NetXPTO - LinkPlanner

Armando Nolasco Pinto

April 3, 2019

Contents

Chapter 1

Preface

Chapter 2

Introduction

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

Chapter 3

Simulator Structure

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

3.1 System

3.2 Blocks

3.3 Signals

List of available signals:

- Signal

PhotonStreamXY

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

PhotonStreamXY_MP

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

3.3.1 Circular Buffer

The signals use a circular buffer to store data. Because standard C++ do not have a circular buffer container (at least up to ISO C++17) one was developed. The circular buffer was developed using the same principles and style of the other STL containers, trying to make sure that the future integration of a standard circular buffer in our code will as easy as possible. In this development we use the following references [**Johnston17**, **Gaspar18**, **Guntheroth18**]. In [**Johnston17**] a simple circular buffer implementation is presented, in [**Gaspar18**] a standard like version of a circular buffer is discussed, and in [**Guntheroth18**] a comparative assessment is presented considering different implementation strategies. We try to follow [**Gaspar18**] as possible.

A circular buffer is a fixed-size container that works in a circular way, the default buffer size is 512. A circular buffer uses a begin and a end pointer to control where data is going to be retrieved (consumed) or added. A full buffer flag is also used to signal the full buffer situation.

Initially, the begin and the end are made to coincide and the full flag is set to false. This is the empty buffer state. When data is added, the end pointer advances. After adding data if the end and the begin pointer coincide the buffer is full.

When data is retrieved, the begin pointer advances. After retrieving data if the begin and end pointer coincide the buffer is empty.

3.4 Log File

3.4.1 Introduction

The Log File allows for a detailed analysis of a simulation. It will output a file containing the timestamp when a block is initialized, the number of samples in the buffer ready to be processed for each input signal, the signal buffer space for each output signal and the amount of time in seconds that took to run each block. Log File is enabled by default, so no change is required. If you want to turn it off, you must call the set method for the logValue and pass $false$ as argument. This must be done before method $run()$ is called, as shown in line 125 of Figure 3.1.

```
115
116     // ##############################################################################################
117     // ########################## System Declaration and Inicialization ##############################
118     // ##############################################################################################
119
120     System MainSystem{ vector<Block*> { &B1, &B2, &B3, &B4, &B5, &B6, &B7, &B8} };
121
122     // ##############################################################################################
123     // ################################### System Run ###############################################
124     // ##############################################################################################
125     MainSystem.setLogValue(false);
126     MainSystem.run();
```

Figure 3.1: Disabling Log File

3.4.2 Parameters

The Log File accepts two parameters: $logFileName$ which correspond to the name of the output file, i.e., the file that will contain all the information listed above and $logValue$ which will enable the Log File if $true$ and will disable it if $false$.

Log File Parameters		
Parameter	**Type**	**Default Value**
logFileName	string	"log.txt"
logValue	bool	true

Available Set Methods		
Parameter	**Type**	**Comments**
setLogFileName(string newName)	void	Sets the name of the output file to the name given as argument
setLogValue(bool value)	void	Sets the value of logValue to the value given as argument

3.4.3 Output File

The output file will contain information about each block. From top to bottom, the output file shows the timestamp (time when the block was started), the number of samples in the buffer ready to be processed for each input signal and the signal buffer space for each output signal. This information is taken before the block has been executed. The amount of time, in seconds, that each block took to run, is also registered. Figure 3.2 shows a portion of an output file. In this example, 4 blocks have been run: MQamTransmitter, LocalOscillator, BalancedBeamSplitter and I_HomodyneReceiver. In the case of the I_HomodyneReceiver block we can see that the block started being ran at 23:27:37 and finished running 0.004 seconds later.

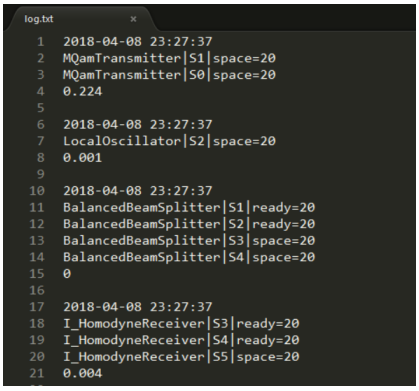


Figure 3.2: Output File Example

Figure 3.3 shows a portion of code that consists in the declaration and inicialization of the I_HomodyneReceiver block. In line 97, we can see that the block has 2 input signals, $S3$ and $S4$, and is assigned 1 output signal, $S5$. Going back to Figure 3.2 we can observe that $S3$ and $S4$ have 20 samples ready to be processed and the buffer of $S5$ is empty.

```
97      I_HomodyneReceiver B4{ vector<Signal*> {&S3, &S4}, vector<Signal*> {&S5} };
98      B4.useShotNoise(true);
99      B4.setElectricalNoiseSpectralDensity(electricalNoiseAmplitude);
100     B4.setGain(amplification);
101     B4.setResponsivity(responsivity);
102     B4.setSaveInternalSignals(true);
103
```

Figure 3.3: I-Homodyne Receiver Block Declaration

The list of the input parameters loaded from a file is presented at the top of the output file, as shown in Figure 3.4.

```
log.txt
1  The following input parameters were loaded from a file:
2  pLength
3  numberOfBitsGenerated
4  bitPeriod
5  shotNoise
6  ------------------------------------------------------------
7  2018-05-15 00:28:42
8  MQamTransmitter|S1|space=20
9  MQamTransmitter|S0|space=20
10 0.107
11
12 2018-05-15 00:28:42
13 LocalOscillator|S2|space=20
14 0.004
```

Figure 3.4: Four input parameters where loaded from a file

3.4.4 Testing Log File

In directory *doc/tex/chapter/simulator_structure/test_log_file/bpsk_system/* there is a copy of the BPSK system. You may use it to test the Log File. The main method is located in file *bpsk_system_sdf.cpp*

3.5 Input Parameters System

3.5.1 Introduction

With the Input Parameters System (IPS) it is possible to read the input parameters from a file.

Format of the Input File

We are going to explain the use of the IPS using as an example the PBSK system. In Figure 3.5, it is possible to observe the contents of the file **input_parameters_0.txt** used to load the values of some of the BPSK system's input parameters. The input file must respect the following properties:

1. Input parameter values can be changed by adding a line in the following format: **paramName=newValue**, where **paramName** is the name of the input parameter and **newValue** is the value to be assigned.
2. IPS supports scientific notation. This notation works for the lower case character **e** and the upper case character **E**.
3. If an input parameters is assigned the wrong type of value, method **readSystemInputParameters()** will throw an exception.
4. Not all input parameters need to be changed.
5. The IPS supports comments in the form of the characters //. The comments will only be recognized if placed at the beginning of a line.

input_parameters_0.txt

```
1  //----------BPSK PARAMETERS--------------
2  //Changes the value of parameter pLength
3  pLength=5
4  //Is able to change the value of different types of parameters
5  //Of type int
6  numberOfBitsGenerated=1e3
7  //Of type double
8  bitPeriod=20e-12
9  //Of type bool
10 shotNoise=false
```

Figure 3.5: Content of file input_parameters_0.txt

Loading Input Parameters From A File

Execute the following command in the Command Line:

some_system.exe <input_file_path> <output_directory>

where **some_system.exe** is the name of the executable generated after compiling the project, **<input_file_path>** is the path to the file containing the new input parameters; **<output_directory>** is the directory where the output signals will be written into.

3.5.2 How To Include The IPS In Your System

In this illustrative example, the code of the BPSK System will be used. To implement the IPS the following requirements must be met:

1. Your system must include **netxpto_20180418.h** or later.
2. A class that will contain the system input parameters must be created. This class must be a derived class of **SystemInputParameters**. In this case the created class is called **BPSKInputParameters**.
3. The created class must have 2 constructors. The implementation of these constructors is the same as **BPSKInputParameters**.

```
BPSKInputParameters();
BPSKInputParameters(int argc, char*argv[]);
```

4. The created class must contain the method **initializeInputParameterMap()**. For every input parameter **addInputParameter(paramName,paramAddress)** must be called, where **paramName** is a string that represents the name of your input parameter and **paramAddress** is the address of your input parameter.

```
void initializeInputParameterMap(){
  //Add parameters
}
```

5. All signals must be instantiated using the constructor that takes as argument, the file name and the folder name, according to the type of signal.

```
Binary S0("S0.sgn", param.getOutputFolderName()) //S0 is a Binary signal
```

6. Method **main** must receive the following arguments.

```
int main(int argc, char*argv[]){...}
```

7. The MainSystem must be instantiated using the following line of code. The ... represent the list of blocks.

```
System MainSystem{ vector<Block*>
     {...},param.getOutputFolderName(),param.getLoadedInputParameters()};
```

The following code represents the input parameters class, **BPSKInputParameters**, and must be changed according to the system you are working on.

```
class BPSKInputParameters : public SystemInputParameters {
public:
  //INPUT PARAMETERS
  int numberOfBitsReceived{ -1 };
  int numberOfBitsGenerated{ 1000 };
  int samplesPerSymbol = 16;
   (...)

  /* Initializes default input parameters */
  BPSKInputParameters() : SystemInputParameters() {
    initializeInputParameterMap();
  }

  /* Initializes input parameters according to the program arguments */
   /* Usage: .\bpsk_system.exe <input_parameters.txt> <output_directory> */
  BPSKInputParameters(int argc, char*argv[]) : SystemInputParameters(argc,argv) {
    initializeInputParameterMap();
    readSystemInputParameters();
  }

  //Each parameter must be added to the parameter map by calling addInputParameter(string,param*)
  void initializeInputParameterMap(){
    addInputParameter("numberOfBitsReceived", &numberOfBitsReceived);
    addInputParameter("numberOfBitsGenerated", &numberOfBitsGenerated);
    addInputParameter("samplesPerSymbol", &samplesPerSymbol);
      (...)
  }
};
```

The method **main** should look similar to the following code.

```
int main(int argc, char*argv[]){

   BPSKInputParameters param(argc, argv);

   //Signal Declaration and Initialization
   Binary S0("S0.sgn", param.getOutputFolderName());
  S0.setBufferLength(param.bufferLength);

  OpticalSignal S1("S1.sgn", param.getOutputFolderName());
  S1.setBufferLength(param.bufferLength);
   (...)

   //System Declaration and Initialization
  System MainSystem{ vector<Block*> { &B1, &B2, &B3, &B4, &B5, &B6, &B7,
       &B8},param.getOutputFolderName(),param.getLoadedInputParameters()};

   //System Run
  MainSystem.run();

  return 0;
}
```

The class **SystemInputParameters**, has the following constructors and methods available:

SystemInputParameters - Constructors	
Constructors	**Comments**
SystemInputParameters()	Creates an object of SystemInputParameters with the default input parameters' values
SystemInputParameters(int argc, char*argv[])	Creates an object of SystemInputParameters and loads the values according to the program arguments passed in the command line

SystemInputParameters - Methods		
Method	**Type**	**Comments**
addInputParameter(string name, int* variable)	void	Adds an input parameter whose value is of type int
addInputParameter(string name, double* variable)	void	Adds an input parameter whose value is of type double
addInputParameter(string name, bool* variable)	void	Adds an input parameter whose value is of type bool
readSystemInputParameters()	void	Reads the input parameters from a file.

