NetXPTO - LinkPlanner

Armando Nolasco Pinto

July 20, 2018

Contents

1 Case Studies **5**

1.1 BPSK Transmission System . . . 6
1.1.1 Theoretical Analysis . . . 6
1.1.2 Simulation Analysis . . . 7
1.1.3 Comparative Analysis . . . 11
Bibliography . . . 13
1.2 Quantum Random Number Generator . . . 14
1.2.1 Theoretical Analysis . . . 14
1.2.2 Simulation Analysis . . . 16
1.2.3 Experimental Analysis . . . 21
1.2.4 Open Issues . . . 23
Bibliography . . . 24
1.3 BB84 with Discrete Variables . . . 25
1.3.1 Protocol Analysis . . . 25
1.3.2 Simulation Analysis . . . 29
1.3.3 Open Issues . . . 40
Bibliography . . . 41
1.4 Arithmetic Encoding & Decoding . . . 42
1.4.1 Encoding Algorithm . . . 42
1.4.2 Decoding Algorithm . . . 44
1.4.3 Encoding and Decoding Simulation Results . . . 45
Bibliography . . . 46

2 Library **46**

2.1 ADC . . . 47
2.2 Add . . . 50
2.3 Balanced Beam Splitter . . . 51
2.4 Bit Error Rate . . . 52
Bibliography . . . 55
2.5 Binary Source . . . 56

2.6 Bit Decider . . . 60
2.7 Clock . . . 61
2.8 Clock_20171219 . . . 63
2.9 Coupler 2 by 2 . . . 66
2.10 Carrier Phase Compensation . . . 67
2.11 Decoder . . . 70
2.12 Discrete To Continuous Time . . . 72
2.13 DSP . . . 74
2.14 Electrical Signal Generator . . . 76
2.14.1 ContinuousWave . . . 76
2.15 Fork . . . 78
2.16 Gaussian Source . . . 79
2.17 MQAM Receiver . . . 81
2.18 IQ Modulator . . . 85
2.19 Local Oscillator . . . 87
2.20 Local Oscillator . . . 89
2.21 MQAM Mapper . . . 92
2.22 MQAM Transmitter . . . 95
2.23 Netxpto . . . 99
2.23.1 Version 20180118 . . . 101
2.24 Alice QKD . . . 102
2.25 Polarizer . . . 104
2.26 Probability Estimator . . . 105
2.27 Bob QKD . . . 108
2.28 Eve QKD . . . 109
2.29 Rotator Linear Polarizer . . . 110
2.30 Optical Switch . . . 112
2.31 Optical Hybrid . . . 113
2.32 Photodiode pair . . . 115
2.33 Photoelectron Generator . . . 118
Bibliography . . . 123
2.34 Pulse Shaper . . . 124
2.35 Quantizer . . . 126
2.36 Resample . . . 129
2.37 Sampler . . . 131
2.38 SNR of the Photoelectron Generator . . . 133
Bibliography . . . 138
2.39 Sink . . . 139
2.40 White Noise . . . 140
2.41 Ideal Amplifier . . . 143
2.42 Arithmetic Encoder . . . 145
2.43 Arithmetic Decoder . . . 147

3 Mathlab Tools **149**
3.1 Generation of AWG Compatible Signals . . . 150
3.1.1 sgnToWfm.m . . . 150
3.1.2 sgnToWfm_20171121.m . . . 151
3.1.3 Loading a signal to the Tekatronix AWG70002A . . . 153

4 Algorithms **157**
4.1 Fast Fourier Transform . . . 158
4.2 Overlap-Save Method . . . 174
4.3 Filter . . . 196
4.4 Hilbert Transform . . . 204

5 Building C++ Projects Without Visual Studio **207**
5.1 Installing Microsoft Visual C++ Build Tools . . . 207
5.2 Adding Path To System Variables . . . 207
5.3 How To Use MSBuild To Build Your Projects . . . 208
5.4 Known Issues . . . 208
5.4.1 Missing ucrtbased.dll . . . 208

6 Git Helper **209**
6.1 Data Model . . . 209
6.2 Refs . . . 211
6.3 Tags . . . 211
6.4 Branch . . . 211
6.5 Heads . . . 211
6.6 Database Folders and Files . . . 211
6.6.1 Objects Folder . . . 211
6.6.2 Refs Folder . . . 212
6.7 Git Spaces . . . 212
6.8 Workspace . . . 213
6.9 Index . . . 213
6.10 Merge . . . 213
6.10.1 Fast-Forward Merge . . . 213
6.10.2 Resolve . . . 213
6.10.3 Recursive . . . 213
6.10.4 Octopus . . . 213
6.10.5 Ours . . . 213
6.10.6 Subtree . . . 213
6.10.7 Custom . . . 213
6.11 Commands . . . 213
6.11.1 Porcelain Commands . . . 213
6.11.2 Pluming Commands . . . 214
6.12 The Configuration Files . . . 215

6.13 Pack Files 215
6.14 Applications 215
6.14.1 Meld 215
6.14.2 GitKraken 215
6.15 Error Messages 215
6.15.1 Large files detected 215

7 Simulating VHDL Programs with GHDL **217**
7.1 Adding Path To System Variables 217
7.2 Using GHDL To Simulate VHDL Programs 218
7.2.1 Requirements 218
7.2.2 Option 1 218
7.2.3 Option 2 218
7.2.4 Simulation Output 218

Chapter 1

Case Studies

1.1 BPSK Transmission System

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

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

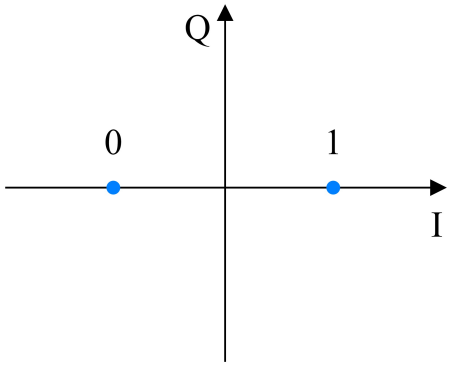


Figure 1.1: BPSK symbol constellation.

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

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

1.1.1 Theoretical Analysis

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

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

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

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

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

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

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

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

The BER is calculated in the following manner,

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

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

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

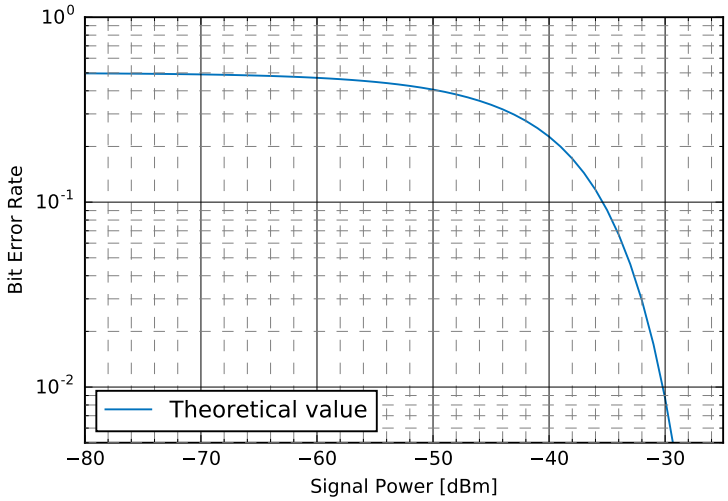


Figure 1.2: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

1.1.2 Simulation Analysis

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

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

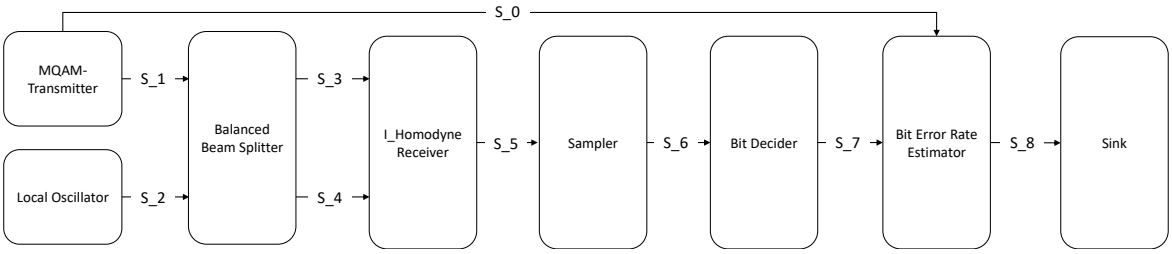


Figure 1.3: Overview of the BPSK system being simulated.

Required files

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

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

System Input Parameters

This system takes into account the following input parameters:

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

Inputs

This system takes no inputs.

Outputs

This system outputs the following objects:

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

Bit Error Rate - Simulation Results

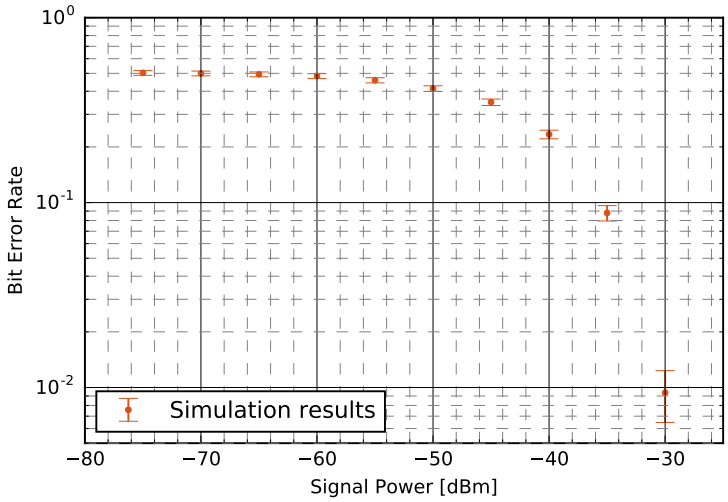


Figure 1.4: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

1.1.3 Comparative Analysis

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

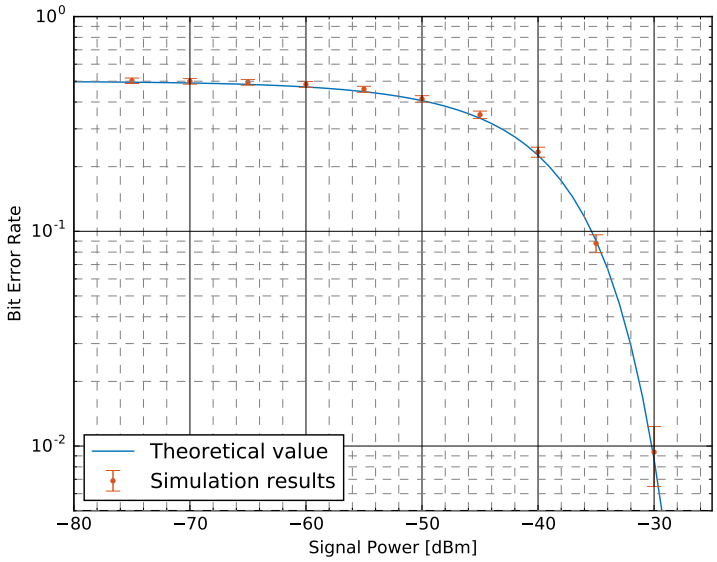


Figure 1.5: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 2.4

References

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

1.2 Quantum Random Number Generator

Students Name	:	Mariana Ramos (12/01/2018 - 11/04/2018)
Goal	:	Simulate and implement an experimental setup of a Quantum Random Number Generator.
Directory	:	sdf/quantum_random_number_generator.

True random numbers are indispensable in the field of cryptography [1]. 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 [2].

In this chapter, it is presented the theoretical, the simulation and the experimental analysis of a quantum random generator based on the use of single photons linearly polarized at $45°$.

1.2.1 Theoretical Analysis

One of the optical processes available as a source of randomness is the splitting of a polarized single photon beam. The principle of operation of the random generator is shown in figure 1.6. Each individual photon coming from the source is linearly polarized at $45°$ and has equal probability of be found in the horizontal (H) or in the vertical (V) output of the PBS. Quantum theory estimates for both cases that the individual choices are truly random, independent one from each other , and with a probability of $1/2$.

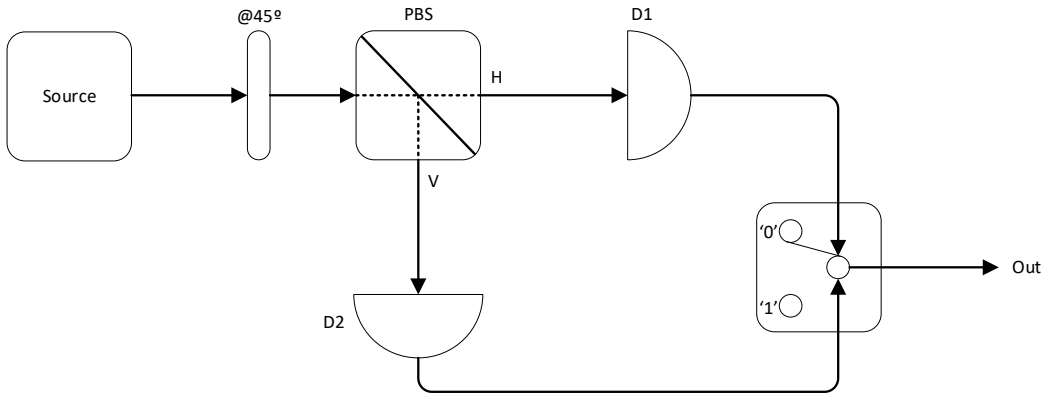


Figure 1.6: Source of randomness with a polarization beam splitter PBS where the incoming light is linearly polarized at $45°$ with respect to the PBS. Figure adapted from [2].

From a classical approach, the information is stored as binary bits that can take the logical value '0' or '1'. From a quantum approach, the information can be stored in quantum

bits or qubits for short. As a consequence of the superposition principle of quantum mechanics, qubits can not only represent the pure '0' or '1' states, but they can also represent a superposition of both. This way, qubits are governed by a quantum wave function ψ. Lets use the Dirac notation to represent the general state of the qubit:

$$|\psi\rangle = C_0|0\rangle + C_1|1\rangle, \tag{1.6}$$

and the normalization condition of $|\psi\rangle$ requires that $|C_0|^2 + |C_1|^2 = 1$. This way, the relative proportion of each of the binary states on a qubit is governed by the amplitude coefficients C_0 and C_1. In the present example, we consider a linear polarization in which the two possible states are orthogonal, such that: $\langle 0|1\rangle = 0$. We define the $|0\rangle$ and $|1\rangle$ states to correspond to the horizontal and vertical polarization states, respectively:

$$\begin{aligned} |\psi\rangle &= C_0|0\rangle + C_1|1\rangle & (1.7)\\ &= C_0|0^\circ\rangle + C_1|90^\circ\rangle. & (1.8) \end{aligned}$$

Amplitude coefficients C_0 and C_1 store the quantum information. Therefore, if one makes a measurement, the result will be '0' with probability $|C_0|^2$ or '1' with probability $|C_1|^2$. Moreover, the state of a single photon can be also described by a wave function as a column vector:

$$|\psi\rangle = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}, \tag{1.9}$$

which will be used in simulation analysis.

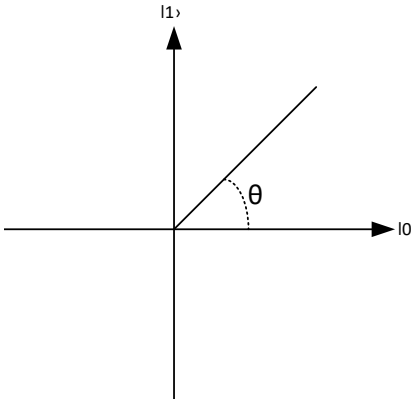


Figure 1.7: Representation of polarization states of a qubit in a bi-dimensional space.

As one can see in figure 1.7 the amplitude coefficients can be written as a function of θ:

$$\begin{aligned} C_0 &= cos(\theta) & (1.10)\\ C_1 &= sin(\theta). & (1.11) \end{aligned}$$

According with the setup presented in figure 1.6 and considering the polarization angle $\theta = 45°$, the single photon has the probability of reach **D1** and outputs a "0" is equals to $|cos(\theta)|^2$ and the probability of reach **D2** and outputs a "1" is equals to $|sin(\theta)|^2$, which in the case $\theta = 45°$ both have the same value equals to 0.5.

1.2.2 Simulation Analysis

The simulation diagram of the setup described in the previous section is presented in figure 1.8. The linear polarizer has an input control signal (S1) which allows to change the rotation angle. Nevertheless, the only purpose is to generate a time and amplitude continuous real signal with the value of the rotation angle in degrees. In addition, the photons are generated by single photon source block at a rate defined by the clock rate. At the end of the simulation there is a circuit decision block which will outputs a binary signal with value "0" if the detector at the end of the horizontal path clicks or "1" if the detector at the end of the vertical path clicks.

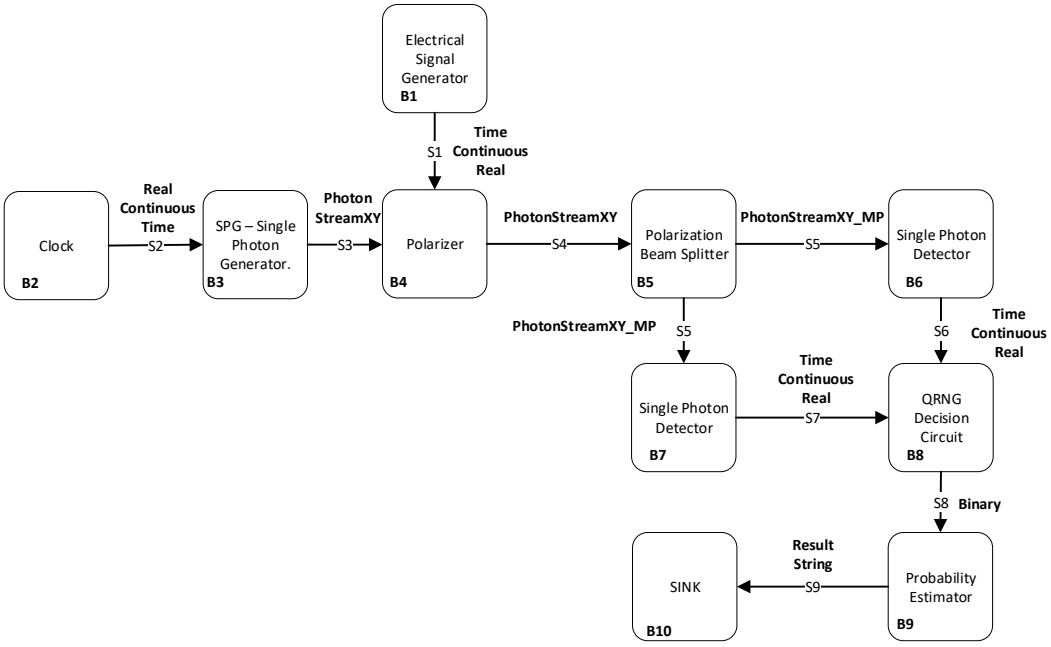


Figure 1.8: Block diagram of the simulation of a Quantum Random Generator.

In table 1.1 are presented the input parameters of the system.

Table 1.1: System Input Parameters

Parameter	Default Value
RateOfPhotons	1e6
NumberOfSamplesPerSymbol	16
PolarizerAngle	45.0

In table 1.2 are presented the system signals to implement the simulation presented in figure 1.8.

Table 1.2: 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 1.3 presents the header files used to implement the simulation as well as the specific parameters that should be set in each block. Finally, table 1.4 presents the source files.

Table 1.3: Header Files

File name	Description	Status
netxpto_20180118.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		✓

Table 1.4: Source Files

File name	Description	Status
netxpto_20180118.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		✓

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 angles of polarization shown in figure 1.9 with different error margin's values since the expected probability changes depending on the polarization angle.

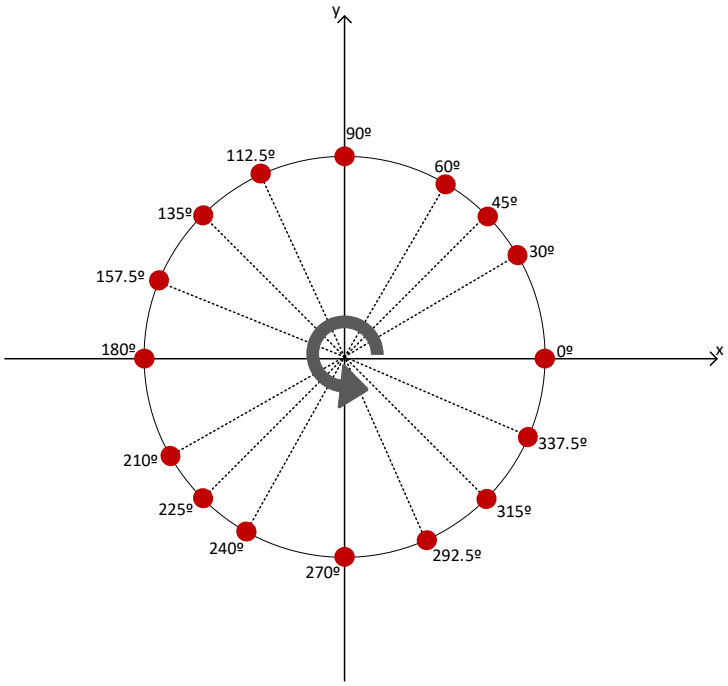


Figure 1.9: Angles used to perform the qrng simulation for $N = 1 \times 10^6$ samples.

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.

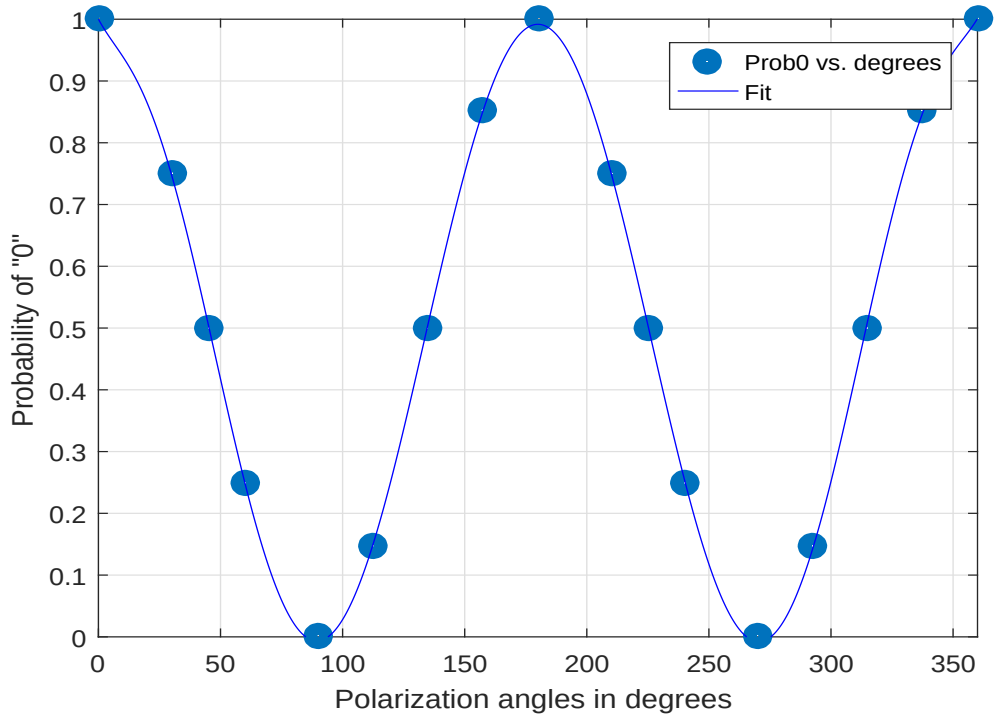


Figure 1.10: Probability of outputs a number "0" depending on the polarization angle.

Figure 1.10 shows the probability of a single photons reaches the detector placed on Horizontal axis depending on the polarization angle of the photon, and this way the output number is "0". The following table shows the goodness of the fit:

SSE:	0.0004785
R-square:	0.9998
Adjusted R-square:	0.9995
RMSE:	0.007734

On the other hand, figure 1.11 shows the probability of a single photon reaches the detector placed on Vertical component of the polarization beam splitter, and this way the output number is "1". As we can see in the figures the two detectors have complementary probabilities, i.e the summation of both values must be equals to 1. One can see that "Probability of 1" behaves almost like a sine function and "Probability of 0" behaves almost like a cosine function with a variable angle.

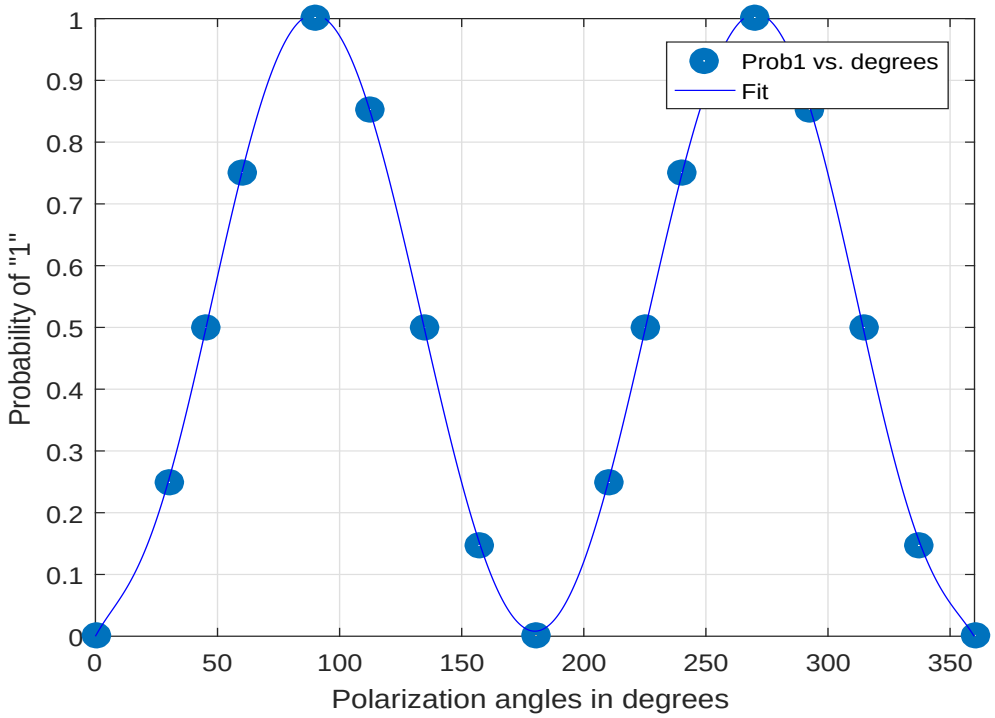


Figure 1.11: Probability of outputs a number "1" depending on the polarization angle.

The goodness of the fit presented in figure 1.11 is shown in the following table:

SSE:	0.0004785
R-square:	0.9998
Adjusted R-square:	0.9995
RMSE:	0.007734

The goodness of the fit is evaluated based on four parameters:

1. The sum of squares due to error (SSE), which measures the total deviation between the fit values and the values that the simulation outputs. This value is calculated from the expression

$$\text{SSE} = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2.$$

A value of SSE closer to 0 means that the model has a small random error component.

2. The R-square measures how good the fit in explaining the data.

$$\text{R-square} = 1 - \frac{\text{SSE}}{\text{SST}},$$

where,

$$\text{SST} = \sum_{i=1}^{n} w_i (y_i - \bar{y}_i)^2.$$

R-square can take a value between 0 and 1. If the value is closer to 1, it means that the fit better explains the total variation in the data around the average.

3. Degrees of freedom adjusted R-square uses the R-square and adjusts it based on the number of degrees of freedom.

$$\text{adjusted R-square} = 1 - \frac{\text{SSE}(n-1)}{\text{SST}(v)},$$

where,

$$v = n - m,$$

where n is the number of values in test and m is the number of fitted coefficients estimated from the values in test. A value of adjusted R-square close to 1 is a indicative factor of a good fit.

4. The root mean square error (RMSE) is also a fit standard error and it can be calculated from:

$$\text{RMSE} = \sqrt{\text{MSE}},$$

where,

$$\text{MSE} = \frac{\text{SSE}}{v}.$$

1.2.3 Experimental Analysis

In order to have a real experimental quantum random number generator, a setup shown in figure 1.12 was built in the lab. To simulate a single photon source we have a CW-Pump laser with 1550 nm wavelength followed by an interferometer Mach-Zenhder in order to have a pulsed beam. The interferometer has an input signal given by a Pulse Pattern Generator. This device also gives a clock signal for the Single Photon Detector (APD-Avalanche Photodiode) which sets the time during which the window of the detector is open. After the MZM there is a Variable Optical Attenuator (VOA) which reduces the amplitude of each pulse until the probability of one photon per pulse is achieved. Next, there is a polarizer controller followed by a Linear Polarized, which is set at 45°, then a Polarization Beam Splitter (PBS) and finally, one detector at the end of each output of the PBS. The output signals from the detector will be received by a Processing Unit. Regarding to acquired the output of the detectors, there is an oscilloscope capable of record 1×10^6 samples.

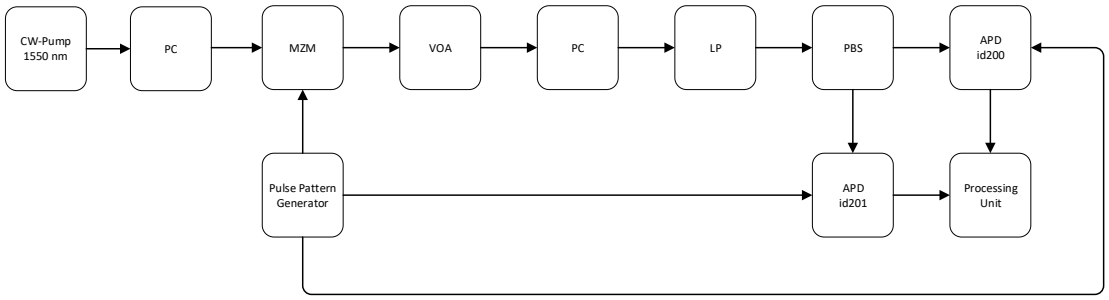


Figure 1.12: Experimental setup to implement a quantum random number generator.

IDQuantique 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.

Since we do not have a single photon source, we must use the Poisson Statistics in order to calculate the best value for a mean of photons per pulse. A weak laser pulse follows a Poissonian Statistics[3]:

$$S_n = e^{-\mu}\frac{\mu^n}{n!}, \tag{1.12}$$

where μ is the average photon number. In addition, the probability of an optical pulse carries one photon at least is:

$$P = 1 - S_0 = 1 - e^{-\mu}. \tag{1.13}$$

On the other hand, the probability of a detector clicks is:

$$P_{click} = P_{det} + P_{dc} + P_{det}P_{dc}, \tag{1.14}$$

where P_{det} is the probability of the detector clicks due a photon which cross its window and P_{dc} is the probability of dark counts. Considering the detector efficiency η_D, the probability of the detector clicks due to a photon is:

$$P_{det} = 1 - e^{-\eta_D\mu}. \tag{1.15}$$

The probability of dark counts is calculated as a ratio between the frequency counts and the trigger frequency when no laser is connected to the detector. Nevertheless, the detector click frequency is

$$f_{click} = f_{trigger}P_{click} \longrightarrow P_{click} = \frac{f_{click}}{f_{trigger}}. \tag{1.16}$$

This way the mean average photon number can be calculate using the following equation:

$$\mu = -\frac{1}{\eta_D}\ln\left[1 - \frac{1}{1-P_{dc}}\left(\frac{f_{click}}{f_{trigger}} - P_{dc}\right)\right] \tag{1.17}$$

1.2.4 Open Issues

- Experimental Implementation.
- Random number validation/standardization.

References

[1] GE Katsoprinakis et al. "Quantum random number generator based on spin noise". In: *Physical Review A* 77.5 (2008), p. 054101.

[2] Thomas Jennewein et al. "A fast and compact quantum random number generator". In: *Review of Scientific Instruments* 71.4 (2000), pp. 1675–1680. DOI: 10.1063/1.1150518.

[3] Mark Fox. *Quantum Optics, an Introdution*. Oxford, University Press, 2006.

1.3 BB84 with Discrete Variables

Students Name	:	Mariana Ramos (7/11/2017 - 9/4/2018) Kevin Filipe (7/11/2017 - 10/11/2017)
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.

1.3.1 Protocol Analysis

Students Name	:	Kevin Filipe (7/11/2017 - 10/11/2017)
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 1.13.

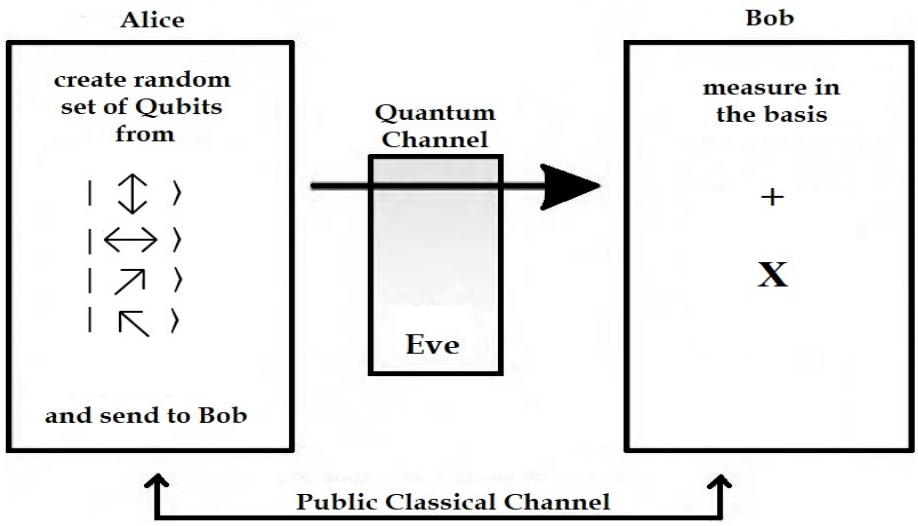


Figure 1.13: Basic QKD Model. Alice and Bob are connected by 2 communication channels, a public quantum channel and a authenticated classical channel, with an eavesdropper, Eve (figure adapted from [2]).

