/algorithms/overlapSave/overlapSave_test".

Step 2 : Find the **overlapSave_test.m** file and open it.
This overlapSave_test.m consists of five sections:
section 1 : It generates the time domain signal and filter impulse response and save them in the form of the text file with the name of *time_domain_data.txt* and *time_domain_filter.txt* respectively in the same folder.
Section 2 : It calculates the length of FFT, data blocks and filter to perform convolution using overlap-save method.
Section 3 : It consists of overlap-save code which first converts the data into the form of matrix with 50% overlap and then performs circular convolution with filter.
Section 4 : It read *overlap_save_data.txt* data file generated by C++ program and compare with MATLAB implementation.
Section 5 : It compares our MATLAB and C++ implementation with the built-in MATLAB function conv().

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generate signal and filter data and save it as a .txt file.
clc
clear all
close all

Fs = 1e5;                   % Sampling frequency
T = 1/Fs;                   % Sampling period
L = 2^12+45;                    % Length of signal
t = (0:L-1)*(5*T);          % Time vector
f = linspace(-Fs/2,Fs/2,L);

%Choose for sig a value between [1, 7]
sig = 2;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
        X = S + 2*randn(size(t));
    case 2
        signal_title = 'Sinusoids with Random Noise';
        S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
        X = S + 2*randn(size(t));
```

```
    case 3
        signal_title = 'Single sinusoids';
        X = sin(2*pi*t);
    case 4
        signal_title = 'Summation of two sinusoids';
        X = sin(2*pi*1205*t) + cos(2*pi*1750*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';
        X = sin(2*pi*250*t).*exp(-12*abs(t));
    case 6
        signal_title = 'Mixed signal 1';
        X = sin(2*pi*10*t).*exp(-t)+sin(2*pi*t)+7*sin(2*pi*+5*t)+7*cos(2*pi*+20*t
    )+5*sin(2*pi*+50*t);
    case 7
        signal_title = 'Mixed signal 2';
        X = 2*sin(2*pi*100*t).*exp(-t)+2.5*sin(2*pi*+250*t)+sin(2*pi*+50*t).*cos
    (2*pi*+20*t)+1.5*sin(2*pi*+50*t).*sin(2*pi*+150*t);
end

Xref = X;
% dlmwrite will generate text file which represents the time domain signal.
%dlmwrite('time_domain_data.txt', X, 'delimiter','\t');
fid=fopen('time_domain_data.txt','w');
fprintf(fid,'%.20f\n',X); % 12-Digit accuracy
fclose(fid);


% Choose for filt a value between [1, 3]
filt = 1;
switch filt
    case 1
        filter_type = 'Impulse response of rcos filter';
        h = rcosdesign(0.25,11,6);
    case 2
        filter_type = 'Impulse response of rrcos filter';
        h = rcosdesign(0.25,11,6,'sqrt');
    case 3
        filter_type = 'Impulse response of Gaussian filter';
        h = gaussdesign(0.25,11,6);
end

%dlmwrite('time_domain_filter.txt', h, 'delimiter','\t');
fid=fopen('time_domain_filter.txt','w');
fprintf(fid,'%.20f\n',h); % 20-Digit accuracy
fclose(fid);


figure;
subplot(211)
plot(t,X)
grid on
title(signal_title)
```

```
axis([min(t) max(t) 1.1*min(X) 1.1*max(X)]);
xlabel('t (Seconds)')
ylabel('X(t)')

subplot(212)
plot(h)
grid on
title(filter_type)
axis([1 length(h) 1.1*min(h) 1.1*max(h)]);
xlabel('Samples')
ylabel('h(t)')

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the length of FFT, data blocks and filter
M = length(h);

if (bitand(M,M-1)==0)
    N = 2 * M; % Where N is the size of the FFT
else
    m =1;
    while(m<=M) % Next value of the order of power 2.
        m = m*2;
    end
    N = m;
end

L = N -M+1;      % Size of data block (50% of overlap)
overlap = N - L;  % size of overlap
Dl = length(X);
extraZeros = 0;
if (mod(Dl,L) == 0)
  X = X;
else
    Dlnew = length(X);
  while (mod(Dlnew,L) ~= 0)
      X = [X 0];
      Dlnew = length(X);
            extraZeros = extraZeros + 1;
    end
end
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MATLAB approach of overlap-save method (First create matrix with
% overlap and then perform convoluion)
zerosForFilter = zeros(1,N-M);
h1=[h zerosForFilter];
H1 = fft(h1);
```

```

x1=X;
nr=ceil((length(x1))/L);

tic
for k=1:nr
    Ma(k,:)=x1(((k-1)*L+1):k*L);
    if k==1
        Ma1(k,:)=[zeros(1,overlap) Ma(k,:)];
    else
        tempVectorM = Ma1(k-1,:);
        overlapData = tempVectorM(L+1:end);
        Ma1(k,:)=[overlapData Ma(k,:)];
    end
    auxfft = fft(Ma1(k,:));
    auxMult = auxfft.*H1;
    Ma2(k,:)=ifft(auxMult);
end

Ma3=Ma2(:,N-L+1:end);
y1=Ma3';
y=y1(:)';
y = y(1:end - extraZeros);
toc
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 4 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read overlap-save data file generated by C++ program and compare with
fullData = load('overlap_save_data.txt');
A=1;
B=A+1;
l=1;
Z=zeros(length(fullData)/2,1);
while (l<=length(Z))
Z(l) = fullData(A)+fullData(B)*1i;
A = A+2;
B = B+2;
l=l+1;
end

figure;
plot(t,real(y))
hold on
plot(t,real(Z),'o')
axis([min(t) max(t) 1.1*min(y) 1.1*max(y)]);
title('Comparision of overlapSave method of MATLAB and C++ ')
legend('MATLAB overlapSave','C++ overlapSave')
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Our MATLAB and C++ implementation test with the built-in conv function of
% MATLAB.
tic
P = conv(Xref,h);
toc
figure
plot(t, P(1:size(Z,1)),'r')
hold on
plot(t,real(Z),'o')
title('Comparision of MATLAB function conv() and overlaSave')
axis([min(t) max(t) 1.1*min(real(Z)) 1.1*max(real(Z))]);
xlabel('t (Seconds)')
ylabel('Z(t) & conv(Xref,h)')
legend('MATLAB function : conv(X,h)','C++ overlapSave')
```

Listing 8.3: overlapSave_test.m code

Step 3 : Choose for a sig and filt value between [1 7] and [1 3] respectively and run the first three sections namely **section 1**, **section 2** and **section 3**.
This will generate a *time_domain_data.txt* and *time_domain_filter.txt* file in the same folder which contains the time domain signal and filter data respectively.

Step 4 : Find the **overlapSave_test.vcxproj** file in the same folder and open it.
In this project file, find *overlapSave_test.cpp* in $SourceFiles$ section and click on it. This file is an example of using $overlapSave$ function. Basically, *overlapSave_test.cpp* file consists of four sections:
Section 1 : It reads the *time_domain_data.txt* and *time_domain_filter.txt* files.
Section 2 : It converts signal and filter data into complex form.
Section 3 : It calls the $overlapSave$ function to perform convolution.
Section 4 : It saves the result in the text file namely *overlap_save_data.txt*.

```
# include "overlapSave_20180208.h"

# include <complex>
# include <fstream>
# include <iostream>
# include <math.h>
# include <stdio.h>
# include <string>
# include <strstream>
# include <algorithm>
# include <vector>

using namespace std;

int main()
{
  ////////////////////////////////// Section 1 ////////////////////////////////////
  ////////////////////////////////////////////////////////////////////////////////
```

```
19  ////// Read the time_domain_data.txt and time_domain_filter.txt files  ///////
    ////////////////////////////////////////////////////////////////////////////
21  ifstream inFile;
    double ch;
23  vector <double> inTimeDomain;
    inFile.open("time_domain_data.txt");
25
    // First data (at 0th position) appled to the ch it is similar to the "cin".
27  inFile >> ch;

29  // It'll count the length of the vector to verify with the MATLAB
    int count = 0;
31
    while (!inFile.eof())
33  {
      // push data one by one into the vector
35    inTimeDomain.push_back(ch);

37    // it'll increase the position of the data vector by 1 and read full vector.s
      inFile >> ch;
39    count++;
    }
41
    inFile.close(); // It is mandatory to close the file at the end.
43
    ifstream inFileFilter;
45  double chFilter;
    vector <double> inTimeDomainFilter;
47  inFileFilter.open("time_domain_filter.txt");
    inFileFilter >> chFilter;
49  int countFilter = 0;

51  while (!inFileFilter.eof())
    {
53    inTimeDomainFilter.push_back(chFilter);
      inFileFilter >> chFilter;
55    countFilter++;
    }
57  inFileFilter.close();