3.6 Documentation

As in any large software system documentation it is critical. The documentation is going to be developed in Latex using WinEdt as the recommend editor. The bibliography is per section, for this to work replace the bibtex by biber, go to the WinEdt Options->Execution Modes->Bibtex and replace bibtex.exe by biber.exe.

Chapter 4

Development Cycle

The NetXPTO-LinkPlanner is a open source project with its core implemented using ISO C++. At the present the followed standard is the ISO C++14.

The developed environment has been the Visual Studio Community 2017, namely release 15.5 and beyond. The Git has been used as the version control system. The NetXPTO-LinkPlanner repository is located in the GitHub site http://github.com/netxpto/linkplanner. Master branch should be considered a functional beta version of the software. Periodically new releases are delivered from the master branch under the branch name R<Release Year>-<Release Number>. The design and integration of the system has been performed by Prof. Armando Pinto.

Chapter 5

Visualizer

The visualizer is based on a customization of the the Matlab SPTOOL (https://www.mathworks.com/help/signal/ref/sptool.html) application.

Chapter 6

Case Studies

6.1 M-QAM Transmission System

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

6.1.1 Introduction

The goal of this project is to simulate a Quadrature Amplitude Modulation transmission system with M points in the constellation diagram (M-QAM) and to perform a Bit Error Rate (BER) measurement that can be compared with theoretical and experimental values. M-QAM systems can encode $\log_2 M$ bits per symbol which means they can transmit higher data rates keeping the same bandwidth. However, because the states are closer together, these systems require a higher signal-to-noise ratio. The Bit Error Rate (BER) is a measurement of how a bit stream is altered by a transmission system due to noise or intersymbol interference.

For $M = 4$ the M-QAM system can be reduced to a Quadrature Phase Shift Keying system (QPSK) system that uses four equispaced points in the constellation diagram, as shown in figure 6.1 where a Gray encoding is assumed [**proakis08**].

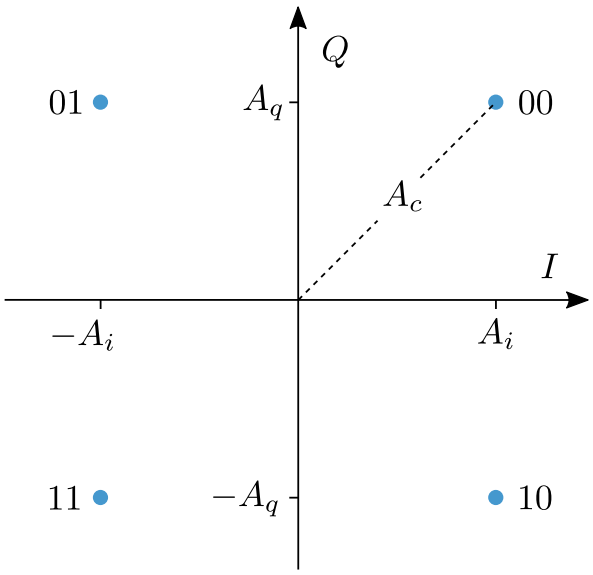


Figure 6.1: A QPSK constellation, assuming $A = A_i = A_q = A_c/\sqrt{2}$.

6.1.2 Theoretical Analysis

6.1.2.1 QPSK Transmitter

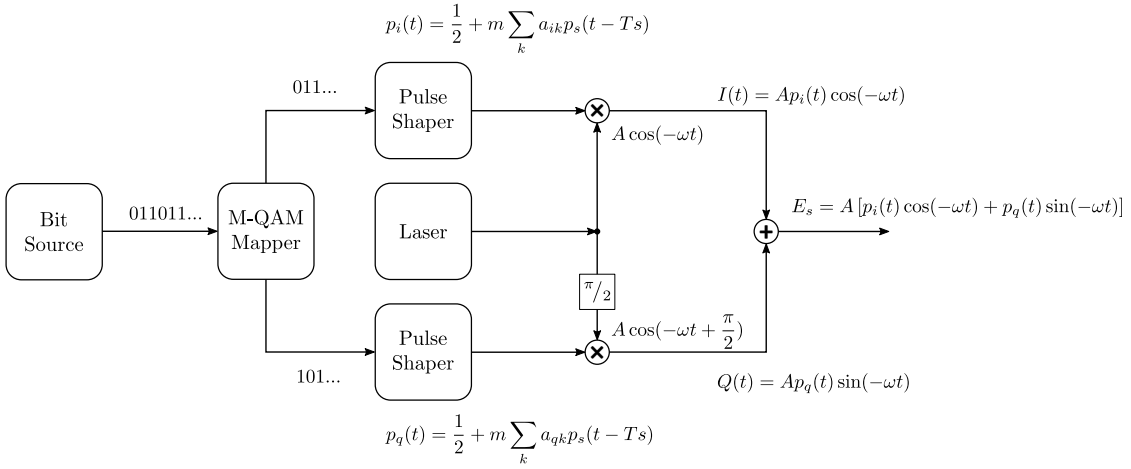


Figure 6.2: Transmitter diagram.

M-QAM is a modulation scheme that takes advantage of two sinusoidal carriers with a phase difference of $\pi/2$. The resultant output consists of a signal with both amplitude and phase variations. For the particular case of M = 4, it can be considered that $A_i = A_q = A$, and therefore $A_c = A\sqrt{2}$. The two carriers, referred to as I (In-phase) and Q (Quadrature), can be represented as [**carlson86**]:

$$I(t) = p_i(t)A\cos(-\omega t) \tag{6.1}$$

$$Q(t) = p_q(t)A\cos\left(-\omega t + \frac{\pi}{2}\right) = p_q(t)A\sin(-\omega t) \tag{6.2}$$

Here, $p_i(t)$ and $p_q(t)$ is the sequence of pulses for each of the components, as defined by:

$$p(t) = \frac{1}{2} + m\sum_k a_k p_s(t - T_s), \qquad a_k = \begin{cases} -1 & \text{if bit} = 1, \\ 1 & \text{if bit} = 0 \end{cases} \tag{6.3}$$

where m is a scaling coefficient, to ensure the following conditions are met

$$\begin{cases} \frac{1}{2} + m\left(\max\left\{p_s(t)\right\}\right) = 1 \\ \frac{1}{2} + mp_s(t) > 0 \end{cases}$$

Following from equations 6.1 and 6.2, we have [**apostol67**, **oo12**]:

$$\begin{aligned} E_s(t) &= I(t) + Q(t) = \\ &= p_i(t)A\cos(-\omega t) + p_q(t)A\sin(-\omega t) \\ &= p_i(t)A\cos(\omega t) - p_q(t)A\sin(\omega t) \\ &= A\sqrt{p_i^2(t) + p_q^2(t)}\cos\left(-\omega t + \phi_s(t)\right), \quad \phi_s(t) = \arctan\left(p_q(t), p_i(t)\right) \end{aligned} \tag{6.4}$$

The *arctan* function shown above is the two argument arctangent function, sometimes also called *atan2*. While the normal arctangent is defined only for the interval $]-\frac{\pi}{2}, +\frac{\pi}{2}]$, the two argument arctangent is a piecewise function defined in the interval $]-\pi, +\pi]$ **[glisson11]**.

$$\arctan(x,y) = \begin{cases} \arctan(y/x), & if x > 0, \\ (\text{sign}\,\{y\})\,(\pi/2) & if x = 0 \text{ and } y \neq 0, \\ \pi + \arctan(y/x) & if x < 0, \\ 0, & if x = 0 \text{ and } y = 0 \end{cases} \tag{6.5}$$

6.1.2.2 QPSK Receiver

We will consider a homodyne receiver with phase diversity. This is a configuration which uses a local oscillator with the same optical frequency, and measures both the in-phase and the quadrature component at the same time.

A typical configuration for this receiver is shown in Figure 6.3. The local oscillator laser uses the same frequency of the incoming signal. It is used in coherent detection to extract the phase and amplitude information contained in the received optical signal. This is done by mixing the LO with the incoming optical signal in a $\pi/2$ optical hybrid. The optical hybrid then outputs 4 different optical signals, where the phase difference between the LO and the signal is at 0, $\pi/2$, π, $3\pi/2$. This difference is relative to their phase difference at the input of the optical hybrid. The signals with a difference of π between themselves are paired and sent to a pair of balanced photodiodes, where each pair measures one of the quadratures. The transimpedance amplifier then amplifies the signal, while converting the current to a voltage to be sampled and quantified in an ADC.

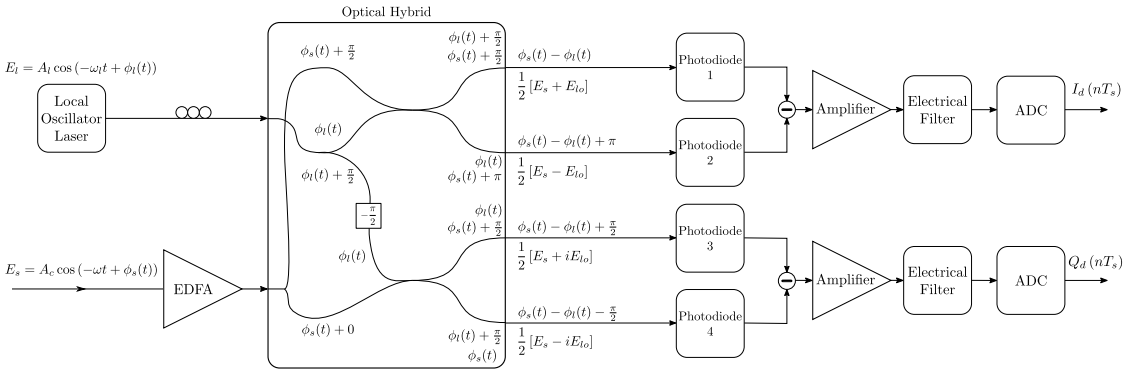


Figure 6.3: Local oscillator and receiver filters diagram. ϕ_a and ϕ_b represent the phase rotations happening inside the optical hybrid.