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

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

The protocol requires the following parameter and it is implemented with the following steps:

Table 1.5: Initial Parameters.

Parameter	**Description**
$M \times N$	Scrambling Matrix M by N
k	Number of revealed bits for BER calculation
α	Confidence level
A	B

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

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

$$\begin{aligned} R_{A2} &= \{0,0,1,0,1,1,1,0,1,1,1,0,1,0,0,0,1,0,1,0\} \\ &= \{+,+,\times,+,\times,\times,\times,+,\times,\times,\times,+,\times,+,+,+,\times,+,\times,+\} \end{aligned}$$

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.

$$\begin{aligned} R_B &= \{0,1,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0\} \\ &= \{+,\times,\times,\times,+,\times,+,+,\times,\times,+,+,\times,\times,+,+,\times,\times,+,+\} \end{aligned}$$

4. Bob performs the incoming photon states measurement, M_B, with its generated random basis, R_B. If the two photon detectors don't click, means the bit was lost during transference due to attenuation. If both photon detectors click, a false positive was detected. In the measurements, M_B, the no-click in both detectors is represented by a -1 and the false positives to -2. The measurements done in rectilinear or diagonal basis are represented by 0 or 1, respectively. This is represented 1.14

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

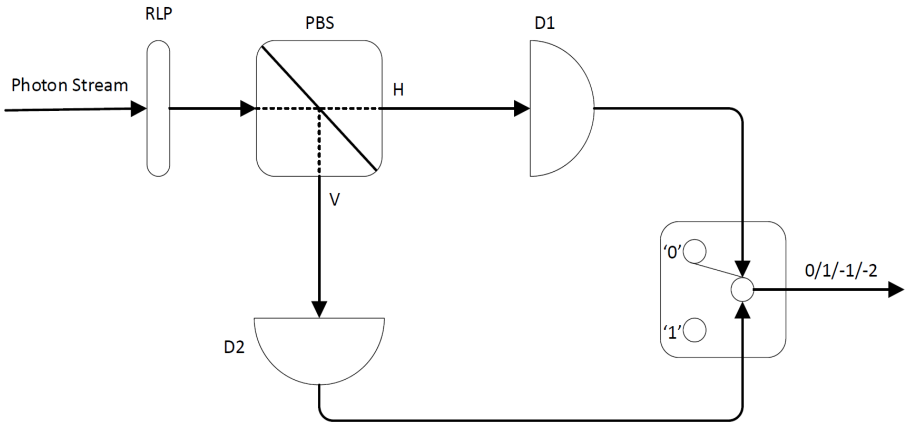


Figure 1.14: Single-Photon Detection block with false-positives, -2, and attenuation, -1, detection depending on D1 and D2 output.

5. After the measurement, Bob sends to Alice, using the classical channel, the used basis values, R_B with the attenuation, -1, and false positives,-2.

6. Alice performs a modified negated XOR, generating a sequence that detects when the same basis she used B_{AB}.

R_{A2}	0	0	1	0	1	1	1	0	1	1	1	0	1	0	0	0	1	0	1	0
R_B	0	1	1	1	-1	1	0	0	-2	1	0	0	-2	1	0	0	1	-1	0	0
B_{AB}	1	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	1	0	0	1

7. Alice sends the B_{AB} sequence to Bob, in which he can correlate with, M_B, and deduce the key K_{AB}.

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

8. Alice then by having knowledge of R_{A2} and B_{AB} performs a scrambling algorithm over the deduced key. It is generated a matrix $M \times N$, according to the input parameter. Assuming a scrambling matrix of 3x4, 1.6. And being the scramble key represented as KS_{AB}

$$KS_B = \{0, 0, 0, 1, 1, 1, 0, 0, 1, 1\}$$

9. Bob uses the same algorithm as Alice and scrambles his key.

Table 1.6: Scrambling matrix

0	1	0	1
0	1	0	1
0	1	-	-

10. Bob then reveals a fixed number of his key to Alice. This number is also an input parameter value, k. With this the Quantum Bit Error Rate (QBER).

To determine the QBER, it is necessary to know the confidence interval parameter, α and the QBER limit, in which states the maximum allowed QBER by the user. Then to verify if the channel is reliable or not, the flowchart presented in figure 1.15.

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

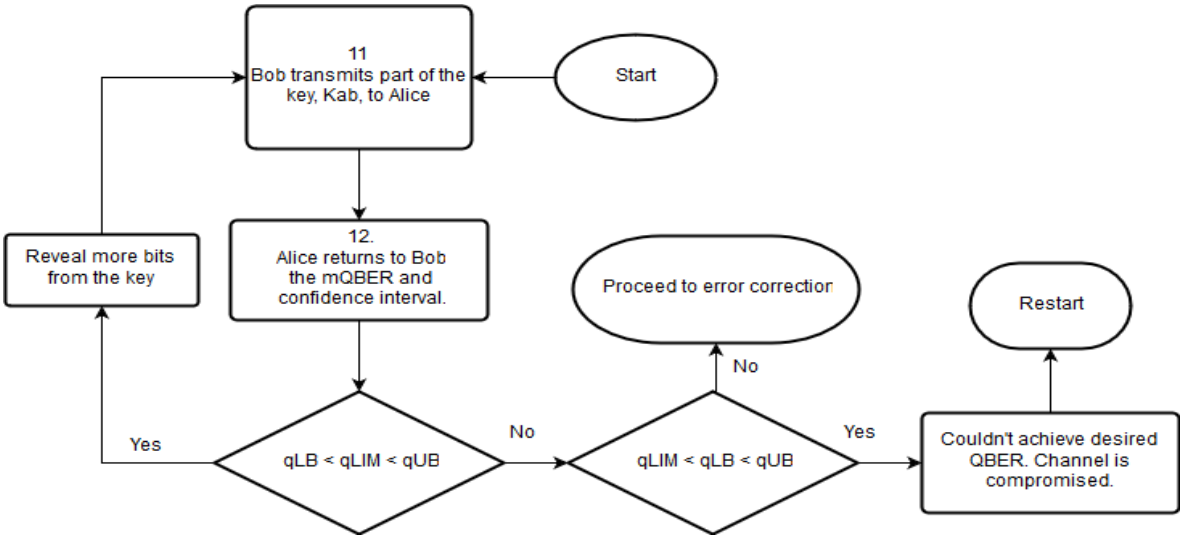


Figure 1.15: Flowchart to determine if the channel is reliable or not.

1.3.2 Simulation Analysis

Students Name	:	Mariana Ramos (7/11/2017 - 9/4/2018)
Goal	:	Perform a simulation of BB84 communication protocol.

In this sub section the simulation setup implementation will be described in order to implement the BB84 protocol. In figure 1.16 a top level diagram is presented. Then it will be presented the block diagram of the transmitter block (Alice) in figure 1.17 and the receiver block (Bob) in figure 1.18. In a first approach, we do not consider the existence of eavesdropper.

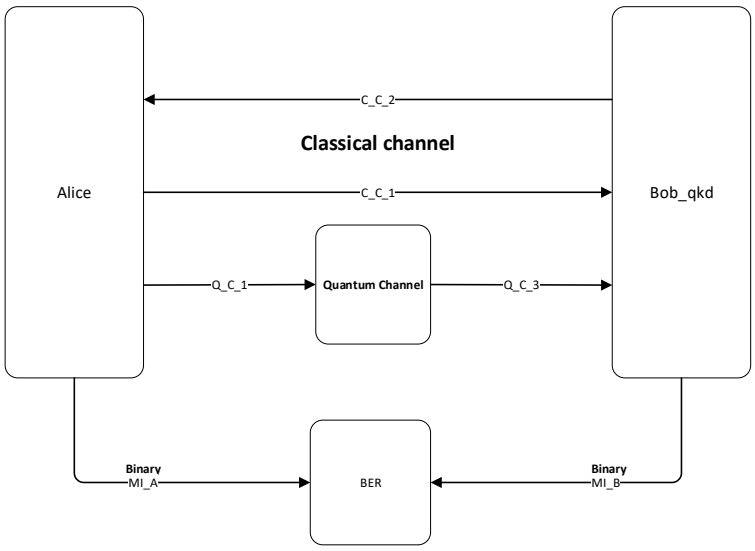


Figure 1.16: Simulation diagram at Alice's side

Figure 1.16 presents the top level diagram of our simulation. The setup contains two parties Alice and Bob, where the communication between them is done throughout two authenticated classical channels and one public quantum channel. In a first approach we will perform the simulation without eavesdropper presence. Furthermore, for bit error rate calculation between Alice and Bob.

In figure 1.17 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

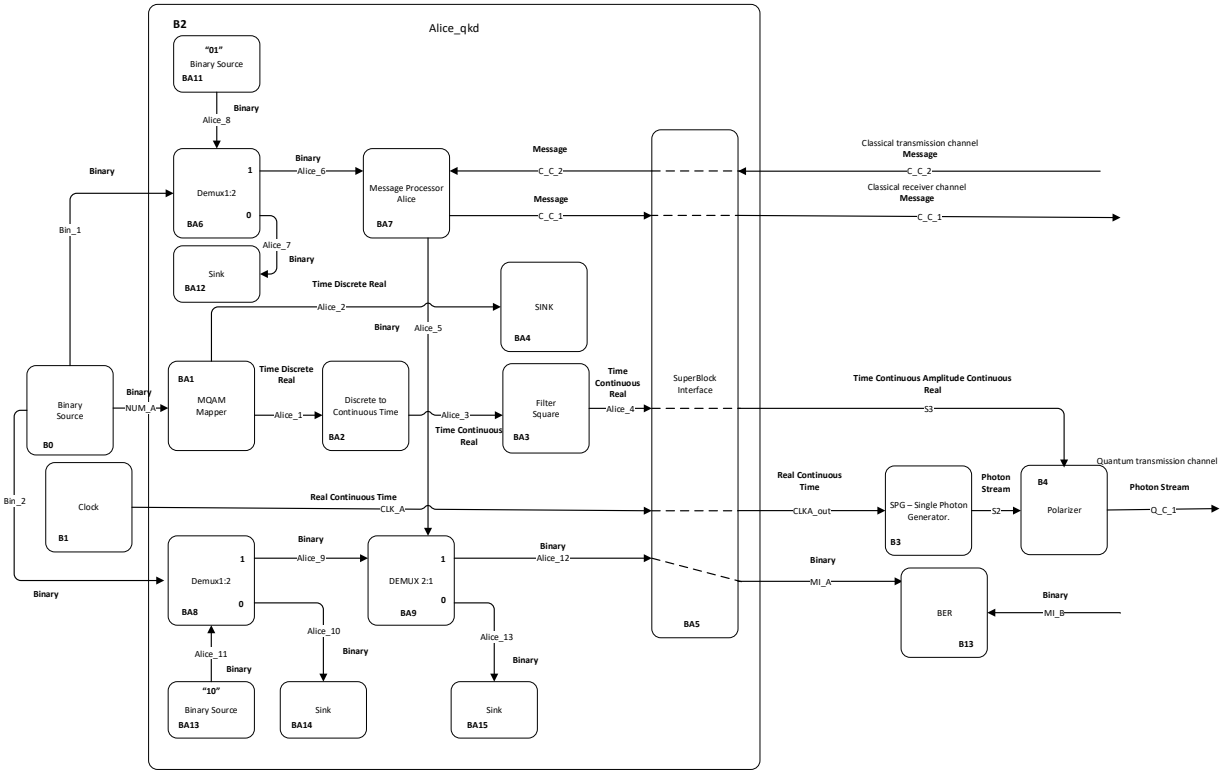


Figure 1.17: Simulation diagram at Alice's side

to Bob. Finally, Processor block will also send a real continuous time signal to single photon generator, in order to generate photons according to this signal, and finally this block also sends to the polarizer a real discrete signal in order to inform the polarizer which basis it should use. Therefore, she has two more blocks for quantum tasks: the single photon generator and the polarizer block which is responsible to encode the photons generated from the previous block and send them throughout a quantum channel from Alice to Bob.

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

In figure 1.17 one can observe a block diagram of the transmitter. As it is shown in the figure, the transmitter must have one block for random number generation (binary source) 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. This block has three outputs which will be inputs for the super block Alice. Furthermore, Alice block is responsible for all logical operations: random single photons state values generation, receive and send messages to the receiver Bob by using the classical channels, binary output for mutual information calculations. Each block of the super block is described in Library chapter. Finally, Alice block will also send a real continuous time signal to single photon generator (clock sets the rate oh photons generation), in order to generate photons polarized in the horizontal axis by default. Therefore, the transmitter has one more block, the polarizer

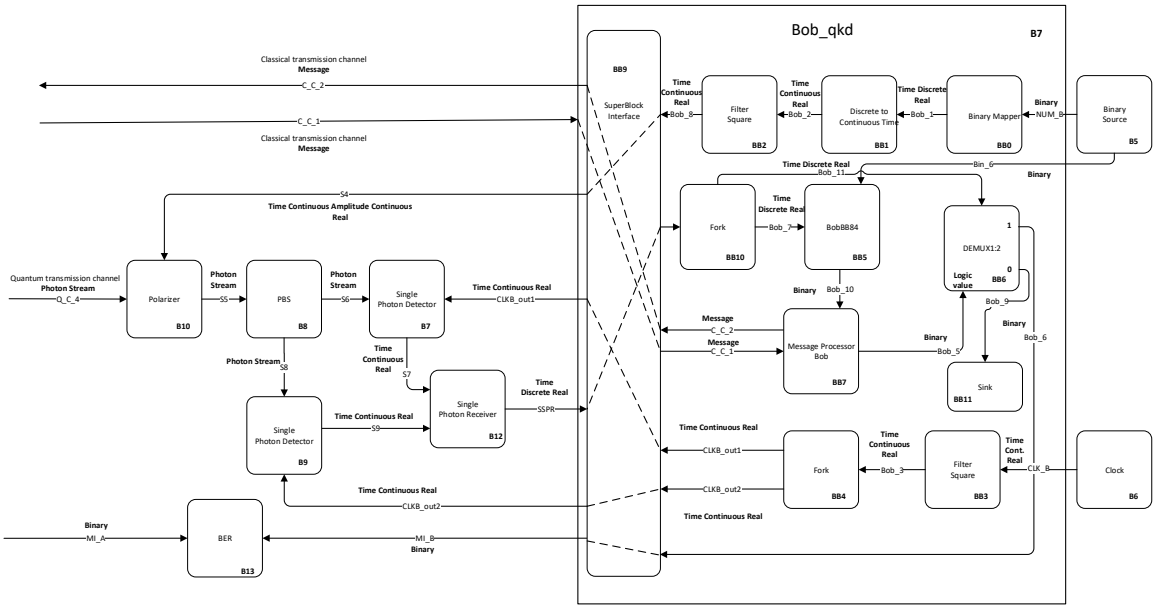


Figure 1.18: Simulation diagram at Bob's side

block, which is responsible to encode the photons generated from the previous block and send them throughout a quantum channel from Alice to Bob.

In figure 1.18 one can see a block diagram of the simulation for receiver (Bob). The receiver 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 transmitter, the receiver has the Bob block responsible for receive and send messages through the classical channel, receive single photons values detection from the single photon detectors, provides a clock signal to the detectors and send binary values for mutual information calculation. Furthermore, the receiver has two blocks for single photon detection (one for horizontal detection and other for vertical detection) which receives from Bob block a real continuous time signal which will set the detection window for the detector and outputs for Bob block the result value for detection. In addition, there is a polarizer which receives from Bob block a time continuous real signal which provides information about the rotation angle. If the basis chosen by Bob is the diagonal basis he sends "$45°$", otherwise sends "$0°$". The polarization beam splitter divides the input photon stream in horizontal component and vertical component.

Table 1.7: System Signals

Signal name	Signal type
NUM_A, NUM_B, Bin_1, Bin_2, Bin_6	Binary
MI_A, MI_B	Binary
CLK_A, CLK_B	TimeContinuousAmplitudeContinuous
CLK_A_out, CLKB_out1, CLKB_out2	TimeContinuousAmplitudeContinuous
S2, S5, S6, S8	PhotonStreamXY
S3, S7, S9	TimeContinuousAmplitudeDiscreteReal
S4	TimeContinuousAmplitudeContinuousReal
C_C_1, C_C_3	Messages
C_C_6, C_C_4	Messages
Q_C_1, Q_C_4	PhotonStreamXY

Table 1.10 presents the system signals as well as them type.

Table 1.8: System Input Parameters

Parameter	Default Value	Description
RateOfPhotons	1K	
iqAmplitudeValues	{-45,0},{0,0},{45,0},{90,0}	
NumberOfSamplesPerSylbom	16	
DetectorWindowTimeOpen	0.2	smaller than 1 ms
DetectorPulseDelay	0.7	in units of ms
DetectorProbabilityDarkCount	0.0	
RotationAngle	0.0	
ElevationAngle	0.0	

Table 1.9: Header Files

File name	Description	Status
netxpto_20180118.h		✓
alice_qkd_20180409.h		✓
binary_source_20180118.h		✓
bob_qkd_20180409.h		✓
clock_20171219.h		✓
discrete_to_continuous_time_20180118.h		✓
m_qam_mapper_20180118.h		✓
polarization_beam_splitter_20180109.h		✓
polarization_rotator_20180113.h		✓
pulse_shaper_20180111.h		✓
single_photon_detector_20180206.h		✓
single_photon_receiver_20180303.h		✓
SOP_modulator_20180319.h		✓
coincidence_detector_20180206.h		✓
single_photon_source_20171218.h		✓
sink_20180118.h		✓
super_block_interface_20180118.h		✓
message_processor_alice_20180205.h		✓
demux_1_2_20180205.h		✓
binary_mapper_20180205.h		✓
bobBB84_20180221.h		✓
message_processor_bob_20180221.h		✓
sampler_20171119.h		✓
optical_attenuator_20180304.h		✓
fork_20180112.h		✓

Table 1.10: Source Files

File name	Description	Status
netxpto_20180118.cpp		✓
bb84_with_discrete_variables_sdf.cpp		✓
alice_qkd_20180409.cpp		✓
binary_source_20180118.cpp		✓
bob_qkd_20180409.cpp		✓
clock_20171219.cpp		✓
discrete_to_continuous_time_20180118.cpp		✓
m_qam_mapper_20180118.cpp		✓
polarization_beam_splitter_20180109.cpp		✓
polarization_rotator_20180113.cpp		✓
pulse_shaper_20180111.cpp		✓
single_photon_detector_20180206.cpp		✓
single_photon_receiver_20180303.cpp		✓
SOP_modulator_20180319.cpp		✓
coincidence_detector_20180206.cpp		✓
single_photon_source_20171218.cpp		✓
sink_20180118.cpp		✓
super_block_interface_20180118.cpp		✓
message_processor_alice_20180205.cpp		✓
demux_1_2_20180205.cpp		✓
binary_mapper_20180205.cpp		✓
bobBB84_20180221.cpp		✓
message_processor_bob_20180221.cpp		✓
sampler_20171119.cpp		✓
optical_attenuator_20180304.cpp		✓
fork_20180112.cpp		✓

Simulation Results

Figure 1.19 represents the block diagram of the first simulation performed between Alice and Bob. This simulation intends to simulate the communication protocol between Alice and Bob until they do the Basis Reconciliation. At this time, it is not taken into account any attack from an eavesdropper. However, as one can learn from theoretical protocol analysis, the attenuation due the fiber losses, dark counts probabilities from single photon detectors and the SOP drift over the quantum channel are all taken into account.

Alice starts by sending a sequence of photons to Bob, and then he measures the photons according to random basis randomly generated by his binary source. After that, he follows the protocol described above until Alice sends to him a string of '0' and '1' where '0' means that both used different basis and '1' means that they used the same basis. Therefore, Alice

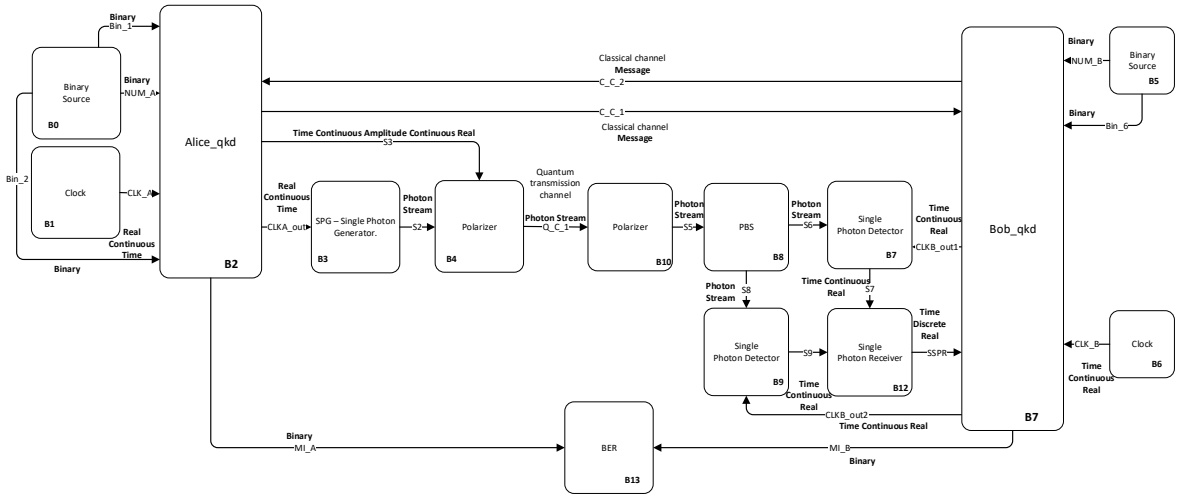


Figure 1.19: Diagram block of simulation performed between Alice and Bob until Basis Reconciliation.

and Bob outputs a binary signal "MI_A" and "MI_B", respectively. In case of no errors occurred in the quantum channel, these signals should be equal in order to both have the same sequence of bits. Furthermore, QBER between the two sequences should be 0. This way, Alice can encode messages using these keys and Bob will be capable of decrypt the message using these symmetric keys. When errors are introduced in quantum channel QBER value will increase as we can see later.

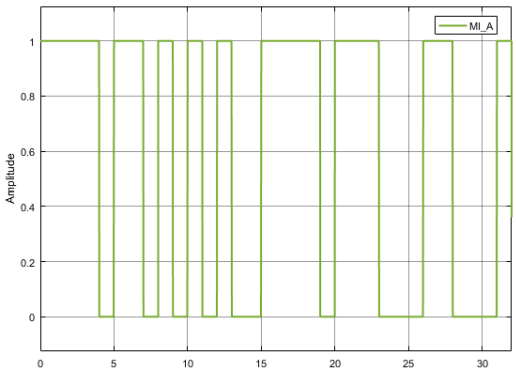


Figure 1.20: MI_A signal.

Figure 1.20 and figure 1.21 represent the sequence of bits which will be used by Alice to encode the messages and the sequence of bits used by Bob to decode the message when no errors in quantum channel are taken into account, respectively. As one can see the two

signals are equal which meets the expected result. In this way, the first step of the protocol has been achieved.

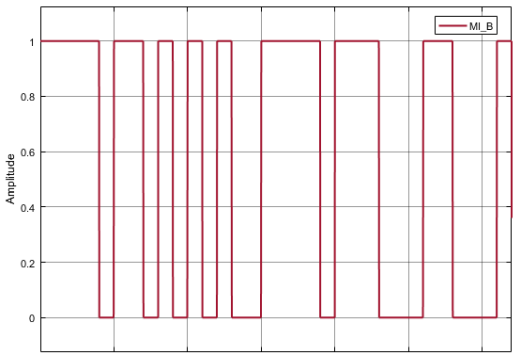


Figure 1.21: MI_B signal.

As one can see in figure **??** a block which calculates QBER is connected to Alice and Bob. This block calculates the QBER between the measurements that Bob performed with the same basis as Alice, based on method described in [3]. Thus, as expected, the QBER is 0% when no errors are taken into account.

Next, some errors due the changes in state of polarization of the single photons transmitted between Alice and Bob were added. This way, a polarization rotator in the middle of the quantum channel was added, which is controlled by a SOP modulator block as it is shown in figure 1.22 with modelled with deterministic [4] and stochastic [5] methods. Additional information about the blocks presented in this quantum channel can be found in library chapter.

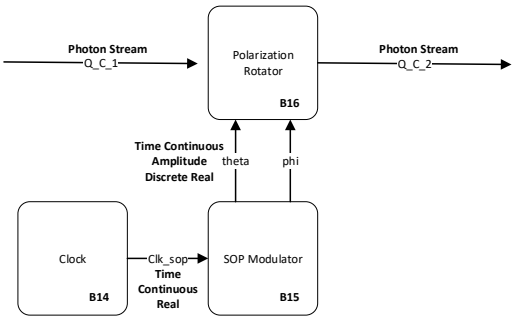


Figure 1.22: Quantum channel diagram.

Now, it is important to calculate the QBER as a function of the rotation angle θ. In order to do that, it was simulated a deterministic SOP modulation, in which the θ angle varies over the time. In figure 1.24 is presented the variation in the value of QBER with respect with theta changes from $0°$ to $45°$. Theoretically, QBER corresponds to the probability of errors in

the channel. Which means that in practice this probability corresponds to the probability of a photon following the wrong path in the polarization beam splitter immediately before the detection circuit.

Figure 1.23: Representation of two orthogonal states rotated by an angle θ.

Figure 1.23 presents the graphical representation of two orthogonal states rotated by an angle θ. This rotation is induced by the SOP modulator block which selects a deterministic θ and ϕ angles that do not change over the time. This same rotation is applied for all sequential samples. From figure 1.23 the theoretical QBER can be calculated using the following equation:

$$QBER = P(0)P(1|0) + P(1)P(0|1). \tag{1.18}$$

Since we have been using a polarization beam splitter 50:50,

$$P(0) = P(1) = \frac{1}{2}.$$

This way,

$$QBER = \frac{1}{2}sin^2(\theta) + \frac{1}{2}sin^2(\theta) \tag{1.19}$$

$$QBER = sin^2(\theta). \tag{1.20}$$

In figure 1.24 are represented two curves: qber calculated from simulated data and qber calculated using theoretical model from equation 1.19. Furthermore, the cross correlation coefficient between the two signals was calculated using a function from MATLAB *xcorr(x,y,'coeff')* which the result is 99.92%. From that, we can conclude that the QBER calculated from simulated data follows the theoretical curve with high correlation. Nevertheless, the error bars presented in figure 1.24 were calculated based on a confidence interval of 95%.

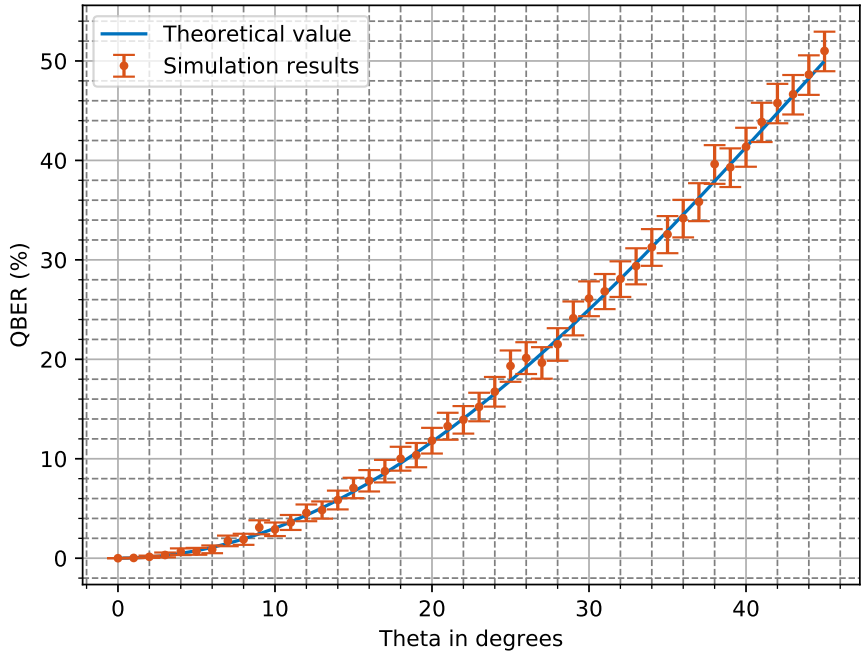


Figure 1.24: Qber evolution in relation with deterministic SOP drift.

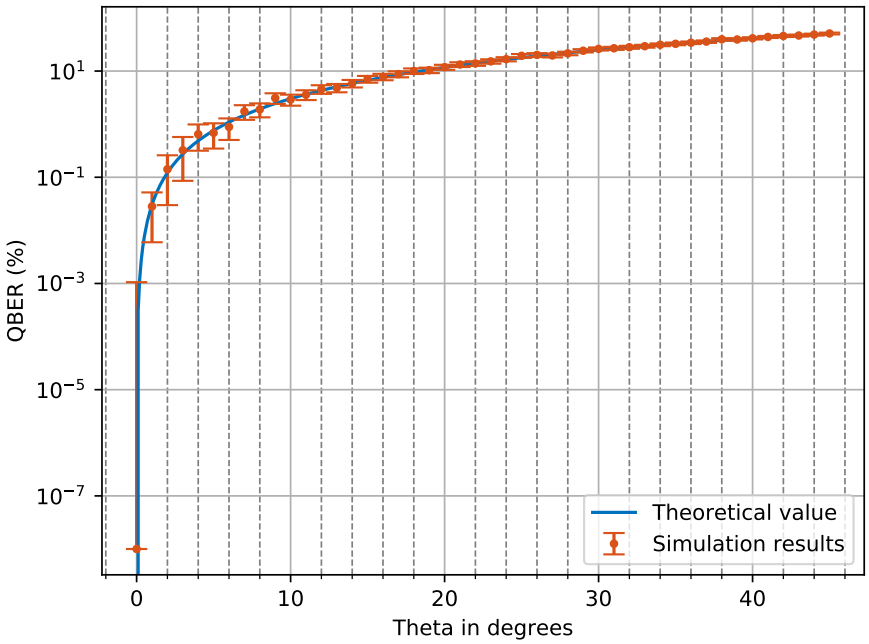


Figure 1.25: Qber evolution in relation with deterministic SOP drift in log scale.

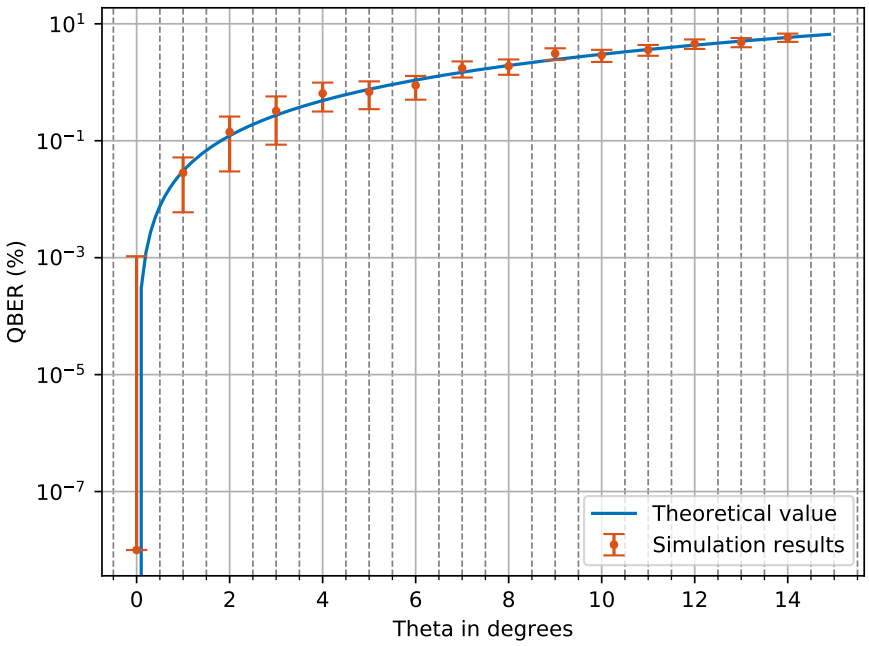


Figure 1.26: Qber evolution in relation with deterministic SOP drift scaled.

1.3.3 Open Issues

1. Implementation of the scrambling algorithm in order to spread the errors.
2. Implementation of the control system for polarization rotations.
3. Implementation of a QBER estimation protocol.
4. Implementation of the cascade for error correction.
5. Implementation of the output which represents the final key that is built.
6. Introduce EVE in simulation as shown in figure 1.27.