59  ////////////////////////////// Section 2 ////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
61  ////////////////////////// Real to complex conversion //////////////////////
    ////////////////////////////////////////////////////////////////////////////
63  ////////////// For signal data  //////////////
    vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
65  vector <complex<double>> fourierTransformed;
    vector <double> re(inTimeDomain.size());
67  vector <double> im(inTimeDomain.size());

69  for (unsigned int i = 0; i < inTimeDomain.size(); i++)
    {
```

```
71      // Real data of the signal
        re[i] = inTimeDomain[i];
73
        // Imaginary data of the signal
75      im[i] = 0;
    }
77  // Next, Real and Imaginary vector to complex vector conversion
    inTimeDomainComplex = reImVect2ComplexVector(re, im);
79
    ////////////// For filter data  //////////////
81  vector <complex<double>> inTimeDomainFilterComplex(inTimeDomainFilter.size());
    vector <double> reFilter(inTimeDomainFilter.size());
83  vector <double> imFilter(inTimeDomainFilter.size());

85  for (unsigned int i = 0; i < inTimeDomainFilter.size(); i++)
    {
87    reFilter[i] = inTimeDomainFilter[i];
      imFilter[i] = 0;
89  }

91  inTimeDomainFilterComplex = reImVect2ComplexVector(reFilter, imFilter);

93  ////////////////////////////////// Section 3 ////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
95  ////////////////////////////// Overlap & save ///////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
97  vector <complex<double>> y;
    y = overlapSave(inTimeDomainComplex, inTimeDomainFilterComplex);
99
    ////////////////////////////////// Section 4 ////////////////////////////////
101 ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////// Save data ////////////////////////////////
103 ////////////////////////////////////////////////////////////////////////////
    ofstream outFile;
105 complex<double> outFileData;
    outFile.precision(20);
107 outFile.open("overlap_save_data.txt");