Before starting, we should clarify how the optical hybrid works. A possible configuration is shown in Figure 6.3, where the mixing is achieved resorting to a few directional couplers. Figure 6.4 shows an example of a directional coupler, where two fiber cores are close enough so that the space between them is comparable to their diameter. In these conditions, the

fundamental modes propagating through each fiber overlap partially, leading to power transfer from one core to the other [**agrawal04**].

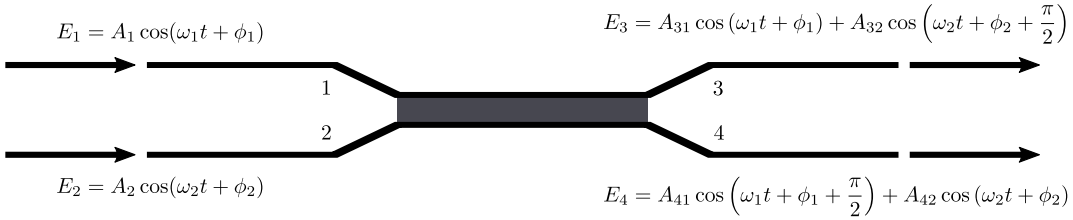


Figure 6.4: Directional coupler diagram.

Assuming a lossless directional coupler as shown Figure 6.4, the relation between input and output power is:

$$P_1 + P_2 = P_3 + P_4 \tag{6.6}$$

With this in mind, the amplitudes A_3 and A_4 on the output of a coupler are related to their inputs A_1 and A_2 by the coupling length L and the coupling coefficient k:

$$\begin{bmatrix} A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} \cos(kL) & i\sin(kL) \\ i\sin(kL) & cos(kL) \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \tag{6.7}$$

Two things can be noticed here: first, the power ratios are dependent on the coupling length. This comes from the supermodes of the fiber coupler, which have different propagation constants [**agrawal04**]. This difference creates a relative phase shift proportional to the coupler length. In addition, a phase shift of $\pi/2$ exists always between the two outputs. This is because the supermodes are the even and odd combinations of the fundamental modes of the coupler.

This phase shift can be taken advantage of to create an optical hybrid, by mixing the components in a way to end up with 4 different combinations of the signal and LO, having relative phase differences of $0, \pi/2, \pi$ and $3\pi/2$.

We can now proceed to analyze the process at the receiver, but for now we will ignore the noise added by the EDFA shown at the beginning of the diagram.

Coherent detection takes the product of the electric fields from the signal and a Local Oscillator, in order to measure the phase difference between them through interference. The electric field of the Local Oscillator is [**kikuchi16**]:

$$E_{lo} = A_{lo} \cos(-\omega_{lo} t + \phi_{lo}) \tag{6.8}$$

The amplitudes A_{lo} and A_c are related to their respective average optical power by:

$$P_s = k{E_s}^2 = k{A_c}^2 \cos^2(-\omega t + \phi_s) = k\frac{{A_c}^2}{2} = k\frac{(A\sqrt{2})^2}{2} = kA^2$$
$$P_{lo} = k{E_{lo}}^2 = k{A_{lo}}^2 \cos^2(-\omega_{lo} t + \phi_{lo}) = k\frac{{A_{lo}}^2}{2}$$

where k is the ratio between the effective beam area and the impedance of free space. The power of the signal entering the EDFA is amplified by a gain factor G_o

$$P_{\text{s\_out}} = G_o P_{\text{s\_in}} \tag{6.9}$$

G_o is the EDFA optical gain, given by [**mukai81**]:

$$G_o = \frac{G_{\text{small}}}{1 + \frac{P_{\text{in}}}{P_{\text{sat}}}} \tag{6.10}$$

where G_{small} is the gain for small signals, P_{in} is the optical power at the EDFA's input and P_{sat} is the EDFA's saturation power. This means that the effective optical gain is lower for input signals with higher optical powers.

As the input optical signal and local oscillator enter the optical hybrid, each of them is split twice in 3-dB couplers. This means that a quarter of the power from each source reaches each of the photodiodes. In addition, as previously explained, they are subject to phase shifts which mix them with differente relative phase differences. We therefore get 4 different combinations from E_s andE_{lo}, the electrical fields of the signal and the LO.

$$\begin{aligned}
E_1 &= \frac{1}{2}\left(\sqrt{G_o}E_s + E_{lo}\right) \\
E_2 &= \frac{1}{2}\left(\sqrt{G_o}E_s - E_{lo}\right) \\
E_3 &= \frac{1}{2}\left(\sqrt{G_o}E_s + iE_{lo}\right) \\
E_4 &= \frac{1}{2}\left(\sqrt{G_o}E_s - iE_{lo}\right)
\end{aligned}$$

Let us assume from now on that $\omega = \omega_{lo}$ and $\phi_{lo} = 0$. In this case, the photocurrent for the each of the previous n combinations is equal to:

$$I_1(t) = \frac{\eta}{4}\left[G_o P_s(t) + P_{lo} + 2\sqrt{G_o P_s(t) P_{lo}}\cos\left(\phi_s(t)\right)\right] \tag{6.11}$$

$$I_2(t) = \frac{\eta}{4}\left[G_o P_s(t) + P_{lo} - 2\sqrt{G_o P_s(t) P_{lo}}\cos\left(\phi_s(t)\right)\right] \tag{6.12}$$

$$I_3(t) = \frac{\eta}{4}\left[G_o P_s(t) + P_{lo} + 2\sqrt{G_o P_s(t) P_{lo}}\sin\left(\phi_s(t)\right)\right] \tag{6.13}$$

$$I_4(t) = \frac{\eta}{4}\left[G_o P_s(t) + P_{lo} - 2\sqrt{G_o P_s(t) P_{lo}}\sin\left(\phi_s(t)\right)\right] \tag{6.14}$$

$$\tag{6.15}$$

where η is the photodiodes' responsivity. Two photocurrents are generated from the combination of the balanced photodiodes, which are then amplified in a transimpedance amplifier with a gain factor G_e.

$$\begin{aligned} V_i(t) &= G_e\left(I_1 - I_2\right) \\ &= G_e\eta\sqrt{G_oP_sP_{lo}}\cos\left(\phi_s(t)\right) \qquad (6.16) \\ V_q(t) &= G_e\left(I_3 - I_4\right) \\ &= G_e\eta\sqrt{G_oP_sP_{lo}}\sin\left(\phi_s(t)\right) \qquad (6.17) \end{aligned}$$

These voltages effectively reconstruct the transmitted optical signal. After going through the electrical filter, they can be sampled into the digital domain for processing.

Assuming perfect synchronization, if the signal is sampled with timing $t = nT_s$, the constellation can be reconstituted perfectly, if there is no intersymbol interference. In this case, according to equations 6.1 and 6.2, $p_i(t)$ and $p_q(t)$ are equal to either 1 or -1. It follows that $\phi_s(t = nT_s) = \frac{\pi}{4} \pm m\frac{\pi}{2}$. If the gain G_e is adjusted so that $G_e\eta\sqrt{G_oP_sP_{lo}} = A_c$, we get:

$$\begin{aligned} V_i(t) &= A_c\cos\left(\phi_s(t)\right), \\ &= \pm\frac{A_c}{\sqrt{2}} = \pm A_i = \pm A \qquad t = nTs \qquad (6.18) \\ V_q(t) &= A_c\sin\left(\phi_s(t)\right), \\ &= \pm\frac{A_c}{\sqrt{2}} = \pm A_q = \pm A \qquad t = nTs \qquad (6.19) \end{aligned}$$

Looking at Figure 6.5, we can see that equations 6.18 and 6.19 describe the four points in the constellation.

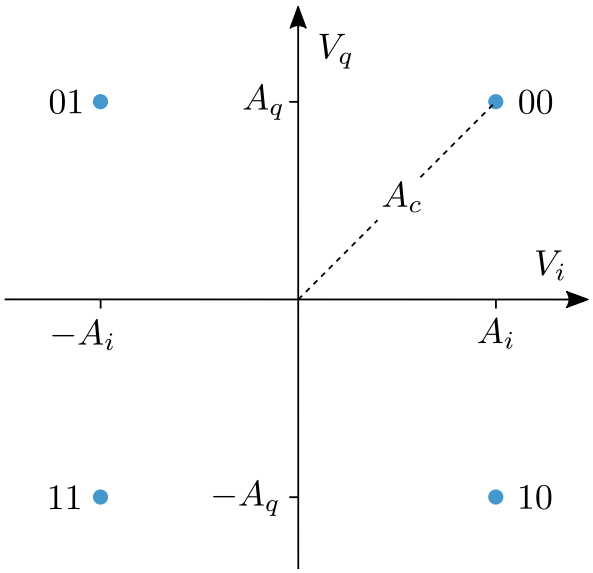


Figure 6.5: A QPSK constellation, assuming $A = A_i = A_q = A_c/\sqrt{2}$.

The previous explanation assumes that the polarization of both the transmitted signal and the local oscillator is aligned. This can be achieved by using a polarization controller, as shown in Figure 6.3. We have also neglected any link losses.

6.1.2.3 ISI

We have assumed the absence of intersymbol interference. As previously mentioned, bits are coded into the symbols of a given constellation. These symbols, associated with a given discrete amplitude, need to be encoded into the I and Q carriers. For this, a given shape must be used to translate the discrete symbols into continuous, finite pulses in the analog domain. The choice of the shape to be used is particularly important for the ISI.