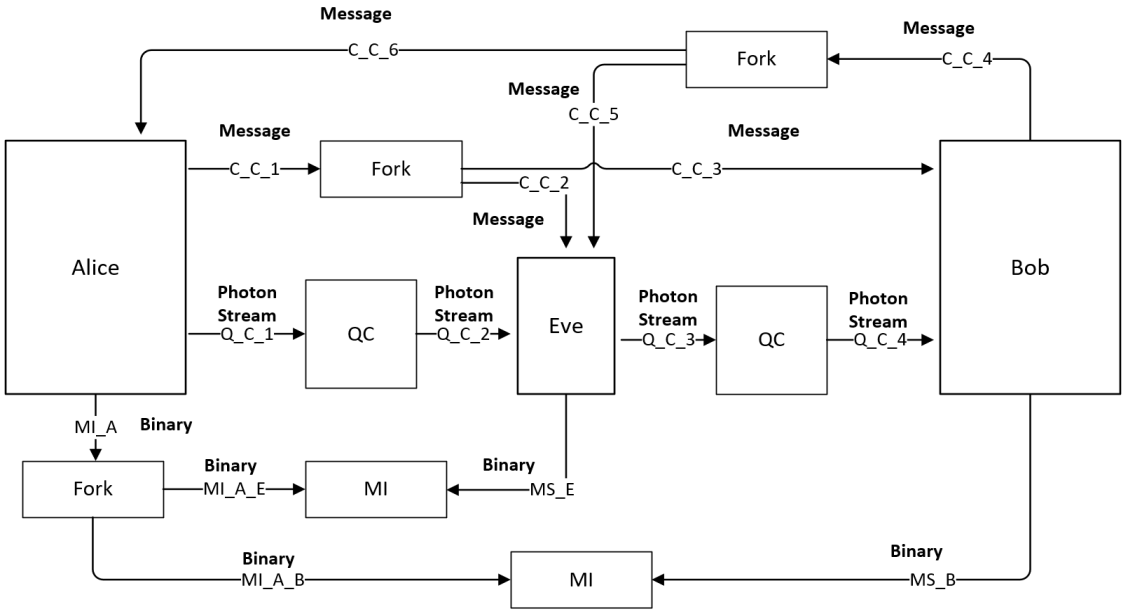


Figure 1.27: Simulation diagram at a top level

7. Experimental Implementation.

References

[1] Charles H Bennet. "Quantum cryptography: Public key distribution and coin tossing". In: *Proc. of IEEE Int. Conf. on Comp., Syst. and Signal Proc., Bangalore, India, Dec. 10-12, 1984*. 1984.

[2] Christopher Gerry and Peter Knight. *Introductory quantum optics*. Cambridge university press, 2005.

[3] Nelson J Muga, Mário FS Ferreira, and Armando N Pinto. "QBER estimation in QKD systems with polarization encoding". In: *Journal of Lightwave Technology* 29.3 (2011), pp. 355–361.

[4] Nelson J Muga and Armando N Pinto. "Extended Kalman filter vs. geometrical approach for stokes space-based polarization demultiplexing". In: *Journal of Lightwave Technology* 33.23 (2015), pp. 4826–4833.

[5] Cristian B Czegledi et al. "Polarization drift channel model for coherent fibre-optic systems". In: *Scientific reports* 6 (2016), p. 21217.

1.4 Arithmetic Encoding & Decoding

Students Name	:	Diogo Barros (46084)
Starting Date	:	July 17, 2018
Goal	:	Integer implementation of Arithmetic encoding and decoding.

Arithmetic encoding is a source coding technique that represents a complete string as a real number in a sub-interval of the unit interval [0,1). Since this interval has unlimited real numbers, it is possible to assign an unique real number to any string of a given length. The coded string is the binary representation of the assigned real value.

Unlike Huffman encoding, a unique code is obtained without the need to generate all codes of all possible strings of the same length, though the implementation of arithmetic coding is significantly more complex to circumvent numeric precision problems.

1.4.1 Encoding Algorithm

Students Name	:	Diogo Barros (17/07/2018 - 20/07/2018)
Goal	:	Arithmetic Encoding Algorithm Description

The first reference to arithmetic encoding was made in [REF]. However, the first practical implementations were only proposed in 1976 by Pasco [REF] and Rissanen [REF]. The block diagram of the algorithm in it's most simple floating point formulation is presented in Fig. 1.28.

The first step before the encoding starts is to compute the cumulative symbol probabilities of the source, given the probabilities of each individual symbol. The initial interval is set to [0,1) and is subdivided into sub intervals based on the symbol probabilities. The required resolution (number of bits) of the registers that represent the limits of the current interval is given by the lowest symbol probability as expressed in (1.21)

$$Resolution = \lceil (-\log_2(\min(P(x_n)))) \rceil + 1 \tag{1.21}$$

The encoding process starts by reading a symbol, identifying the corresponding sub interval in the current interval and updating the variables that save the current interval limits by using (1.22)-(1.24).

$$delta = lim\_high - lim\_low + 1 \tag{1.22}$$

$$lim\_high = lim\_low + \lfloor delta \cdot count(symb + 1)/total\_count \rfloor - 1 \tag{1.23}$$

$$lim\_low = lim\_low + \lfloor delta \cdot count(symb)/total\_count \rfloor \tag{1.24}$$

At this point we need to perform two operations. The first is to check if we are able to output any code bit. The second is to rescale the interval if its amplitude is small enough. These operations are done based on the three conditions (1.25), (1.26) and (1.27).

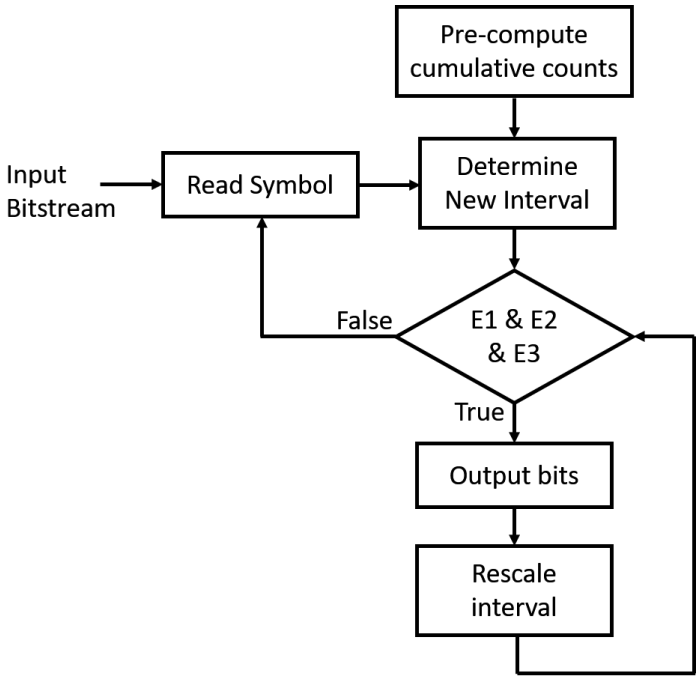


Figure 1.28: Block diagram of the algorithm for arithmetic encoding.

The algorithm keeps track of how many times condition (1.27) is met in succession and increments an internal extra bit counter.

$$E1 = (lim\_low < 0.5) \;\&\&\; (lim\_high < 0.5) \tag{1.25}$$

$$E2 = (lim\_low > 0.5) \;\&\&\; (lim\_high > 0.5) \tag{1.26}$$

$$E3 = (lim\_low > 0.25) \;\&\&\; (lim\_high < 0.75) \tag{1.27}$$

When (1.25) is met, the algorithm outputs a '0' and a number of extra '1' bits equal to the number of times (1.27) was met previously and the extra bit counter is reset. When (1.25) is met, the algorithm outputs a '1' and a number of '0' bits equal to the number of times (1.27) was met previously and the extra bit counter is reset. In the practical implementation, these conditions are tested by comparing the most significant bits of each register.

If any of these conditions is met, the amplitude of the interval is lower than 0.25 and needs to be rescaled by a factor of two, depending on which condition is met, as described in (1.28), (1.29) and (1.30).

$$E1: \; lim\_low = 2 \cdot lim\_low; \; lim\_high = 2 \cdot lim\_high; \tag{1.28}$$

$$E2: \; lim\_low = 2 \cdot (lim\_low - 0.5); \; lim\_high = 2 \cdot (lim\_high - 0.5); \tag{1.29}$$

$$E3: \; lim\_low = 2 \cdot (lim\_low - 0.25); \; lim\_high = 2 \cdot (lim\_high - 0.25); \tag{1.30}$$

The interval keeps on rescaling until none of the conditions is met, after which a new symbol is read and the process repeats until all symbols are coded.

1.4.2 Decoding Algorithm

Students Name	:	Diogo Barros (18/07/2018 - 20/07/2018)
Goal	:	Arithmetic Decoding Algorithm Description.

The decoding operations are the same as the ones performed by the coding algorithm with the exception of symbol identification and how the output bits are determined. The simplified block diagram of the encoder algorithm is presented in Fig. 1.29.

The decoder uses the same bit resolution as that of the encoder and starts by reading that number of bits from the coded bit stream, that becomes the "tag" value. The tag is then normalized as expressed in (1.31) and used to find the next symbol that was coded. This is done by finding the interval in the cumulative probability vector that contains it. Once the symbol is identified the code that corresponds to it can be sent to the output.

$$tag\_norm = \lfloor((tag - lim\_low + 1) \cdot total\_count - 1)/(lim\_high - lim\_low + 1)\rfloor \quad (1.31)$$

The interval is updated in the same way as in the encoder and the conditions (1.25), (1.26) and (1.27) are computed to rescale the interval. Additionally the tag value is updated based on which of these conditions is met.

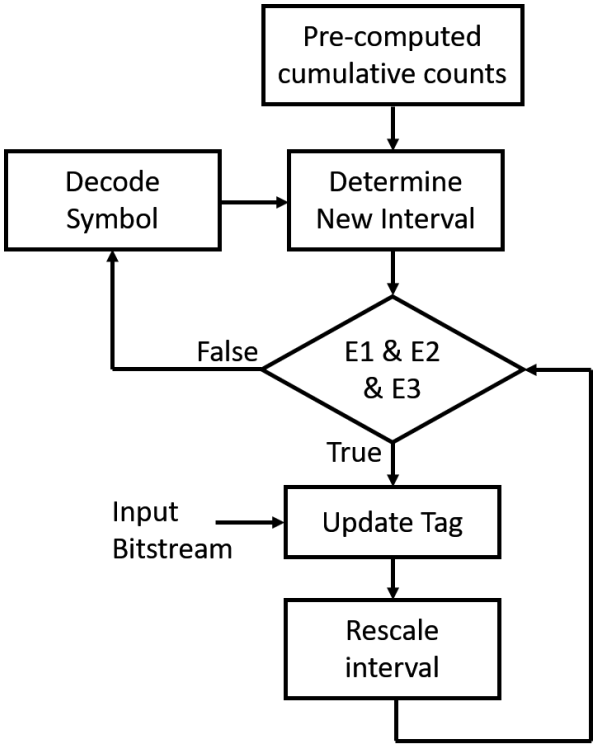


Figure 1.29: Block diagram of the algorithm for arithmetic decoding.

If either (1.25) or (1.26) is met, a new bit is read from the coded bit stream and added to the tag value multiplied by two. In the implemented code this is done right shifting the tag

bits by one (discarding the most significant bit) and adding the new bit to it. If (1.27) is met, the tag is updated in the same way but the new most significant bit is negated. When none of the conditions is met, a new symbol is decoded using the tag value normalized and the process continues until all symbols are decoded.

1.4.3 Encoding and Decoding Simulation Results

To test the implemented encoding and decoding algorithms, the system presented in Fig.1.30 was created in simulation. The system contains a binary source, the arithmetic encoding and decoding blocks and a bit-error-ratio computation block to ensure that the information is not degraded.

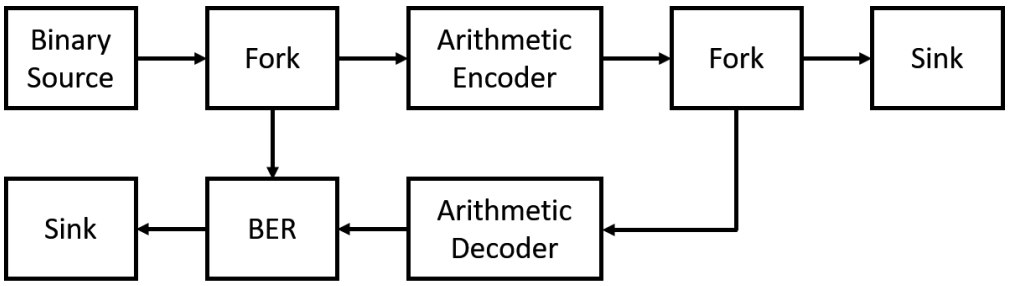


Figure 1.30: Block diagram of the floating point algorithm for arithmetic decoding.

A test simulation with a total of 45360 bits and a probability of zero of 0.1 was performed. the number of bits of the encoder was changed from 2 to 5 and the code efficiency, given by (1.32), was computed for each value. The results are presented in Fig.1.31. As expected, the code efficiency of the arithmetic algorithm is very close to optimum and increases as the number of coding bits increases.

$$\eta = \frac{H(n)}{L} \tag{1.32}$$

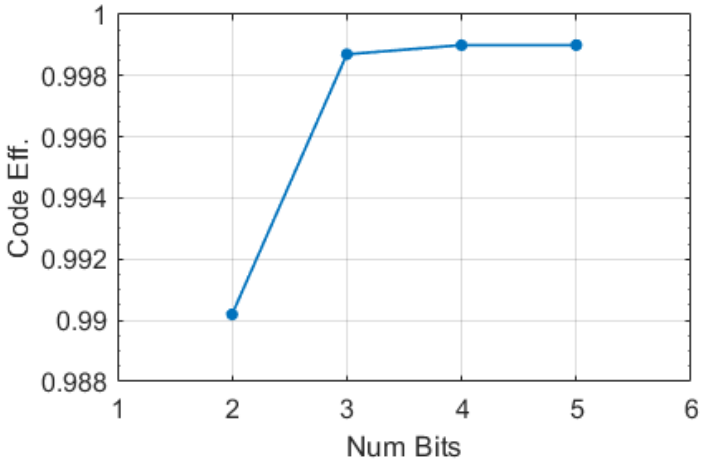


Figure 1.31: Code efficiency variation with the number of coding bits.

Chapter 2

Library

2.1 ADC

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

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

Input Parameters

Parameter	Unity	Type	Values	Default
samplingPeriod	–	double	any	$--$
rFactor	–	double	any	1
resolution	bits	double	any	inf
maxValue	volts	double	any	1.5
minValue	volts	double	any	-1.5

Table 2.1: ADC input parameters

Methods

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

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

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

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

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

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

void setQuantizerMaxValue(double maxvalue) B03.setMaxValue(maxvalue); B04.setMaxValue(maxvalue);

void setQuantizerMinValue(double minvalue) B03.setMinValue(minvalue); B04.setMinValue(minvalue);

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.2 Add

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

Input Parameters

This block takes no parameters.

Functional Description

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

Input Signals

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

Output Signals

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

2.3 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,

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

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \tag{2.2}$$

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

Input Signals

Number: 1 or 2
Type: Complex

Output Signals

Number: 2
Type: Complex

2.4 Bit Error Rate

Header File : bit_error_rate_*.h
Source File : bit_error_rate_*.cpp
Version : 20171810 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
alpha	double	0.05
m	integer	0
lMinorant	double	1×10^{-10}

Version : 20181424 (Mariana Ramos)

Name	Type	Default Value
alpha	double	0.05
m	integer	0
lMinorant	double	1×10^{-10}
midRepType	MidReportType	Cumulative

Methods

- BitErrorRate(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setConfidence(double P) { alpha = 1-P; }
- void setMidReportSize(int M) { m = M; }
- void setLowestMinorant(double lMinorant) { lowestMinorant=lMinorant; }

Version : 20181424 (Mariana Ramos)

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

- void initialize(void);
- bool runBlock(void);
- void setConfidence(double P) { alpha = 1-P; }
- void setMidReportSize(int M) { m = M; }
- void setLowestMinorant(double lMinorant) { lowestMinorant=lMinorant; }
- void setMidReportType(MidReportType mrt) { midRepType = mrt; };

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. In version **20180424** this block can operate mid-reports using a CUMULATIVE mode, in which the BER is calculated in a cumulative way taking into account all received bits, coincidences and errors, or in a RESET mode, in which at each **m** bits the number of received bits and coincidence bits is reset for the BER calculation.

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:

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \tag{2.3}$$

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

$N_T > 40$ [1]:

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

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

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

References

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

2.5 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	$1.0/100e9$

Table 2.2: 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 2.1 numbered in this order). Some of these require wrap.

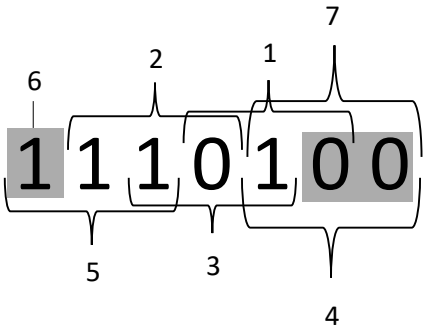


Figure 2.1: Example of a pseudorandom sequence with a pattern length equal to 3.

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 2.2.

Sugestions for future improvement

Implement an input signal that can work as trigger.

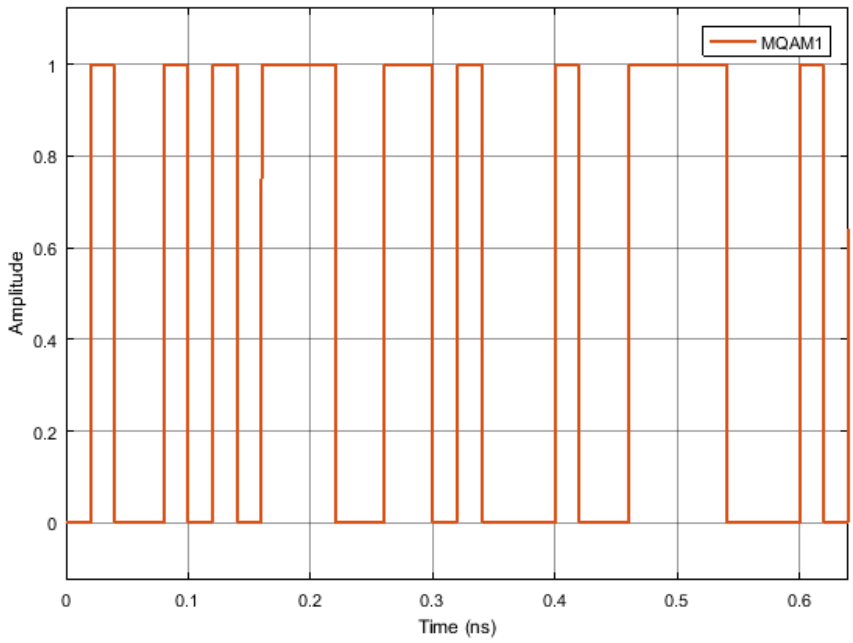


Figure 2.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

2.6 Bit Decider

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

Input Parameters

Name	Type	Default Value
decisionLevel	double	0.0

Functional Description

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

Input Signals

Number: 1
Type: Real signal (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

2.7 Clock

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

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

Input Parameters

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

Table 2.3: Binary source input parameters

Methods

Clock()

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

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per)

void setSamplingPeriod(double sPeriod)

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

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

Examples

Sugestions for future improvement

2.8 Clock_20171219

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

Input Parameters

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

Methods

Clock()

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

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per) double getClockPeriod()

void setClockPhase(double pha) double getClockPhase()

void setSamplingPeriod(double sPeriod) double getSamplingPeriod()

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

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

Examples

Sugestions for future improvement

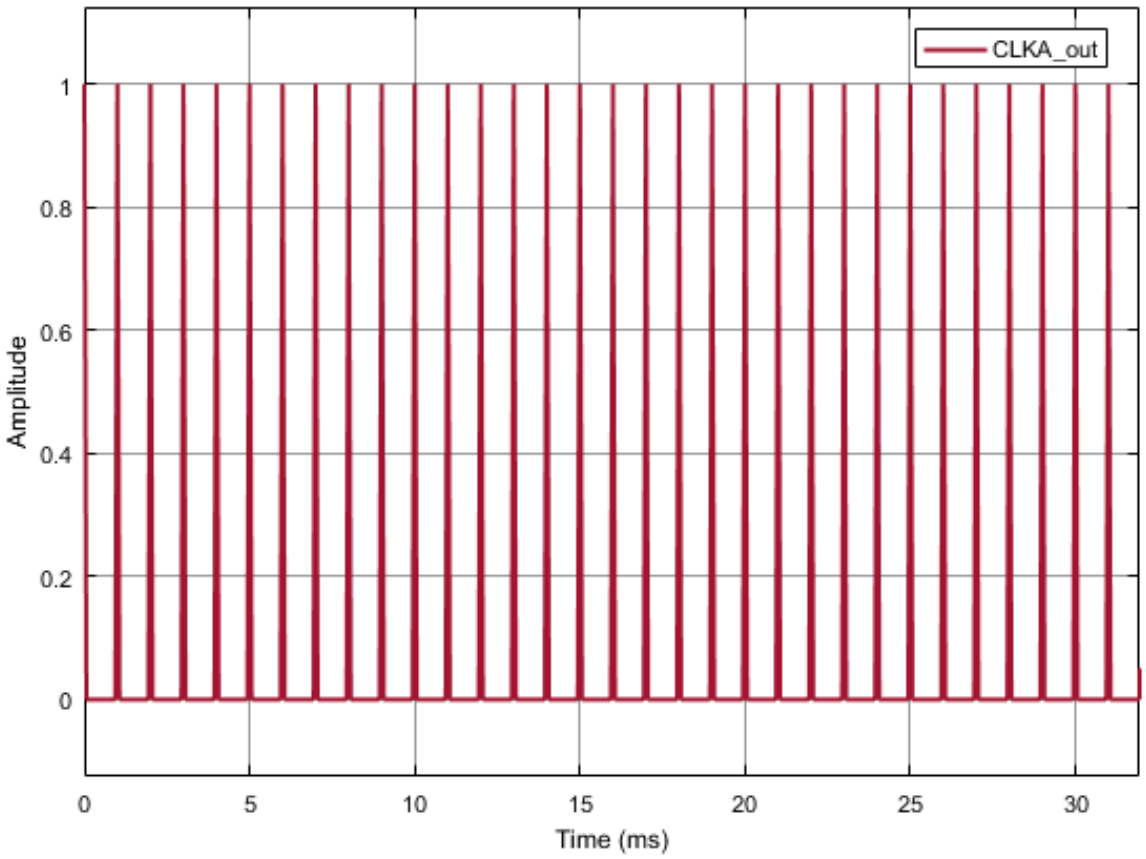


Figure 2.3: Example of the output signal of the clock without phase shift.

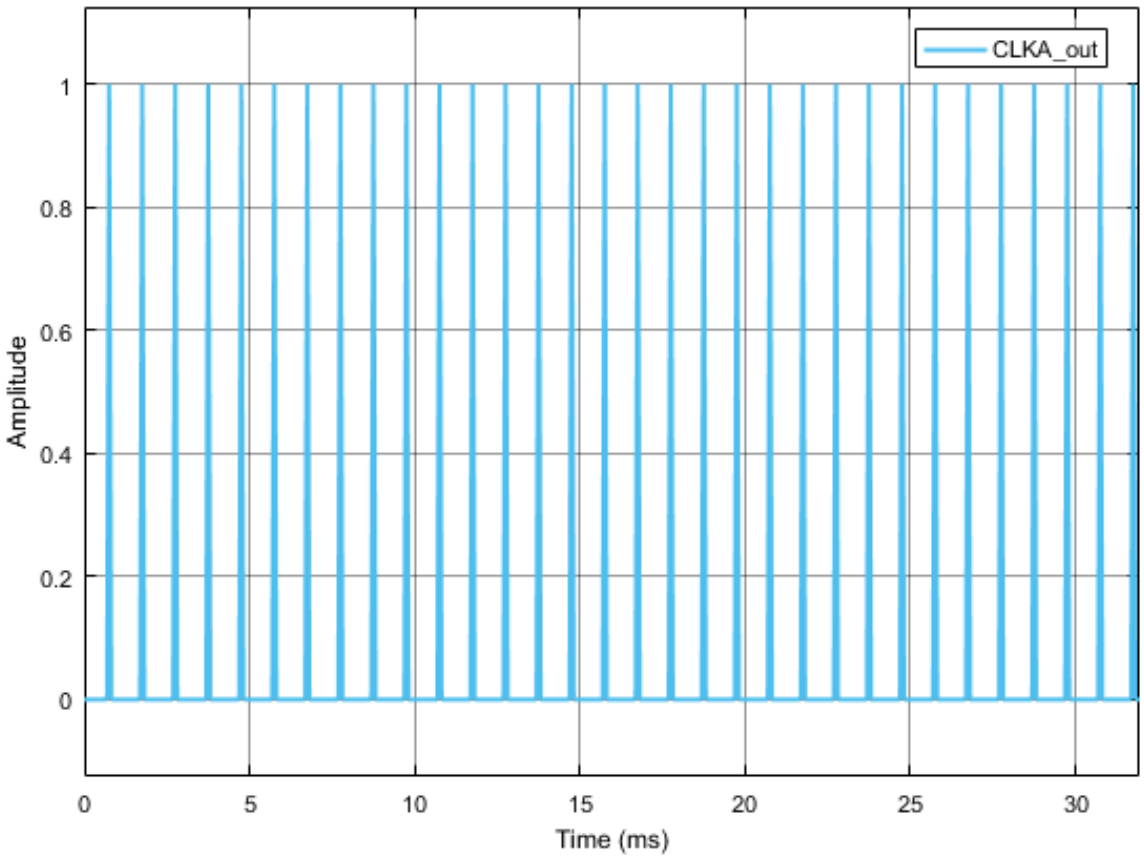


Figure 2.4: Example of the output signal of the clock with phase shift.

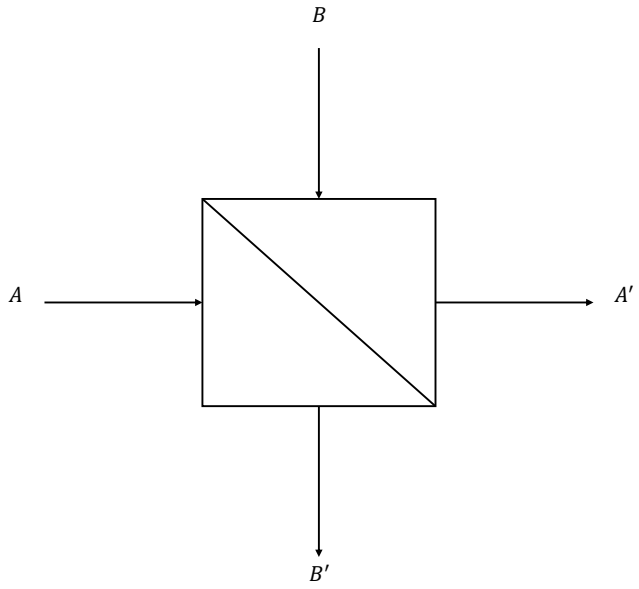


Figure 2.5: 2x2 coupler

2.9 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} . \begin{bmatrix} A \\ B \end{bmatrix} \tag{2.6}$$

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

$$T = \sqrt{1 - \eta_R} \tag{2.7}$$

$$R = \sqrt{\eta_R} \tag{2.8}$$

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

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

2.10 Carrier Phase Compensation

Header File	:	cpe_∗.h
Source File	:	cpe_∗.cpp
Version	:	20180423 (Celestino Martins)

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

Input Parameters For VV Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
methodType	string	VV	VV
samplingPeriod	double	any	0.0
symbolPeriod	double	any	0.0

Table 2.4: CPE input parameters

Input Parameters For BPS Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
NtestPhase	int	any	32
methodType	string	BPS	VV
samplingPeriod	double	any	0.0
symbolPeriod	double	any	0.0

Table 2.5: CPE input parameters

Methods

CPE() ;

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod) samplingPeriod = sPeriod;

void setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;

void setnTaps(double ntaps) nTaps = ntaps;

double getnTaps() return nTaps;

void setTestPhase(double nTphase) nTestPhase = nTphase;

double getTestPhase() return nTestPhase;

void setmethodType(string mType) methodType = mType;

double getmethodType() return methodType;

Functional description

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

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.11 Decoder

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

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

Input Parameters

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

Table 2.6: Binary source input parameters

Methods

Decoder()

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

void initialize(void)

bool runBlock(void)

void setM(int mValue)

void getM()

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues>getIqAmplitudes()

Functional description

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

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

Input Signals

Number: 1

Type: Electrical complex (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Binary

Examples

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

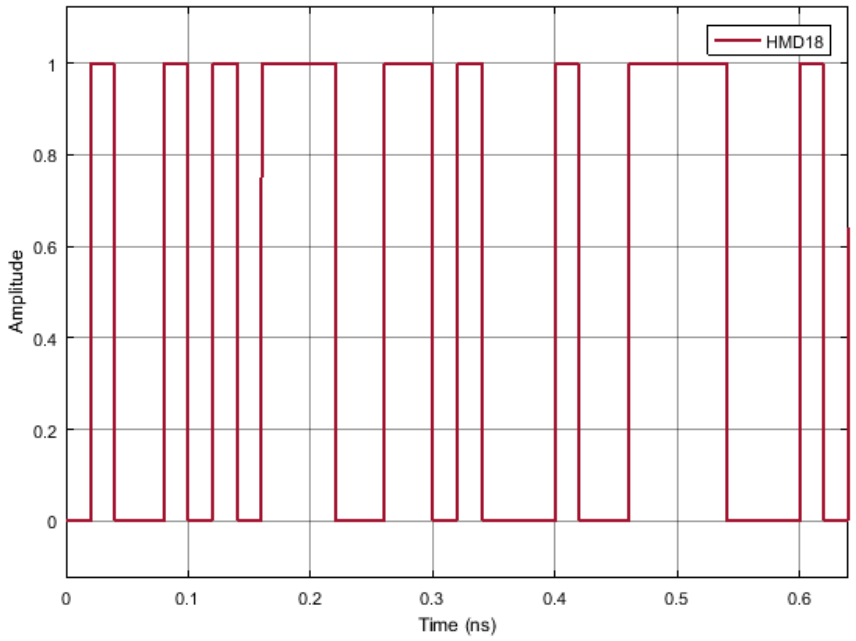


Figure 2.6: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

2.12 Discrete To Continuous Time

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

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

Input Parameters

Parameter	Type	Values	Default
numberOfSamplesPerSymbol	int	any	8

Table 2.7: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)

int const getNumberOfSamplesPerSymbol(void)

Functional Description

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

Input Signals

Number : 1

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

Output Signals

Number : 1

Type : Sequence of Dirac delta functions (ContinuousTimeDiscreteAmplitude)

Example

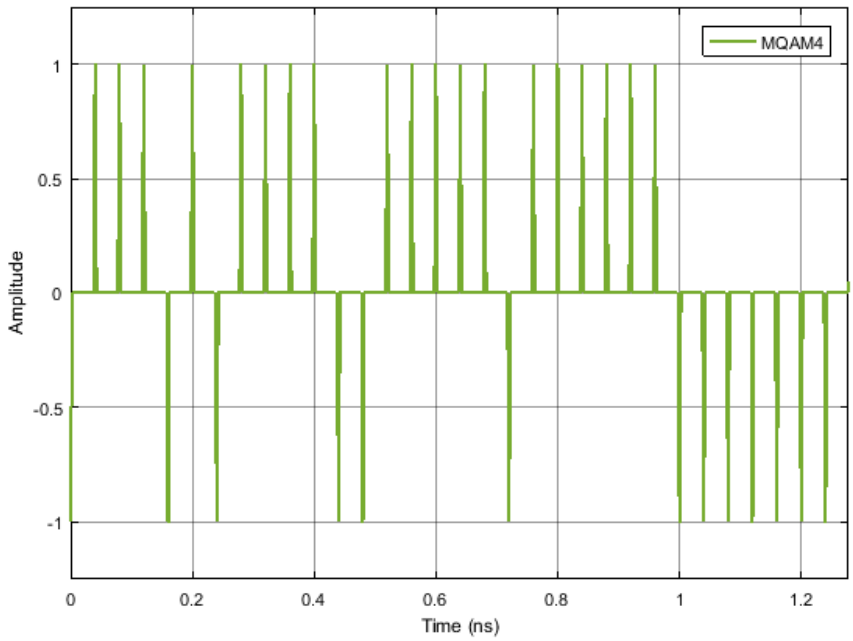


Figure 2.7: Example of the type of signal generated by this block for a binary sequence 0100...

2.13 DSP

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

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

Input Parameters

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

Table 2.8: DSP input parameters

Methods

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

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

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

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

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

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

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.14 Electrical Signal Generator

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

2.14.1 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

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

Methods

ElectricalSignalGenerator() {};

void initialize(void);

bool runBlock(void);

void setFunction(ElectricalSignalFunction fun) ElectricalSignalFunction getFunction()

void setSamplingPeriod(double speriod) double getSamplingPeriod()

void setSymbolPeriod(double speriod) double getSymbolPeriod()

void setGain(double gvalue) double getGain()

Functional description

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

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

Input Signals

Number: 0

Type: No type

Output Signals

Number: 1

Type: TimeContinuousAmplitudeContinuous

Examples

Sugestions for future improvement

Implement other functions according to the needs.

2.15 Fork

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

Input Parameters

— NA —

Input Signals

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

Output Signals

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

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

Functional Description

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

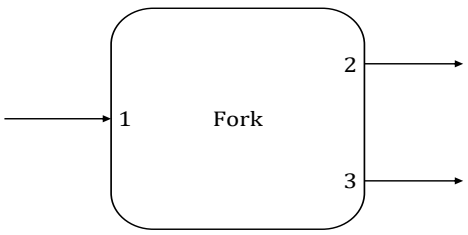


Figure 2.8: Fork

2.16 Gaussian Source

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

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

Input Parameters

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

Table 2.9: Gaussian source input parameters

Methods

GaussianSource()

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

void initialize(void);

bool runBlock(void);

void setAverage(double Average) ;

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Continuous signal (TimeDiscreteAmplitudeContinuousReal)

Examples

Sugestions for future improvement

2.17 MQAM Receiver

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

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

This block of code simulates the reception and demodulation of an optical signal (which is the input signal of the system) outputing a binary signal. A simplified schematic representation of this block is shown in figure 2.9.