109 for (unsigned int i = 0; i <y.size(); i++)
    {
111   outFile << y[i].real() << endl;
      outFile << y[i].imag() << endl;
113 }
    outFile.close();
115
    cout << "Execution finished! Please hit enter to exit." << endl;
117 getchar();
    return 0;
119 }
```

Listing 8.4: overlapSave_test.cpp code

Step 5 : Now, go to the **overlapSave_test.m** and run section 4 and 5.

It'll display the graphs of comparative analysis of the MATLAB and C++ implementation of overlapSave program and also compares results with the MATLAB conv() function.

Resultant analysis of various test signals

1. Signal with two sinusoids and random noise

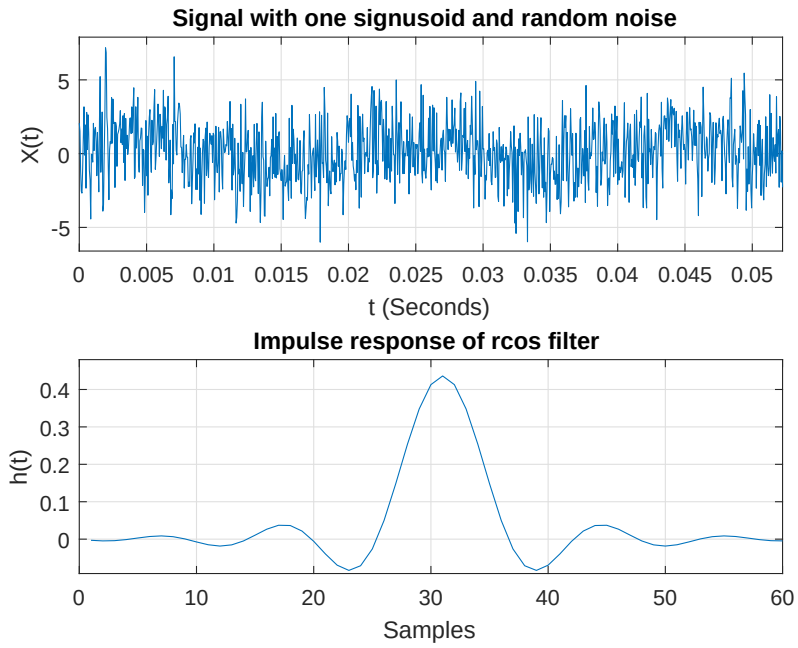


Figura 8.16: Random noise and two sinusoids signal & Impulse response of rcos filter

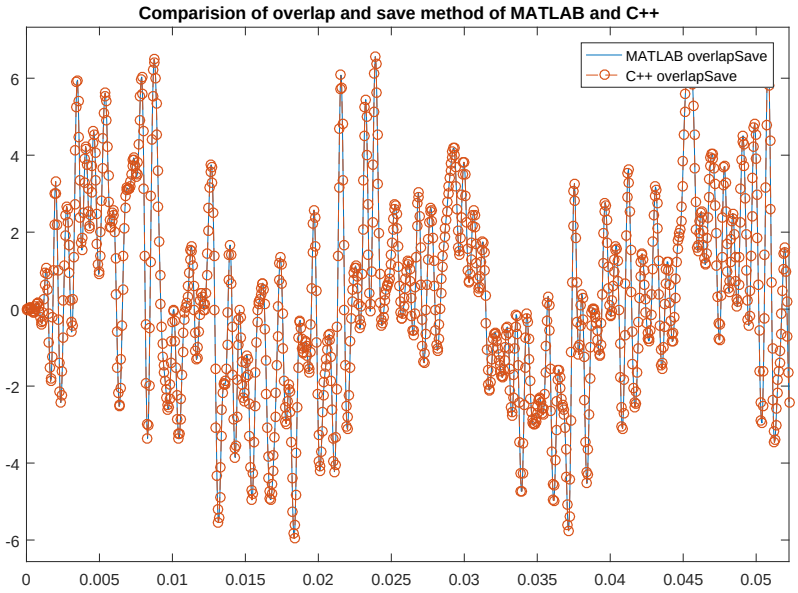


Figura 8.17: MATLAB and C++ comparison

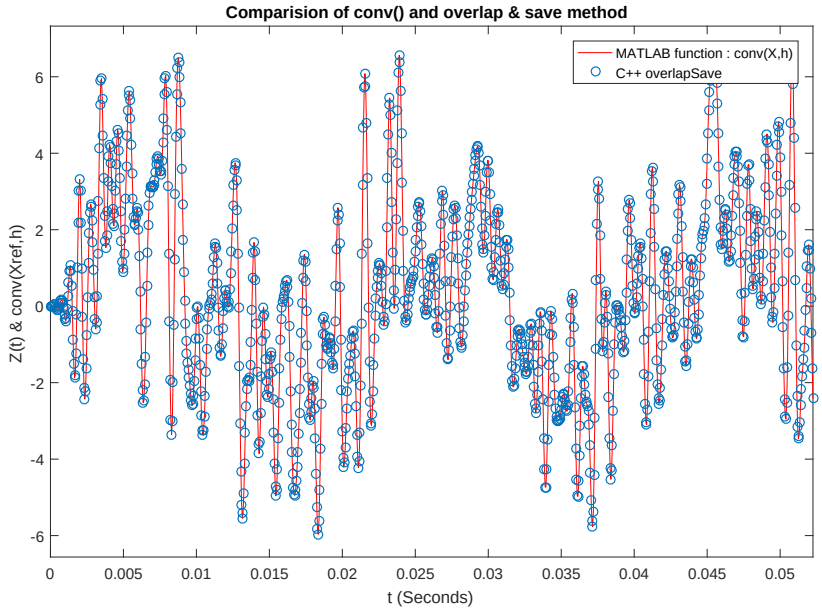


Figura 8.18: MATLAB function conv() and C++ overlapSave comparison

2. Mixed signal2

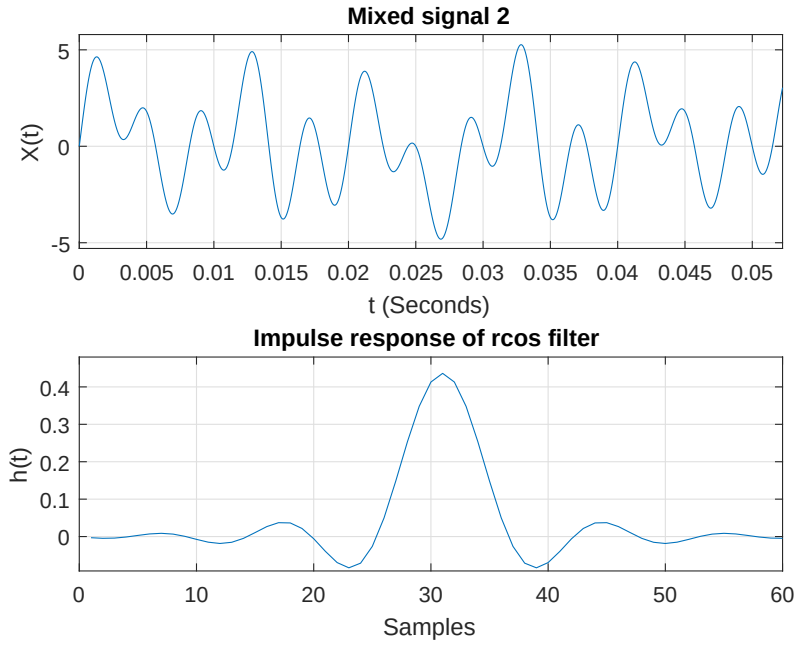


Figura 8.19: Mixed signal2 & Impulse response of rcos filter

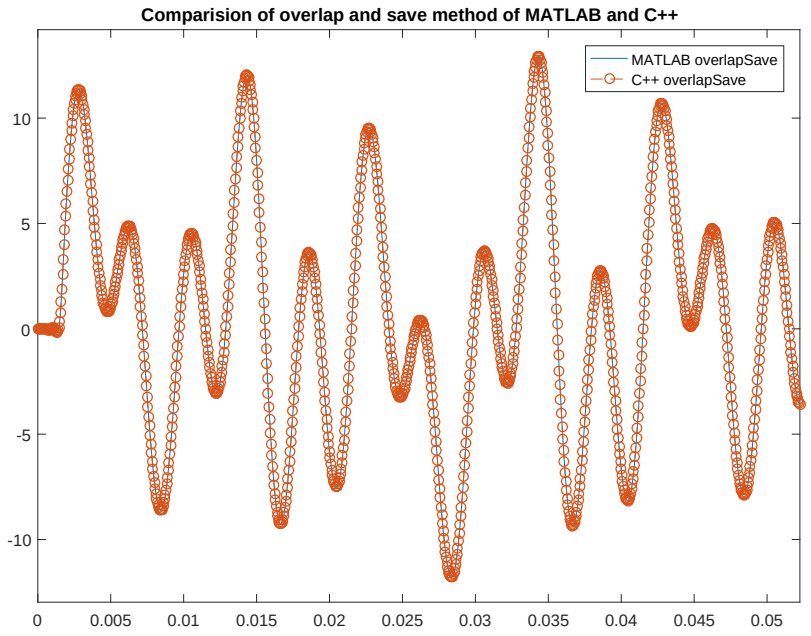


Figura 8.20: MATLAB and C++ comparison

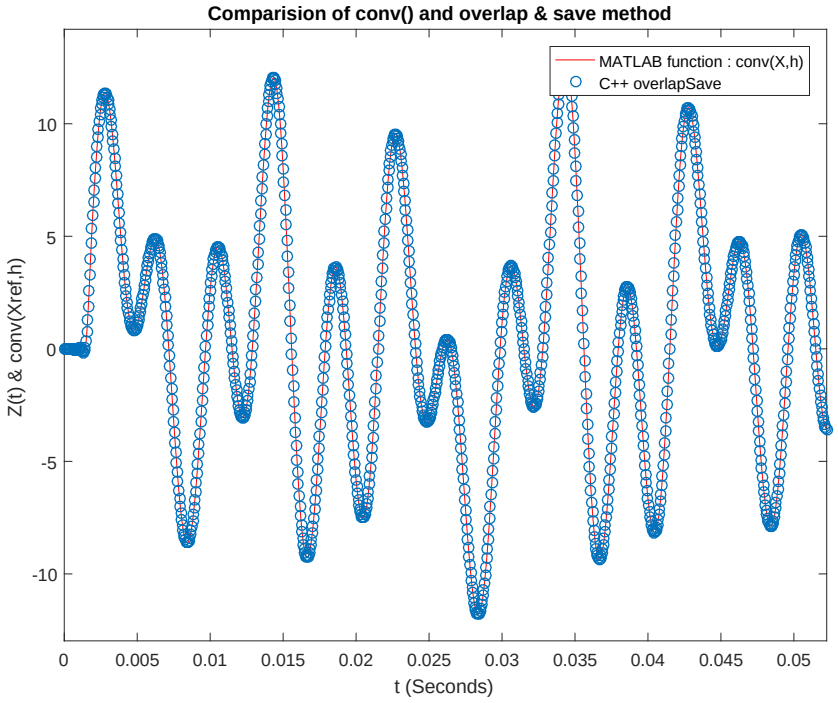


Figura 8.21: MATLAB function conv() and C++ overlapSave comparison

3. Sinusoid with exponent

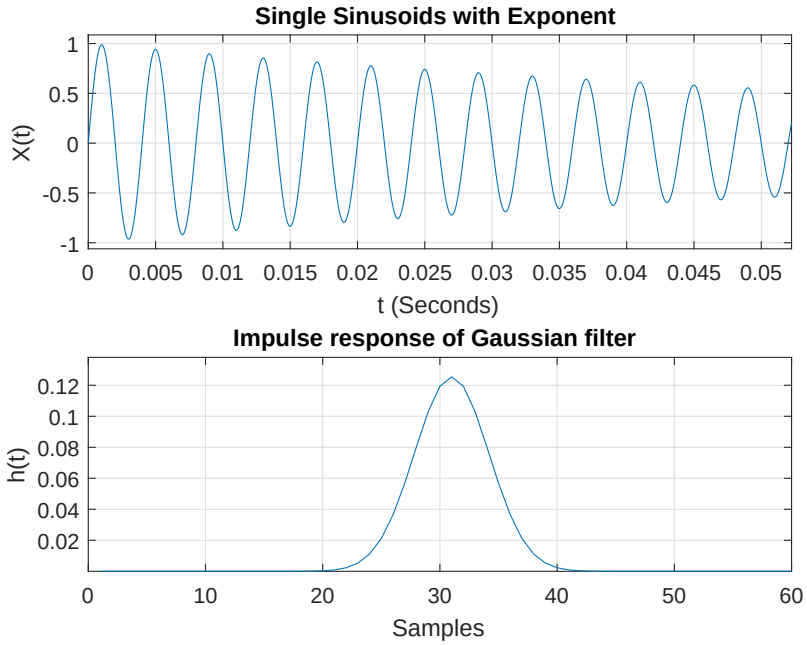


Figura 8.22: Sinusoid with exponent & Impulse response of Gaussian filter

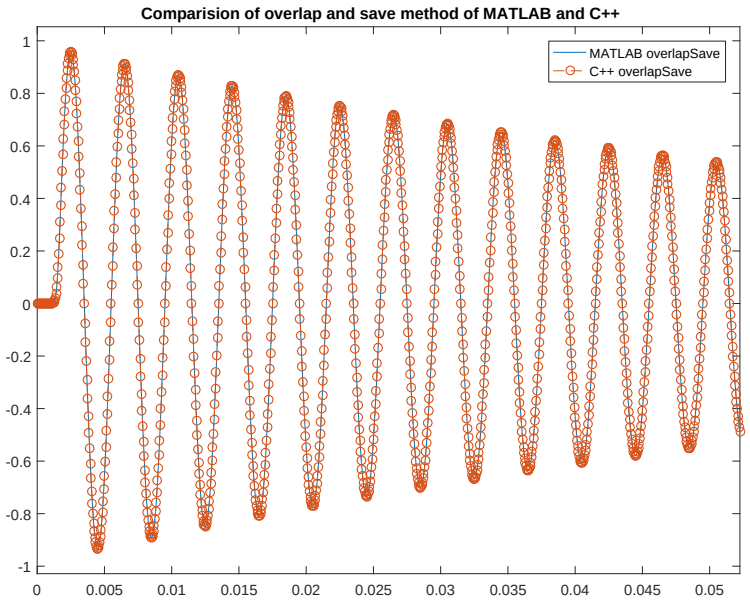


Figura 8.23: MATLAB and C++ comparison

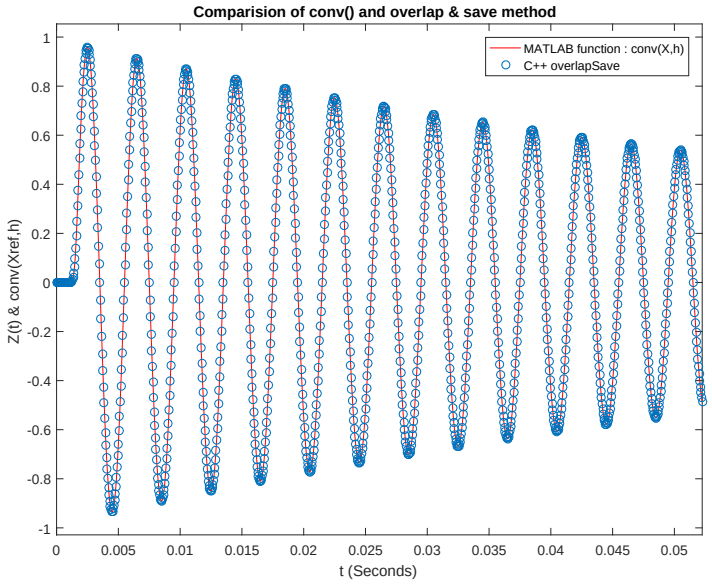


Figura 8.24: MATLAB function conv() and C++ overlapSave comparison

References

[1] Blahut, R.E. *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, MA, 1985.

[2] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.

8.3 Filter

Header File	:	filter_*.h
Source File	:	filter_*.cpp
Version	:	20180201 (Romil Patel)

Test setup

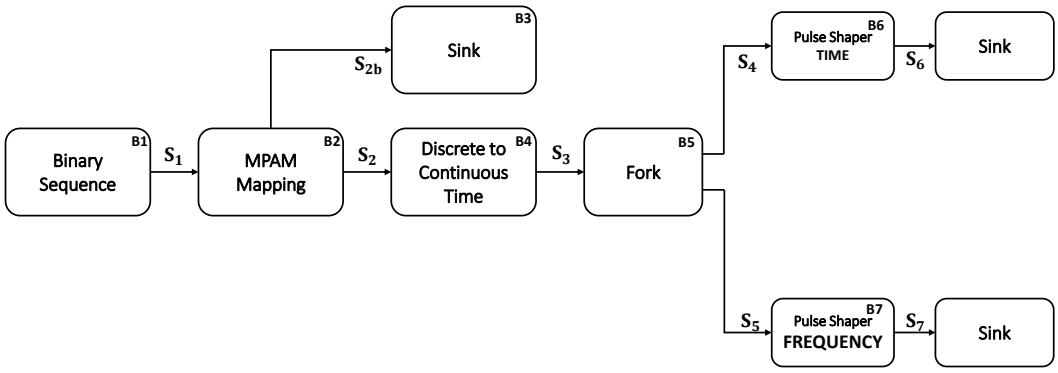


Figura 8.25: Filter test setup

Test example

This section explains the how to use the pulse shaping filter to work in the time and frequency domain.
Step 1 : Open the folder namely **filter_test** by following the path "/algorithms/filter/filter_test".
Step 2 : Find the **filter_test.vcxproj** file in the same folder and open it.
In this project file, find *filter_test.cpp* in $SourceFiles$ section and click on it. This file is an example of using $PulseShaper$ in the time and frequency domain.