Each symbol is represented by its own pulse, which is dependent on the symbol period T_s. In each pulse, there is an optimal instant t_s, where we can sample the signal in order to perfectly identify the contained symbol. These instants are separated by a time equal to the symbol period. In order to negate intersymbol interference, the signal for each pulse should be equal to zero every time a different symbol is sampled, that is, at all instants $t_s \pm nT_s$. If this condition is verified, the value of the signal at $t = t_s$ is dependent only on the symbol transmitted at that time, not being influenced by the surrounding pulses. This is shown in Figure 6.7, where we can see that at all instants nT_s, the reponse is zero for all pulses but one. Consequently, at those instants, the signal obtained from the combination of all pulses is exactly equal to the peak of the pulse transmitted at that time. Those instants are also shown at the center of the eye diagram shown in Figure 6.7, where t_s is represented, and

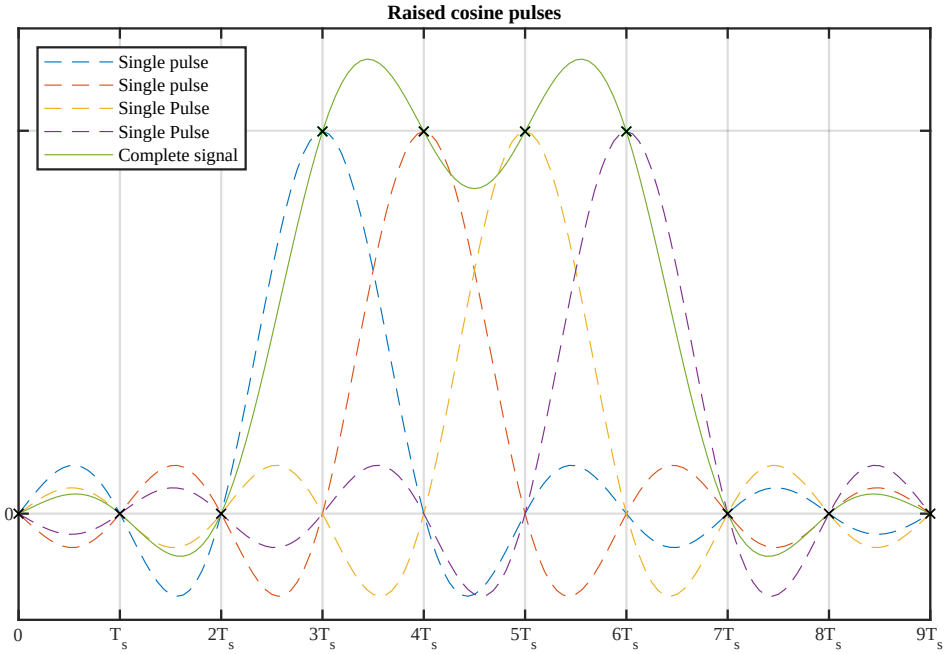


Figure 6.6: Signal generated by four sequential raised cosine pulses. It can be seen that at $t = nT_s$, every pulse except one equals exactly zeros, showing that there is no interference. In addition, at those times, the value of the signal is coincident with the peak of the individual pulses.

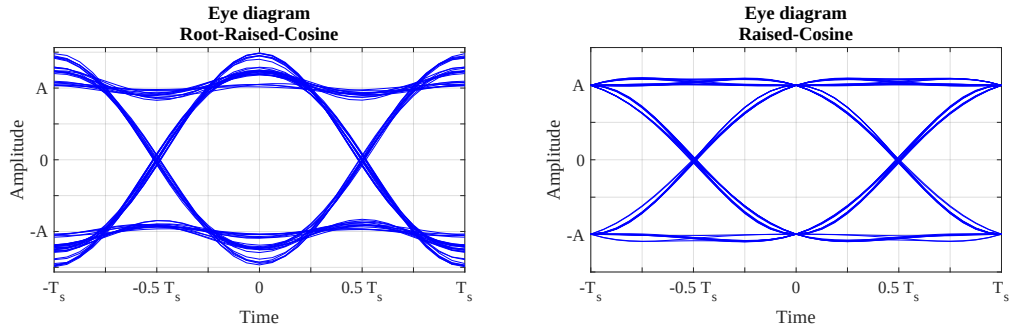


Figure 6.7: Eye diagrams of two signals without noise: on the left, shaped with a single root-raised-cosine filter and affected by ISI; on the right, shaped using a raised-cosine filter, showing no signs of ISI.

perfect coincidence in the signal at sampling time is shown.

The raised-cosine is a shape commonly used for pulse shaping, as it does not create any intersymbol interference. This is shown in Figure 6.7. The time domain response for raised-cosine filter is given by [**nguyen09**]

$$h_{RC}(t) = \frac{\sin(\pi t/T_s)}{(\pi t/T_s)} \frac{\cos(\pi \beta t/T_s)}{1-(4\beta^2 t^2/T_s^2)} \tag{6.20}$$

where β is the roll-off factor and T_s is the symbol period.

Optimal detection often requires the implementation of matched filters [**schwartz90**]. This is because in addition to avoiding ISI, it is desirable to maximize the SNR before sampling the signal, and in order to achieve both these objectives simultaneously, a matched filter should be used. This is done by implementing in the receiver a filter similar to the one used to initially shape the pulses. The pulse-shaping process becomes divided between the transmission and reception stages. Provided that an appropprate filter is chosen, the resulting signal can still be free of ISI, while reducing the noise power in the final sampled signal.

The first filter, implemented on the transmitter, will convert the discrete symbols into continuous shapes, so that it can be transmitted on a modulated signal. The filter at the receiver finalizes the shaping process, making the signal reach the desired shape for symbol identification. In addition, it decreases the amount of noise affecting the signal.

With matched filtering, the raised-cosine can be implemented by having the filters at the transmitter and receiver be root-raised-cosine filters. The end result will be signal with the same shape as if a raised cosine filter were used at the transmitter.

The root-raised-cosine is defined as a filter for which the square of the frequency response is equal to the frequency response of the raised cosine filter. Its time domain impulse response given by[**nguyen09**]:

$$h_{RRC}(t) = \frac{(4\beta t/T_s)\cos\left[\pi(1+\beta)t/T_s\right] + \sin\left[\pi(1-\beta)t/T_s\right]}{(\pi t/T_s)\left[1-(4\beta t/T_s)^2\right]} \tag{6.21}$$

6.1.2.4 Noise

So far we have neglected to include any source of noise in this analysis. Several possible noise sources can be considered, either in the optical or in the electrical part of the system. As shown in Figure 6.3, for now we will only consider the existence of optical noise, and we will consider that the noise added is ASE which could originate from a EDFA. The ASE field can be represented as a sum of terms as follows [**crognale14**, **olsson89**]:

$$E_{\text{ASE}x} = \sum_{k=-B_\nu}^{B_\nu} \sqrt{2n_{0o}\delta\nu}\cos\left((-\omega + 2\pi k\delta\nu)t + \phi_{kx}\right) \tag{6.22}$$

$$E_{\text{ASE}y} = \sum_{k=-B_\nu}^{B_\nu} \sqrt{2n_{0o}\delta\nu}\sin\left((-\omega + 2\pi k\delta\nu)t + \phi_{ky}\right) \tag{6.23}$$

where $B_\nu = B_{\text{opt}}/(2\delta\nu)$, B_{opt} is the optical bandwidth and $\delta\nu$ is small frequency interval. We can write an expression for the optical noise spectral density n_{0o} considering the use of a number N_A of cascaded EDFA's. Assuming that all amplifiers are similar and equally spaced, and that the gain in each EDFA is just enough to compensate the fiber losses in the distance between them, we have [**agrawal12**]:

$$n_{0o} = N_A n_{\text{sp}}(G_o - 1)h\nu \tag{6.24}$$

Here, n_{sp} is the EDFA's spontaneous emission factor, $h\nu$ is the photon energy. In additon, ϕ_{kx} and ϕ_{ky} are independent uniform random variables representing random phases for each of the terms in the summation, and G_o is the amplifiers' optical gain as defined in equation 6.10. This field is added to equation 6.4, giving rise to three noise terms from different interactions:

- ASE-LO, from the interaction with the local oscillator;
- ASE-SIG, from the interaction with the transmitted signal;
- ASE-ASE, from the interaction with itself.

For now, we will consider only the ASE-LO component, which is close enough to additive white Gaussian noise. In addition, it is typically the strongest, as usually the LO power is much greater than either the signal or the ASE. The variance, or noise power, generated by this component at each pair of balanced photodiodes is given by [**crognale14**]:

$$\sigma^2_{\text{ASE}-\text{LO}} = 2\eta^2 P_{\text{LO}} n_{0o} B_N = 2\eta^2 P_{\text{LO}} N_A n_{\text{sp}}(G_o - 1)h\nu B_N \tag{6.25}$$

Here, B_N is the electrical noise bandwidth. Notice that the total noise power is proportional to this bandwidth, so keeping it closer to the signal's bandwidth limits the total noise to a minimum. This noise is also amplified at the transimpedance amplifier, so we get a voltage variance equal to:

$$\sigma^2_{\text{ASE}-\text{LO}} = 2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu B_N \tag{6.26}$$

Considering the generated noise to be independent from the signal, let $x(t)$ be the signal at the ADCs, sampled at a given instant t for either the in-phase or quadrature component. $x(t)$ has two components:

$$x(t) = s(t) + n(t) \tag{6.27}$$

where the component $s(t)$ corresponds to the received signal, and $n(t)$ corresponds to the noise component. For $t = nT_s$, as mentioned in equations 6.18 and 6.19, $s(t) = \pm A$. The noise component, being AWGN, has constant power spectral density, and follows a Gaussian distribution with zero mean and variance equal to the noise power as described in equation 6.26.

It's possible to remove the exceeding noise through filtering. This is done by the matched filter, as mentioned in the previous section. The filter essentially decreases the bandwidth, removing all the noise at frequencies not contained in $f < \left|1/(2T_s)\right|$, while not causing intersymbol interference in the transmitted signal.