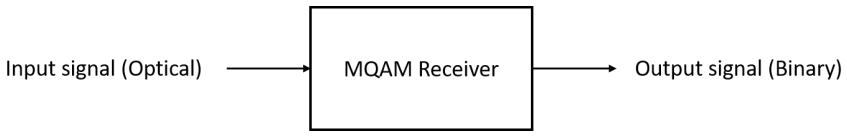


Figure 2.9: Basic configuration of the MQAM receiver

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

In can also be seen from figure 2.10 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*.

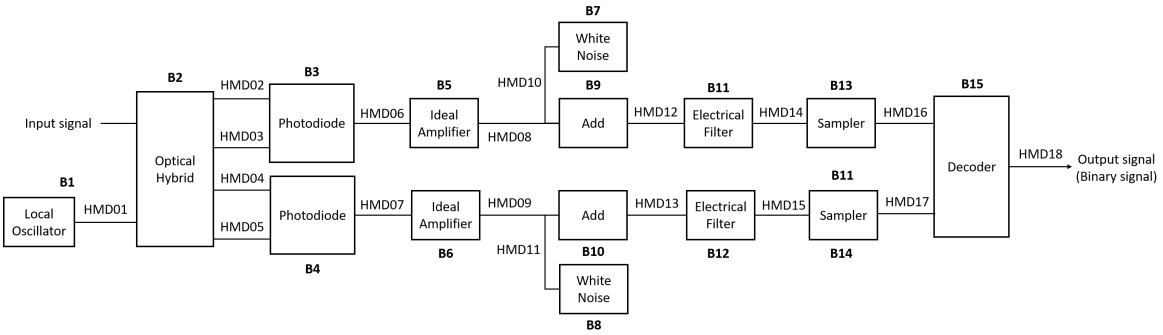


Figure 2.10: Schematic representation of the block homodyne receiver.

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 2.15) 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*

Table 2.10: List of input parameters of the block MQAM receiver

Methods

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

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setLocalOscillatorSamplingPeriod(double sPeriod)

void setLocalOscillatorOpticalPower(double opticalPower)

void setLocalOscillatorOpticalPower_dBm(double opticalPower_dBm)

void setLocalOscillatorPhase(double lOscillatorPhase)

void setLocalOscillatorOpticalWavelength(double lOscillatorWavelength)

void setSamplingPeriod(double sPeriod)

void setResponsivity(t_real Responsivity)

void setAmplification(t_real Amplification)

void setNoiseAmplitude(t_real NoiseAmplitude)

void setImpulseResponseTimeLength(int impResponseTimeLength)

void setFilterType(PulseShaperFilter fType)

void setRollOffFactor(double rOffFactor)

void setClockPeriod(double per)

void setSamplesToSkip(int sToSkip)

Input Signals

Number: 1

Type: Optical signal

Output Signals

Number: 1

Type: Binary signal

Example

Sugestions for future improvement

2.18 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

Table 2.11: 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

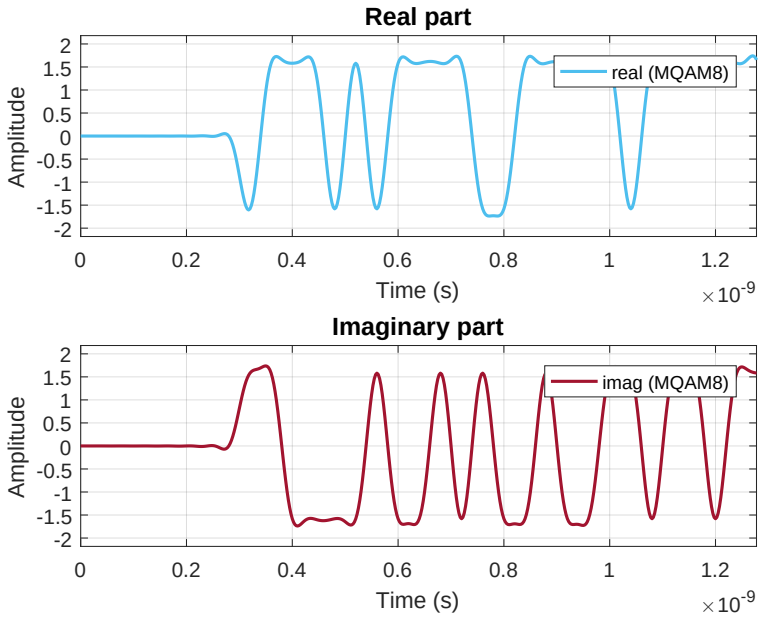


Figure 2.11: Example of a signal generated by this block for the initial binary signal 0100...

2.19 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

Table 2.12: 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

2.20 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

Table 2.13: Local oscillator input parameters

Methods

LocalOscillator()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setPhase(double lOscillatorPhase);

void setLaserLinewidth(double laserLinewidth);

double getLaserLinewidth();

void setLaserRIN(double LOlaserRIN);

double getLaserRIN();

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal

Examples

Sugestions for future improvement

2.21 MQAM Mapper

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

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

Input Parameters

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

Table 2.14: 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 2.12.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

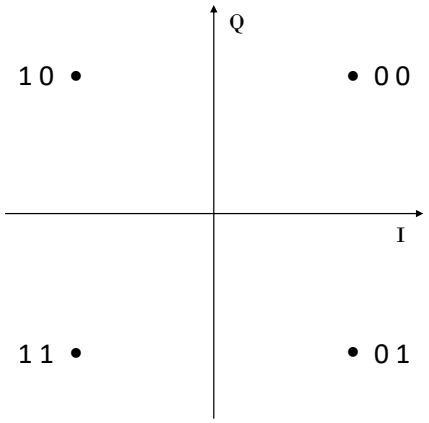


Figure 2.12: Constellation used to map the signal for m=4

Output Signals

Number : 2

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

Example

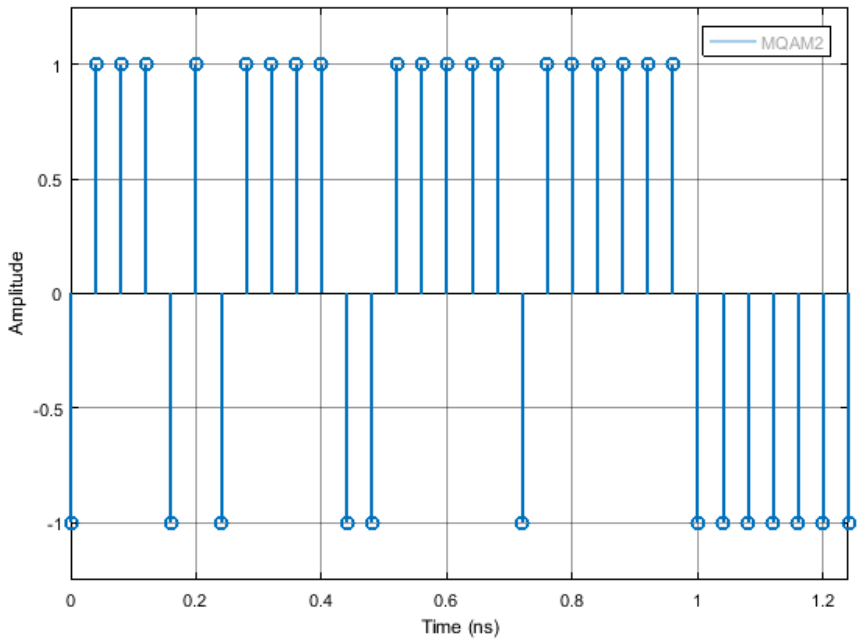


Figure 2.13: Example of the type of signal generated by this block for the initial binary signal 0100...

2.22 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 2.14.

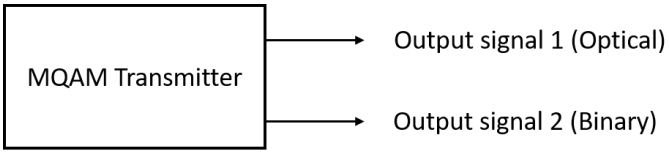


Figure 2.14: Basic configuration of the MQAM transmitter

Functional description

This block generates an optical signal (output signal 1 in figure 2.15). The binary signal generated in the internal block Binary Source (block B1 in figure 2.15) 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 2.15).

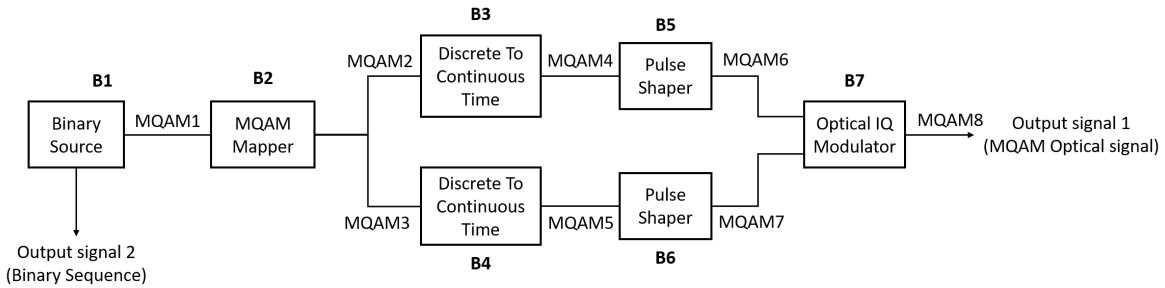


Figure 2.15: Schematic representation of the block MQAM transmitter.

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 2.15.

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

Table 2.15: 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.

2.23 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

Table 2.16: 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

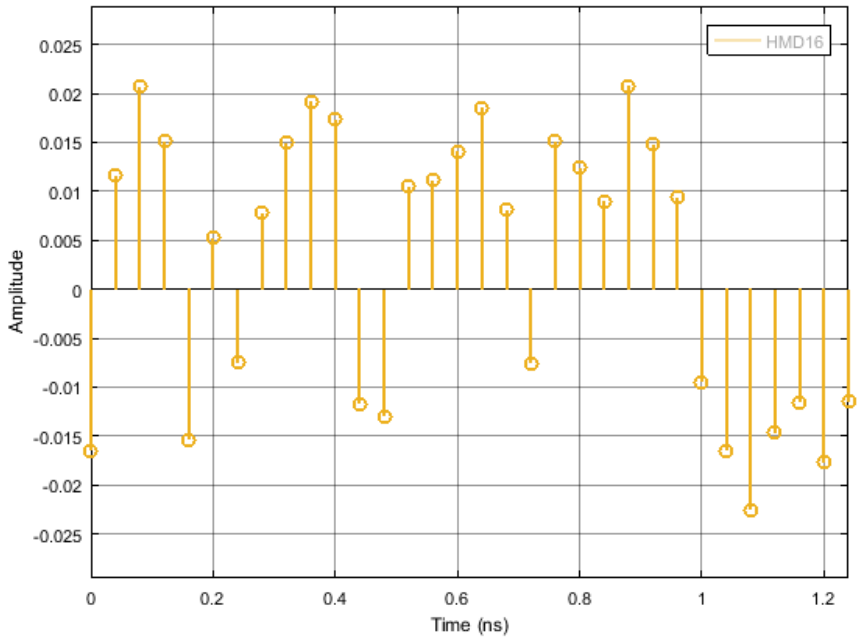


Figure 2.16: Example of the output signal of the sampler

2.23.1 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

2.24 Alice QKD

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

Input Parameters

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

Methods

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

Functional description

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

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

Input Signals

Number : 3

Type : Binary, Real Continuous Time and Messages signals.

Output Signals

Number : 3

Type : Binary, Real Discrete Time and Messages signals.

Examples

Sugestions for future improvement

2.25 Polarizer

This block is responsible of changing the polarization of the input photon stream signal by using the information from the other real time discrete input signal. This way, this block accepts two input signals: one photon stream and other real discrete time signal. The real discrete time input signal must be a signal discrete in time in which the amplitude can be 0 or 1. The block will analyse the pairs of values by interpreting them as basis and polarization direction.

Input Parameters

- m{4}
- Amplitudes { {1,1}, {-1,1 }, {-1,-1 }, { 1,-1} }

Methods

Polarizer (vector <Signal*> &inputSignals, vector <Signal*>&outputSignals) : Block(inputSignals, outputSignals) {};
void initialize(void);
bool runBlock(void);
void setM(int mValue);
void setAmplitudes(vector <t_iqValues> AmplitudeValues);

Functional description

Considering m=4, this block 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

2.26 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

$$\begin{aligned} X(\omega) &= 1 \quad \text{,if } \omega \in A \\ X(\omega) &= 0 \quad \text{,if } \omega \in \bar{A} \end{aligned} \tag{2.9}$$

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 $P(X = 0)$,

$$\begin{aligned} P(X = 1) = &\quad P(A) &&= p \\ P(X = 0) = &\quad P(\bar{A}) &&= 1 - p \end{aligned} \tag{2.10}$$

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

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

$$\frac{X - np}{\sqrt{np(1-p)}} \backsim N(0, 1). \tag{2.11}$$

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:

$$\frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} \backsim N(0, 1) \tag{2.12}$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} < z_{\alpha/2}\} &\approx 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})} < np - X < z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})}\} &\approx 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}\} &\approx 1 - \alpha \end{aligned} \tag{2.13}$$

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

Input Parameters

- zscore
  (double)
- fileName
  (string)

Methods

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

void initialize(void);

bool runBlock(void);

void setProbabilityExpectedX(double probx) double getProbabilityExpectedX()

void setProbabilityExpectedY(double proby) double getProbabilityExpectedY()

void setZScore(double z) double getZScore()

Functional description

This block receives an input binary signal with values "0" or "1" and it calculates the probability of having each number according with the number of samples acquired. This probability is calculated using the following formulas:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \tag{2.14}$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \tag{2.15}$$

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

$$\text{ME} = \text{z}_{\text{score}} \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \tag{2.16}$$

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,

$$\begin{aligned} \text{ME} &= \pm z_{\alpha/2}\frac{\sigma}{\sqrt{N}} \\ \text{ME} &= \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{N}}, \end{aligned} \tag{2.17}$$

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

2.27 Bob QKD

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

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

2.28 Eve QKD

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

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

2.29 Rotator Linear Polarizer

This block accepts a Photon Stream signal and a Real discrete time signal. It produces a photon stream by rotating the polarization axis of the linearly polarized input photon stream by an angle of choice.

Input Parameters

- m{2}
- axis { {1,0}, {$\frac{\sqrt{2}}{2}$,$\frac{\sqrt{2}}{2}$ } }

Methods

RotatorLinearPolarizer(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) : Block(inputSignals, outputSignals) {};
void initialize(void);
bool runBlock(void);
void setM(int mValue);
void setAxis(vector <t_iqValues> AxisValues);

Functional description

This block accepts the input parameter m, which defines the number of possible rotations. In this case m=2, the block accepts the rectilinear basis, defined by the first position of the second input parameter axis, and the diagonal basis, defined by the second position of the second input parameter axis. This block rotates the polarization axis of the linearly polarized input photon stream to the basis defined by the other input signal. If the discrete value of this signal is 0, the rotator is set to rotate the input photon stream by $0°$, otherwise, if the value is 1, the rotator is set to rotate the input photon stream by an angle of $45°$.

Input Signals

Number : 2

Type : Photon Stream and a Sequence of 0's and '1s (DiscreteTimeDiscreteAmplitude)

Output Signals

Number : 1

Type : Photon Stream

Examples

Sugestions for future improvement

2.30 Optical Switch

This block has one input signal and two input signals. Furthermore, it accepts an additional input binary input signal which is used to decide which of the two outputs is activated.

Input Parameters

No input parameters.

Methods

OpticalSwitch(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals) : Block(inputSignals, outputSignals) {};
void initialize(void);
bool runBlock(void);

Functional description

This block receives an input photon stream signal and it decides which path the signal must follow. In order to make this decision it receives a binary signal (0's and 1's) and it switch the output path according with this signal.

Input Signals

Number : 1

Type : Photon Stream

Output Signals

Number : 2

Type : Photon Stream

Examples

Sugestions for future improvement

2.31 Optical Hybrid

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

This block simulates an optical hybrid. It accepts two input signals corresponding to the signal and to the local oscillator. It generates four output complex signals separated by $90°$ in the complex plane. Figure 2.17 shows a schematic representation of this block.

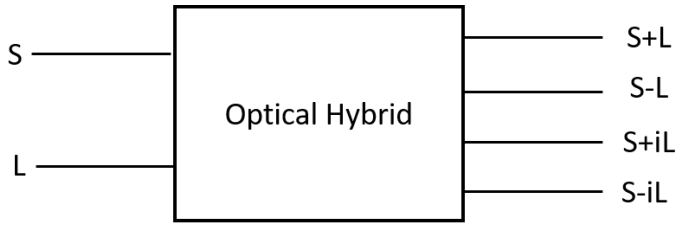


Figure 2.17: Schematic representation of an optical hybrid.

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

Table 2.17: Optical hybrid input parameters

Methods

OpticalHybrid()

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

void initialize(void)

bool runBlock(void)

void setOutputOpticalPower(double outOpticalPower)

void setOutputOpticalPower_dBm(double outOpticalPower_dBm)

void setOutputOpticalWavelength(double outOpticalWavelength)

void setOutputOpticalFrequency(double outOpticalFrequency)

void setPowerFactor(double pFactor)

Functional description

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

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 4

Type: Optical (OpticalSignal)

Examples

Sugestions for future improvement

2.32 Photodiode pair

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

This block simulates a block of two photodiodes assembled like in figure 2.18. 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.

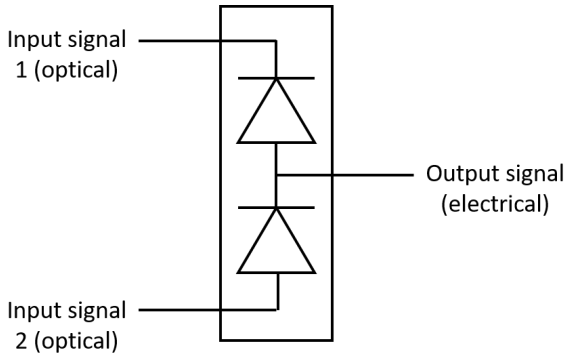


Figure 2.18: Schematic representation of the physical equivalent of the photodiode code block.

Input Parameters

- responsivity{1}
- outputOpticalWavelength{ 1550e-9 }
- outputOpticalFrequency{ SPEED_OF_LIGHT / wavelength }

Methods

Photodiode()

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

void initialize(void)

bool runBlock(void)

void setResponsivity(`t_real` Responsivity)

Functional description

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

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

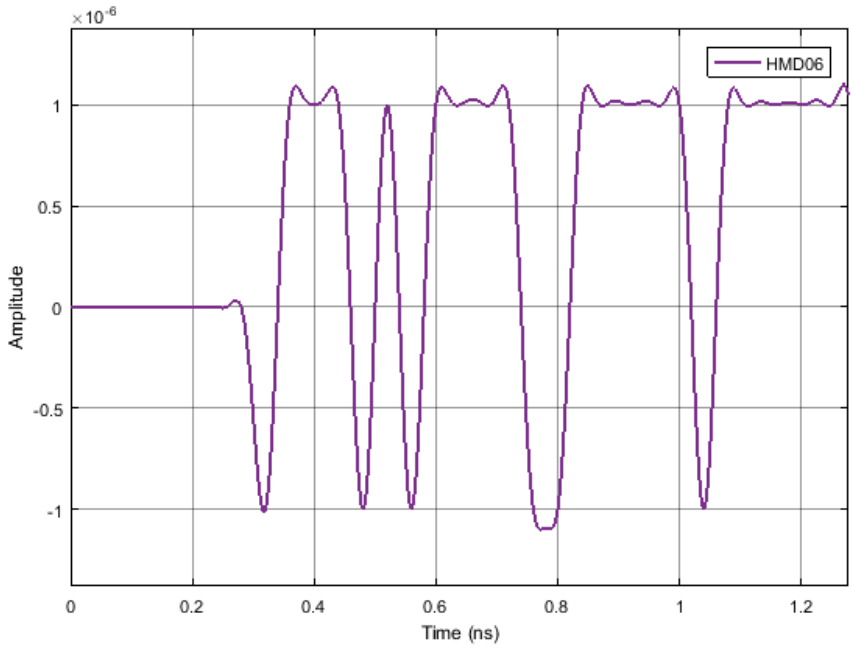


Figure 2.19: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

2.33 Photoelectron Generator

Header File	:	photoelectron_generator_*.h
Source File	:	photoelectron_generator_*.cpp
Version	:	20180302 (Diamantino Silva)

This block simulates the generation of photoelectrons by a photodiode, performing the conversion of an incident electric field into an output current proportional to the field's instantaneous power. It is also capable of simulating shot noise.

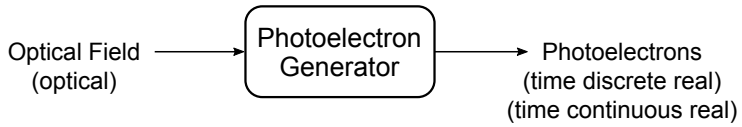


Figure 2.20: Schematic representation of the photoelectron generator code block

Theoretical description

The operation of a real photodiode is based on the photoelectric effect, which consists on the removal of one electron from the target material by a single photon, with a probability η. Given an input beam with an optical power $P(t)$ in which the photons are around the wavelength λ, the flux of photons $\phi(t)$ is calculated as [1]

$$\phi(t) = \frac{P(t)\lambda}{hc} \tag{2.18}$$

therefore, the mean number of photons in a given interval $[t, t+T[$ is

$$\bar{n}(t) = \int_t^{t+T} \phi(\tau)\, \mathrm{d}\tau \tag{2.19}$$

But the actual number of photons in a given time interval, $n(t)$, is random. If we assume that the electric field is generated by an ideal laser with constant power, then $n(t)$ will follow a Poisson distribution

$$p\,(n) = \frac{\bar{n}^n \exp(-\bar{n})}{n!} \tag{2.20}$$

where $n = n(t)$ and $\bar{n} = \bar{n}(t)$.
For each incident photon, there is a probability η of generating a phototelectron. Therefore, we can model the generation of photoelectrons during this time interval, as a binomial process where the number of events is equal to the number of incident photons, $n(t)$, and the rate of success is η. If we combine the two random processes, binomial photoelectron generation after poissonian photon flux, the number of output photoelectrons in this time interval, $m(t)$, will follow [1]

$$m \sim \text{Poisson}\,(\eta\bar{n}) \tag{2.21}$$

with $\bar{m} = \eta\bar{n}$ where $m = m(t)$.

Functional description

The input of this block is the electric field amplitude, $A(t)$, with sampling period T. The first step consists on the calculation the instantaneous power. Given that the input amplitude is a baseband representation of the original signal, then $P(t) = 4|A(t)|^2$. From this result, the average number of photons $\bar{n}(t) = TP(t)\lambda/hc$.
If the shot-noise is negleted, then the output number of photoelectrons, $n_e(t)$ in the interval, will be equal to

$$m(t) = \eta\overline{n(t)} \tag{2.22}$$

If the shot-noise is considered, then the output fluctuations will be simulated by generating a value from a Poissonian random number generator with mean $\eta\overline{n(t)}$

$$m(t) \sim \text{Poisson}\left(\eta\overline{n(t)}\right) \tag{2.23}$$

Input Parameters

Parameter	Default Value	Description
efficiency	1.0	Photodiode's quantum efficiency.
shotNoise	false	Shot-noise off/on.

Methods

PhotoelectronGenerator()

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

void initialize(void)

bool runBlock(void)

void setEfficiency(`t_real` efficiency)

void getEfficiency()

void setShotNoise(`bool` shotNoise)

void getShotNoise()

Input Signals

Number: 1

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal or TimeContinuousAmplitudeContinuousReal)

Examples

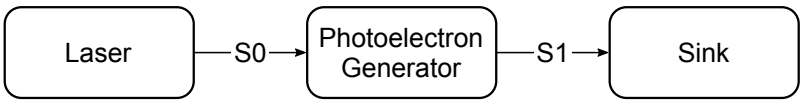
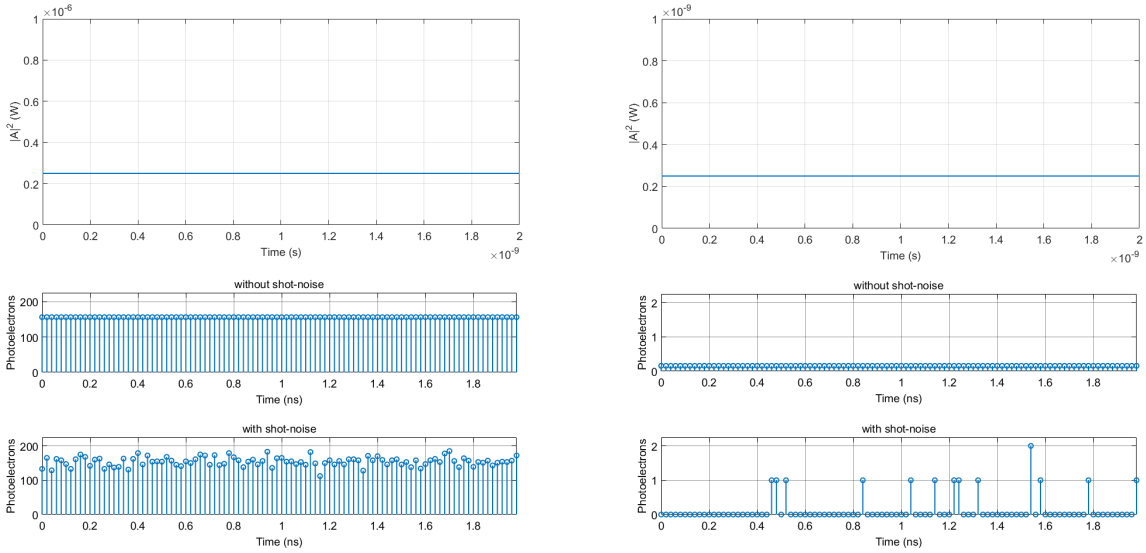


Figure 2.21: Constant power simulation setup

To test the output of this block, we recreated the results of figure $11.2 - 3$ in [1]. We started by simulating the constant optical power case, in which the local oscillator power was fixed to a constant value. Two power levels were tested, $P = 1\mu W$ and $P = 1nW$, using a sample period of 20 picoseconds and photoelectron generator efficiency of 1.0. The simulation code is in folder `lib \photoelectron_generator \photoelectron_generator_test_constant`. The following plots show the number of output electrons per sample when the shot noise is ignored or considered



(a) $P = 1\mu W$ (b) $P = 1nW$

Figure 2.22: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 2.22 shows clearly that turning the quantum noise on or off will produce a signal with or without variance, as predicted. If we compare this result with plot (a) in [1], in particular $P = 1nW$, we see that they are in conformance, with a slight difference, where a sample has more than one photoelectron. In constrast with the reference result, where only single events are represented, the $P = 1\mu W$ case shows that all samples account many photoelectrons. Given it's input power, multiple photoelectron generation events will occur during the sample time window. Therefore, to recreate the reference result, we just need to reduce the sample period until the probability of generating more than 1 photoelectron per sample goes to 0.

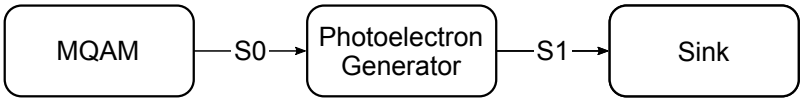


Figure 2.23: Variable power simulation setup

To recreate plot (b) in [1], a more complex setup was used, where a series of states are generated and shaped by a MQAM, creating a input electric field with time-varying power. The simulation code is in folder `lib \photoelectron_generator \photoelectron_generator_test_variable`.

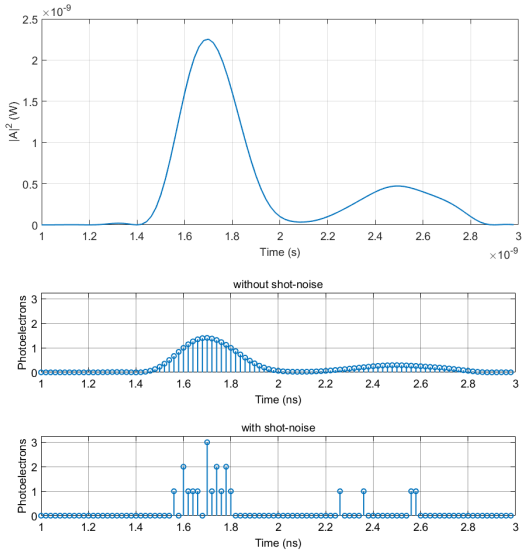


Figure 2.24: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 2.24 shows that the output without shot-noise is following the input power perfectly, apart from a constant factor. In the case with shot-noise, we see a that there are only output samples in high power input samples.
These results are not following strictly plot (b) in [1], because has already discussed previously, in the high power input samples, we have a great probability of generating many photoelectrons.

Sugestions for future improvement

References

[1] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of photonics*. Wiley series in pure and applied optics. New York (NY): John Wiley & Sons, 1991.

2.34 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

Table 2.18: 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

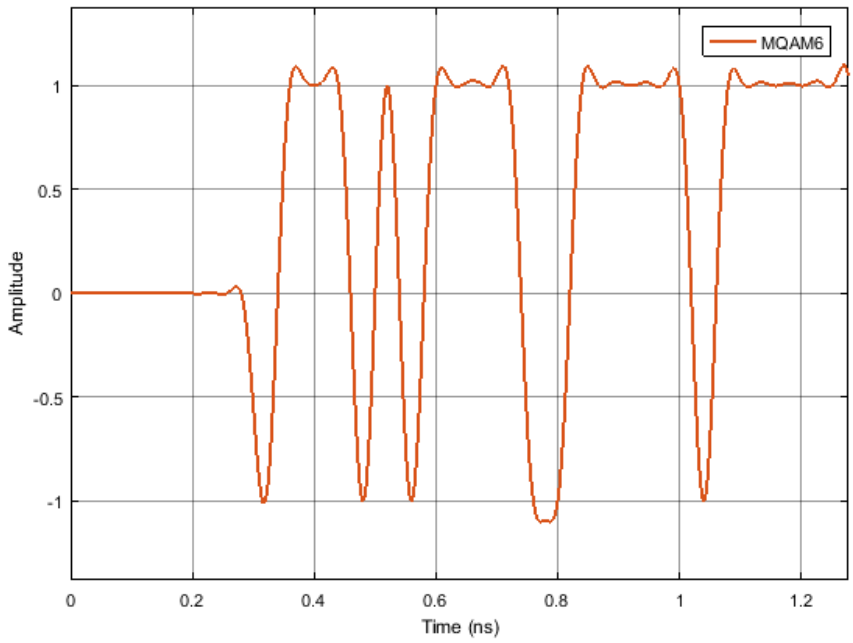


Figure 2.25: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

2.35 Quantizer

Header File	:	quantizer_∗.h
Source File	:	quantizer_∗.cpp
Version	:	20180423 (Celestino Martins)

This block simulates a quantizer, where the signal is quantized into discrete levels. Given a quantization bit precision, $resolution$, the outputs signal will be comprise $2^{nBits} - 1$ levels.

Input Parameters

Parameter	Unity	Type	Values	Default
resolution	bits	double	any	inf
maxValue	volts	double	any	1.5
minValue	volts	double	any	-1.5

Table 2.19: Quantizer input parameters

Methods

Quantizer() ;

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod) samplingPeriod = sPeriod;

void setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;

void setResolution(double nbits) resolution = nbits;

double getResolution() return resolution;

void setMinValue(double maxvalue) maxValue = maxvalue;

double getMinValue() return maxValue;

void setMaxValue(double maxvalue) maxValue = maxvalue;

double getMaxValue() return maxValue;

Functional description

This block can performs the signal quantization according to the defined input parameter *resolution*.

Firstly, the parameter *resolution* is checked and if it is equal to the infinity, the output signal correspond to the input signal. Otherwise, the quantization process is applied. The input signal is quantized into $2^{resolution-1}$ discrete levels using the standard *round* function.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.36 Resample

Header File	:	resample_∗.h
Source File	:	resample_∗.cpp
Version	:	20180423 (Celestino Martins)

This block simulates the resampling of a signal. It receives one input signal and outputs a signal with the sampling rate defined by sampling rate, which is externally configured.

Input Parameters

Parameter	Type	Values	Default
rFactor	double	any	inf
samplingPeriod	double	any	0.0
symbolPeriod	double	any	1.5

Table 2.20: Resample input parameters

Methods

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

void initialize(void); bool runBlock(void);

void setSamplingPeriod(double sPeriod) samplingPeriod = sPeriod; void setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;

void setOutRateFactor(double OUTsRate) rFactor = OUTsRate; double getOutRateFactor() return rFactor;

Functional description

This block can performs the signal resample according to the defined input parameter *rFactor*. It resamples the input signal at rFactor times the original sample rate.