```
# include "netxpto.h"
# include "binary_source.h"
# include "m_qam_mapper.h"
# include "discrete_to_continuous_time.h"
# include "sink.h"
```

```
# include "filter.h"
# include "fork.h"
# include "overlapSave_20180208.h"


int main() {

  // ##############################################################################
  // ###################### System Input Parameters ################################
  // ##############################################################################

  BinarySourceMode sourceMode{ PseudoRandom };
  int patternLength{ 5 };
  double bitPeriod{ 1.0 / 2.5e9 };
  vector<t_iqValues> iqAmplitudes{ { { 0,0 },{ 1,0 },{ 2,0 },{ 3,0 } } };
  int numberOfBits{ 1000 };      // -1 will generate long bit sequence.
  int numberOfSamplesPerSymbol{ 16 };
  double rollOffFactor{ 0.3 };
  int impulseResponseTimeLength{ 16 };
  int transferFunctionFrequencyLength{ 16 };

  // ##############################################################################
  // ################ Signals Declaration and Inicialization ################
  // ##############################################################################

  Binary S1{ "S1.sgn" };                // Binary Sequence
  TimeDiscreteAmplitudeDiscreteReal S2{"S2.sgn"};  // MPAM Signal
  TimeDiscreteAmplitudeDiscreteReal S2b{"S2b.sgn"};// Not used (Q signal)
  TimeContinuousAmplitudeDiscreteReal S3{"S3.sgn"};// Discrete to continious time
  TimeContinuousAmplitudeContinuousReal S4{"S4.sgn"};// Pulse Shapping filter
  TimeContinuousAmplitudeContinuousReal S5{"S5.sgn"};// Hilbert filter
  TimeContinuousAmplitudeContinuousReal S6{"S6.sgn"};
  TimeContinuousAmplitudeContinuousReal S7{"S7.sgn"};

  // ##############################################################################
  // ############### Blocks Declaration and Inicialization ###################
  // ##############################################################################

  BinarySource B1{ vector<Signal*> {}, vector<Signal*> { &S1} };
  B1.setMode(sourceMode);
  B1.setPatternLength(patternLength);
  B1.setBitPeriod(bitPeriod);
  B1.setNumberOfBits(numberOfBits);

  MQamMapper B2{ vector<Signal*> { &S1 }, vector<Signal*> { &S2, &S2b } };
  B2.setIqAmplitudes(iqAmplitudes);

  Sink B3{ vector<Signal*> { &S2b }, vector<Signal*> {} };

  DiscreteToContinuousTime B4{ vector<Signal*> { &S2 }, vector<Signal*> { &S3 }
    };
  B4.setNumberOfSamplesPerSymbol(numberOfSamplesPerSymbol);
```

```
57
   Fork B5{ vector<Signal*> { &S3 }, vector<Signal*> { &S4, &S5 } };
59
   PulseShaper B6{ vector<Signal*> { &S4 }, vector<Signal*> { &S6 } };
61 B6.setRollOffFactor( rollOffFactor);
   B6.setImpulseResponseTimeLength(impulseResponseTimeLength);
63 B6.setSeeBeginningOfImpulseResponse(false);
   B6.setFilterDomainType("time");
65 B6.setFilterType(RaisedCosine);

67
   PulseShaper B7{ vector<Signal*> { &S5 }, vector<Signal*> { &S7 } };
69 B7.setRollOffFactor(rollOffFactor);
   B7.setTransferFunctionFrequencyLength(transferFunctionFrequencyLength);
71 B7.setSeeBeginningOfTransferFunction(false);
   B7.setFilterDomainType("frequency");
73 B7.setFilterType(RaisedCosine);

75
   Sink B8{ vector<Signal*> { &S6 }, vector<Signal*> {} };
77 Sink B9{ vector<Signal*> { &S7 }, vector<Signal*> {} };

79


81 // ##############################################################################
   // ################ System Declaration and Inicialization ######################
83 // ##############################################################################

85 System MainSystem{ vector<Block*> { &B1, &B2, &B3, &B4, &B5, &B6, &B7, &B8, &B9
     } };

87 // ##############################################################################
   // ############################## System Run ####################################
89 // ##############################################################################

91 MainSystem.run();
   cout << "\nExecution Finished, Please hit Enter to exit!";
93 getchar();
   return 0;
95 }
```

Listing 8.5: filter_test.cpp code

Step 3 : Set the domain of the filter to *time* or *frequency* using **setFilterDomianType** and select type of the filter to *RaisedCosine, Gaussian* or *Square* using **setFilterType**.

Step 4 : Run the code and compare the resulting signals **S6.sgn** and **S7.sgn** using visualizer. The *filter.h* file contains two class namely **FilterRoot** and **PulseShaperFilter**.

```
1 # ifndef FILTER_H_
  # define FILTER_H_
```

```
# include <vector>
# include "netxpto.h"


/*
class FilterRoot : public Block {

public:

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

  void initializeFilterRoot(void);
  bool runBlock(void);
  void terminate(void) {};

  void setSaveImpulseResponse(bool sImpulseResponse) { saveImpulseResponse =
    sImpulseResponse; }; // saveImpulseResponse
  bool getSaveImpulseResponse(void) { return saveImpulseResponse; };

  void setSeeBeginningOfImpulseResponse(bool sBeginning) {
    seeBeginningOfImpulseResponse = sBeginning; }; //
    seeBeginningOfImpulseResponse
  bool const getSeeBeginningOfImpulseResponse() { return
    seeBeginningOfImpulseResponse; };


  void setSaveTransferFunction(bool sTransferFunction) { saveTransferFunction =
    sTransferFunction; }; // saveTransferFunction
  bool getSaveTransferFunction(void) { return saveTransferFunction; };

  void setSeeBeginningOfTransferFunction(bool sBeginning) {
    seeBeginningOfTransferFunction = sBeginning; }; //
    seeBeginningOfTransferFunction
  bool const getSeeBeginningOfTransferFunction() { return
    seeBeginningOfTransferFunction; };


  void setImpulseResponseLength(int iResponseLength) { impulseResponseLength =
    iResponseLength; }; // impulseResponseLength
  int const getImpulseResponseLength() { return impulseResponseLength; }

  void setTransferFunctionLength(int iTransferFunctionLength) {
    transferFunctionLength = iTransferFunctionLength; }; // transferFunctionLength
  int const getTransferFunctionLength() { return transferFunctionLength; };

private:

  vector<t_real> impulseResponse;
  int impulseResponseLength;
```

```
  bool saveImpulseResponse{ true };
  string impulseResponseFilename{ "impulse_response.imp" };
  bool seeBeginningOfImpulseResponse{ true };

  vector<t_complex> transferFunction;
  int transferFunctionLength;
  bool saveTransferFunction{ true };
  string transferFunctionFilename{ "transfer_function.tfn" };
  bool seeBeginningOfTransferFunction{ true };

  string filterDomain;