Let us consider a signal as described in Equation 6.27, with $a(t) = A_p\, p(t)$, where A_p is the peak amplitude of the signal and $p(t)$ is the shape of the pulse. In addition, let $n(t)$ be AWGN of spectral density $G_n(f) = n_0/2$. Finally, assume that the filter at the receiver has a frequency response $H(f)$. The filter is similar to the shape of the pulse , but reversed in time and shifted by a time delay, such that it maximizes the SNR [**carlson86**]. The energy contained in the pulse that enters the receiver filter depends on the pulse amplitude and on its shape, and it is given by:

$$E_p = \int_{-\infty}^{\infty} |A(f)|^2 df = A_p^2 \int_{-\infty}^{\infty} |P(f)|^2 df \tag{6.28}$$

The amplitude of the signal component after the receiver filter $H(f)$, at a given sampling time $t = t_0 + t_d$, is:

$$A = \mathfrak{F}^{-1}\left[H(f)A(f)\right]\big|_{t=t_0+t_d} = A_p \int_{-\infty}^{+\infty} H(f)P(f)e^{+j\omega t_d} df \tag{6.29}$$

Similarly, the noise power at the receiver filter output is given by:

$$N_o = \int_{-\infty}^{+\infty} |H(f)|^2 G_n(f) df = \frac{n_0}{2} \int_{-\infty}^{+\infty} |H(f)|^2 \tag{6.30}$$

This means that the peak signal power to mean noise power ratio at the filter output is given by:

$$\begin{aligned} \frac{A^2}{N_o} &= A_p^2 \frac{\left| \int_{-\infty}^{+\infty} |H(f)P(f)e^{j\omega t_d} df \right|^2}{\int_{-\infty}^{+\infty} |H(f)|^2 G_n(f) df} \\ &= A_p^2 \frac{\left| \int_{-\infty}^{+\infty} |H(f)P(f)e^{j\omega t_d} df \right|^2}{\frac{n_0}{2} \int_{-\infty}^{+\infty} |H(f)|^2 df} \end{aligned} \tag{6.31}$$

It can be shown that a matched filter maximizes the ratio above, so that it becomes [**carlson86**] :

$$\frac{A^2}{N_o} = \frac{A_p^2}{n_0/2}\int_{-\infty}^{+\infty}|P(f)|^2 df = \frac{A_p^2}{n_0/2}\int_{-\infty}^{+\infty}p(t)^2 dt \tag{6.32}$$

Thus, substituting from equation 6.28, the following relation can be verified for the signal after the matched filter:

$$\frac{A^2}{N_o} = \frac{2E_b}{n_0} \tag{6.33}$$

Here, A is the amplitude of $s(t)$ at the output of the matched filter, N_o is the corresponding noise power, E_b is the energy per bit and $n_0/2$ is the two-sided noise spectral density, obtained from equation 6.26 from $n_0 = \sigma^2_{\text{ASE}-\text{LO}}/B_N$.

$x(t) = s(t) + n(t)$ remains a valid representation of the signal for each of the quadratures after the matched filter, even if $s(t)$ and $n(t)$ have changed. In addition, $n(t)$ remains additive Gaussian noise. Therefore, $x(t)$ after the matched filter, at time $t = nT_s$ is also a Gaussian random variable, and has mean A^2 and variance N_o. This relation will show itself useful during the analysis of the error probability in the following section.

6.1.2.5 Bit error rate

For each quadrature, an error occurs in one of two situations: when a 0 is transmitted but a 1 is identified, or a 1 is transmitted and a 0 is identified.

As previously mentioned, the values sampled at sampling time $t = nT_s$, which are used to identify the received symbols, follow a Gaussian distribution. Using the constellation from Figure 6.1, with a decision boundary halfway between A and $-A$, an error occurs in two situations:

$$\begin{cases} x(t) < 0, & \text{if } s(t) = A \\ x(t) > 0, & \text{if } s(t) = -A \end{cases} \quad\Longrightarrow\quad \begin{cases} n(t) < -A, & \text{if } s(t) = A \\ n(t) > A, & \text{if } s(t) = -A \end{cases} \tag{6.34}$$

This is illustrated in Figure 6.8, where the probability of error is shown by the colored area under the curves [**schwartz90**].

The probability of bit error (assuming equal emission probabilities for both bits) can be expressed as:

$$\begin{aligned} P_{be} &= P_0 P_{e0} + P_1 P_{e1} = \frac{1}{2}P_{e0} + \frac{1}{2}P_{e1} \\ &= \frac{1}{2}\Big[\Pr\Big(n(t) < -A\Big) + \Pr\Big(n(t) > A\Big)\Big] \end{aligned} \tag{6.35}$$

P_0 and P_1 are the probabilities of the transmitted bit being a 0 or 1, respectively, and P_{e0} and P_{e1} are the respective probabilities of error. It follows that the probability of bit error can be described using the Q-function, or alternatively, the complementary error function.

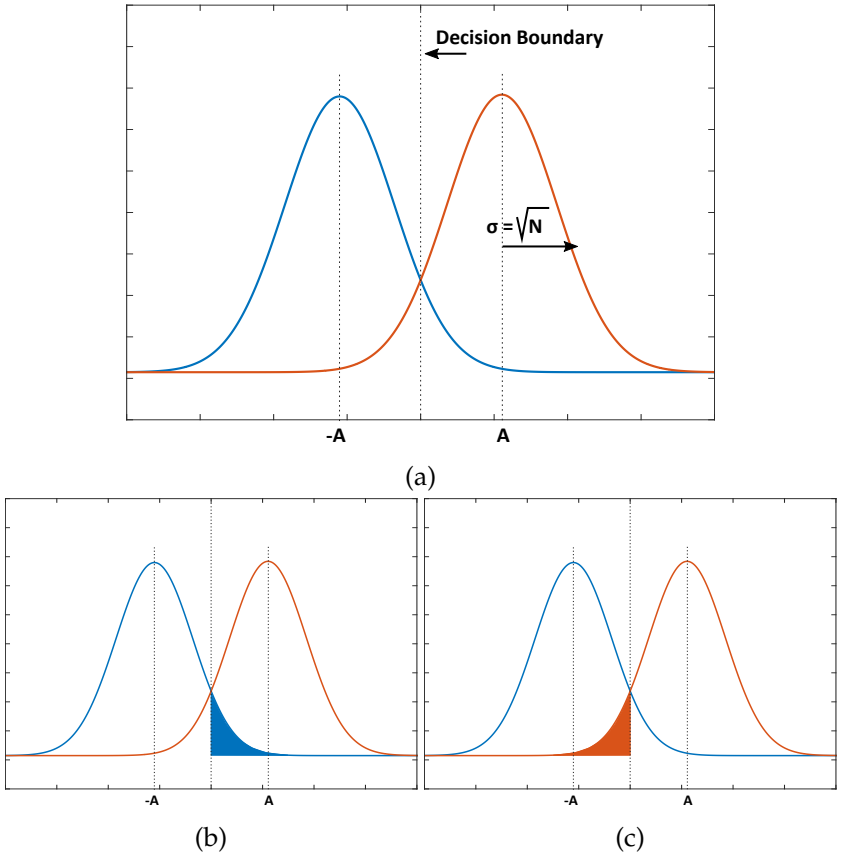


Figure 6.8: Probability density functions for $x(t) = s(t) + n(t)$, with $s(t) = \pm A$ at the time of sampling, and $n(t)$ as a Gaussian variable of zero mean and variance N. The colored areas below the curves on the right represent the probability of error for each transmitted bit.

$$P_{be} = Q\left(\frac{A}{\sqrt{N_o}}\right) = \frac{1}{2}\text{erfc}\left(\frac{A}{\sqrt{2N_o}}\right) \tag{6.36}$$

The probability of bit error is also known as bit error rate, or BER.

We have worked on the probability of bit error for each quadrature independently. However, assuming a gray code, and knowing that the carriers are uncorrelated, an error in each carrier is independent from the other [**nguyen09**]. This can be verified by noting that the even bits are dependent only on one of the quadratures, and the odd bits depend on the other. Therefore, as half the bits are even and half are odd, and assuming that the constellation is symmetrical, it follows that the error rate is the same as for each quadrature independently.

$$\begin{aligned}
\text{BER} &= P_{\text{odd}} P_{\text{oddError}} + P_{\text{even}} P_{\text{evenError}} \\
&= \frac{1}{2} P_{\text{oddError}} + \frac{1}{2} P_{\text{evenError}} \\
&= \frac{1}{2} \left(Q\left(\frac{A_{even}}{\sqrt{N_{even}}} \right) + Q\left(\frac{A_{odd}}{\sqrt{N_{odd}}} \right) \right) \\
&= Q\left(\frac{A}{\sqrt{N_o}} \right)
\end{aligned} \tag{6.37}$$

Looking back at equation 6.33, we can also define the probability of error as a function of the energy per bit:

$$P_{be} = Q\left(\sqrt{\frac{2E_b}{n_0}} \right) = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{E_b}{n_0}} \right) \tag{6.38}$$

While the BER can be calculated by analyzing each quadrature independently, the probability of symbol error is different. In this case, a symbol error happens whether there is an error in one of the quadratures or in both. There is no distinction between these cases. Therefore, the calculation of the symbol error rate requires a different approach. Let us first define the probability of correctly identifying a symbol.

$$P_C = (1 - P_{be})^2 \tag{6.39}$$

From this, the probability of symbol error is:

$$\begin{aligned}
P_{Se} &= 1 - P_C \\
&= 1 - \left(1 - Q\left(\frac{A}{\sqrt{N_o}} \right) \right)^2 \\
&= 2Q\left(\frac{A}{\sqrt{N_o}} \right) \left[1 - \frac{1}{2} Q\left(\frac{A}{\sqrt{N_o}} \right) \right] \\
&= 2Q\left(\sqrt{\frac{2E_b}{n_0}} \right) \left[1 - \frac{1}{2} Q\left(\sqrt{\frac{2E_b}{n_0}} \right) \right]
\end{aligned} \tag{6.40}$$

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

We can now write the expected bit error rate as a function of the system parameters. To get E_b, we can start by picking up the equations for the signal waveforms which are sampled and measured at the ADC, described in Equations 6.16 and 6.17.

$$E_b(t) = G_e^2 \eta^2 G_o P_s P_{lo} \frac{T_s}{2} \tag{6.41}$$

To get n_0, we start with equation 6.26, which gives the noise variance at the ADC's. The noise power spectral density will be the ration of this variance to the electrical bandwidth of the receiver. So we get:

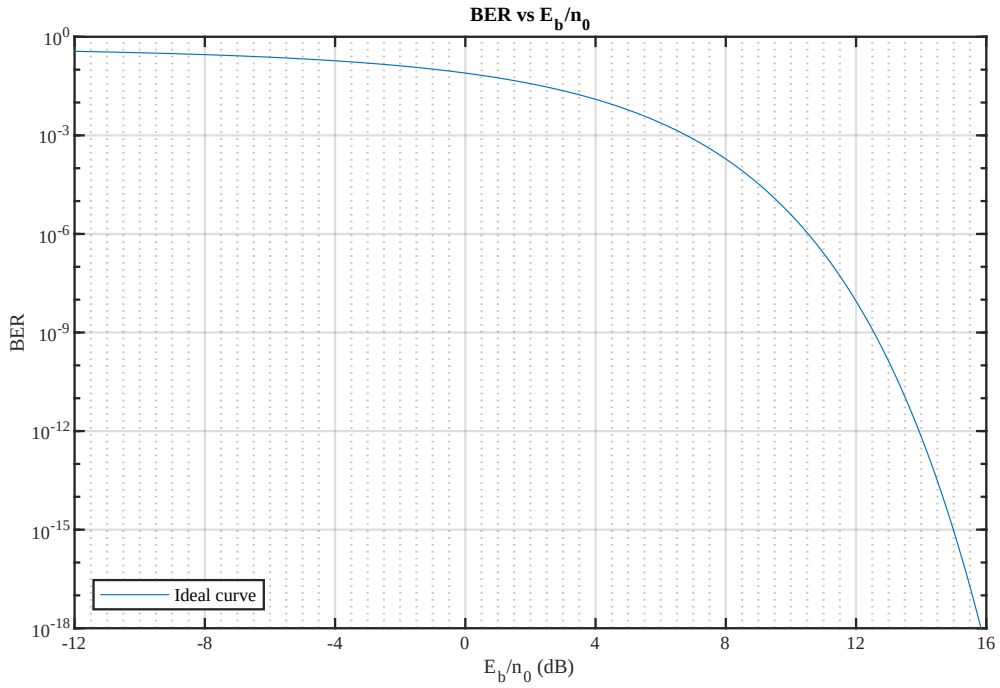


Figure 6.9: QPSK theoretical BER curve as a function of the ratio of energy per bit and the noise spectral density in dB.

$$\begin{aligned} n_0 &= \frac{2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu B_N}{B_N} \\ &= 2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu \\ &= 2\eta^2 G_e^2 P_{\text{LO}} n_{0o} \end{aligned} \tag{6.42}$$

6.1.3 Simulation Analysis

The M-QAM transmission system is a set of blocks that simulate the modulation, transmission and demodulation of an optical signal using M-QAM modulation. It is composed of several complex blocks: a transmitter, a receiver, a noise source, an addition block, a sink and a block that performs a Bit Error Rate (BER) measurement. The schematic representation of the system is presented in Figures 6.10 to 6.13. The simulation currently implements a QPSK system (M=4).

6.1.3.1 Functional description

A complete description of the M-QAM transmitter and M-QAM homodyne receiver blocks can be found in the *Library* chapter of this document as well as a detailed description of the independent blocks that compose these blocks. The M-QAM transmitter generates one or two optical signals by encoding a binary string using M-QAM modulation. It also outputs a binary signal that is used to perform the BER measurement. The optical signal is then sent through a fiber block, which attenuates it, simulating transmission over a given distance. Aftwerwards, as EDFA block is present to be used as an optical preamplifier. The amplified optical signal, with added ASE noise, is then sent to the homodyne receiver.

The homodyne receiver requires two optical inputs: one of the signal itself, and another from a local oscillator perform the signal demodulation. It receives, processes and decodes the received signal and afterwards outputs the reconstructed bitstream. This signal is compared to the binary signal generated by the transmitter in order to estimate the Bit Error Rate (BER). The files used are summarized in tables 6.1 and 6.2. All the blocks and sub-blocks used are included in the tables.

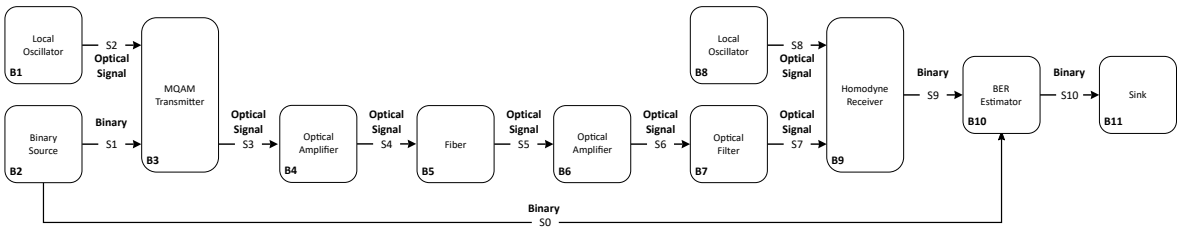


Figure 6.10: Top-Layer Schematic representation of the simulated MQAM system.

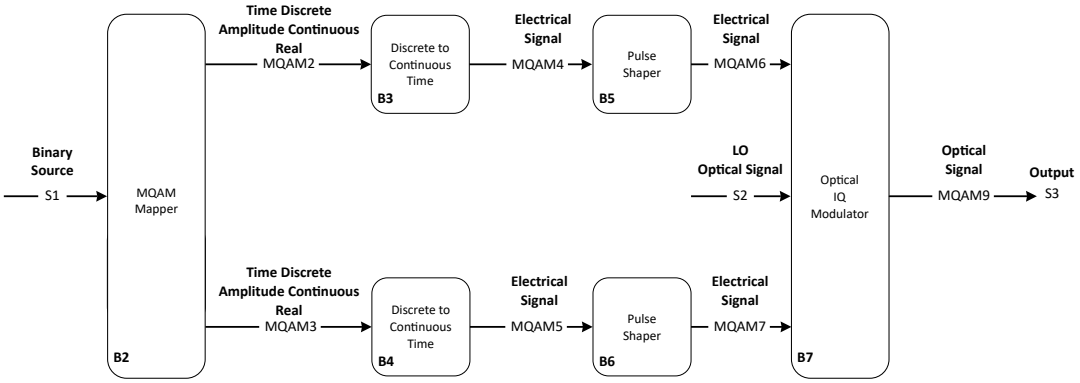


Figure 6.11: Schematic representation of the MQAM Transmitter block.

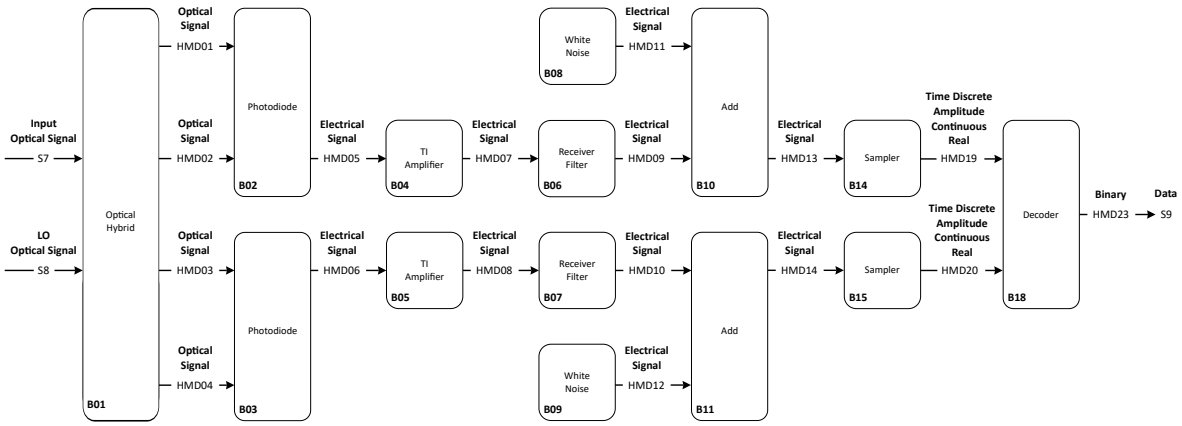


Figure 6.12: Simplified schematic representation of the Homodyne Receiver block.

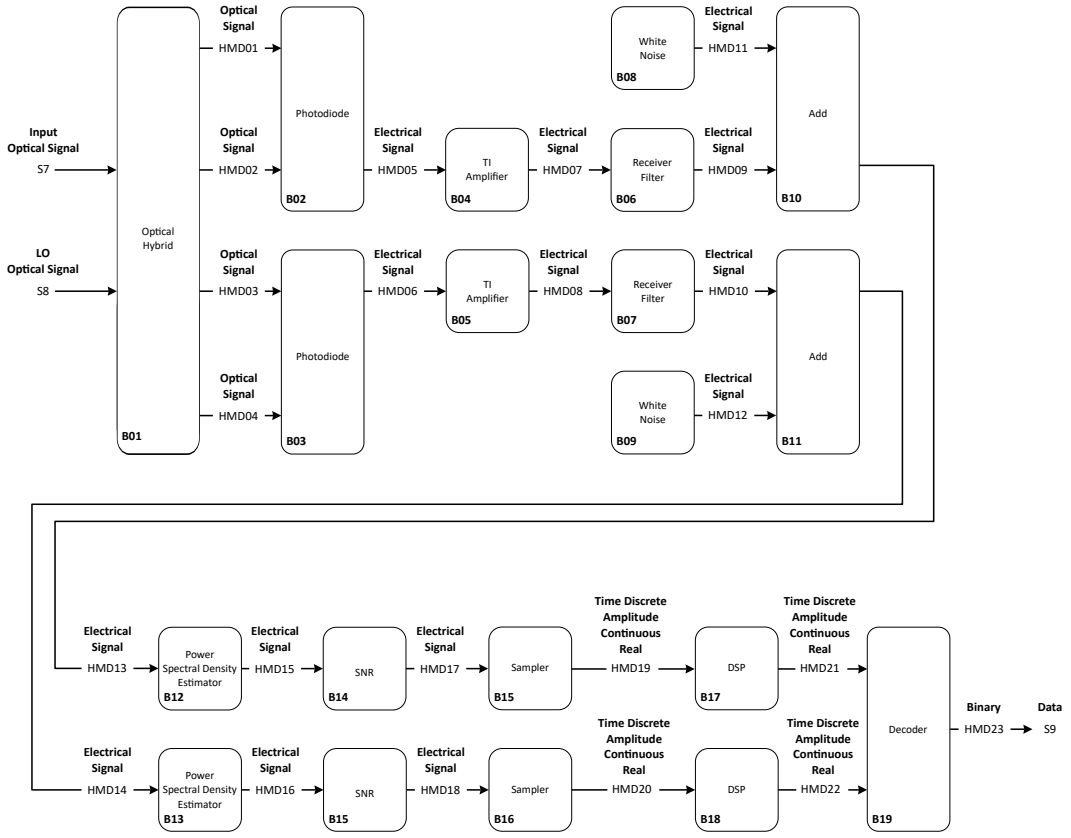


Figure 6.13: Schematic representation of the Homodyne Receiver block, with the DSP block (currently not implemented) and two optional blocks for measuring the SNR and power spectral density.