Firstly, the parameter *nBits* is checked and if it is greater than 1 it is performed a linear interpolation, increasing the input signal original sample rate to rFactor times.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

2.37 Sampler

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

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

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

Input Parameters

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

Table 2.21: 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

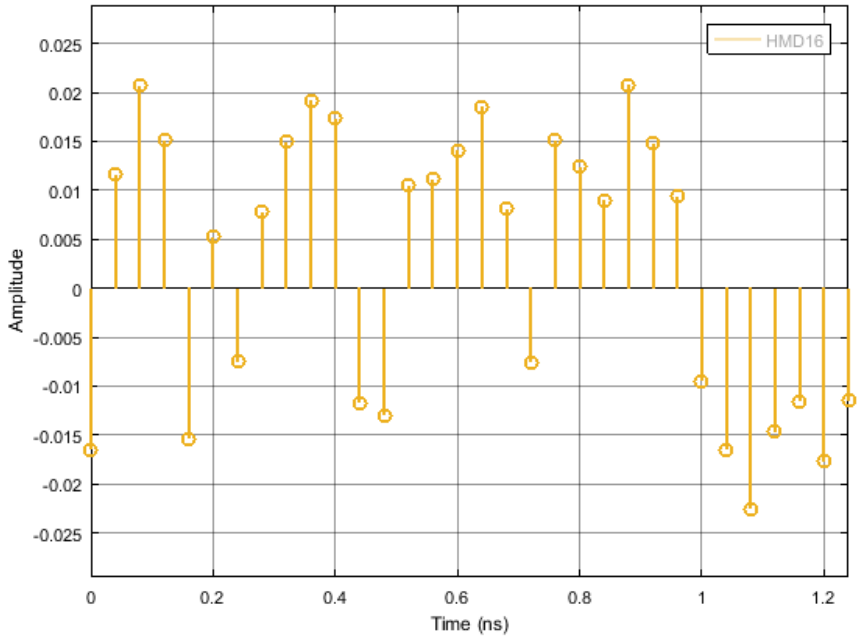


Figure 2.26: Example of the output signal of the sampler

Sugestions for future improvement

2.38 SNR of the Photoelectron Generator

Header File	:	srn_photoelectron_generator_*.h
Source File	:	snr_photoelectron_generator_*.cpp
Version	:	20180309 (Diamantino Silva)

This block estimates the signal to noise ratio (SNR) of a input stream of photoelectrons, for a given time window.

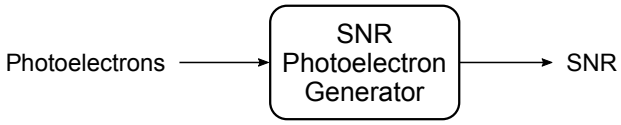


Figure 2.27: Schematic representation of the SNR of the Photoelectron Generator code block

Theoretical description

The input of this block is a stream of samples, y_j, each one of them corresponding to a number of photoelectrons generated in a time interval Δt. These photoelectrons are usually the output of a photodiode (photoelectron generator). To calculate the SNR of this stream, we will use the definition used in [1]

$$\text{SNR} = \frac{\bar{n}^2}{\sigma_n^2} \tag{2.24}$$

in which $\bar{n}$ is the mean value and σ_n^2 is the variance of the photon number in a given time interval T.
To apply this definition to our input stream, we start by separate it's samples in contiguous time windows with duration T. Each time window k is defined as the time interval $[kT, (k+1)T[$. To estimate the SNR for each time window, we will use the following estimators for the mean, μ_k, and variance, s_k^2 [2]

$$\mu_k = \langle y \rangle_k \qquad\qquad s_k^2 = \frac{N}{N-1}\left(\langle y^2 \rangle_k - \langle y \rangle_k^2\right) \tag{2.25}$$

were $\langle y^n \rangle_k$ is the n moment of the k window given by

$$\langle y^n \rangle_k = \frac{1}{N_k} \sum_{j=j_{\min}(k)}^{j_{\max}(k)} y_j^n \tag{2.26}$$

in which

$$j_{min}(k) = \lceil t_k/\Delta t \rceil \tag{2.27}$$
$$j_{max}(k) = \lceil t_{k+1}/\Delta t \rceil - 1 \tag{2.28}$$
$$N_k = j_{max}(k) - j_{min}(k) + 1 \tag{2.29}$$
$$t_k = kT \tag{2.30}$$

where $\lceil x \rceil$ is the ceiling function.
In our implementation, we define two variables, $S_1(k)$ and $S_2(k)$, corresponding to the sum of the samples and the sum of the squares of the sample in the time interval k. These two sums are related to the moments as

$$S_1(k) = N_k \langle y \rangle_k \tag{2.31}$$
$$S_2(k) = N_k \langle y^2 \rangle_k \tag{2.32}$$

Using these two variables, we can rewrite μ_k and s_k^2 as

$$\mu_k = \frac{S_1(k)}{N_k} \qquad s_k^2 = \frac{1}{N_k - 1}\left(S_2(k) - \frac{1}{N_k}(S_1(k))^2\right) \tag{2.33}$$

The signal to noise ratio of the time interval k, SNR_k, can be expressed as

$$\text{SNR}_k = \frac{\mu_k^2}{\sigma_k^2} = \frac{N_k - 1}{N_k}\frac{(S_1(k))^2}{N_k S_2(k) - (S_1(k))^2} \tag{2.34}$$

One particularly important case is the phototelectron stream resulting from the conversion of a laser photon stream by a photodiode (photoelectron generator). The resulting SNR will be [1]

$$\text{SNR} = \eta\bar{n} \tag{2.35}$$

in which η is the photodiode quantum efficiency.

Functional description

This block is designed to operate in time windows, dividing the input stream in contiguous sets of samples with a duration `tWindow` $= T$. For each time window, the general process consists in accumulating the input sample values and the square of the input sample values, and calculating the SNR of the time window based on these two variables.
To process this accumulation, the block uses two state variables, `aux_sum1` and `aux_sum2`, which hold the accumulation of the sample values and accumulation of the square of sample values, respectively.
The block starts by calculating the number of samples it has to process for the current time window, using equations 2.28, 2.29 and 2.30. If the duration of `tWindow` is 0, then we assume that this time window has infinite time (infinite samples). The values of `aux_sum1` and `aux_sum2` are set to 0, and the processing of the samples of current window begins.
After processing all the samples of the time window, we obtain $S1(k)$ and $S2(k)$ from the

state variables as $S1(k) =$ `aux_sum1` and $S_2(k) =$ `aux_sum2`, and proceed to the calculation of the SNR$_k$, using equation 2.34.
If the simulation ends before reaching the end of the current time window, we calculate the SNR$_k$, using the current values of `aux_sum1`, `aux_sum2` for $S_1(k)$ and $S_2(k)$, and the number of samples already processed, `currentWindowSample`, for N_k.

Input Parameters

Parameter	Default Value	Description
windowTime	0	SNR time window.

Methods

SnrPhotoelectronGenerator()

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

void initialize(void)

bool runBlock(void)

void setTimeWindow(`t_real` timeWindow)

Input Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Examples

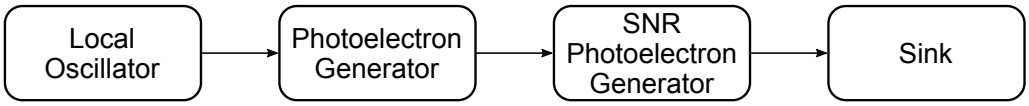
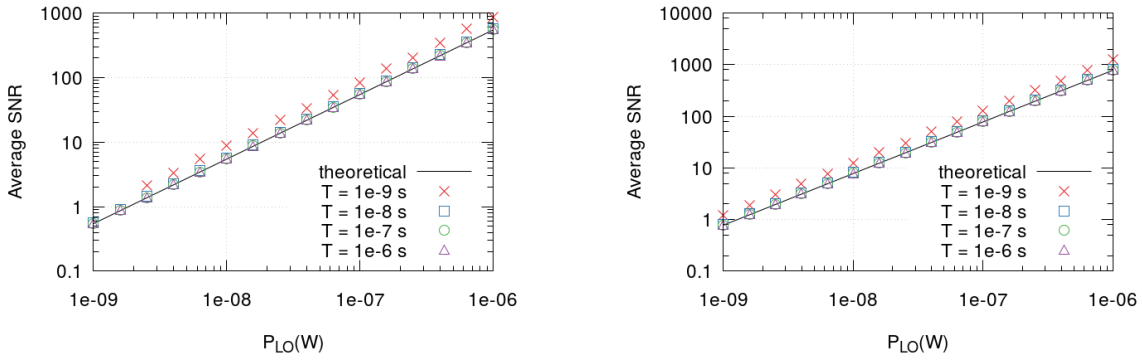


Figure 2.28: Simulation setup

To confirm the block's correct output, we have designed a simulation setup which calculates the SNR of a stream of photoelectrons generated by the detection of a laser photon stream by a photodiode.
The simulation has three main parameters, the power of the local oscillator, P_{LO}, the duration of the time window, T, and the photodiode's quantum efficiency, η. For each combination of these three parameters, the simulation generates 1000 SNR samples, during which all parameters stay constant. The final result is the average of these SNR samples.
The simulations were performed with a sample time $\Delta t = 10^{-10} s$.



(a) $\eta = 0.7$ (b) $\eta = 1$

Figure 2.29: Theoretical and simulated results of the average SNR, for two photodiode efficiencies.

The plots in 2.29 show the comparison between the theoretical result 2.35 and the simulation results. We see that for low values of T, the average SNR shows a systematic deviation form the theoretical resul, but for $T > 10^{-6}s$ (10000 samples per time window), the simulation result shows a very good agreement with the theoretical result.

The simulations also show a lack of average SNR results when low power, low efficiency and small time window are combined (see plot 2.29a). This is because in those conditions, the probability of having a time windows with no photoelectrons, creating a invalid SNR, is very high, which will prevent the calculation of the average SNR.
We can estimate the probability of calculating a valid SNR average by calculating the probability of no time window having 0 phototelectrons, $p_{ave} = (1-q)^M$, in which q is the probability of a time window having all it's samples equal to 0 and M is the number of time windows. We know that the input stream follows a Poisson distribution with mean $\bar{m}$, therefore $q = (\exp(-\bar{m}))^N$, in which $\bar{m} = \eta P\lambda/hc$ and $N = T/\Delta t$, is the average number of samples per time window. Using this result, we obtain the probability of calculating a valid SNR average as

$$p_{ave} = (1 - \exp(-N\bar{m}))^M \tag{2.36}$$

Block problems

Future work

The block could also output a confidence interval for the calculated SNR. Given that the output of the Photoelectron Generator follows a Poissonian distribuition when the shot noise is on, the article "Confidence intervals for signal to noise ratio of a Poisson distribution" by Florence George and B.M. Kibria [3], could be used as a reference to implement such feature.

References

[1] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of photonics*. Wiley series in pure and applied optics. New York (NY): John Wiley & Sons, 1991.

[2] S. W. Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.

[3] G. Florence and K. B. Golam. "Confidence intervals for signal to noise ratio of a Poisson distribution". In: *American Journal of Biostatistics* 2.2 (2011), p. 44.

2.39 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

Table 2.22: 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

2.40 White Noise

Header File	:	white_noise_20180420.h
Source File	:	white_noise_20180420.cpp

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

1. DefaultDeterministic
2. RandomDevice
3. Selected

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

Input Parameters

Parameter	Type	Values	Default
seedType	enum	DefaultDeterministic, RandomDevice, Selected	RandomDevice
spectralDensity	real	> 0	1.5×10^{-17}
seed	int	$\in [1, 2^{32} - 1]$	1
samplingPeriod	double	> 0	1.0

Table 2.23: White noise input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNoiseSpectralDensity(double SpectralDensity) spectralDensity = SpectralDensity;

double const getNoiseSpectralDensity(void) return spectralDensity;

void setSeedType(SeedType sType) seedType = sType; ;

SeedType const getSeedType(void) return seedType; ;

void setSeed(int newSeed) seed = newSeed;

int getSeed(void) return seed;

Functional description

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

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

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

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

The noise is obtained from a gaussian distribution with zero mean and a given variance. The variance is equal to the noise power, which can be calculated from the spectral density n_0 and the signal's bandwidth B, where the bandwidth is obtained from the defined sampling time T.

$$N = n_0 B = n_0 \frac{2}{T} \tag{2.37}$$

If the signal is complex, the noise is calculated independently for the real and imaginary parts, and the spectral density value is divided by two, to account for the two-sided noise spectral density.

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: RealValue, ComplexValue or ComplexValueXY

Examples

Random Mode

Suggestions for future improvement

2.41 Ideal Amplifier

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

Input Parameters

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

Table 2.24: Ideal Amplifier input parameters

Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;

Functional description

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

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

2.42 Arithmetic Encoder

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

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

Input Parameters

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

Table 2.25: Arithmetic encoder block input parameters.

Methods

bool runBlock(void)

void initialize(void);

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

Functional description

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

Input Signals

Number: 1

Output Signals

Number: 1

Type: binary

Examples

Sugestions for future improvement

2.43 Arithmetic Decoder

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

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

Input Parameters

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

Table 2.26: Arithmetic decoding block input parameters.

Methods

bool runBlock(void)

void initialize(void);

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

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

Chapter 3

Mathlab Tools

3.1 Generation of AWG Compatible Signals

Students Name	:	Francisco Marques dos Santos
	:	Romil Patel
Goal	:	Convert simulation signals into waveform files compatible with the laboratory's Arbitrary Waveform Generator
Version	:	sgnToWfm.m (**Student Name**: Francisco Marques dos Santos)
	:	sgnToWfm_20171119.m (**Student Name**: Romil Patel)

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.

3.1.1 sgnToWfm.m

Structure of a function

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

Inputs

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

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

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

Outputs

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

data: A vector with the signal data.

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

samplingPeriod: Sampling period of the signal.

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

numberOfSymbols: Number of symbols retrieved from the signal.

samplingRate: Sampling rate of the signal.

Functional Description

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

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

Using one argument:

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

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

Using two arguments:

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

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

Using three arguments:

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

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

3.1.2 sgnToWfm_20171121.m

Structure of a function

[dataDecimate, data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate, samplingRateDecimate] = sgnToWfm_20171121(fname_sgn, nReadr, fname_wfm)

Inputs

Same as discussed above in the file sgnToWfm.m.

Outputs

The output of the function sgnToWfm_20171121.m contains eight different parameters. Among those eight different parameters, six output parameters are the same as discussed above in the function sgnToWfm.m and remaining two parameters are the following:

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

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

«««< HEAD

Functional Description

The functional description is same as discussed above in sgnToWfm.m. =======

Outputs

The output of the function version 20171121 contains eight different parameters. Among those eight parameters, six output parameters are the same as discussed above in the version 20170930 and remaining two parameters are the following:

Name of output signals	Description
dataDecimate	A vector which contains decimated signal data by an appropriate decimation factor to make it compatible with the AWG.
samplingRateDecimate	Reduced sampling rate which is compatible with AWG. (i.e. less than 16 GSa/s).

Decimation factor calculation

The flowchart for calculating the decimation factor is as follows:

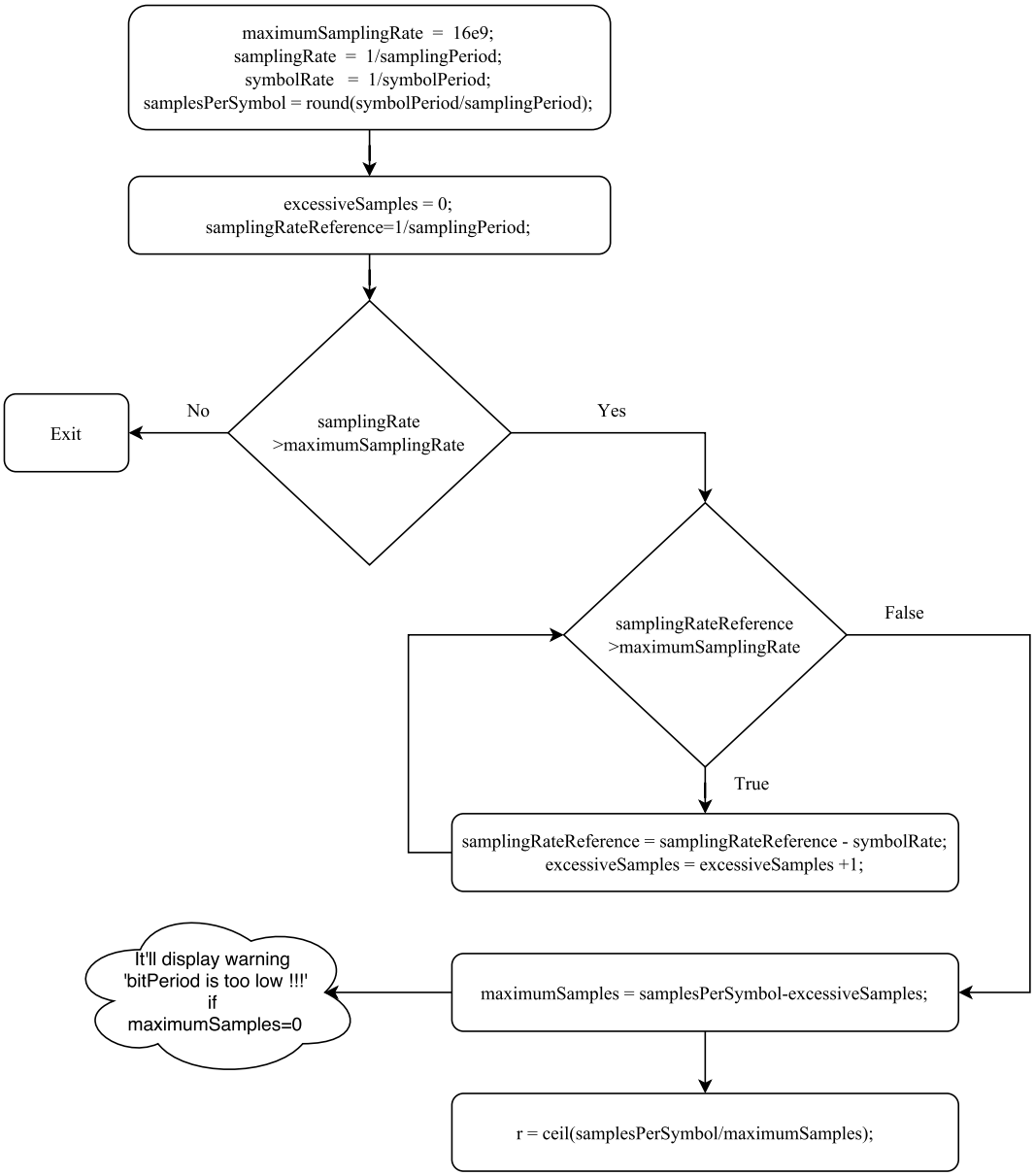


Figure 3.1: Flowchart to calculate decimation factor

»»»»> Develop.Romil

3.1.3 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).

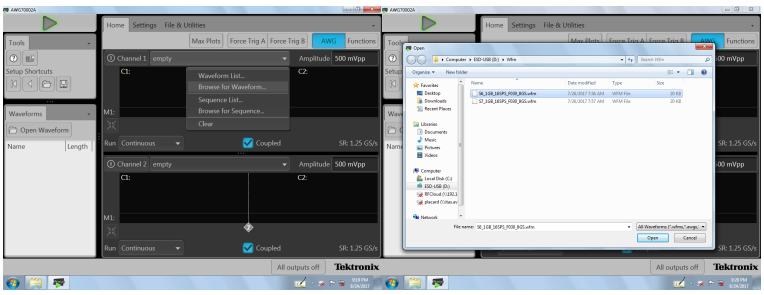


Figure 3.2: Selecting your waveform in the AWG

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

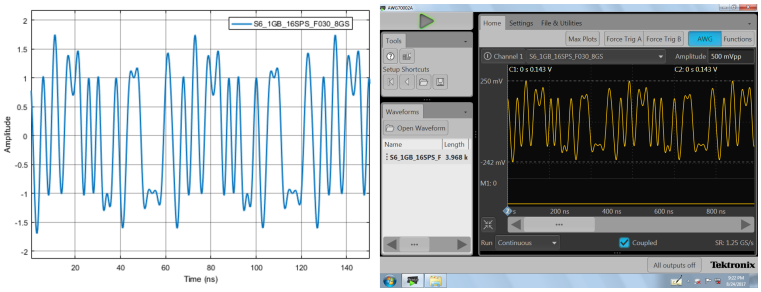


Figure 3.3: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

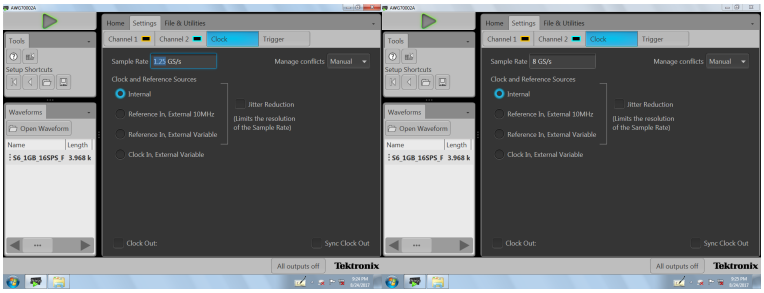


Figure 3.4: Configuring the right sampling rate

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

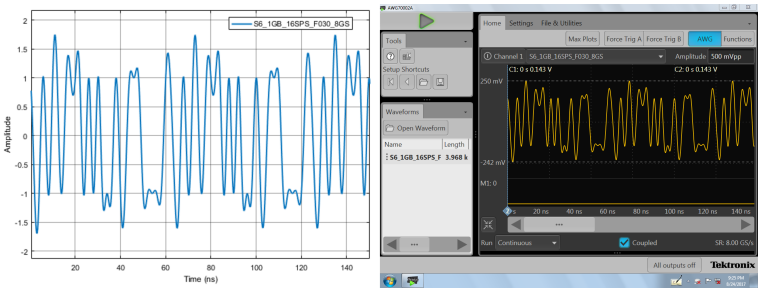


Figure 3.5: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

Chapter 4

Algorithms

4.1 Fast Fourier Transform

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

Algorithm

The algorithm for the FFT will be implemented according with the following expression,

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \ e^{-i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{4.1}$$

Similarly, for IFFT,

$$x_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X_k \ e^{i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{4.2}$$

From equations 4.1 and 4.2, 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,

$$y = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x \ e^{mi2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{4.3}$$

where, x is an input complex signal, y is the output complex signal and m equals -1 or 1 for FFT and IFFT, respectively. An optimized fft function is also implemented without the $1/\sqrt{(N)factor}$, see below in the optimized fft section.

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 4.1 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 [1]. 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 [2].

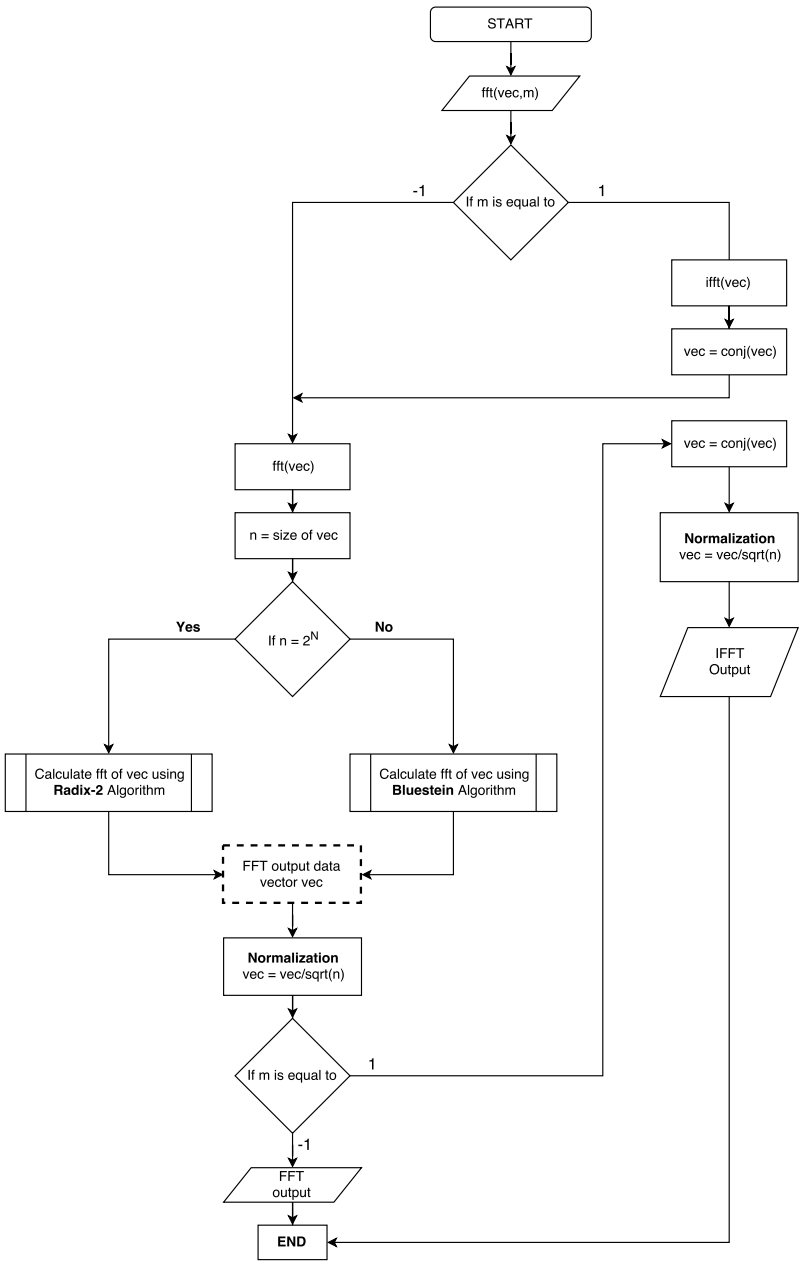


Figure 4.1: Top level architecture of FFT algorithm

Radix-2 algorithm

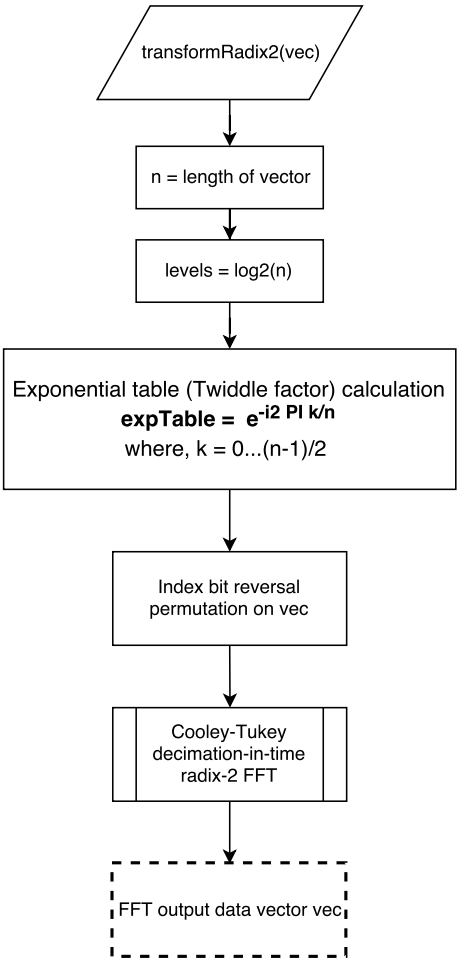


Figure 4.2: Radix-2 algorithm

Cooley-Tukey algorithm

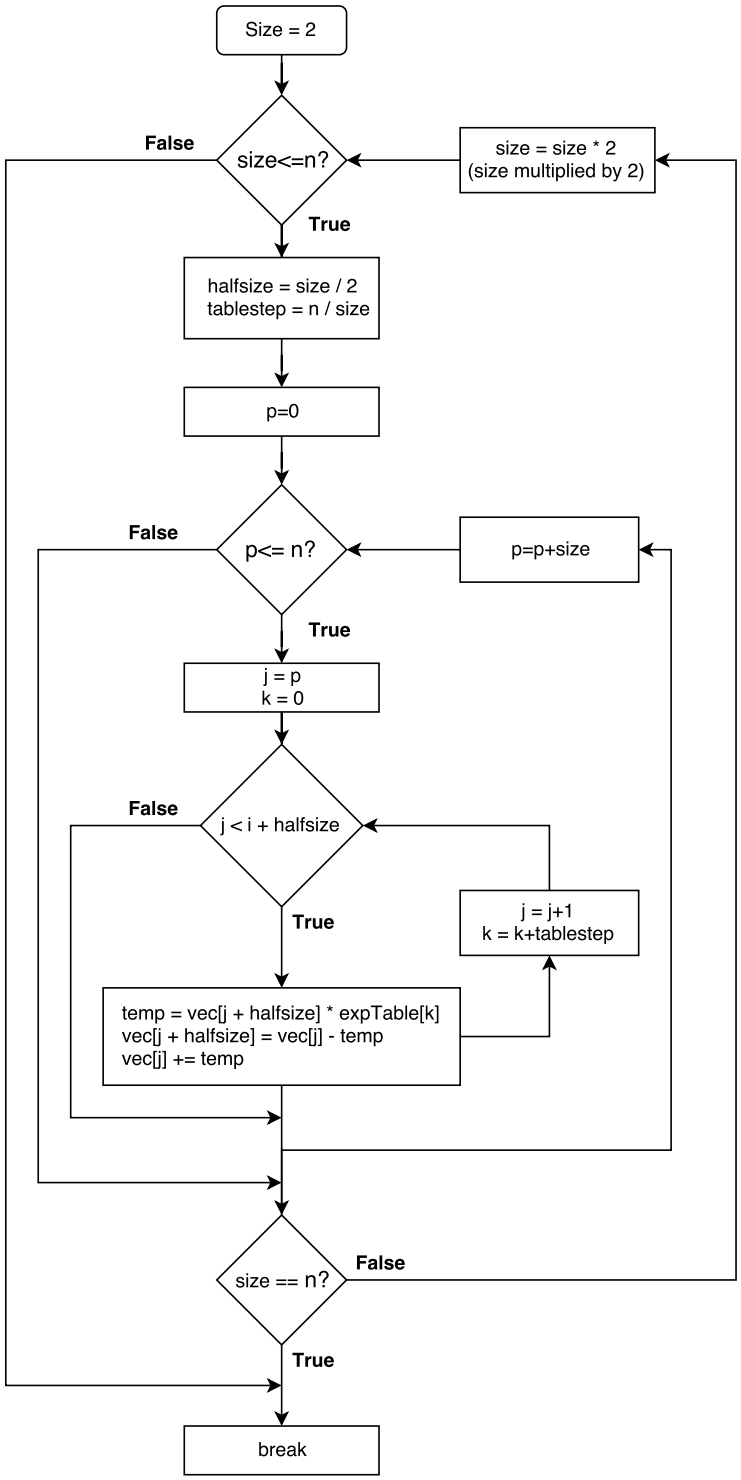


Figure 4.3: Cooley-Tukey algorithm

Bluestein algorithm

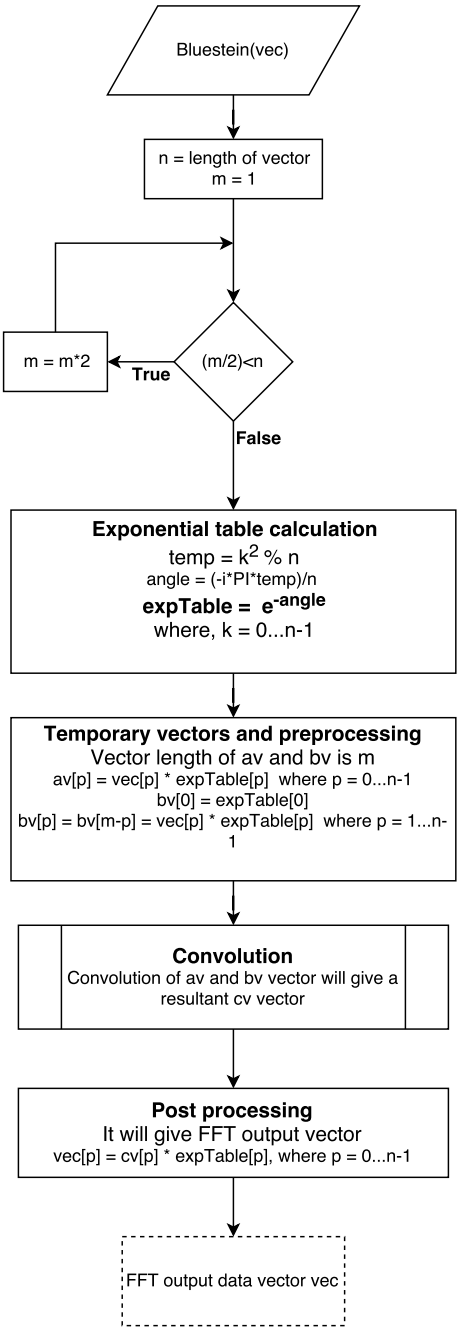


Figure 4.4: Bluestein algorithm

Convolution algorithm

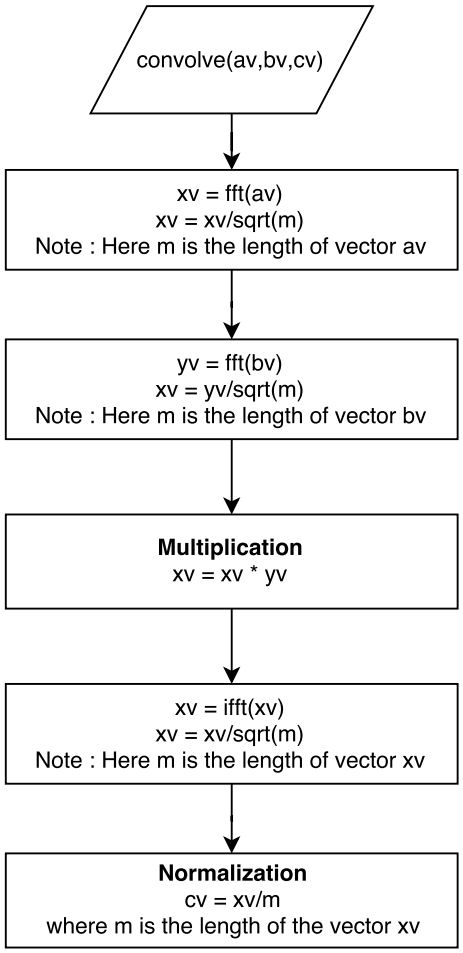


Figure 4.5: Circular convolution algorithm

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 = 2^10;               % 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*t) + cos(2*pi*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';
        X = sin(2*pi*200*t).*exp(-abs(70*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);
    case 8
        signal_title = 'Sinusoid tone';
        X = cos(2*pi*100*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 (s)')
ylabel('X(t)')
grid on