  //FREQ
  vector<t_complex> previousCopy;
  int K{ 0 };



};
*/


class FIR_Filter : public Block {

public:

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

  void initializeFIR_Filter(void);

  bool runBlock(void);

  void terminate(void) {};

  void setSaveImpulseResponse(bool sImpulseResponse) { saveImpulseResponse =
    sImpulseResponse; };
  bool getSaveImpulseResponse(void) { return saveImpulseResponse; };

  void setImpulseResponseLength(int iResponseLength) { impulseResponseLength =
    iResponseLength; };
  int const getImpulseResponseLength() { return impulseResponseLength; }

  void setSeeBeginningOfImpulseResponse(bool sBeginning) {
    seeBeginningOfImpulseResponse = sBeginning; };
  bool const getSeeBeginningOfImpulseResponse() { return
    seeBeginningOfImpulseResponse; };

private:
```

```
91    // State Variable
      vector<t_real> impulseResponse;
93    vector<t_real> delayLine;

95    /* Input Parameters */
      int impulseResponseLength;              // filter order + 1 (filter order =
        number of delays)
97    bool saveImpulseResponse{ true };
      string impulseResponseFilename{ "impulse_response.imp" };
99    bool seeBeginningOfImpulseResponse{ true };


101
  };
103


105 class FD_Filter : public Block {

107   /* State Variable */

109   vector<t_real> inputBufferTimeDomain;
      vector<t_real> outputBufferTimeDomain;
111
      int inputBufferPointer{ 0 };
113   int outputBufferPointer{ 0 };

115   /* Input Parameters */
      bool saveTransferFunction{ true };
117   string transferFunctionFilename{ "transfer_function.tfn" };
      int transferFunctionLength{ 128 };
119
      int inputBufferTimeDomainLength{ transferFunctionLength };
121   int outputBufferTimeDomainLength{ transferFunctionLength };

123 public:
      /* State Variable */
125   vector<t_complex> transferFunction;

127   /* Methods */
      FD_Filter() {};
129   FD_Filter(vector<Signal *> &InputSig, vector<Signal *> OutputSig) :Block(
        InputSig, OutputSig) {};

131   void initializeFD_Filter(void);

133   bool runBlock(void);

135   void terminate(void) {};

137   void setInputBufferTimeDomainLength(int iBufferTimeDomainLength) {
        inputBufferTimeDomainLength = iBufferTimeDomainLength; };
      int const getInputBufferTimeDomainLength() { return inputBufferTimeDomainLength
        ; }
```

```
139
    void setOutputBufferTimeDomainLength(int oBufferTimeDomainLength) {
      outputBufferTimeDomainLength = oBufferTimeDomainLength; };
141 int const getOutputBufferTimeDomainLength() { return
      outputBufferTimeDomainLength; }

143 void setInputBufferPointer(int iBufferPointer) { inputBufferPointer =
      iBufferPointer; };
    int const getInputBufferPointer() { return inputBufferPointer; }
145
    void setOutputBufferPointer(int oBufferPointer) { outputBufferPointer =
      oBufferPointer; };
147 int const getOutputBufferPointer() { return outputBufferPointer; }

149 void setSaveTransferFunction(bool sTransferFunction) { saveTransferFunction =
      sTransferFunction; };
    bool getSaveTransferFunction(void) { return saveTransferFunction; };
151
    void setTransferFunctionLength(int iTransferFunctionLength) {
      transferFunctionLength = iTransferFunctionLength; };
153 int const getTransferFunctionLength() { return transferFunctionLength; };

155 };

157
////////////////////////////////////////////////////////////////////////////
159 ////////////////////////////// pulseShaper ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
161 /*enum PulseShaperFilter { RaisedCosine, Gaussian, Square, RaisedCosineTfn,
      GaussianTfn, SquareTfn};

163 class PulseShaper : public FilterRoot
    {
165   // Input Parameters
      PulseShaperFilter filterType{ RaisedCosine };
167   int impulseResponseTimeLength{ 16 };    // in units of symbol period
      int transferFunctionFrequencyLength{ 16 };
169   double rollOffFactor{ 0.9 };        // Roll-off factor
      bool passiveFilterMode{ false };
171   string filterDomainType{ "time" };

173 public:

175   // Methods
      PulseShaper() :FilterRoot() {};
177   PulseShaper(vector<Signal *> &InputSig, vector<Signal *> OutputSig) :FilterRoot
        (InputSig, OutputSig) {};

179   void initialize(void);

181   void setImpulseResponseTimeLength(int impResponseTimeLength) {
        impulseResponseTimeLength = impResponseTimeLength; };
```

```
  int const getImpulseResponseTimeLength(void) { return impulseResponseTimeLength
    ; };
183
  void setTransferFunctionFrequencyLength(int transFunctionLength) {
    transferFunctionFrequencyLength = transFunctionLength; };
185 int const getTransferFunctionFrequencyLength(void) { return
    transferFunctionFrequencyLength; };


187
  void setFilterType(PulseShaperFilter fType) { filterType = fType; };
189 PulseShaperFilter const getFilterType(void) { return filterType; };

191 void setRollOffFactor(double rOffFactor) { rollOffFactor = rOffFactor; };
  double const getRollOffFactor() { return rollOffFactor; };
193
  void usePassiveFilterMode(bool pFilterMode) { passiveFilterMode = pFilterMode;
    };
195
  void setFilterDomainType(string fDomain) { filterDomainType = fDomain; };
197 string getFilterDomainType() { return filterDomainType; };
};*/
199 #endif
```

Listing 8.6: filter.h code

The code of the *filter.cpp* contains thee definitions namely; initialize, initialize_FilterRoot and runBlock.

```
# include "filter.h"
2 # include "overlapSave_20180208.h"
# include <algorithm>
4 # include <vector>
# include <algorithm>
6



8
void FilterRoot::initializeFilterRoot(void)
10 {
  if (filterDomain == "time")
12 {
    ////////////////////////////////// TIME
    //////////////////////////////////
14  //
    ////////////////////////////////////////////////////////////////////////////
    outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
16  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
    outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;
18


20  if (!getSeeBeginningOfImpulseResponse()) {
      int aux = (int)(((double)impulseResponseLength) / 2) + 1;
22    outputSignals[0]->setFirstValueToBeSaved(aux);
    }
```

```
24
    if (!getSeeBeginningOfTransferFunction()) {
26    int aux = (int)(((double)transferFunctionLength) / 2) + 1;
      outputSignals[0]->setFirstValueToBeSaved(aux);
28  }

30
    delayLine.resize(impulseResponseLength, 0);
32
    if (saveImpulseResponse)
34  {
      ofstream fileHandler("./signals/" + impulseResponseFilename, ios::out);
36    fileHandler << "// ### HEADER TERMINATOR ###\n";

38    t_real t;
      double samplingPeriod = inputSignals[0]->samplingPeriod;
40    for (int i = 0; i < impulseResponseLength; i++)
      {
42      t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
        fileHandler << t << " " << impulseResponse[i] << "\n";
44    }
      fileHandler.close();
46  }

48 }

50 else
    ////////////////////////////////// FREQ
    //////////////////////////////////////
52  //
    ////////////////////////////////////////////////////////////////////////////
  {
54  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
    outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
56  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

58  if (!getSeeBeginningOfImpulseResponse()) {
      int aux = (int)(((double)impulseResponseLength) / 2) + 1;
60    outputSignals[0]->setFirstValueToBeSaved(aux);
    }
62
    if (!getSeeBeginningOfTransferFunction()) {
64    int aux = (int)(((double)transferFunctionLength) / 2) + 1;
      outputSignals[0]->setFirstValueToBeSaved(aux);
66  }

68  if (saveTransferFunction)
    {
70    ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
      fileHandler << "// ### HEADER TERMINATOR ###\n";
72
      double samplingPeriod = inputSignals[0]->samplingPeriod;
```

```
74        t_real fWindow = 1 / samplingPeriod;
          t_real df = fWindow / transferFunction.size();
76
          t_real f;
78        for (int k = 0; k < transferFunction.size(); k++)
          {
80          f = -fWindow / 2 + k * df;
            fileHandler << f << " " << transferFunction[k] << "\n";
82        }
          fileHandler.close();
84      }

86    }
};
88

90  bool FilterRoot::runBlock(void) {

92    ////////////////////////////////// TIME /////////////////////////////////////
      ////////////////////////////////////////////////////////////////////////////
94    if (filterDomain == "time")
      {
96      int ready = inputSignals[0]->ready();
        int space = outputSignals[0]->space();
98      int process = min(ready, space);
        if (process == 0) return false;
100