6.1.3.2 Required files

Source Files		
File	**Comments**	**Status**
add_20190215.cpp		✓
binary_source_20190215.cpp		✓
bit_error_rate_20190215.cpp		✓
decoder_20190215.cpp		✓
discrete_to_continuous_time_20190215.cpp		✓
electrical_filter_20190215.h		✓
edfa_20190215.cpp		✓
fft_20180208.cpp		✓
fiber_20190215.cpp		✓
homodyne_receiver_withoutLO_20190215.cpp	[1]	✓
ideal_amplififer_20190215.cpp		✓
iq_modulator_20190215.cpp		✓
local_oscillator_20190215.cpp		✓
m_qam_mapper_20190215.cpp		✓
m_qam_system_sdf.cpp	[2]	✓
m_qam_transmitter_20190215.cpp		✓
netxpto_20190215.cpp	[2]	✓
optical_hybrid_20190215.cpp		✓
photodiode_old_20190215.cpp		✓
power_spectral_density_estimator_20190215.cpp		✓
pulse_shaper_20190215.cpp		✓
sampler_20190215.cpp		✓
sink_20190215.cpp		✓
snr_estimator_20190215.cpp		✓
super_block_interface_20190215.cpp	[2]	✓
white_noise_20190215.cpp		✓
window_20180704.cpp		✓

Table 6.1: [1] The library entry is under a different name, *m_qam_receiver*;
[2] No library entry as it is a main or general purpose file, not a specific block.

Header Files		
File	**Comments**	**Status**
add_20190215.h		✓
binary_source_20190215.h		✓
bit_error_rate_20190215.h		✓
decoder_20190215.h		✓

discrete_to_continuous_time_20190215.h		✓
electrical_filter_20190215.h		✓
edfa_20190215.h		✓
fft_20180208.h		✓
fiber_20190215.h		✓
homodyne_receiver_withoutLO_20190215.h	[1]	✓
ideal_amplifier_20190215.h		✓
iq_modulator_20190215.h		✓
local_oscillator_20190215.h		✓
m_qam_mapper_20190215.h		✓
m_qam_transmitter_20190215.h		✓
netxpto_20190215.h	[2]	✓
optical_hybrid_20190215.h		✓
photodiode_old_20190215.h		✓
power_spectral_density_estimator_20190215.h		✓
pulse_shaper_20190215.h		✓
sampler_20190215.h		✓
sink_20190215.h		✓
snr_estimator_20190215.h		✓
super_block_interface_20190215.h	[2]	✓
white_noise_20190215.h		✓
window_20180704.h		✓

Table 6.2: [1] The library entry is under a different name, *m_qam_receiver*
[2] No library entry as it is a main or general purpose file, not a specific block.

6.1.3.3 Input Parameters

Table 6.3: Input parameters

Parameter	Type	Description
numberOfBitsGenerated	t_integer	Determines the number of bits to be generated by the binary source
samplingRate	double	The simulation sampling rate
symbolRate	double	The symbol rate of the main signal
samplesPerSymbol	t_integer	Number of samples per symbol. Defined from the samplingRate and symbolRate.
symbolPeriod	double	Period of the main signal

bitPeriod	double	Periodicity of bits in the main signal
prbsPatternLength	int	Determines the length of the pseudorandom sequence Pattern (used only when the binary source is operated in *PseudoRandom* mode)
bitPeriod	t_real	Temporal interval occupied by one bit
rollOffFactor_shp	t_real	Roll-off factor of the pulse shaper filter
rollOffFactor_out	t_real	Roll-off factor of the output filter
shaperFilter	enum	Type of filter used in Pulse Shaper
outputFilter	enum	Type of filter used in output filter
seedType	enum	Method of seeding noise generators
seedArray	array<int,2>	Seeds to initialize noise generators
signalOutputPower_dBm	t_real	Determines the power of the output optical signal in dBm
numberOfBitsReceived	int	Determines when the simulation should stop. If -1 then it only stops when there is no more bits to be sent
symbolPeriod	double	Calculated from symbolRate
fiberLength_m	t_real	Optical fiber length
attentuationCoefficient	t_real	Optical fiber attenuation coefficient
dispersionCoefficient	t_real	Optical fiber dispersion coefficient
opticalGain_dB	t_real	Optical gain of the EDFA
noiseFigure	t_real	Noise figure of the EDFA
localOscillatorPower_dBm	t_real	Power of the local oscillator
localOscillatorPhase	t_real	Phase of the local Oscillator

responsivity	t_real	Responsivity of the photodiodes (1 corresponds to having all optical power transformed into electrical current)
amplification	t_real	Amplification provided by the ideal amplifier
thermalNoiseSpectralDensity	t_real	Noise spectral density added after the electrical filter
amplifierNoiseSpectralDensity	t_real	Electrical noise spectral density added before the electrical filter
elFilterType	enum	Type of the electrical filter: generated low pass or defined by coefficients
elFilterOrder	enum	Order of the electrical filter
impulseResponseArr	t_real[]	Array of coefficients of the electrical filter.
iqAmplitudeValues	vector<t_iqValues>	Determines the constellation used to encode the signal in IQ space
samplesToSkip	t_integer	Number of samples to be skipped by the *sampler* block
confidence	t_real	Determines the confidence limits for the BER estimation
midReportSize	t_integer	
bitSourceMode	enum	Mode of generating the bitstream for the signal
electricalSNRMethod	enum	Mode for calculating the SNR prior to the matched filter
filteredSNRMethod	enum	Mode for calculating the SNR after the matched filter
SNRignoreSamples	int	Number of samples to initially ignore when calculating the SNR
SNRsegmentSize	int	Size of each segment used for estimating the SNR
powerSpectralDensityOverlap	double	Percentage of the signal to overlap when averaging periodograms in power spectral density estimation

powerSpectralDensitySegment	int	Size of segment used for calculating each periodogram
powerSpectralDensityInterval	int	Number of samples to acquire before estimating the power spectral density
bufferLength	t_integer	Corresponds to the number of samples that can be processed in each run of the system

6.1.3.4 Output Files

Table 6.4: Output Files

Files	**Description**
Signal.sgn	Files with corresponding signal data generated in the simulation.
BER.txt	Results from bit_error_rate block.
log.txt	Log file from simulation.
params.txt	Input parameter list.
PowerSpectralDensity.txt	Power spectral density estimate from optical signal.
PowerSpectralDensity2.txt	Power spectral density estimate from electrical signal.
SNR.txt	SNR estimate before matched filter.
SNRFiltered.txt	SNR estimate after matched filter.
impulse_response.imp	Impulse response from electrical filter.
out_filter.imp	Impulse response from matched filter.
pulse_shaper.imp	Pulse shaper impulse response.

6.1.3.5 Simulation results - ISI

In this section we will explore the signals generated during the simulation. The general scheme of the simulation is shown in Figures 6.10, 6.11 and 6.13. Every signal generated during the simulation is identified in those diagrams.

We will start by analyzing the intersymbol interference (ISI) in the signals generated on the simulation. For this purpose, we will turn of all noise sources. We will be using root-raised cosines at the pulse shaper (*B5* and *B6* on MQAM transmitter) and at the matched filter (*B18* and *B19* at the homodyne receiver). With this configuration, we should obtain a perfect copy of the transmitted constellation, affected only by a scaling factor (which could be removed by adjusting the *amplification* parameter).

The parameters used to obtain the plots and eye diagrams in this section are displayed in Table 6.5.

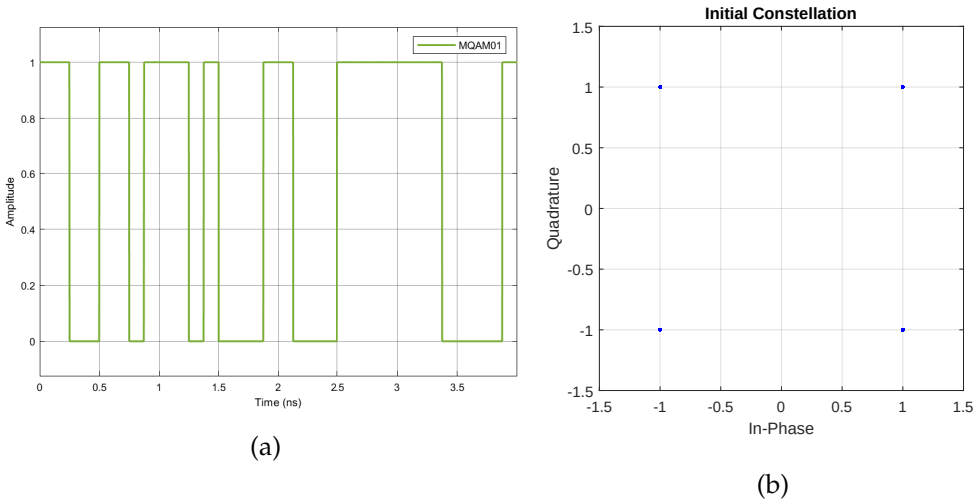
Table 6.5: Simulation parameters

Parameter	**Value**	**Units**
numberOfBitsGenerated	100×10^3	
samplingRate	64×10^9	Hz
symbolRate	4×10^9	Bd
samplesPerSymbol	16	
symbolPeriod	250×10^{-12}	s
bitPeriod	125×10^{-12}	s
signalOutputPower_dBm	-10	dBm
localOscillatorPower_dBm	0	dBm
localOscillatorPhase	0	rad
nBw	18×10^9	Hz
amplification	3162.28	
responsivity	1	A/W
outputFilter	RootRaisedCosine	
shaperFilter	RootRaisedCosine	
rollOffFactor_out	0.9	
rollOffFactor_shp	0.9	
seedType	RandomDevice	
numberOfBitsReceived	-1	
elFilterType	Defined	
elFilterOrder	20	
opticalGain_dB	0	dB
noiseFigure	0	dB
preFilterNoiseSpectralDensity	0	W/Hz
elNoiseSpectralDensity	0	W/Hz
bufferLength	512	
bitSourceMode	Random	

confidence	0.95	

We will start by showing the signals present in the *m_qam_transmitter* block, shown in Figure 6.11. This is the block where the signals are generated and modulated, and remains unaltered for all cases shown here. Therefore, to avoid repetition, these signals will only be shown here, as their properties remain similar for all the other simulation that will be examined further on.