% 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');
grid on
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');
grid on

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 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

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

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

Listing 4.1: 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"

```

```
# include <complex>
# include <fstream>
# include <iostream>
# include <math.h>
# include <stdio.h>
# include <string>
# include <strstream>
# 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++)
```

```
  {
    re[i] = inTimeDomain[i]; // Real data of the signal
  }

  // Next, Real and Imaginary vector to complex vector conversion
  inTimeDomainComplex = reImVect2ComplexVector(re, im);

  // calculate FFT
  clock_t begin = clock();
  fourierTransformed = fft(inTimeDomainComplex,-1,1); // Optimized
  clock_t end = clock();
  double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;

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

  ////////////////////////////////// Section 4 ////////////////////////////////////
  ////////////////////////////////// Display Section //////////////////////////////
  ///////////////////////////////////////////////////////////////////////////////
  for (unsigned int i = 0; i < fourierTransformed.size(); i++){
    cout << fourierTransformed[i] << endl;  // Display all FFT calculated data
  }
  cout << "\n\nTime elapsed to calculate FFT : " << elapsed_secs << " seconds"<<
    endl;
  cout <<"\nTotal length of of data :"<< count << endl;
  getchar();
  return 0;
}
```

Listing 4.2: 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

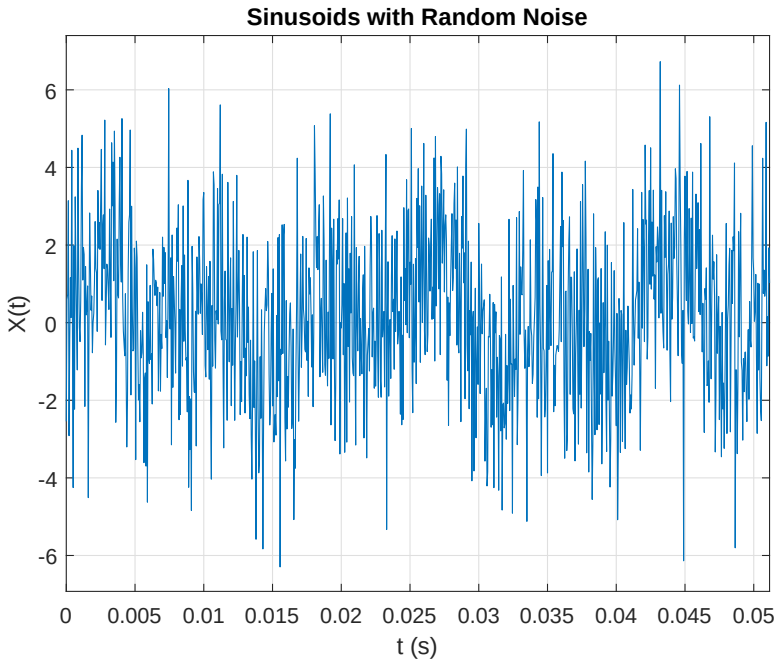


Figure 4.6: Random noise and two sinusoids

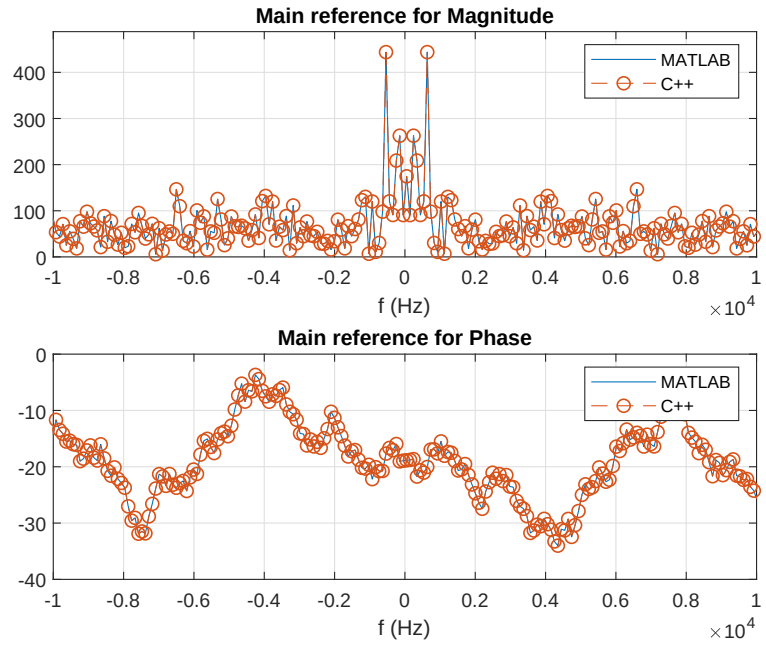


Figure 4.7: MATLAB and C++ comparison

2. Sinusoid with an exponent

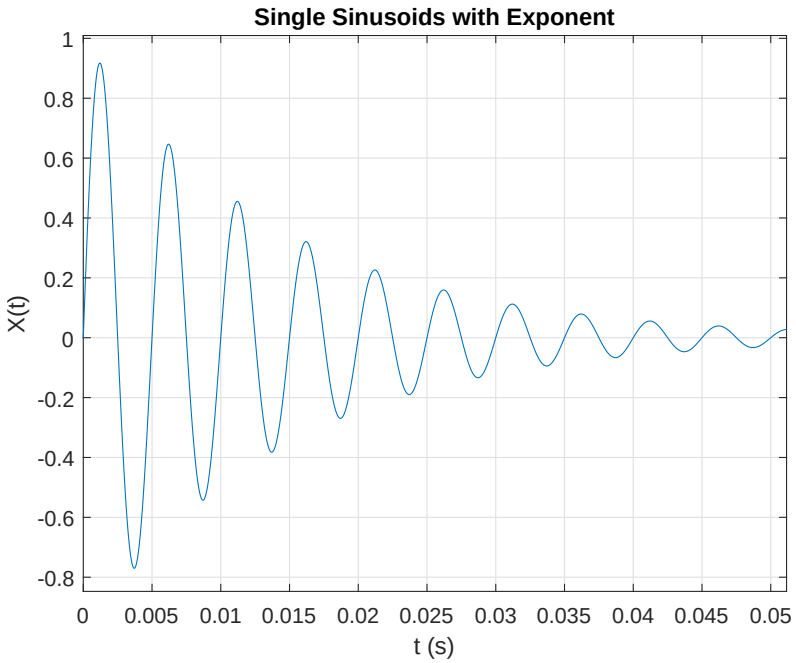


Figure 4.8: Sinusoids with exponent

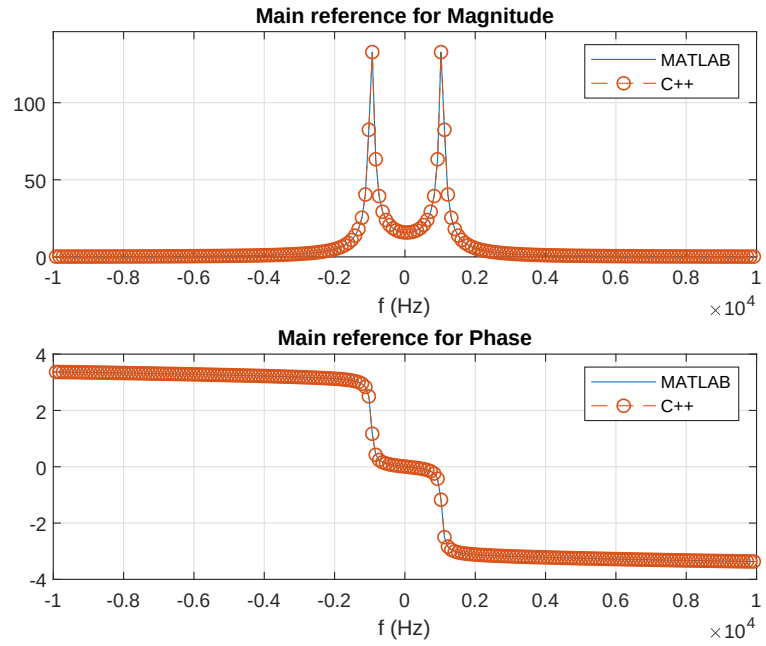


Figure 4.9: MATLAB and C++ comparison

3. Mixed signal

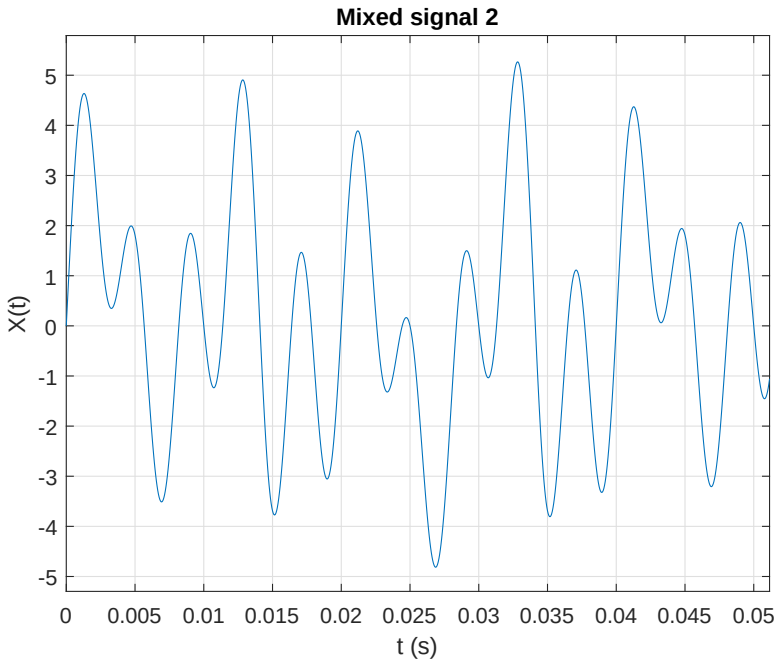


Figure 4.10: mixed signal

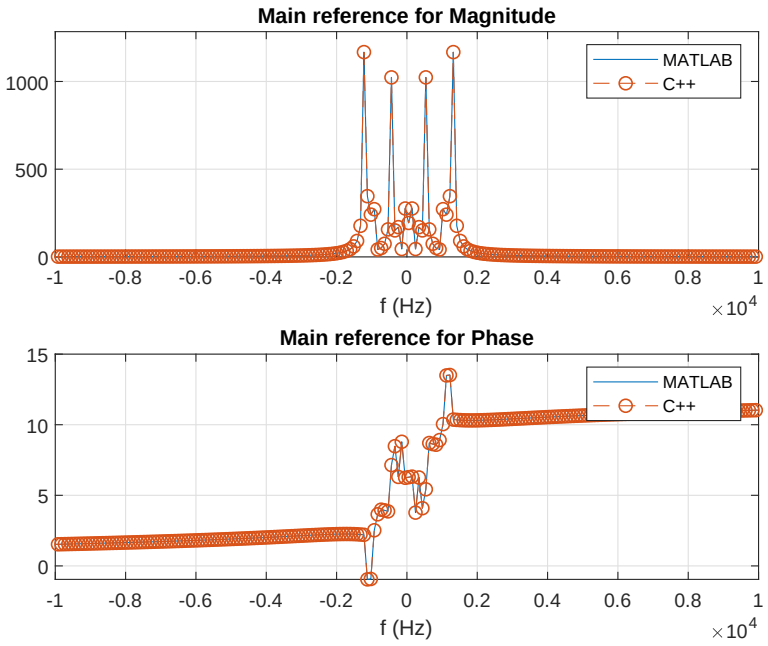


Figure 4.11: MATLAB and C++ comparison

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.

Optimized FFT

Algorithm

The algorithm for the optimized FFT will be implemented according with the following expression,

$$X_k = \sum_{n=0}^{N-1} x_n \ e^{mi2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{4.4}$$

Similarly, for IFFT,

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \ e^{mi2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{4.5}$$

where, X_k is the Fourier transform of x_n, and m equals -1 or 1 for FFT and IFFT, respectively.

Function description

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

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

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

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

If we manipulate the optimized FFT and IFFT functions as $y = fft(x, -1, 0)$ and $x = fft(y, 1, 0)$ then it'll calculate the FFT and IFFT as discussed in equation 4.1 and 4.2 respectively.

Comparative analysis

The following table displays the comparative analysis of time elapsed by FFT and optimized FFT for the various length of the data sequence. This comparison performed on the computer having configuration of 16 GB RAM, i7-3770 CPU @ 3.40GHz with 64-bit Microsoft Windows 10 operating system.

Length of data	Optimized FFT	FFT	MATLAB
2^{10}	0.011 s	0.012 s	0.000485 s
$2^{10} + 1$	0.174 s	0.179 s	0.000839 s
2^{15}	0.46 s	0.56 s	0.003470 s
$2^{15} + 1$	6.575 s	6.839 s	0.004882 s
2^{18}	4.062 s	4.2729 s	0.016629 s
$2^{18} + 1$	60.916 s	63.024 s	0.018992 s
2^{20}	18.246 s	19.226 s	0.04217 s
$2^{20} + 1$	267.932 s	275.642 s	0.04217 s

4.2 Overlap-Save Method

Header File	:	overlap_save_*.h
Source File	:	overlap_save_*.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 [3, 4] :
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,

$$y_i(n) = x_i(n) \circledast h(n). \tag{4.6}$$

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 4.12 it is illustrated an example of overlap-save method.

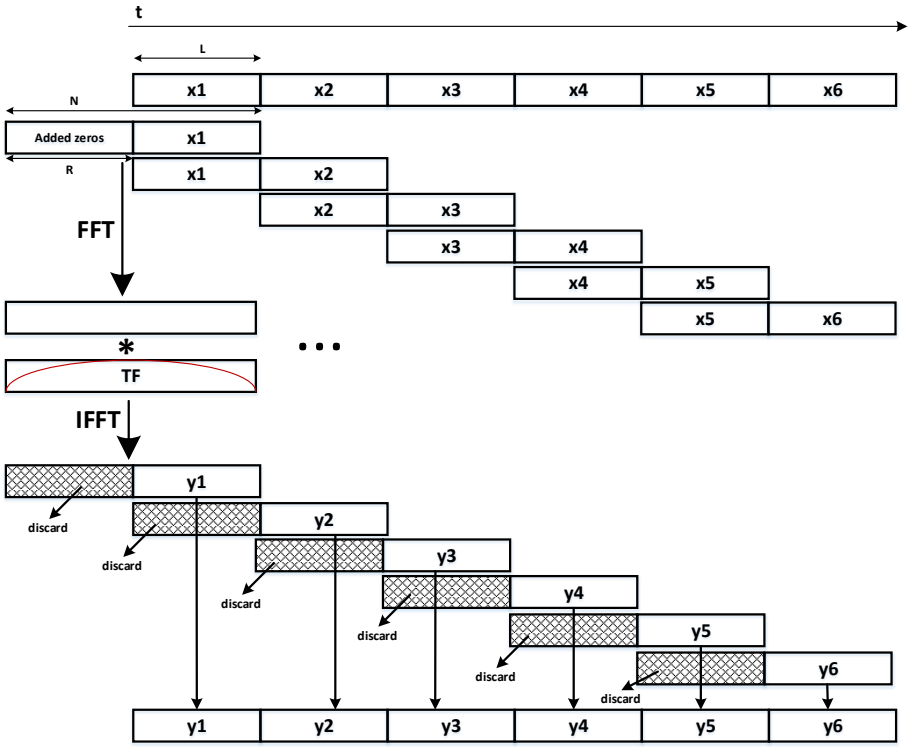


Figure 4.12: Illustration of Overlap-save method.

Function description

Traditionally, overlap-save method performs the convolution (More precisely, circular convolution) between discrete time-domain signal $x(n)$ and the filter impulse response $h(n)$. Here the length of the signal $x(n)$ is greater than the length of the filter $h(n)$. To perform convolution between the time domain signal $x(n)$ with the filter $h(n)$, include the header file overlap_save_*.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.
The one noticeable thing in the traditional way of implementation of overlap-save is that it cannot work with the real-time system. Therefore, to make it usable in the real-time environment, one more $overlapSave$ function with three input parameters was implemented and used along with the traditional overlap-save method. The structure of the new function is as follows,

$$y(n) = overlapSave(x_m(n), x_{m-1}(n), h(n))$$

Here, $x_m(n)$, $x_{m-1}(n)$ and $h(n)$ are of the C++ type vector< complex<double> > and the length of each of them are arbitrary. However, the combined length of $x_{m-1}(n)$ and $x_m(n)$ must be greater than the length of $h(n)$.

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 signal $x(n)$ 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 real-time overlap-save method

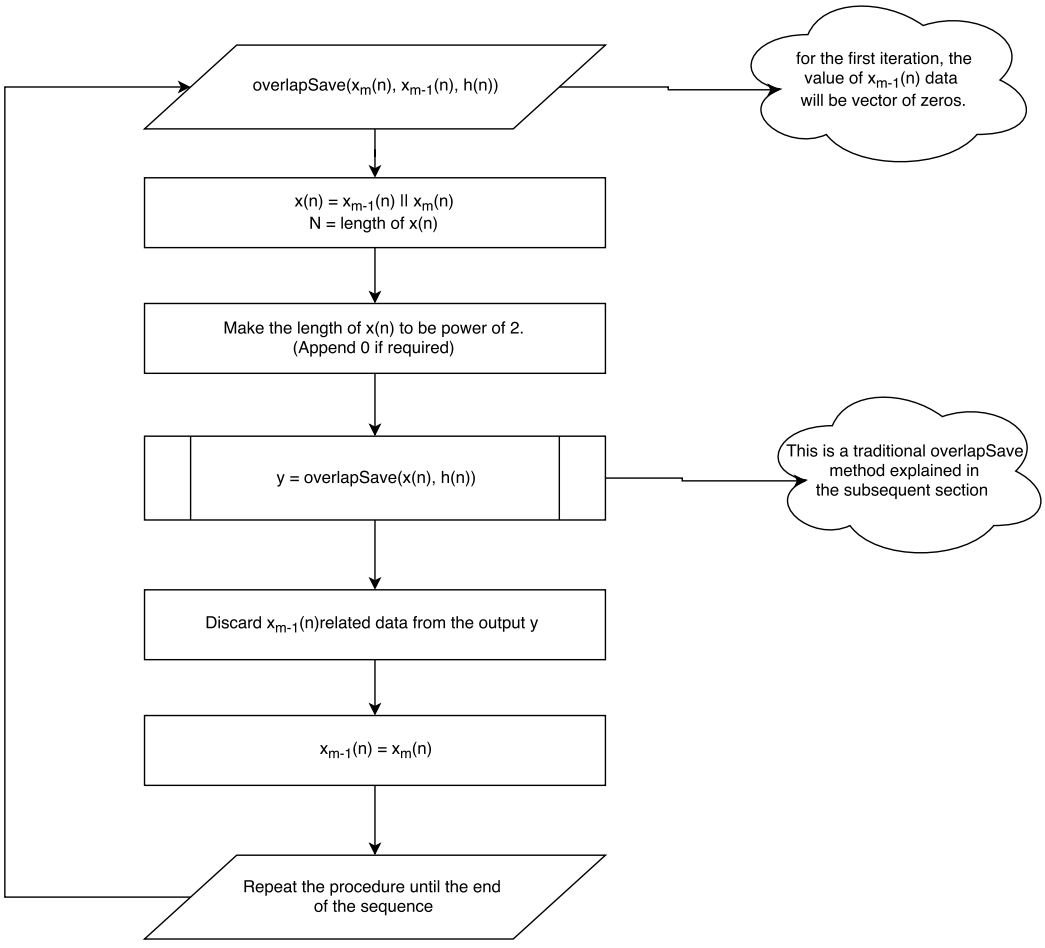


Figure 4.13: Flowchart for real-time overlap-save method

Flowchart of traditional overlap-save method

The following three flowcharts describe the logical flow of the traditional overlap-save method with two inputs as $overlapSave(x(n), h(n))$. In the flowchart, $x(n)$ and $h(n)$ are regarded as *inTimeDomainComplex* and *inTimeDomainFilterComplex* respectively.

1. Decide length of FFT, data block and filter

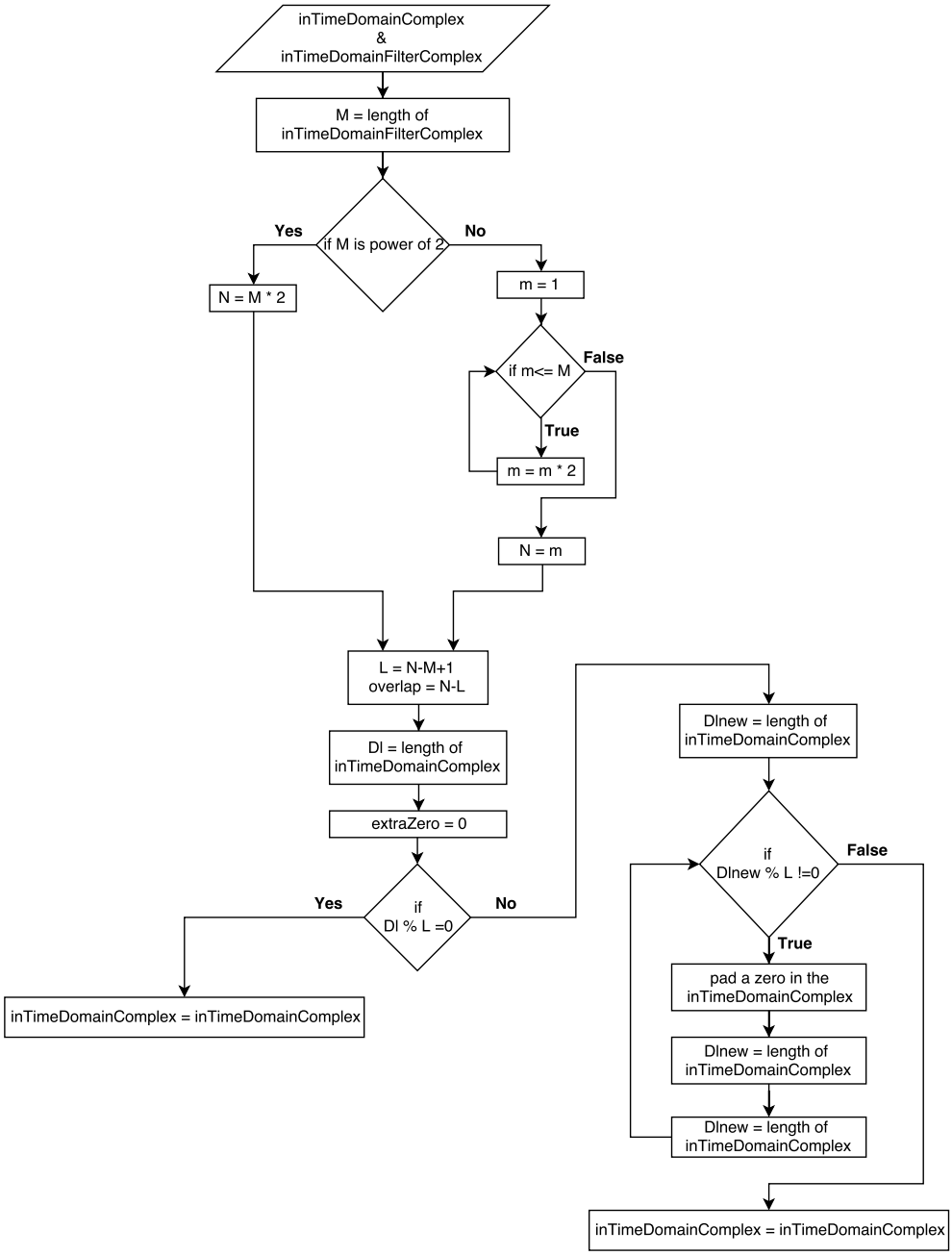


Figure 4.14: Flowchart for calculating length of FFT, data block and filter

2. Create matrix with overlap

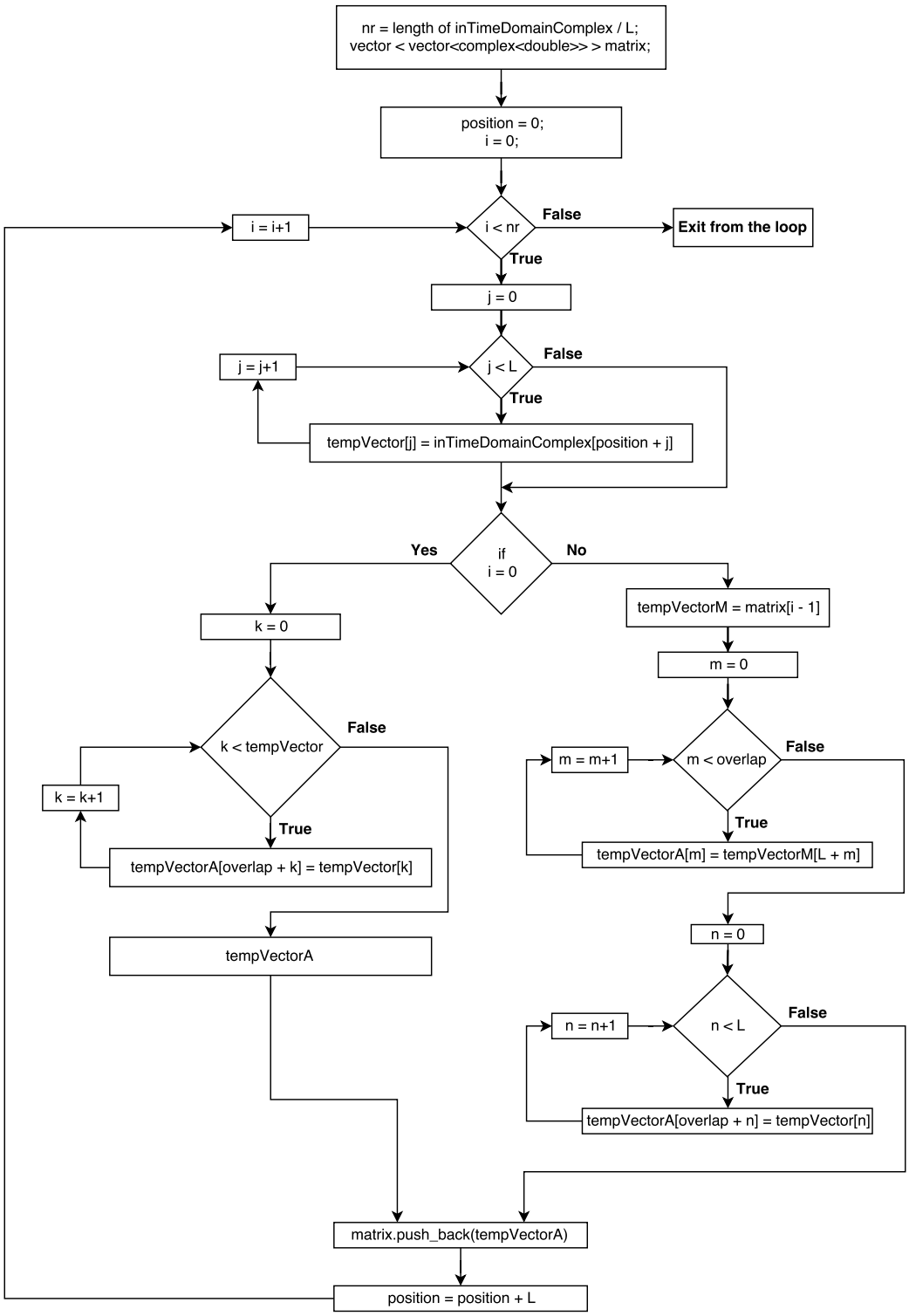


Figure 4.15: Flowchart of creating matrix with overlap

3. Convolution between filter and data blocks

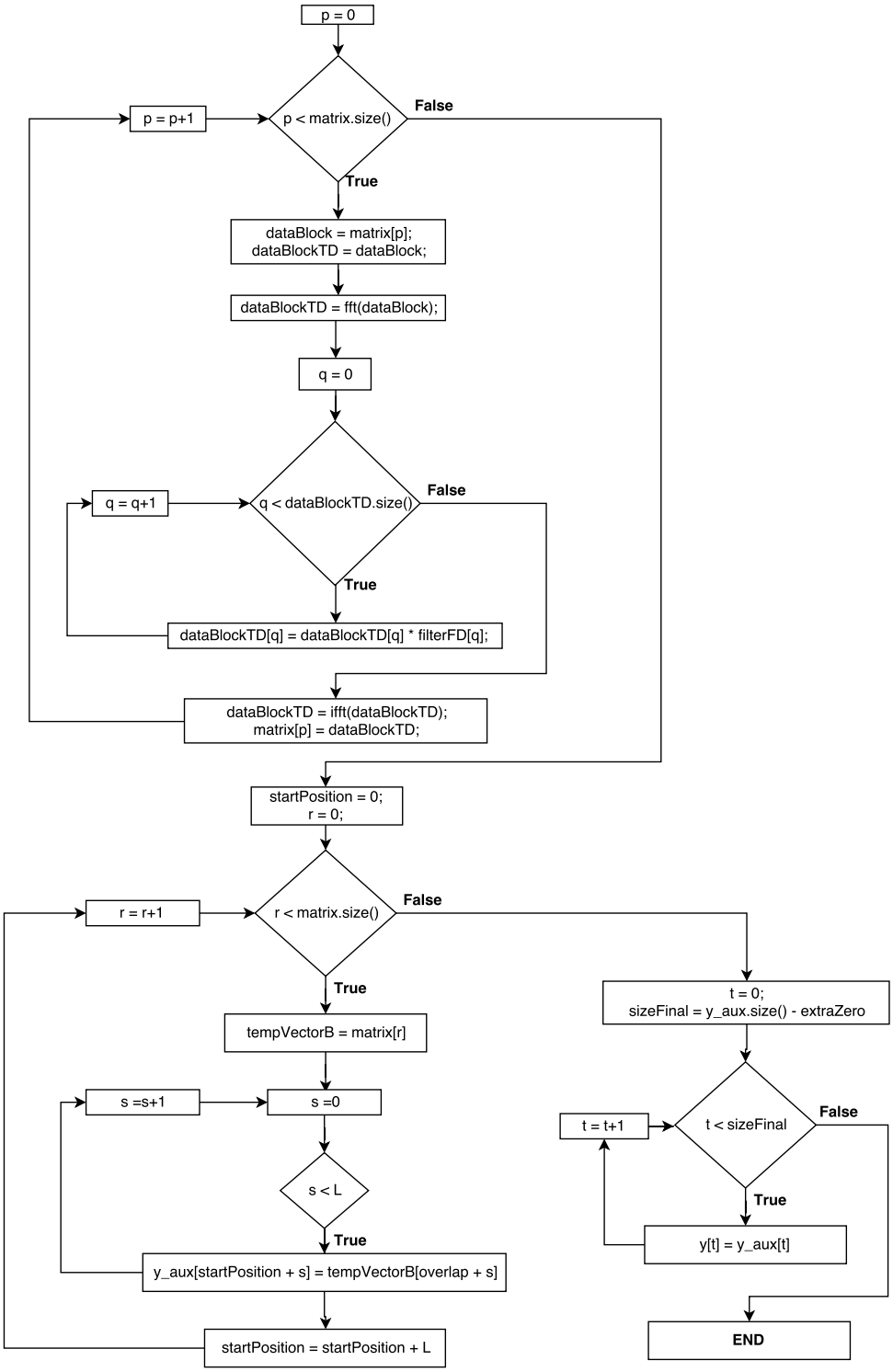


Figure 4.16: Flowchart of the convolution

Test example of traditional overlap-save function

This sections explains the steps of comparing our C++ based $overlapSave(x(n), h(n))$ function with the MATLAB overlap-save program and MATLAB's built-in conv() function.

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 = 2^10;                   % 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 = 5;
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(-70*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 (s)')
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)]);
xlabel('t (Seconds)')
ylabel('y(t)')
title('Comparision of overlapSave method of MATLAB and C++ ')
legend('MATLAB overlapSave','C++ overlapSave')
grid on
%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 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('y(t)')
legend('MATLAB function : conv(X,h)','C++ overlapSave')
grid on
```

Listing 4.3: 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 "overlap_save_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;
```

```
15 int main()
   {
17   ////////////////////////////// Section 1 ////////////////////////////////
     ////// Read the time_domain_data.txt and time_domain_filter.txt files  ///////
19   ////////////////////////////////////////////////////////////////////////////
     ifstream inFile;
21   double ch;
     vector <double> inTimeDomain;
23   inFile.open("time_domain_data.txt");

25   // First data (at 0th position) appled to the ch it is similar to the "cin".
     inFile >> ch;
27
     // It'll count the length of the vector to verify with the MATLAB
29   int count = 0;

31   while (!inFile.eof())
     {
33     // push data one by one into the vector
       inTimeDomain.push_back(ch);
35
       // it'll increase the position of the data vector by 1 and read full vector.s
37     inFile >> ch;
       count++;
39   }