102     for (int i = 0; i < process; i++)
        {
104       t_real val;
          (inputSignals[0])->bufferGet(&val);
106
          if (val != 0)
108       {
            vector<t_real> aux(impulseResponseLength, 0.0);
110         transform(impulseResponse.begin(), impulseResponse.end(), aux.begin(),
      bind1st(multiplies<t_real>(), val));
            transform(aux.begin(), aux.end(), delayLine.begin(), delayLine.begin(),
      plus<t_real>());
112       }
          outputSignals[0]->bufferPut((t_real)(delayLine[0]));
114       rotate(delayLine.begin(), delayLine.begin() + 1, delayLine.end());
          delayLine[impulseResponseLength - 1] = 0.0;
116     }

118     return true;
      }
120

      else
122     ////////////////////////////////// FREQ /////////////////////////////////
        ////////////////////////////////////////////////////////////////////////
```

```
124  {
       bool alive{ false };
126
       int ready = inputSignals[0]->ready();
128    int space = outputSignals[0]->space();
       int process = min(ready, space);
130    if (process == 0) return false;

132    vector<t_real> re(process);
       vector<t_real> im(process);
134    vector<t_complex> IN1(process);
       vector<t_complex> OUTaux;
136    vector<t_complex> OUT;
       t_real input;
138    vector<double> inputBufferTimeDomain(process);
       vector<t_complex>  currentCopyAux;
140
       ////////////////////// previousCopy & currentCopy ////////////////////
142    //////////////////////////////////////////////////////////////////////
       for (int i = 0; i < process; i++)              // Get the Input signal
144    {
         inputSignals[0]->bufferGet(&input);
146      inputBufferTimeDomain.at(i) = input;
       }
148

150    for (int i = 0; i < process; i++)
       {
152      re[i] = inputBufferTimeDomain.at(i); // Real part of input
         im[i] = 0;         // Imaginary part which is manipulated as "0"
154    }
       currentCopyAux = reImVect2ComplexVector(re, im);// currentCopy complex form
156

158    vector<t_complex> pcInitialize(process);
       if (K == 0)                  // For the first data block only
160    {
         previousCopy = pcInitialize;       // PREVIOUS COPY
162    }

164    // size modification of currentCopyAux to currentCopy.
       vector<t_complex> currentCopy(previousCopy.size());
166    for (unsigned int i = 0; i < currentCopyAux.size(); i++)
       {
168      currentCopy[i] = currentCopyAux[i];
       }
170
       /////////////////////////// Filter Data "hn" ///////////////////////////
172    //////////////////////////////////////////////////////////////////////
       vector<t_real> rehn(impulseResponse.size());
174    vector<t_real> imhn(impulseResponse.size());
```

```
176     for (int i = 0; i < impulseResponse.size(); i++)
        {
178       rehn[i] = impulseResponse.at(i);// Real part of input
          imhn[i] = 0;     // Imaginary part which is manipulated as "0"
180     }

182     vector<t_complex> hn = reImVect2ComplexVector(rehn, imhn); // filter hn
        complex form

184                                     ////////////////////// OverlapSave in Realtime
         ////////////////////////
                                        //
        ////////////////////////////////////////////////////////////////////
186     OUTaux = overlapSaveRealTime(currentCopy, previousCopy, hn);

188     previousCopy = currentCopy;
        K = K + 1;
190
        // Discard the overlap data
192     for (int i = 0; i < process; i++)
        {
194       OUT.push_back(OUTaux[previousCopy.size() + i]);
        }
196
        // Bufferput
198     for (int i = 0; i < process; i++)
        {
200       t_real val;
          val = OUT[i].real();
202       outputSignals[0]->bufferPut((t_real)(val));
        }
204
        return true;
206
    }
208 };

210 ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////// pulseShaper ////////////////////////////////////
212 ////////////////////////////////////////////////////////////////////////////
    void raisedCosine(vector<t_real> &impulseResponse, vector<t_complex> &
        transferFunction, int impulseResponseLength, int transferFunctionLength,
        double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
        passiveFilterMode, string &filterDomain);
214 void gaussian(vector<t_real> &impulseResponse, vector<t_complex> &
        transferFunction, int impulseResponseLength, int transferFunctionLength,
        double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
        passiveFilterMode, string &filterDomain);
    void square(vector<t_real> &impulseResponse, vector<t_complex> &transferFunction,
         int impulseResponseLength, int transferFunctionLength, double samplingPeriod,
         double symbolPeriod, string &filterDomain);
216
```

```
void raisedCosineTfn(vector<t_real> &impulseResponse, vector<t_complex> &
    transferFunction, int impulseResponseLength, int transferFunctionLength,
    double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
    passiveFilterMode, string &filterDomain);
void gaussianTfn(vector<t_real> &impulseResponse, vector<t_complex> &
    transferFunction, int impulseResponseLength, int transferFunctionLength,
    double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
    passiveFilterMode, string &filterDomain);

void PulseShaper::initialize(void) {

  double samplingPeriod = inputSignals[0]->samplingPeriod;
  double symbolPeriod = inputSignals[0]->symbolPeriod;

  impulseResponseLength = (int)floor(impulseResponseTimeLength * symbolPeriod /
    samplingPeriod);
  transferFunctionLength = (int)floor(transferFunctionFrequencyLength *
    symbolPeriod / samplingPeriod);

  impulseResponse.resize(impulseResponseLength);
  transferFunction.resize(transferFunctionLength);
  filterDomain = filterDomainType;

  switch (getFilterType()) {

  case RaisedCosine:
    raisedCosine(impulseResponse, transferFunction, impulseResponseLength,
    transferFunctionLength, rollOffFactor, samplingPeriod, symbolPeriod,
    passiveFilterMode, filterDomain);
    break;
  case Gaussian:
    gaussian(impulseResponse, transferFunction, impulseResponseLength,
    transferFunctionLength, rollOffFactor, samplingPeriod, symbolPeriod,
    passiveFilterMode, filterDomain);
    break;
  case Square:
    square(impulseResponse, transferFunction, impulseResponseLength,
    transferFunctionLength, samplingPeriod, symbolPeriod, filterDomain);
    break;
  case RaisedCosineTfn:
    raisedCosineTfn(impulseResponse, transferFunction, impulseResponseLength,
    transferFunctionLength, rollOffFactor, samplingPeriod, symbolPeriod,
    passiveFilterMode, filterDomain);
    break;
  case GaussianTfn:
    gaussianTfn(impulseResponse, transferFunction, impulseResponseLength,
    transferFunctionLength, rollOffFactor, samplingPeriod, symbolPeriod,
    passiveFilterMode, filterDomain);
    break;
  };

  impulseResponseLength = (int)impulseResponse.size();     // It is due to update
```

```
    the size of impulseResponseLength
  transferFunctionLength = (int)transferFunction.size();  // It is due to update
    the size of transferFunctionLength

  initializeFilterRoot();
}

////////////////////////////////////////////////////////////
////////////////////// Impulse Responses ///////////////////////
////////////////////////////////////////////////////////////
void raisedCosine(vector<t_real> &impulseResponse, vector<t_complex> &
    transferFunction, int impulseResponseLength, int transferFunctionLength,
    double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
    passiveFilterMode, string &filterDomain) {
  double sinc;
  double gain{ 0 };
  ofstream dataImpulseT("dataImpulseT.txt");

  for (int i = 0; i < impulseResponseLength; i++) {
    t_real t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
    dataImpulseT << t << "\n";
    if (t != 0) {
      sinc = sin(PI * t / symbolPeriod) / (PI * t / symbolPeriod);
    }
    else {
      sinc = 1;
    }
    impulseResponse[i] = sinc*cos(rollOffFactor*PI*t / symbolPeriod) / (1 - (4.0
    * rollOffFactor * rollOffFactor * t * t) / (symbolPeriod * symbolPeriod));
    gain = gain + impulseResponse[i];
  };
  dataImpulseT.close();

  if (passiveFilterMode)
  {
    for (int i = 0; i < impulseResponseLength; i++)
    {
      impulseResponse[i] = impulseResponse[i] / gain;
    }
  }


  vector<t_real> re(impulseResponse.size());
  vector<t_real> im(impulseResponse.size());

  for (unsigned int i = 0; i < impulseResponse.size(); i++)
  {
    re[i] = impulseResponse[i];
  }

  vector<complex<double>> impulseResponseAux = reImVect2ComplexVector(re, im);
  transferFunction = fft(impulseResponseAux);
```

```
298
   /////////////// Normalization of Response///////////////
300 vector<t_real> absTransferFunction(transferFunction.size());
   for (int i = 0; i < transferFunction.size(); i++) { absTransferFunction[i] =
     abs(transferFunction[i]); }
302 double maxValue = *std::max_element(absTransferFunction.begin(),
     absTransferFunction.end());

304
   for (unsigned int i = 0; i < impulseResponseAux.size(); i++)
306 {
     transferFunction[i] = (1 / static_cast<double>(maxValue))*(static_cast<double
     >(samplingPeriod / symbolPeriod)*transferFunction[i] * (sqrt(static_cast<
     double>(impulseResponseAux.size()))));
308 }