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

43   ifstream inFileFilter;
     double chFilter;
45   vector <double> inTimeDomainFilter;
     inFileFilter.open("time_domain_filter.txt");
47   inFileFilter >> chFilter;
     int countFilter = 0;
49
     while (!inFileFilter.eof())
51   {
       inTimeDomainFilter.push_back(chFilter);
53     inFileFilter >> chFilter;
       countFilter++;
55   }
     inFileFilter.close();
57
     ////////////////////////////// Section 2 ////////////////////////////////
59   ////////////////////////// Real to complex conversion //////////////////////
     ////////////////////////////////////////////////////////////////////////////
61   ////////////// For signal data  //////////////
     vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
63   vector <complex<double>> fourierTransformed;
     vector <double> re(inTimeDomain.size());
65   vector <double> im(inTimeDomain.size());
```

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

83    for (unsigned int i = 0; i < inTimeDomainFilter.size(); i++)
      {
85      reFilter[i] = inTimeDomainFilter[i];
        imFilter[i] = 0;
87    }
      inTimeDomainFilterComplex = reImVect2ComplexVector(reFilter, imFilter);
89
      ////////////////////////////////// Section 3 ////////////////////////////////
91    ////////////////////////////// Overlap & save ///////////////////////////////
      ////////////////////////////////////////////////////////////////////////////
93    vector <complex<double>> y;
      y = overlapSave(inTimeDomainComplex, inTimeDomainFilterComplex);
95
      ////////////////////////////////// Section 4 ////////////////////////////////
97    ////////////////////////////////// Save data ////////////////////////////////
      ////////////////////////////////////////////////////////////////////////////
99    ofstream outFile;
      complex<double> outFileData;
101   outFile.precision(20);
      outFile.open("overlap_save_data.txt");
103
      for (unsigned int i = 0; i <y.size(); i++)
105   {
        outFile << y[i].real() << endl;
107     outFile << y[i].imag() << endl;
      }
109   outFile.close();

111   cout << "Execution finished! Please hit enter to exit." << endl;
      getchar();
113   return 0;
}
```

Listing 4.4: 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

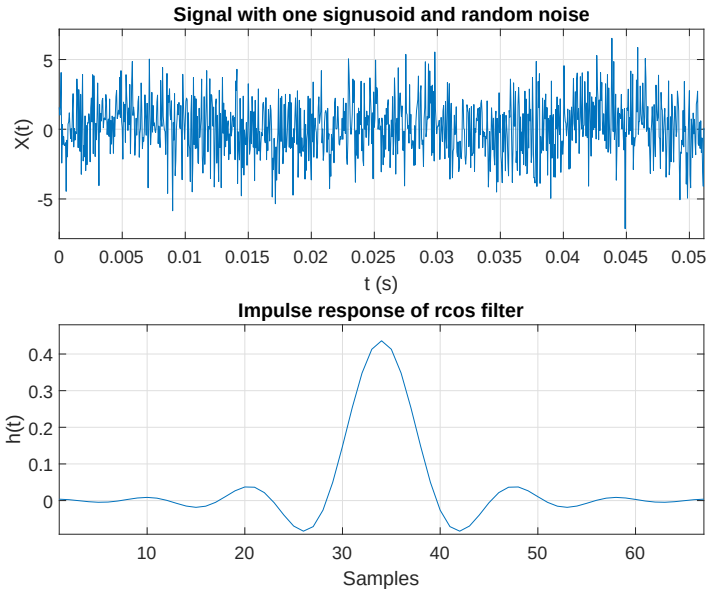


Figure 4.17: Random noise and two sinusoids signal & Impulse response of rcos filter

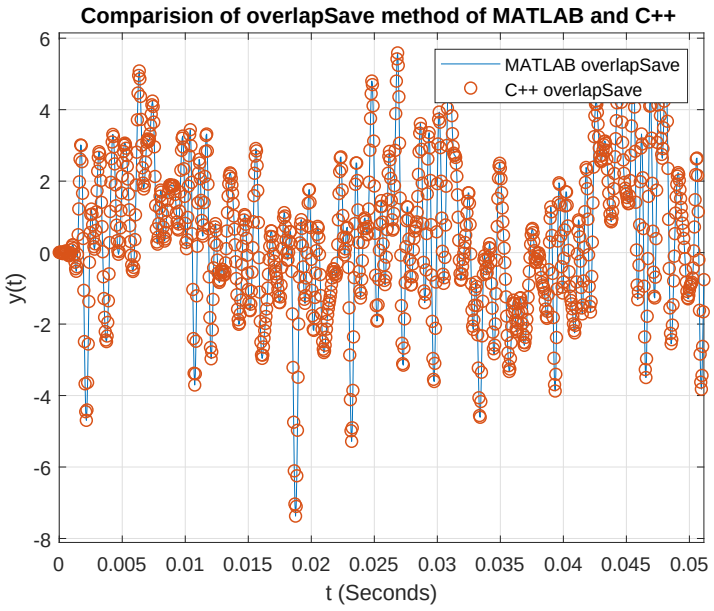


Figure 4.18: MATLAB and C++ comparison

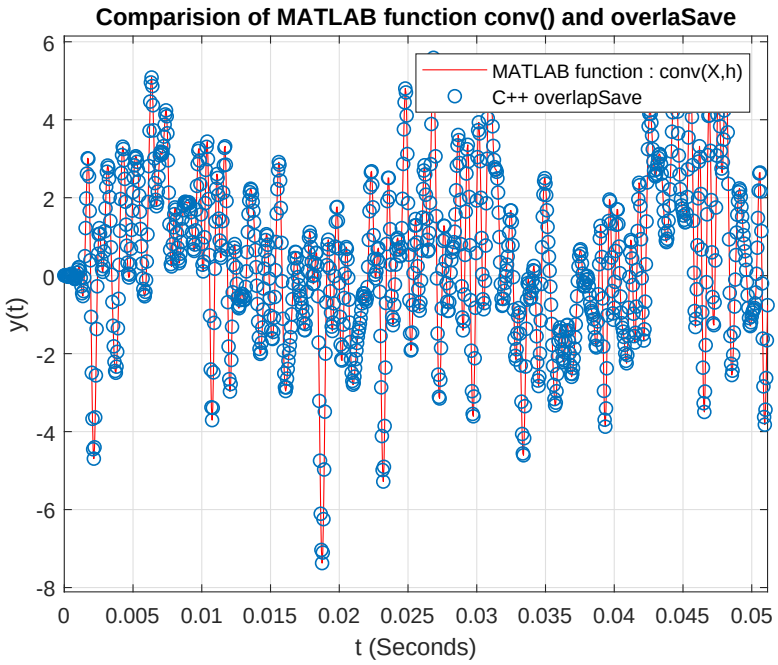


Figure 4.19: MATLAB function conv() and C++ overlapSave comparison

2. Mixed signal2

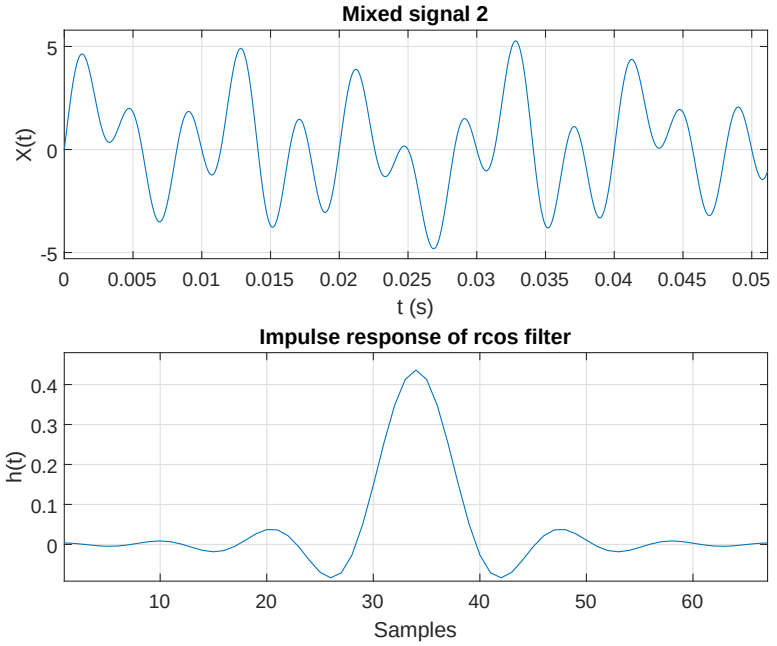


Figure 4.20: Mixed signal2 & Impulse response of rcos filter

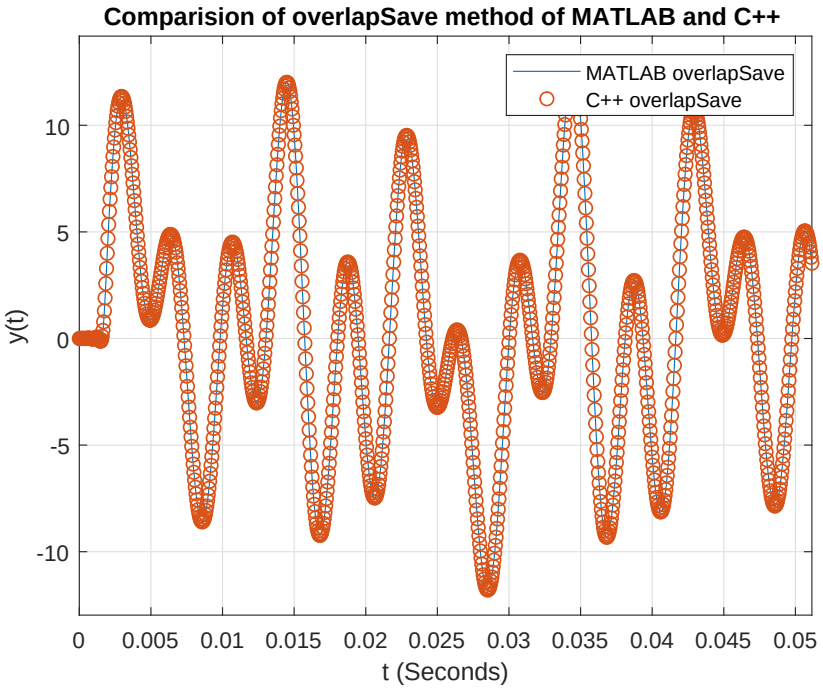


Figure 4.21: MATLAB and C++ comparison

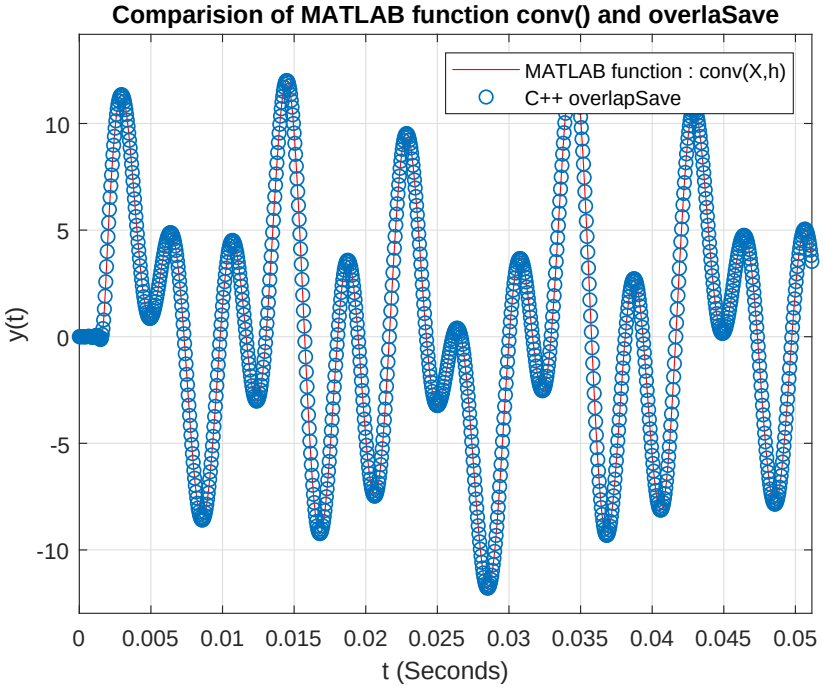


Figure 4.22: MATLAB function conv() and C++ overlapSave comparison

3. Sinusoid with exponent

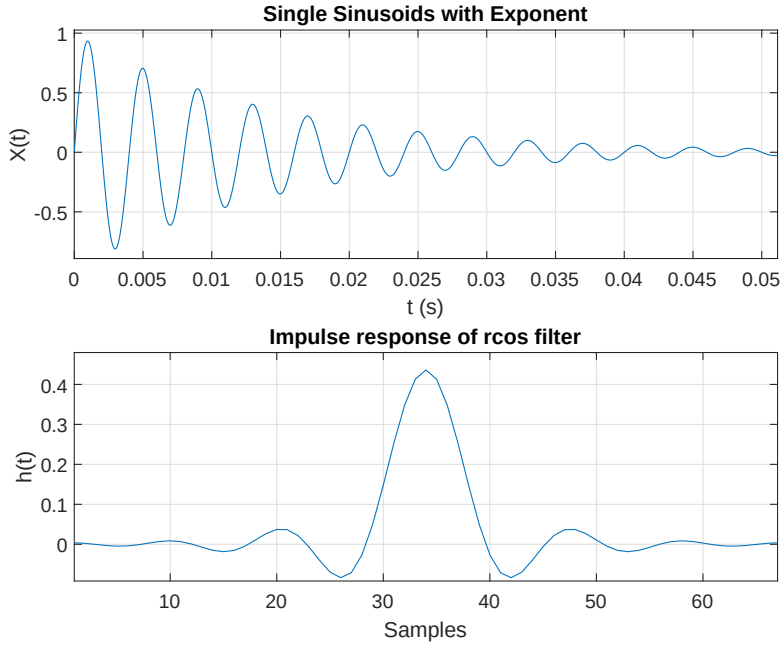


Figure 4.23: Sinusoid with exponent & Impulse response of Gaussian filter

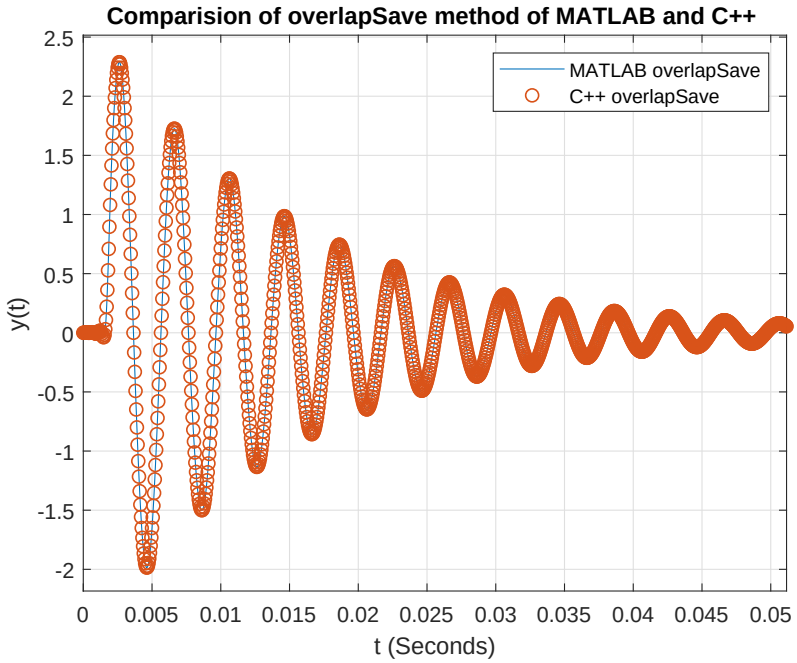


Figure 4.24: MATLAB and C++ comparison

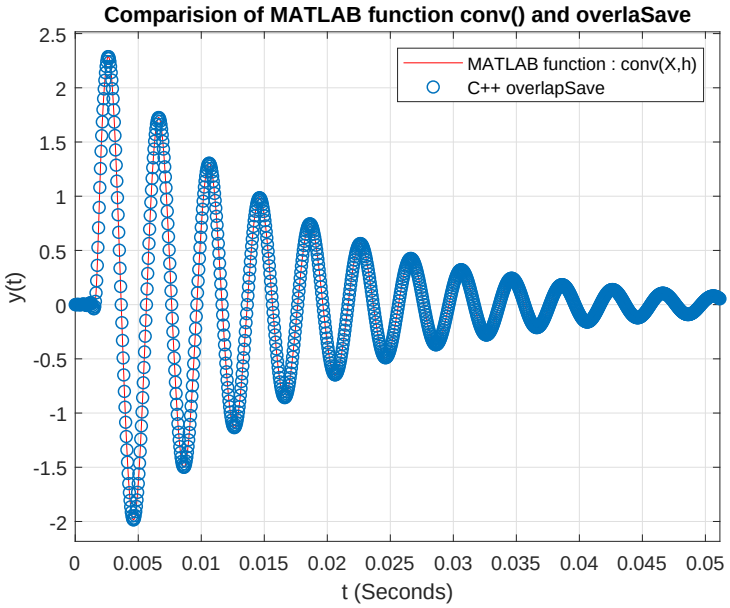


Figure 4.25: MATLAB function conv() and C++ overlapSave comparison

Test example of real-time overlap-save function with Netxpto simulator

This section explains the steps of comparing real-time overlap-save method with the time-domain filtering. The structure of the real-time overlap-save function $overlapSave(x_{m-1}(n), x_m(n), h(n))$ requires an impulse response $h(n)$ of the filter. There are two methods to feed the impulse response to the real-time overlap-save function:
Method 1. The impulse response $h(n)$ of the filter can be fed using the time-domain impulse response formula of the filter.
Method 2. Write the transfer function of the filter and convert it into the impulse response using Fourier transform method.
Here, this example uses the method 2 to feed the impulse response of the filter. In order to compare the result, follow the steps given below:

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

Step 2 : Find the **overlapSaveRealTime_test.vcxproj** file and open it.
In this project file, find *filter_20180306.cpp* in $SourceFiles$ section and click on it. This file includes the several definitions of the two different filter class namely **FIR_Filter** and **FD_Filter** for filtering in time-domain and frequency-domain respectively. In this file, **FD_Filter::runBlock** displays the logic of real-time overlap-save method.

```
# include "filter_20180306.h"
2
```

```
////////////////////////////////////////////////////////////
////////////////////// FIR_Filter ////////////////////////// TIME DOMAIN
////////////////////////////////////////////////////////////

void FIR_Filter::initializeFIR_Filter(void) {

  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

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

  delayLine.resize(impulseResponseLength, 0);

  if (saveImpulseResponse) {
    ofstream fileHandler("./signals/" + impulseResponseFilename, ios::out);
    fileHandler << "// ### HEADER TERMINATOR ###\n";

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

};


bool FIR_Filter::runBlock(void) {

  int ready = inputSignals[0]->ready();
  int space = outputSignals[0]->space();
  int process = min(ready, space);
  if (process == 0) return false;

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

```
    delayLine[impulseResponseLength - 1] = 0.0;
  }

  return true;
};


////////////////////////////////////////////////////////////
////////////////////////// FD_Filter ////////////////////////// FREQUENCY DOMAIN
////////////////////////////////////////////////////////////
void FD_Filter::initializeFD_Filter(void)
{
  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

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

  if (saveTransferFunction)
  {
    ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
    fileHandler << "// ### HEADER TERMINATOR ###\n";

    double samplingPeriod = inputSignals[0]->samplingPeriod;
    t_real fWindow = 1 / samplingPeriod;
    t_real df = fWindow / transferFunction.size();

    t_real f;
    for (int k = 0; k < transferFunction.size(); k++)
    {
      f = -fWindow / 2 + k * df;
      fileHandler << f << " " << transferFunction[k] << "\n";
    }
    fileHandler.close();
  }
}

bool FD_Filter::runBlock(void)
{
  bool alive{ false };

  int ready = inputSignals[0]->ready();
  int space = outputSignals[0]->space();
  int process = min(ready, space);
  if (process == 0) return false;

  ////////////////////// previousCopy & currentCopy ////////////////////
  //////////////////////////////////////////////////////////////////////
  vector<double> re(process); // Get the Input signal
```

```
  t_real input;
  for (int i = 0; i < process; i++){
    inputSignals[0]->bufferGet(&input);
    re.at(i) = input;
  }

  vector<t_real> im(process);
  vector<t_complex> currentCopyAux = reImVect2ComplexVector(re, im);

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

  // size modification of currentCopyAux to currentCopy.
  vector<t_complex> currentCopy(previousCopy.size());
  for (unsigned int i = 0; i < currentCopyAux.size(); i++){
    currentCopy[i] = currentCopyAux[i];
  }

  /////////////////////////// Filter Data "hn" ///////////////////////////
  ////////////////////////////////////////////////////////////////////////
  vector<t_complex> impulseResponse;
  impulseResponse = transferFunctionToImpulseResponse(transferFunction);
  vector<t_complex> hn = impulseResponse;

  ////////////////////// OverlapSave in Realtime //////////////////////
  ////////////////////////////////////////////////////////////////////
  vector<t_complex> OUTaux = overlapSave(currentCopy, previousCopy, hn);

  previousCopy = currentCopy;
  K = K + 1;

  // Remove the size modified data (opppsite to "currentCopyAux to currentCopy")
  vector<t_complex> OUT;
  for (int i = 0; i < process; i++){
    OUT.push_back(OUTaux[previousCopy.size() + i]);
  }

  // Bufferput
  for (int i = 0; i < process; i++){
    t_real val;
    val = OUT[i].real();
    outputSignals[0]->bufferPut((t_real)(val));
  }

  return true;
}
```

Listing 4.5: filter_20180306.cpp code

Step 3 : Next, open **overlapSaveRealTime_test.cpp** file in the same project and run it. Graphically, this files represents the following Figure 4.26.

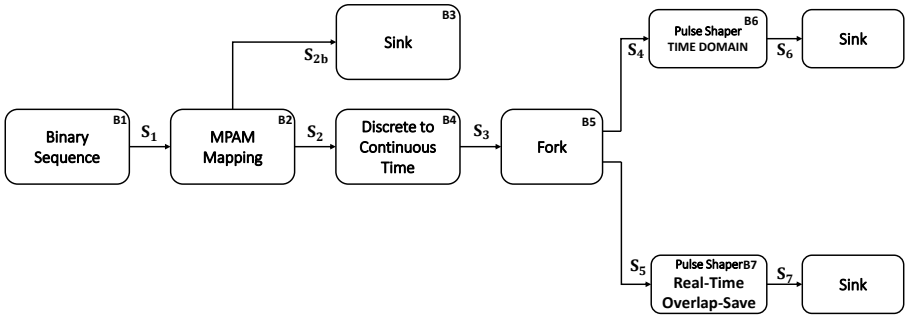


Figure 4.26: Real-time overlap-save example setup

Step 4 : Open the MATLAB visualizer and compare the signal **S6.sgn** and **S7.sgn** as shown in Figure 4.27.

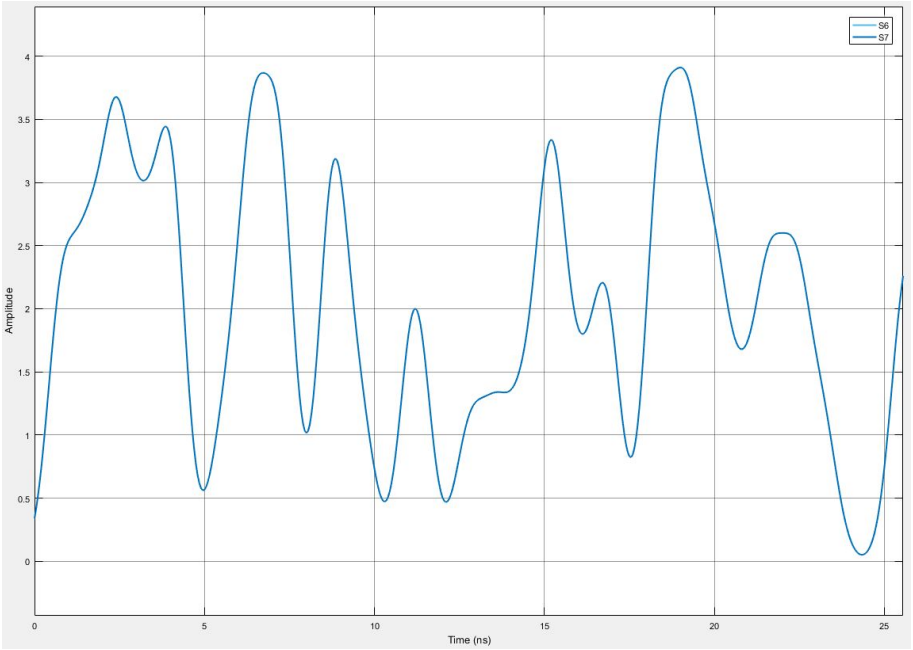


Figure 4.27: Comparison of signal S6 and S7

4.3 Filter

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

In order to filter any signal, a new generalized version of the filter namely $filter\_*.h$ & $filter\_*.cpp$ is programmed which facilitate to filtering in both time and frequency domain. Basically, $filter\_*.h$ file contains the declaration two distinct class namely **FIR_Filter** and **FD_Filter** which help to perform filtering in time-domain (using impulse response) and frequency-domain (using transfer function), respectively (see Figure 4.28). The $filter\_*.cpp$ file contains the definitions of all the functions declared in the **FIR_Filter** and **FD_Filter**.
In the Figure 4.28, the function **bool** $runblock(void)$ in the transfer function based **FD_Filter**

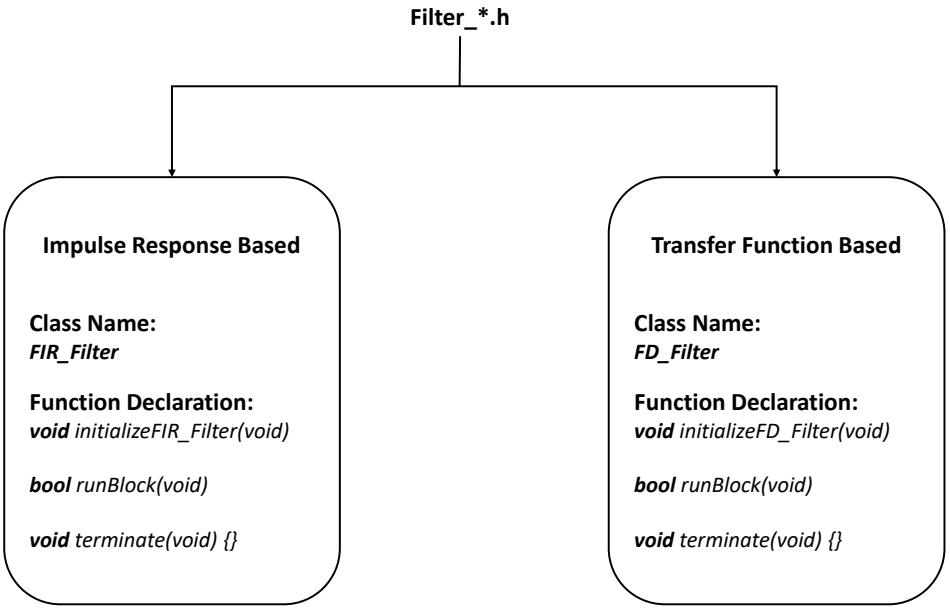


Figure 4.28: Filter class

class, is the declaration of the real-time overlap-save method for filtering in the frequency domain. On the other hand, the function **bool** $runblock(void)$ in the **FIR_Filter** class is the declaration of the function to facilitate filtering in the time domain [5, 6] .
All those function declared in the $filter\_*.h$ file are defined in the $filter\_*.cpp$ file. The definition of **bool** $runblock(void)$ function for both the classes are the following,

```
2
bool FIR_Filter::runBlock(void) {
4
```

```
  int ready = inputSignals[0]->ready();
  int space = outputSignals[0]->space();
  int process = min(ready, space);
  if (process == 0) return false;

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

  return true;
};
```

Listing 4.6: Definition of **bool FIR_Filter::runBlock(void)**

```
bool FD_Filter::runBlock(void)
{
  bool alive{ false };

  int ready = inputSignals[0]->ready();
  int space = outputSignals[0]->space();
  int process = min(ready, space);
  if (process == 0) return false;

  ////////////////////// previousCopy & currentCopy ////////////////////
  ///////////////////////////////////////////////////////////////////////
  vector<double> re(process); // Get the Input signal
  t_real input;
  for (int i = 0; i < process; i++){
    inputSignals[0]->bufferGet(&input);
    re.at(i) = input;
  }

  vector<t_real> im(process);
  vector<t_complex> currentCopyAux = reImVect2ComplexVector(re, im);

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

  // size modification of currentCopyAux to currentCopy.
  vector<t_complex> currentCopy(previousCopy.size());
  for (unsigned int i = 0; i < currentCopyAux.size(); i++){
    currentCopy[i] = currentCopyAux[i];
```

```
  }

  /////////////////////////// Filter Data "hn" ////////////////////////
  ////////////////////////////////////////////////////////////////////
  vector<t_complex> impulseResponse;
  impulseResponse = transferFunctionToImpulseResponse(transferFunction);
  vector<t_complex> hn = impulseResponse;

  ////////////////////// OverlapSave in Realtime ///////////////////////
  ////////////////////////////////////////////////////////////////////
  vector<t_complex> OUTaux = overlapSave(currentCopy, previousCopy, hn);

  previousCopy = currentCopy;
  K = K + 1;

  // Remove the size modified data (opppsite to "currentCopyAux to currentCopy")
  vector<t_complex> OUT;
  for (int i = 0; i < process; i++){
    OUT.push_back(OUTaux[previousCopy.size() + i]);
  }