310
   ofstream dataImpulseIMP("dataImpulseIMP.txt");
312 ofstream dataImpulseTFN("dataImpulseTFN.txt");
   for (unsigned int i = 0; i < impulseResponse.size(); i++)
314 {
     dataImpulseIMP << impulseResponse[i] << "\n";
316  dataImpulseTFN << transferFunction[i].real() << "\n";
   }
318 dataImpulseIMP.close();
   dataImpulseTFN.close();
320
};
322
void gaussian(vector<t_real> &impulseResponse, vector<t_complex> &
    transferFunction, int impulseResponseLength, int transferFunctionLength,
    double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
    passiveFilterMode, string &filterDomain) {
324 double gauss;
   double pulsewidth = 5e-10;
326 double gain{ 0 };
   /*for (int i = 0; i < impulseResponseLength; i++) {
328 t_real t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
   gauss = exp(-t*t / (pulsewidth*pulsewidth / 36));
330 impulseResponse[i] = gauss;
   gain = gain + impulseResponse[i];
332 };*/
   t_real Ts = symbolPeriod;
334 t_real T = samplingPeriod;
   t_real BTs = 1;
336 t_real a = (sqrt(log(2) / 2)/BTs)*Ts;

338
   ofstream gaussImpulseT("gaussImpulseT.txt");
340 for (int i = 0; i < impulseResponseLength; i++)
   {
342   t_real t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
```

```
344     //gauss = (sqrt(PI)/a)*(exp(-(PI*t / a)*(PI*t / a)));
        gauss = (exp(-(PI*t / a)*(PI*t / a)));
346     impulseResponse[i] = gauss;
        gaussImpulseT << t << "\n";
348     gain = gain + impulseResponse[i];
    };
350 gaussImpulseT.close();


352
    if (passiveFilterMode)
354 {
      for (int i = 0; i < impulseResponseLength; i++)
356   {
        impulseResponse[i] = impulseResponse[i] / gain;
358   }
    }
360
    ////////////////////////////////////////////////////// TF
      ////////////////////////////////////////////////////////////////////////////////////////////////////


362
    vector<t_real> im(impulseResponse.size());
364 vector<t_complex> impulseResponseAux = reImVect2ComplexVector(impulseResponse,
      im);
    transferFunction = fft(impulseResponseAux);
366
    /////////////// Normalization of Response///////////////
368 vector<t_real> absTransferFunction(transferFunction.size());
    for (int i = 0; i < transferFunction.size(); i++) { absTransferFunction[i] =
      abs(transferFunction[i]);}
370 double maxValue = *std::max_element(absTransferFunction.begin(),
      absTransferFunction.end());


372
    for (unsigned int i = 0; i < impulseResponseAux.size(); i++)
374 {
      transferFunction[i] = (1/ static_cast<double>(maxValue))*(static_cast<double
      >(samplingPeriod / symbolPeriod)*transferFunction[i] * (sqrt(static_cast<
      double>(impulseResponseAux.size()))));
376 }

378 /////////////// Write Response///////////////
    ofstream gaussImpulseIMP("gaussImpulseIMP.txt");
380 ofstream gaussImpulseTFN("gaussImpulseTFN.txt");
    for (unsigned int i = 0; i < impulseResponse.size(); i++)
382 {
      gaussImpulseIMP << impulseResponse[i] << "\n";
384   gaussImpulseTFN << transferFunction[i].real() << "\n";
    }
386 gaussImpulseIMP.close();
    gaussImpulseTFN.close();
```

```
388 };

390 void square(vector<t_real> &impulseResponse, vector<t_complex> &transferFunction,
        int impulseResponseLength, int transferFunctionLength, double samplingPeriod,
        double symbolPeriod, string &filterDomain) {

392   int samplesPerSymbol = (int)(symbolPeriod / samplingPeriod);

394   for (int k = 0; k < samplesPerSymbol; k++) {
        impulseResponse[k] = 1.0;
396   };

398   for (int k = samplesPerSymbol; k < impulseResponseLength; k++) {
        impulseResponse[k] = 0.0;
400   };

402   vector<t_real> re(impulseResponse.size());
      vector<t_real> im(impulseResponse.size());
404
      for (unsigned int i = 0; i < impulseResponse.size(); i++)
406   {
        re[i] = impulseResponse[i];
408   }

410   vector<t_complex> impulseResponseAux = reImVect2ComplexVector(re, im);
      transferFunction = fft(impulseResponseAux);
412
      for (unsigned int i = 0; i < impulseResponseAux.size(); i++)
414   {
        transferFunction[i] = transferFunction[i] * (sqrt(static_cast<double>(
        impulseResponseAux.size())));
416   }

418 };

420
////////////////////////////////////////////////////////////
422 ////////////////////// Transfer Function ///////////////////////
////////////////////////////////////////////////////////////
424 void raisedCosineTfn(vector<t_real> &impulseResponse, vector<t_complex> &
        transferFunction, int impulseResponseLength, int transferFunctionLength,
        double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
        passiveFilterMode, string &filterDomain)
{
426   t_real Ts = symbolPeriod;
      t_real T = samplingPeriod;
428
      t_real fWindow = 1 / samplingPeriod;
430   t_real df = fWindow / transferFunctionLength;

432   t_real f;
      vector<t_real> fSpan;
```

```
434   ofstream dataTransferF("dataTransferF.txt");
      for (int k = 0; k < transferFunctionLength; k++)
436   {
        f = -fWindow / 2 + k * df;
438     fSpan.push_back(f);
        dataTransferF << f << "\n";
440   }
      dataTransferF.close();
442

444   vector<t_real> transferFunctionReal;
      for (unsigned int i = 0; i < fSpan.size(); i++)
446   {
        // Calculate the transferFunctionAux
448     t_real negativeLimit = (1 - rollOffFactor) / (2 * Ts);
        t_real positiveLimit = (1 + rollOffFactor) / (2 * Ts);
450

        if (abs(fSpan[i]) <= negativeLimit)
452     {
          transferFunctionReal.push_back(1);
454     }
        else if (negativeLimit < abs(fSpan[i]) && abs(fSpan[i]) <= positiveLimit)
456     {
          t_real value;
458       value = 0.5*(1 + cos((PI*Ts / rollOffFactor) * (abs(fSpan[i]) - ((1 -
      rollOffFactor) / (2 * Ts)))));
          transferFunctionReal.push_back(value);
460     }
        else
462     {
          transferFunctionReal.push_back(0);
464     }


466
      }
468   ofstream dataTransferTFN("dataTransferTFN.txt");
      for (unsigned int i = 0; i < transferFunctionReal.size(); i++)
470   {
        dataTransferTFN << transferFunctionReal[i] << "\n";
472   }
      dataTransferTFN.close();
474

      // convert transferFunctionAux to complex value
476   vector<t_complex> transferFunctionComplex;
      vector<t_real> imTfn(transferFunctionReal.size());
478   transferFunctionComplex = reImVect2ComplexVector(transferFunctionReal, imTfn);

480   transferFunction = transferFunctionComplex;

482   // Calculate the impulse response of the filter
      vector<t_complex> impulseResponseComplex(transferFunctionComplex.size());
484   vector<t_real> impulseResponseAux(transferFunctionComplex.size());
```

```
486   transferFunctionComplex = ifftshift(transferFunctionComplex); // ifftshift
      impulseResponseComplex = ifft(transferFunctionComplex);      // ifft
488   impulseResponseComplex = fftshift(impulseResponseComplex);    // fftshift

490
      // Multiply it with the (symbolPeriod/samplingPeriod) to adjust the shape
492   vector<t_real> absImpulseResponse(impulseResponseComplex.size());
      for (int i = 0; i < impulseResponseComplex.size(); i++) { absImpulseResponse[i]
        = (impulseResponseComplex[i].real()); }
494   double maxValue1 = *std::max_element(absImpulseResponse.begin(),
        absImpulseResponse.end());

496   for (unsigned int i = 0; i < transferFunctionComplex.size(); i++) {
        //impulseResponseAux[i] = (sqrt(2)*PI)* impulseResponseComplex[i].real() / (
        sqrt(static_cast<double>(transferFunctionComplex.size())));
498     impulseResponseAux[i] = (1 / maxValue1)*(symbolPeriod / samplingPeriod)*
        absImpulseResponse[i] / (sqrt(static_cast<double>(transferFunctionComplex.size
        ())));
      }
500
      impulseResponse = impulseResponseAux;
502   impulseResponseLength = (int)impulseResponse.size();

504   ofstream dataTransferIMP("dataTransferIMP.txt");
      for (unsigned int i = 0; i < impulseResponse.size(); i++) {
506     dataTransferIMP << impulseResponse[i] << "\n";
      }
508   dataTransferIMP.close();

510 }

512 void gaussianTfn(vector<t_real> &impulseResponse, vector<t_complex> &
      transferFunction, int impulseResponseLength, int transferFunctionLength,
      double rollOffFactor, double samplingPeriod, double symbolPeriod, bool
      passiveFilterMode, string &filterDomain)
    {
514   t_real Ts = symbolPeriod;
      t_real T = samplingPeriod;
516   t_real fWindow = 1 / T;
      t_real df = fWindow / transferFunctionLength;
518
      t_real f;
520   vector<t_real> fSpan;
      ofstream gaussTransferF("gaussTransferF.txt");
522   for (int k = 0; k < transferFunctionLength; k++) {
        f = -fWindow / 2 + k * df;
524     fSpan.push_back(f);
        gaussTransferF << f << "\n";
526   }
      gaussTransferF.close();
528
```

```
  t_real BTs = 1;
530 t_real a = (sqrt(log(2) / 2) / BTs)*Ts;

532 vector<t_real> transferFunctionReal;
  for (unsigned int i = 0; i < fSpan.size(); i++)
534 {
    // Calculate the transferFunctionAux
536   t_real value = exp(-a*a*fSpan[i] * fSpan[i]);
    //t_real value = exp(-(PI*PI*fSpan[i] * fSpan[i] / a));
538   transferFunctionReal.push_back(value);
  }
540
  ofstream gaussTransferTFN("gaussTransferTFN.txt");
542 for (unsigned int i = 0; i < transferFunctionReal.size(); i++) {
    gaussTransferTFN << transferFunctionReal[i] << "\n";
544 }
  gaussTransferTFN.close();
546
  // convert transferFunctionAux to complex value
548 vector<t_complex> transferFunctionComplex;
  vector<t_real> imTfn(transferFunctionReal.size());
550 transferFunctionComplex = reImVect2ComplexVector(transferFunctionReal, imTfn);

552 transferFunction = transferFunctionComplex;

554 // Calculate the impulse response of the filter
  vector<t_complex> impulseResponseComplex(transferFunctionComplex.size());
556 vector<t_real> impulseResponseAux(transferFunctionComplex.size());

558 transferFunctionComplex = ifftshift(transferFunctionComplex); // ifftshift
  impulseResponseComplex = ifft(transferFunctionComplex);     // ifft
560 impulseResponseComplex = fftshift(impulseResponseComplex);    // fftshift


562




564

  vector<t_real> absImpulseResponse(impulseResponseComplex.size());
566 for (int i = 0; i < impulseResponseComplex.size(); i++) { absImpulseResponse[i]
     = (impulseResponseComplex[i].real()); }
  double maxValue1 = *std::max_element(absImpulseResponse.begin(),
    absImpulseResponse.end());
568
  // Multiply it with the (symbolPeriod/samplingPeriod) to adjust the shape
570
  for (unsigned int i = 0; i < transferFunctionComplex.size(); i++) {
572   //impulseResponseAux[i] = (sqrt(2)*PI)* impulseResponseComplex[i].real() / (
    sqrt(static_cast<double>(transferFunctionComplex.size())));
    impulseResponseAux[i] = (1/ maxValue1)*(symbolPeriod/samplingPeriod)*
    absImpulseResponse[i] / (sqrt(static_cast<double>(transferFunctionComplex.size
    ())));
574 }
```

```
576
    impulseResponse = impulseResponseAux;
578 impulseResponseLength = (int)impulseResponse.size();

580 ofstream gaussTransferIMP("gaussTransferIMP.txt");
    for (unsigned int i = 0; i < impulseResponse.size(); i++) {
582   gaussTransferIMP << impulseResponse[i] << "\n";
    }
584 gaussTransferIMP.close();


586
}
```

Listing 8.7: filter.cpp code

Results

The following two graphs displays the result of **S6.sgn** and **S7.sgn**.

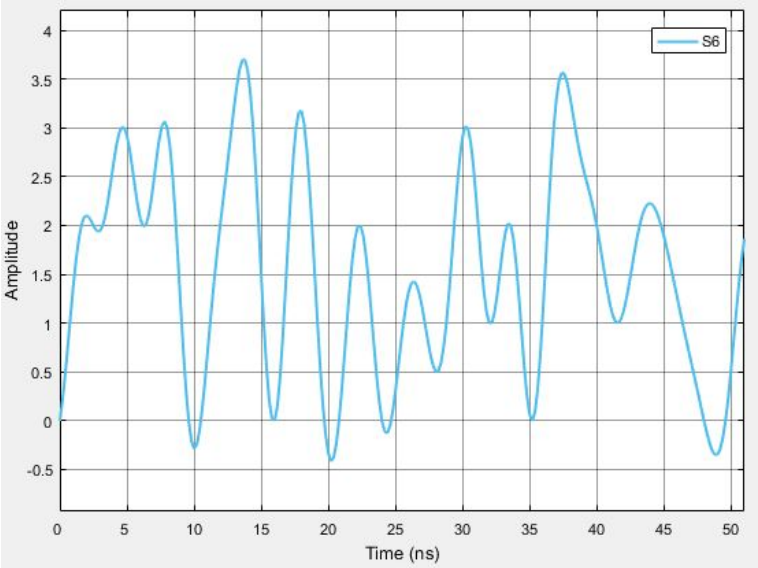


Figura 8.26: S6.sgn signal

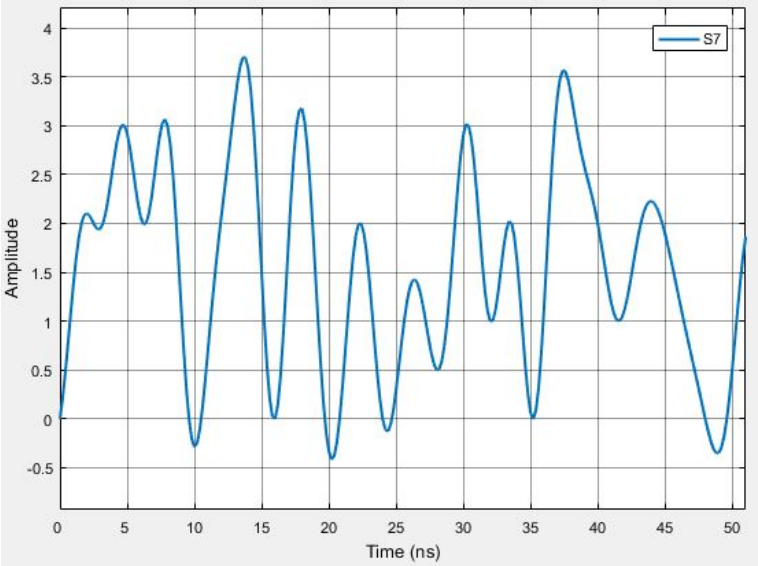


Figura 8.27: S7.sgn signal

Remarks

Capítulo 9

Building C++ projects without Visual Studio

This is a guide on how to build C++ projects without having Microsoft Visual Studio installed. All the necessary files will be available in the /msbuild folder on this repository.

9.1 Install Microsoft Visual C++ Build Tools

Run the file **visualcppbuildtools_full.exe** and follow all the setup instructions;

9.2 Adding Path to System Variables

Please follow this step-by-step tutorial carefully.

1. Open the **Control Panel**.
2. Select the option **System and Security**.
3. Select the option **System**.
4. Select **Advanced System Settings** on the menu on the left side of the window.
5. This should have opened another window. Click on **Environment variables**.
6. Check if there is a variable called 'Path' in the **System Variables** (bottom list).
7. If it doesn't exist, create a new variable by pressing **New** in **System Variables** (bottom list). Insert the name **Path** as the name of the variable and enter the following value **C:\Windows\Microsoft.Net\Framework\v4.0.30319**. Jump to step 10.
8. If it exists, click on the variable 'Path' and press **Edit**. This should open another window;
9. Click on **New** to add another value to this variable. Enter the following value: **C:\Windows\Microsoft.Net\Framework\v4.0.30319**.
10. Press **Ok** and you're done.

9.3 How to use MSBuild to build your projects

You are now able to build (compile and link) your C++ projects without having Visual Studio installed on your machine. To do this, please follow the instructions below:

1. Open the **Command Line** and navigate to your project folder (where the .vcxproj file is located).
2. Enter the command **msbuild <filename>**, where <filename> is your .vcxproj file. Ex: **msbuild project.vcxproj**;

After building the project, the .exe file should be generated automatically.

9.4 Known issues

9.4.1 Missing ucrtbased.dll

In order to solve this issue, please follow the instructions below:

1. Navigate to **C:\Program Files (x86)\Windows Kits\10\bin\x86\ucrt**
2. Copy the following file: **ucrtbased.dll**
3. Paste this file in the following folder: **C:\Windows\System32**
4. Paste this file in the following folder: **C:\Windows\SysWOW64**

Attention:you need to paste the file in BOTH of the folders above.