  // Bufferput
  for (int i = 0; i < process; i++){
    t_real val;
    val = OUT[i].real();
    outputSignals[0]->bufferPut((t_real)(val));
  }

  return true;
}
```

Listing 4.7: Definition of **bool FD_Filter::runBlock(void)**

Both the class of the filter discussed above are the root class for the filtering operation in time and frequency domain. To perform filtering operation, we have to include $filter\_*.h$ and $filter\_*.cpp$ in the project. These filter root files require either *impulse response* or *transfer function* of the filter to perform filtering operation in time domain and frequency domain respectively. In the next section, we'll discuss an example of pulse shaping filtering using the proposed filter root class.

Example of pulse shaping filtering

This section explains how to use **FIR_Filter** and **FD_Filter** class for the pulse shaping using the impulse response and the transfer function, respectively and it also compares the resultant output of both methods. The impulse response for the **FIR_Filter** class will be generated by a *pulse_shaper.cpp* file and the transfer function for the **FD_Filter** will be generated by a *pulse_shaper_fd_20180306.cpp* file and applied to the **bool** $runblock(void)$ block as shown in Figure 4.29.

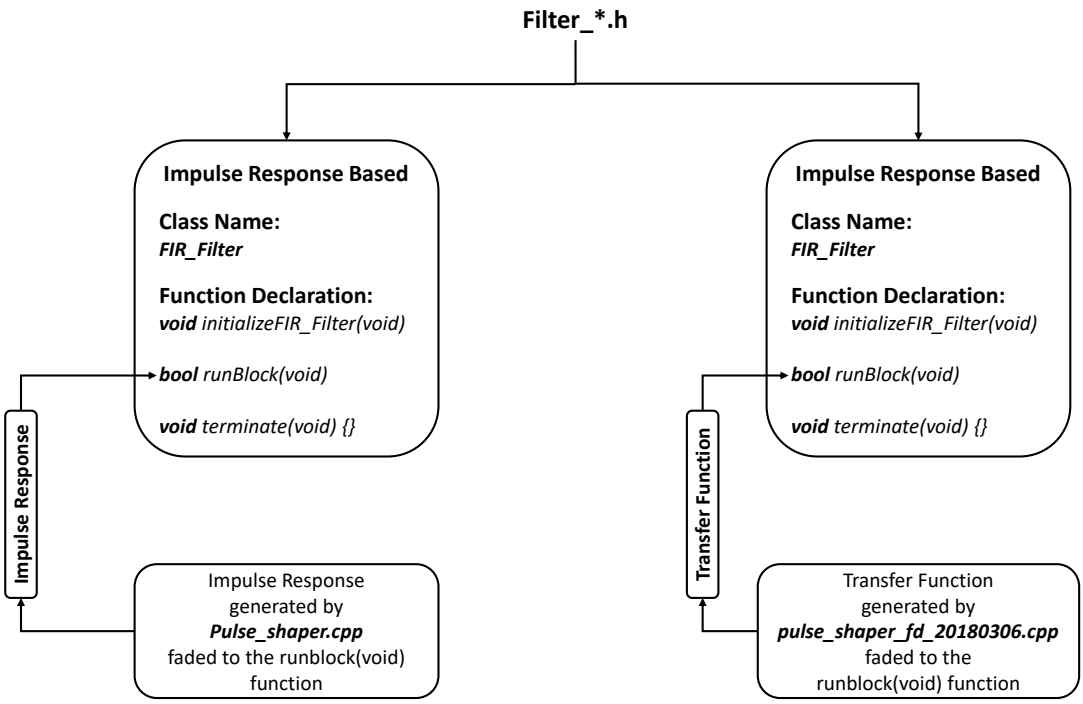


Figure 4.29: Pulse shaping using **filter_*.h**

Example of pulse shaping filtering : Procedural steps

This section explains the steps of filtering a signal with the various pulse shaping filter using its impulse response and transfer function as well. It also displays the comparison between the resultant output generated by both the methods. In order to conduct the experiment, follow the steps given below:

Step 1 : In the directory, 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 represents the simulation set-up as shown in Figure 4.30.

Step 3 : Check how **PulseShaper** and **PulseShaperFd** blocks are implemented.
Check the appendix for the various types of pulse sapping techniques and what are the different parameters used to adjust the shape of the pulse shaper.

Step 4 : Run the *filter_test.cpp* code and compare the signals **S6.sgn** and **S7.sgn** using visualizer.
Here, we have used three different types of pulse shaping filter namely, raised cosine, root raised cosine and Gaussian pulse shaper. The following Figure 4.31, 4.32 and 4.33 display

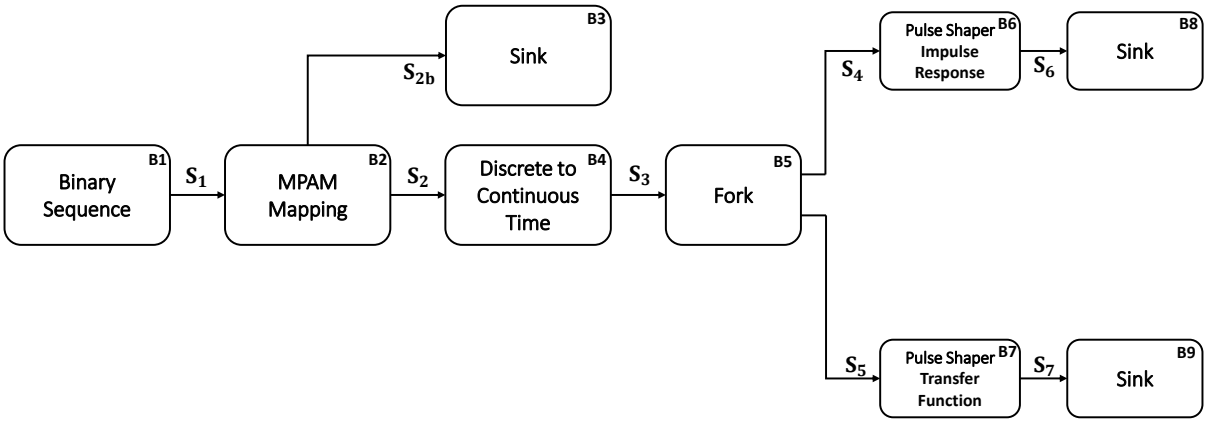


Figure 4.30: Filter test setup

the comparison of the output signals **S6.sgn** and **S7.sgn** for the raised cosine, root raised cosine and Gaussian pulse shaping filter, respectively.

Case 1 : Raised cosine

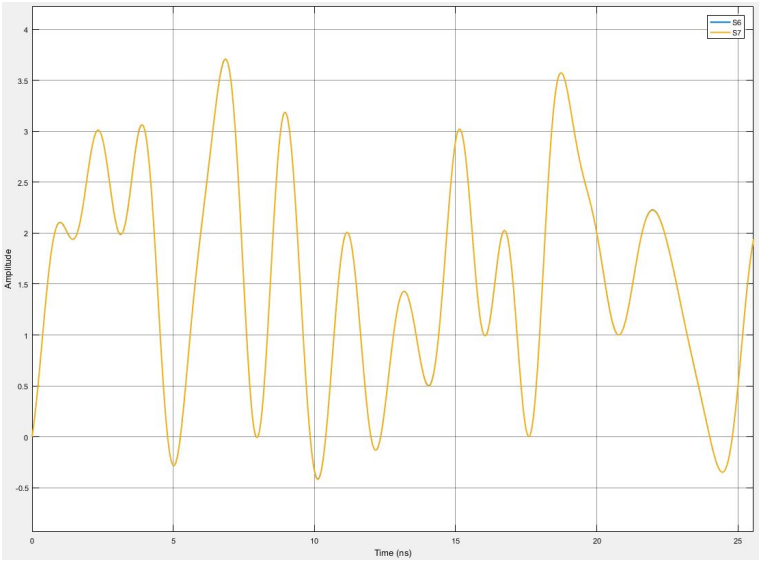


Figure 4.31: Raised cosine pulse shaping results comparison

Case 2 : Root raised cosine

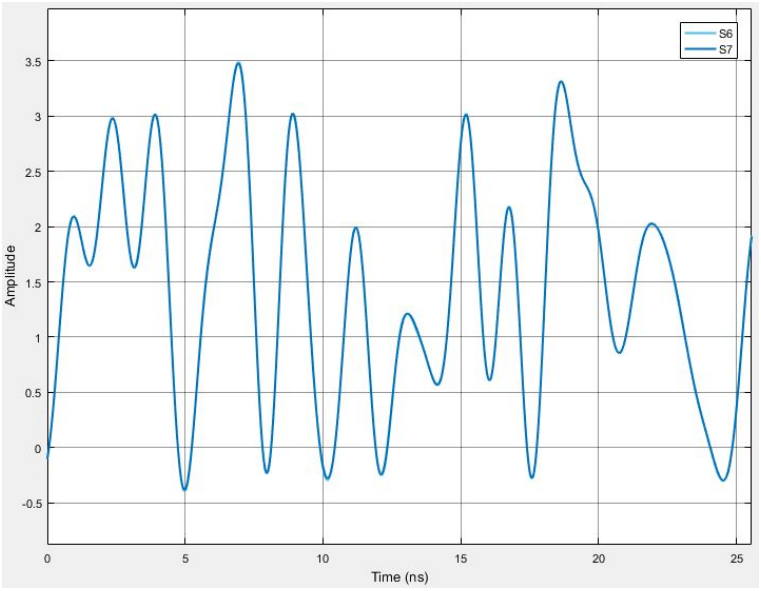


Figure 4.32: Root raised cosine pulse shaping result

Case 3 : Gaussian

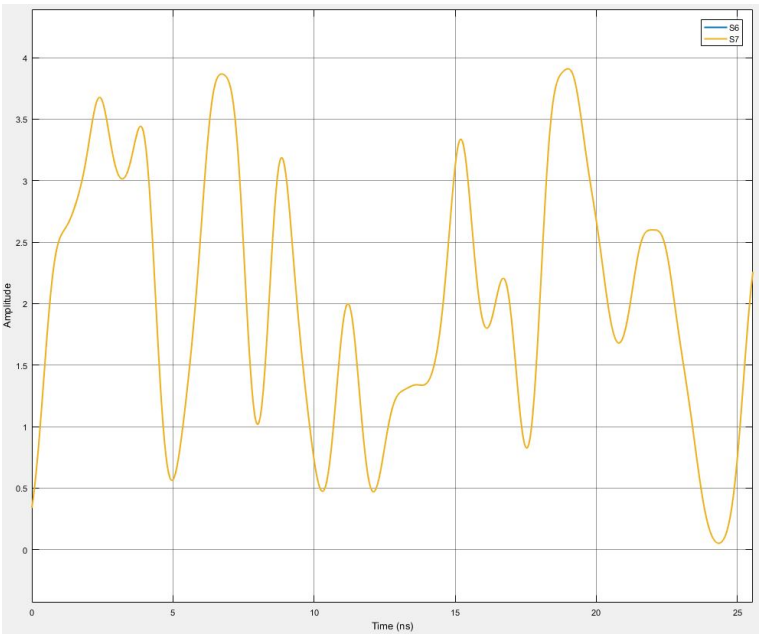


Figure 4.33: Gaussian pulse shaping results comparison

APPENDICES

A. Raised cosine pulse shaper

The raised cosine pulse shaping filter has a transfer function given by,

$$H_{RC}(f) = \begin{cases} 1 & \text{for } |f| \leq \dfrac{1-\beta}{2T_s} \\ \dfrac{1}{2}\left[1 + cos\left(\dfrac{\pi Ts}{\beta}\left[|f| - \dfrac{1-\beta}{2T_s}\right]\right)\right] & \text{for } \dfrac{1-\beta}{2T_s} < |f| \leq \dfrac{1+\beta}{2T_s} \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

The parameter, β is the roll-off factor of the raised cosine filter. The impulse response of the raised cosine filter is given by,

$$h_{RC}(t) = \frac{sin(\pi t/T_s)}{\pi t/Ts}\frac{cos(\pi\beta t/Ts)}{1 - 4\beta^2 t^2/T_s^2} \tag{4.8}$$

B. Root raised cosine pulse shaper

The raised cosine pulse shaping filter has a transfer function given by,

$$H_{RC}(f) = \begin{cases} 1 & \text{for } |f| \leq \dfrac{1-\beta}{2T_s} \\ \sqrt{\dfrac{1}{2}\left[1 + cos\left(\dfrac{\pi Ts}{\beta}\left[|f| - \dfrac{1-\beta}{2T_s}\right]\right)\right]} & \text{for } \dfrac{1-\beta}{2T_s} < |f| \leq \dfrac{1+\beta}{2T_s} \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

The parameter, β is the roll-off factor of the raised cosine filter. The impulse response of the root raised cosine filter is given by,

$$h_{RRC}(t) = \begin{cases} \dfrac{1}{T_s}\left(1 + \beta(\dfrac{4}{\pi} - 1)\right) & \text{for } t = 0 \\ \dfrac{\beta}{T_s\sqrt{2}}\left[\left(1 + \dfrac{2}{\pi}\right)sin\left(\dfrac{\pi}{4\beta}\right) + \left(1 - \dfrac{2}{\pi}\right)cos\left(\dfrac{\pi}{4\beta}\right)\right] & \text{for } t = \dfrac{T_s}{4\beta} \\ \dfrac{1}{Ts}\dfrac{sin\left[\pi\dfrac{t}{T_s}(1-\beta)\right] + 4\beta\dfrac{t}{Ts}cos\left[\pi\dfrac{t}{T_s}(1+\beta)\right]}{\pi\dfrac{t}{T_s}\left[1 - \left(4\beta\dfrac{t}{T_s}\right)^2\right]} & \text{otherwise} \end{cases} \tag{4.10}$$

C. Gaussian pulse shaper

The Gaussian pulse shaping filter has a transfer function given by,

$$H_G(f) = exp(-\alpha^2 f^2) \tag{4.11}$$

The parameter α is related to B, the 3-dB bandwidth of the Gaussian shaping filter is given by,

$$\alpha = \frac{\sqrt{ln2}}{\sqrt{2}B} = \frac{0.5887}{B} \tag{4.12}$$

From the equation 4.12, as α increases, the spectral occupancy of the Gaussian filter decreases. The impulse response of the Gaussian filter can be given by,

$$h_G(t) = \frac{\sqrt{\pi}}{\alpha} exp(-\frac{\pi^2}{\alpha^2} t^2) \tag{4.13}$$

From the equation 4.12, we can also write that,

$$\alpha = \frac{0.5887}{BT_s} T_s \tag{4.14}$$

Where, BT_s is the 3-dB bandwidth-symbol time product which ranges from $0 \leq BT_s \leq 1$ given as the input parameter for designing the Gaussian pulse shaping filter.

4.4 Hilbert Transform

Header File	:	hilbert_filter_*.h
Source File	:	hilbert_filter_*.cpp
Version	:	20180306 (Romil Patel)

What is the purpose of Hilbert transform?

The Hilbert transform facilitates the formation of analytical signal. An analytic signal is a complex-valued signal that has no negative frequency components, and its real and imaginary parts are related to each other by the Hilbert transform.

$$s_a(t) = s(t) + i\hat{s}(t) \tag{4.15}$$

where, $s_a(t)$ is an analytical signal and $\hat{s}(t)$ is the Hilbert transform of the signal $s(t)$. Such analytical signal can be used to generate Single Sideband Signal (SSB) signal.

Transfer function for the discrete Hilbert transform

There are two approached to generate the analytical signal using Hilbert transformation method. First method generates the analytical signal $S_a(t)$ directly, on the other hand, second method will generate the $\hat{s}(t)$ signal which is multiplied with i and added to the $s(t)$ to generate the analytical signal $S_a(t)$.

Method 1 :

The discrete time analytical signal $S_a(t)$ corresponding to $s(t)$ is defined in the frequency domain as [7]

$$S_a(f) = \begin{cases} 2S(f) & \text{for } f > 0 \\ S(f) & \text{for } f = 0 \\ 0 & \text{for } f < 0 \end{cases} \tag{4.16}$$

which is inverse transformed to obtain an analytical signal $S_a(t)$.

Method 2 :

The discrete time Hilbert transformed signal $\hat{s}(t)$ corresponding to $s(t)$ is defined in the frequency domain as [8]

$$\hat{S}(f) = \begin{cases} -iS(f) & \text{for } f > 0 \\ 0 & \text{for } f = 0 \\ iS(f) & \text{for } f < 0 \end{cases} \tag{4.17}$$

which is inverse transformed to obtain a Hilbert transformed signal $\hat{S}(t)$. To generate an analytical signal, $\hat{S}(t)$ is added to the $S(t)$ to get the equation 4.15.

Real-time Hilbert transform : Proposed logical flow

To understand the new proposed method, consider that the signal consists of 2048 samples and the **bufferLength** is 512. Therefore, by considering the **bufferLength**, we will process the whole signal in four consecutive blocks namely A, B, C and D; each with the length of 512 samples as shown in Figure 4.34. The filtering process will start only after acquiring first

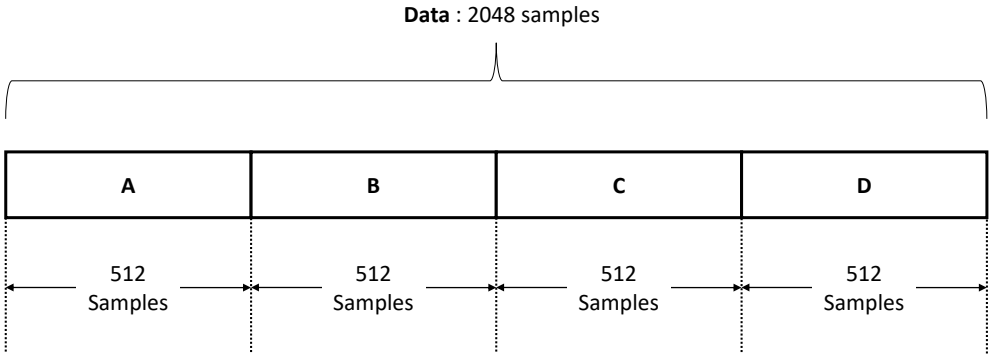


Figure 4.34: Logical flow

two blocks A and B (see iteration 1 in Figure 4.35), which introduces delay in the system. In the iteration 1, $x(n)$ consists of 512 front Zeros, block A and block B which makes the total length of the $x(n)$ is 512 x $3 = 1536$ symbols. After applying filter to the $x(n)$, we will capture the data which corresponds to the block A only and discard the remaining data from each side of the filtered output.

In the next iteration 2, we'll use **previousCopy** A and B along with the **currentCopy** "C"

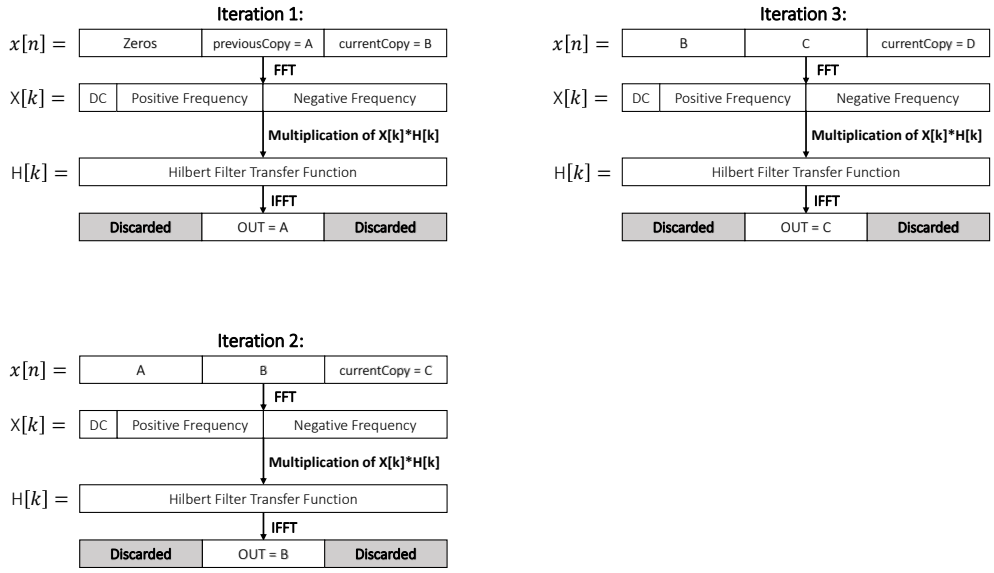


Figure 4.35: Logical flow of real-time Hilbert transform

and process the signal same as we did in iteration and we will continue the procedure until the end of the sequence.

Real-time Hilbert transform : Test setup

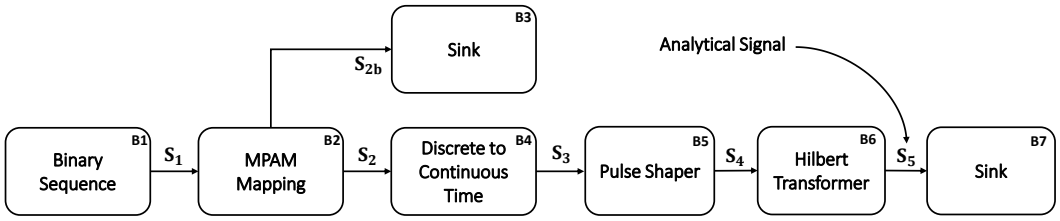


Figure 4.36: Test setup for the real time Hilbert transform

Real-time Hilbert transform : Results

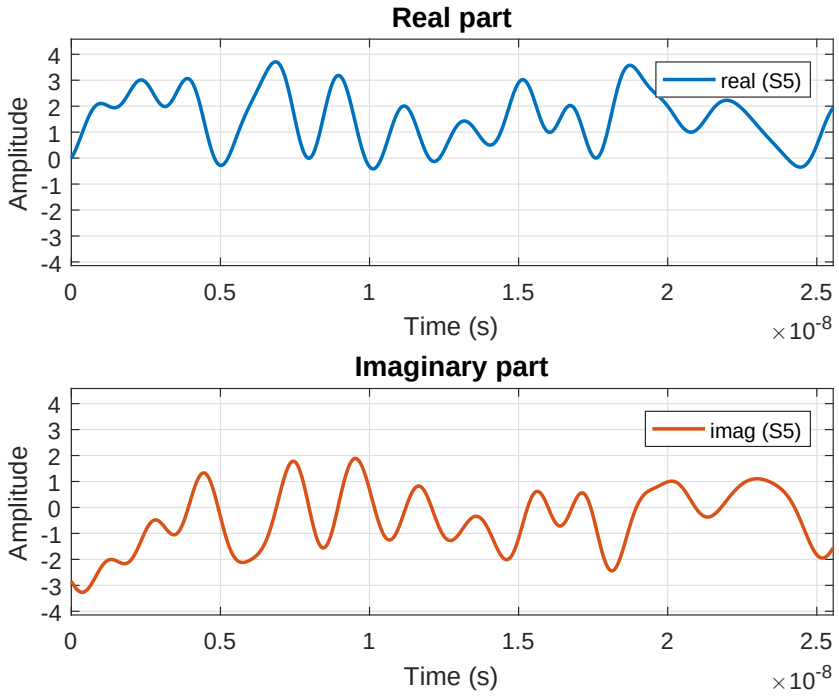


Figure 4.37: Complex analytical signal $S5$ with its real part $S4$ and imaginary part $\hat{S}4$

Remark : Here, we have used method 1 to generate analytical signal using Hilbert transform.

Chapter 5

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.

5.1 Installing Microsoft Visual C++ Build Tools

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

5.2 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.

5.3 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> /tv:14.0 /p:PlatformToolset=v140,TargetPlatformVersion=8.1,OutDir=".\"
   , where <filename> is your .vcxproj file.

After building the project, the .exe file should be automatically generated to the current folder.
The signals will be generated into the sub-directory **\signals**, which must already exist.

5.4 Known Issues

5.4.1 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.

Chapter 6

Git Helper

Git creates and maintains a database that store versions of a repository, i.e. versions of a folder. To create this database for a specific folder the Git application must be installed on the computer. Open the Git console program and go to the specific folder and execute the following command:

git init

The Git database is created and stored in the folder *.git* in the root of your repository. The Git commands allow you to manipulate this database.

6.1 Data Model

To understand Git is fundamental to understand the Git data model.

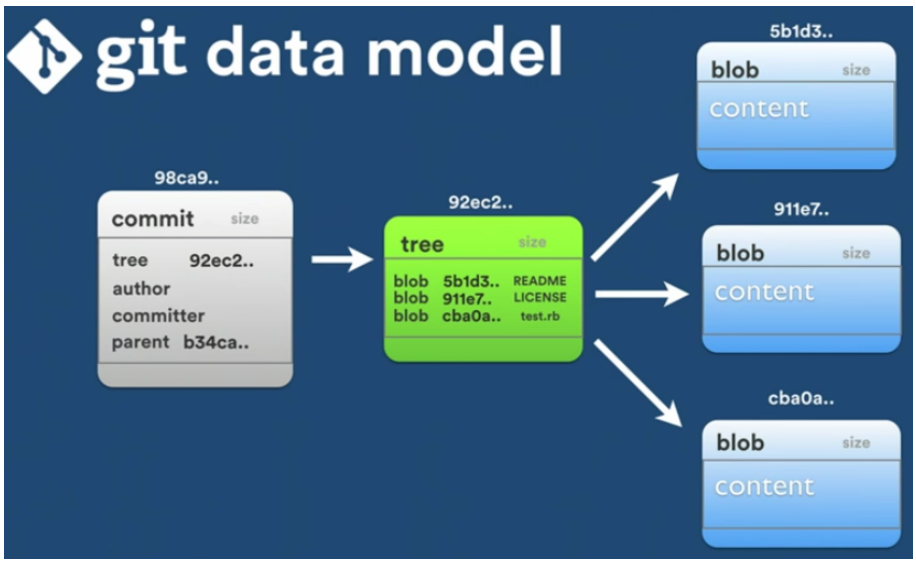


Figure 6.1: Git data model.

Git manipulates the following objects:

- commits - text files that store a description of the repository;
- trees - text files that store a description of a folder;

- blobs - the files that exist in your repository.

The objects are stored in the folder *.git/objects*. Each store object is identified by its SHA1 hash value, i.e. 20 bytes which identifies unequivocally the object. Note that 20 bytes can be represented by a 40 characters hexadecimal string. The identifier of each object is the 40 characters hexadecimal string. Each particular object is stored in a sub-folder inside the *.git/objects*. The name of the sub-folder is the two most significative characters of the SHA1 hash value. The name of the file that is inside the sub-folder is the remanning thirty eight characters of the SHA1 hash value. The Git stores all committed versions of a file. The Git maintains a contend-addressable file systems, i.e. a file system in which the files can be accessed based on its contend.

A commit object is identified by a SHA1 hash value, and has the following information: a pointer for a tree (the root of your repository), a pointer for the previous commit, the author of the commit, the committer and a commit message. The author is the person who did the work. The committer is the person who validate the work and who apply the work by doing the commit. By doing this difference git allow both to get the credit, the author and the committer. Example of a commit file contend:

tree 2c04e4bad1e2bcc0223239e65c0e6e822bba4f16
parent bd3c8f6fed39a601c29c5d101789aaa1dab0f3cd
author NetXPTO <netxpto@gmail.com> 1514997058 +0000
committer NetXPTO <netxpto@gmail.com> 1514997058 +0000

Here goes the commit message.

A tree object is identified by a SHA1 hash value, and has a list of blobs and trees that are inside that tree. Example of a tree file contend:

100644	blob	bdb0cabc87cf50106df6e15097dff816c8c3eb34	.gitattributes
100644	blob	50492188dc6e12112a42de3e691246dafdad645b	.gitignore
100644	blob	8f564c4b3e95add1a43e839de8adbfd1ceccf811	bfg-1.12.16.jar
040000	tree	de44b36d96548240d98cb946298f94901b5f5a05	doc
040000	tree	8b7147dbfdc026c78fee129d9075b0f6b17893be	garbage
040000	tree	bdfcd8ef2786ee5f0f188fc04d9b2c24d00d2e92	include
040000	tree	040373bd71b8fe2fe08c3a154cada841b3e411fb	lib
040000	tree	7a5fce17545e55d2faa3fc3ab36e75ed47d7bc02	msbuild
040000	tree	b86efba0767e0fac1a23373aaf95884a47c495c5	mtools
040000	tree	1f981ea3a52bccf1cb00d7cb6dfdc687f33242ea	references
040000	tree	86d462afd7485038cc916b62d7cbfc2a41e8cf47	sdf
040000	tree	13bfce10b78764b24c1e3dfbd0b10bc6c35f2f7b	things_to_do
040000	tree	232612b8a5338ea71ab6a583d477d41f17ebae32	visualizerXPTO
040000	tree	1e5ee96669358032a4a960513d5f5635c7a23a90	work_in_progress

A blob is identified by a SHA1 hash value, and has the file contend compressed. A git header

and tailor is added to each file and the file is compressed using the zlib library. The git header is just the file type, file size and the \NUL caracter, for instance "blob 13\NUL", the tailor is just the \n caracter. The blob is stored as a binary file.

6.2 Refs

SHA1 hash values are hard to memorize by humans. To make life easier to humans we use refs. A ref associate a name, easier to memorize by humans, with a SHA1 hash value. Therefore refs are pointers to objects. Refs are implementes by text files, the name of the file is the name of the ref and inside the file is a string with the SHA1 hash value.

There are different type of refs. Some are static, for instance the tags, others are actualized automatically, for instance the branches.

6.3 Tags

A tag is just a ref for a specific commit. A tag do not change over time.

6.4 Branch

A branch is a ref that points for a commit that is originated by a divergence from a common point. A branch is automatically actualized so that it always points for the most recent commit of that branch.

6.5 Heads

Heads is a pointer for the commit where you are.

6.6 Database Folders and Files

6.6.1 Objects Folder

Git stores the database and the associated information in a set of folders and files inside the the folder *.git* in the root of your repository.

The folder *.git/objects* stores information about all objects (commits, trees and blobs). The objects are stored in files inside folders. The name of the folders are the 2 first characters of the SHA1 40 characters hexadecimal string. The name of the files are the other 38 hexadecimal characters of the SHA1. The information is compressed to save same space but it can be access using some applications.

6.6.2 Refs Folder

The *.git/refs* folder has inside the following folders *heads*,*remotes*, and *tags*. The *heads* has inside a ref for all local branches of your repository. The *remotes* folder has inside a set of folders with the name of all remote repositories, inside each folder is a ref for all branches in that remote repository. The *tag* folder has a ref for each tag.

6.7 Git Spaces

Git uses several spaces.

- workspace - is your directories where you are working;
- index - when you record changes before commit them;
- blobs - the files that exist in your repository.

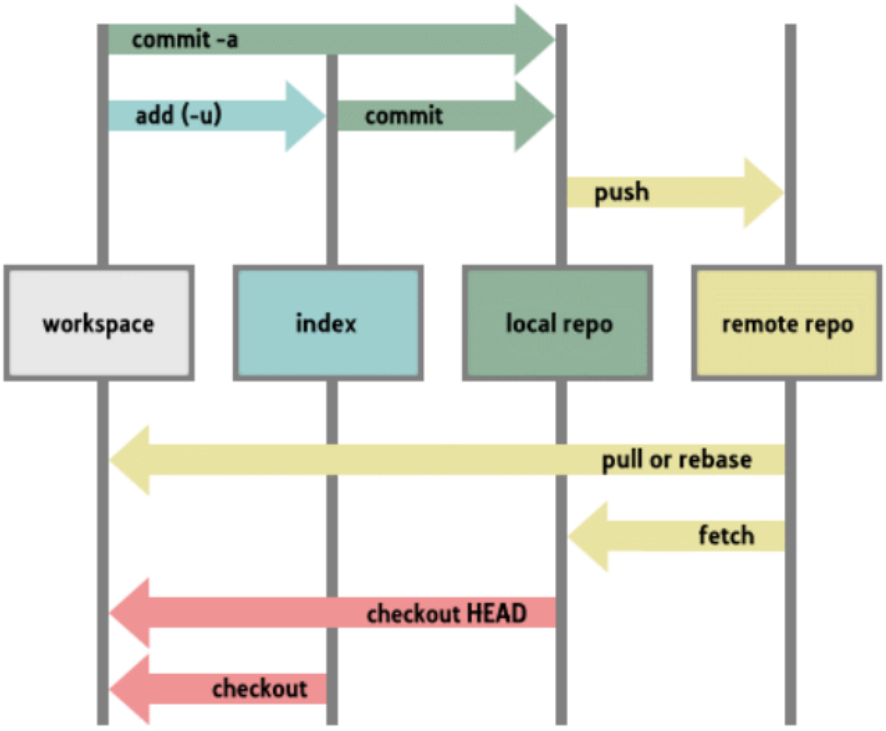


Figure 6.2: Git spaces.

6.8 Workspace

6.9 Index

6.10 Merge

Merge is a fundamental concept to git. It is the way you consolidate your work.

6.10.1 Fast-Forward Merge

6.10.2 Resolve

6.10.3 Recursive

6.10.4 Octopus

6.10.5 Ours

6.10.6 Subtree

6.10.7 Custom

6.11 Commands

6.11.1 Porcelain Commands

git add
git add, store a file that was changed to the index.
git bisect
git branch
git branch –set-upstream-to=<remote>/<branch> <local branch>, links a local branch with a branch in a remote repository.

git cat-file
git cat-file -t <hash>
shows the type of the objects
git cat-file -p <hash>
shows the contend of the object
git clone
git diff
git diff shows the changes in the working space.
git diff –name-only shows the changes in the working space, only file names.
git diff –cached shows the changes in the index space.
git diff –cached –name-only shows the changes in the index space, only file names.
git fecth -all

git init

git init, used to initialize a git repository. It creates the *.git* folder and all its subfolders and files. The subfolders are *objects*, *refs*, ... There are also the following files *HEAD*.

git log

shows a list of commits from reverse time order (goes back on time), i.e. shows the history of your repository. The history of a repository can be represented by a directed acyclic graph (dag for short), pointing in the forward direction in time.

options

–graph

shows a graphical representation of the repository commits history.

git rebase

git rebase <branch2 or commit2>, finds a common point between the current branch and branch2 or commit2, reapply all commits of your current branch from that divergent points on top of branch2 or commit2, one by one.

git reset

git reset –soft HEAD 1, moves one commit back but keeps all modified files.

git reset –hard HEAD 1, moves one commit back and cleans all the modified files.

git reflog

Keep a log file with all commands from the last 90 days.

git show

Shows what is new in the last commit.

git stash

Stash is a global branch where you can store the present state.

git stash, save the present state of your repository.

git stash –list, shows what is in the stash.

git status

6.11.2 Pluming Commands

git count-object

git count-object -H, counts all object and shows the result in a readable form (-H, human).

git gc

Garbage collector. Eliminates all objects that are not referenced, i.e. has no reference associated with.

git gc –prune=all

git hash-object

git hash-object -w <file>, calculates the SHA1 hash value of a file and write it in the *.git/objects* folder.

git cat-files

git cat-files -p <sha1>, shows the contend of a file in a readable format (flag -p, pretty format).

git cat-files -t <sha1>, shows the type of a file.

git update-index

git update-index –add <file name>, creates the hash and adds the <file_name> to the index.

git ls-files
git ls-files –stage, shows all files that you are tracking.
git write-tree
git commit-tree
git update-ref
git update-ref refs/heads/<branch name> <commit sha1 value>, creates a branch that points to the <commit sha1 value>.
git verify-pack

6.12 The Configuration Files

There is a config file for each repository that is stored in the *.git/* folder with the name *config*.

There is a config file for each user that is stored in the *c:/users/<user name>/* folder with the name *.gitconfig*.

To open the *c:/users/<user name>/.gitconfig* file type:

git config –global -e

6.13 Pack Files

Pack files are binary files that git uses to save data and compress your repository. Pack files are generated periodically by git or with the use of gc command.

6.14 Applications

6.14.1 Meld

6.14.2 GitKraken

6.15 Error Messages

6.15.1 Large files detected

Clean the repository with the BFG Repo-Cleaner.

Run the Java program:

java -jar bfg-1.12.16.jar −−strip-blobs-bigger-than 100M

This program is going to remote from your repository all files larger than 100MBytes. After

do:

git push −−force.

Chapter 7

Simulating VHDL Programs with GHDL

This guide will help you simulate VHDL programs with the open-source simulator GHDL.

7.1 Adding Path To System Variables

Please follow this step-by-step tutorial:

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 your absolute path to the folder **\LinkPlanner\vhdl_simulation\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\vhdl_simulation\ghdl\bin**.
   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 your absolute path to the folder **\LinkPlanner\vhdl_simulation\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\vhdl_simulation\ghdl\bin**.
10. Press **Ok** and you're done.

7.2 Using GHDL To Simulate VHDL Programs

This guide will only cover the simulation of the VHDL module in this repository.
This simulation will take an .sgn file and output its binary information, removing the header.
There are two ways to simulate this module.

7.2.1 Requirements

Place a .sgn file in the directory **\vhdl_simulation\input_files** and rename it to SIGNAL.sgn

7.2.2 Option 1

Execute the batch file **simulation.bat**, located in the directory **\vhdl_simulation** in this repository.

7.2.3 Option 2

Open the **Command Line** and navigate to your project folder (where the .vhd file is located).
Execute the following commands:

ghdl -a –std=08 signal_processing.vhd

ghdl -a –std=08 vhdl_simulation.vhd

ghdl -e –std=08 vhdl_simulation

ghdl -r –std=08 vhdl_simulation

Additional information: The first two commands are used to compile the program and will generate .cf files. Do not remove these file until the simulation is complete.
The third command is used to elaborate the simulation.
The last command is used to run the simulation. If you want to simulate the same program again, you will just need to execute this command (as long as you don't delete the .cf files).

7.2.4 Simulation Output

The simulation will output the file SIGNAL.sgn. This file will contain all the processed binary information of the input file, with the header.