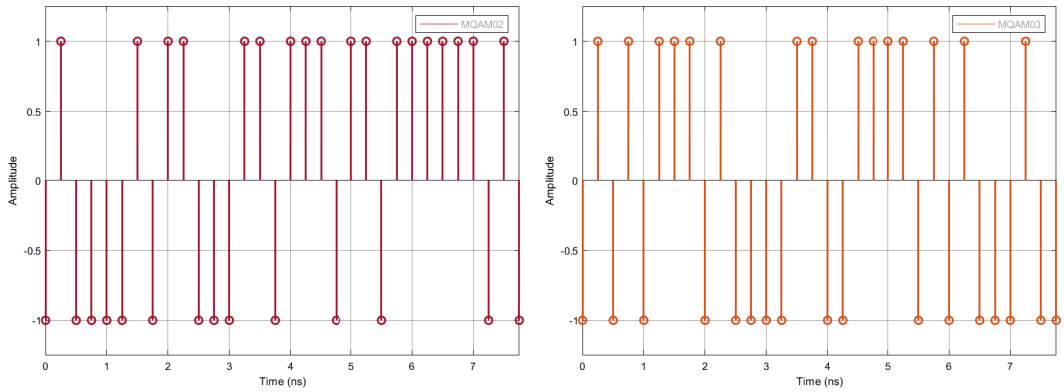
First, the initial bitstream is generated. This is the pseudorandom array of ones and zeros which will be transmitted, and later compared with the decoded signal.



(a) (b)

Figure 6.14: Eye diagram of initial bitstream MQAM01 (pseudorandom ones and zeros), and respective constellation to be used.

This series of bits needs to be encoded into the corresponding coordinate points, according to the chosen constellation and modulation format. As we are using a 4 point QAM modulation, these coordinate points are (1,1), (1,-1), (-1,-1) and (-1,1). Mapping the bits to these points is done in the *MQAM_mapper* block. This block receives the binary sequence and outputs two signals, discrete in time and value. Each bit is alternately coded into one of the output signals, according to the chosen constellation. In this case, the values are either 1 or -1. Each signal contains the values ultimately used to modulate either the in-phase or quadrature component.

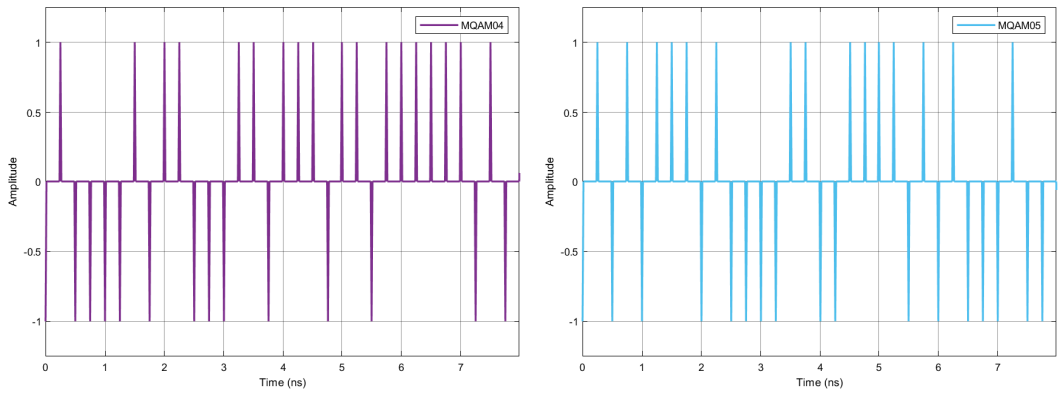


(a) (b)

Figure 6.15: Signals MQAM02 (a) and MQAM03 (b), containing the values encoded and distributed to the in-phase and quadrature components.

As mentioned before, the signals MQAM02 and MQAM03 are discrete in both value and time. However, in order to be used for modulating the optical signal, they need to be continuous in time. Also, they have to be assigned a given continuous shape that can be modulated onto the optical signal.

The first of these requirements is fixed with the *discrete_to_continuous_time* block. This block takes the discrete sequence of values generated on the previous block and outputs an equivalent where each value (or symbol) has a proper timing. Therefore, it outputs the previous sequence of values, but in an array continuous in time, with each value separated by an amount of time equal to the desired symbol period.

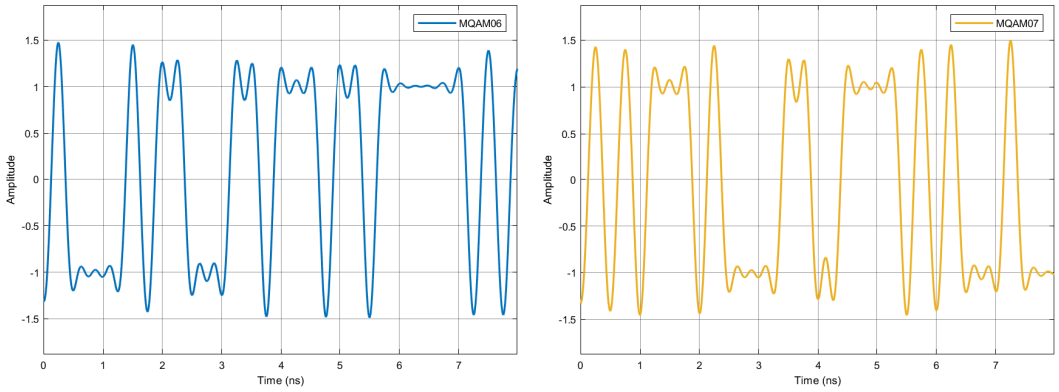


(a) (b)

Figure 6.16: Signals MQAM04 (a) and MQAM05 (b), containing the values in a signal with continuous time.

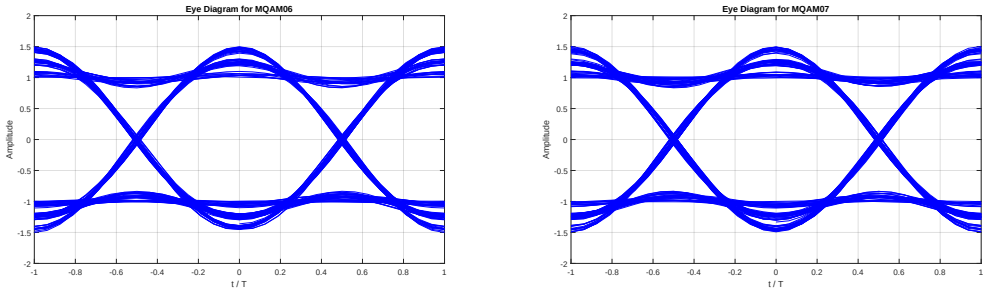
The signals still need to be assigned a continuous shape in order to modulate them into the optical signal. This is done with a pulse shaper, which acts as a FIR filter shaping the discretely-valued sequences of symbols. As mentioned in the beginning of the section, the

chosen filter is a root-raised-cosine.



(a) (b)

Figure 6.17: Signals MQAM06 (a) and MQAM07 (b), containing the signals to be modulated, already shaped with a root-raised-cosine filter.



(a) (b)

Figure 6.18: Eye diagrams for MQAM06 (a) and MQAM07 (b), shaped with root-raised-cosines.

It can be seen in the eye diagrams that the signal is not free from ISI. However, as mentioned in section 6.1.2.3, the shaping is done at the transmitter and at the receiver. We shall see further ahead that the signal will be free of ISI at sampling time.

Being continuous in time and in value, signals MQAM06 and MQAM07 are then ready to be passed on to the *iq_modulator* block, where they are modulated into an optical signal, and then transmitted.

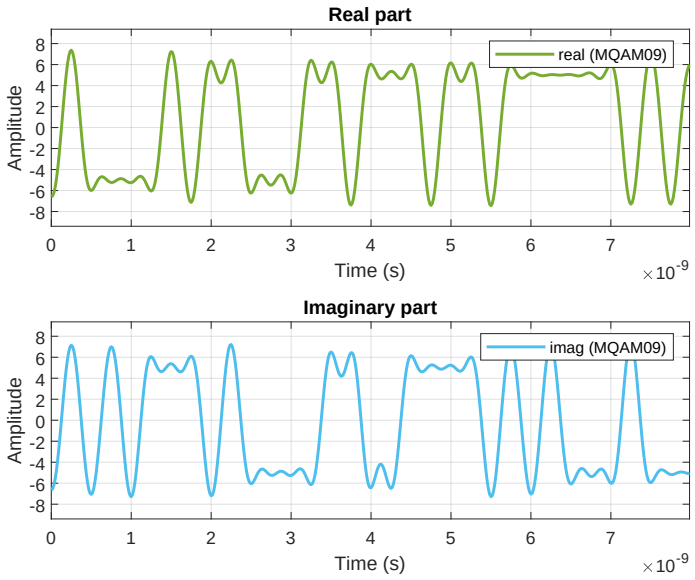


Figure 6.19: Signal MQAM09, modulated optical signal.

MQAM01 and MQAM09 are the output signals of the *m_qam_transmitter* block. They are equal to the top level signals S0 and S1, respectively. Normally the S1 signal is now directed to an *optical_fiber* block, which is then connected to an *edfa* block. However, in this simulation they are both configured to have no effect, and so we shall not show them here. The optical signal then proceeds to be used as input to the *homodyne_receiver_withoutLO* block, along with S4, an optical signal acting as local oscillator.

We shall now explore the process inside the *homodyne_receiver_withoutLO* block. The inputs of the block are mixed inside an *optical_hybrid* block. These outputs are a mix of the transmitted optical signal and the local oscillator, as explained in section 6.1.2.2.

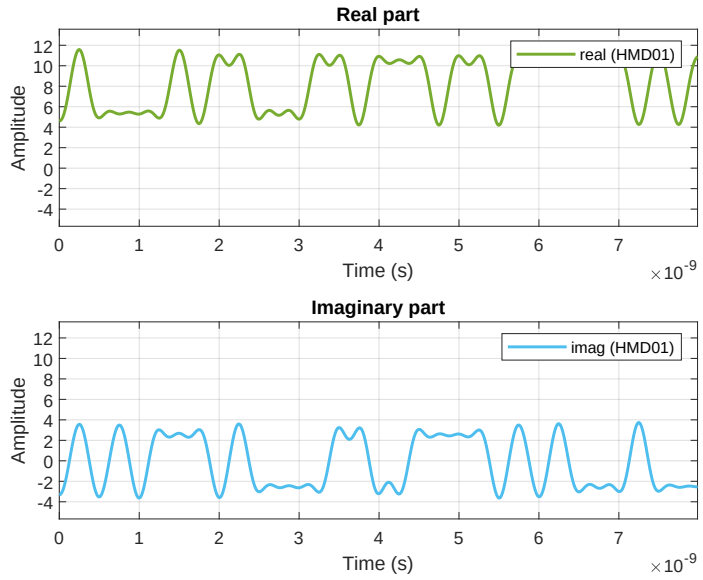


Figure 6.20: HMD01, output 1 optical hybrid.

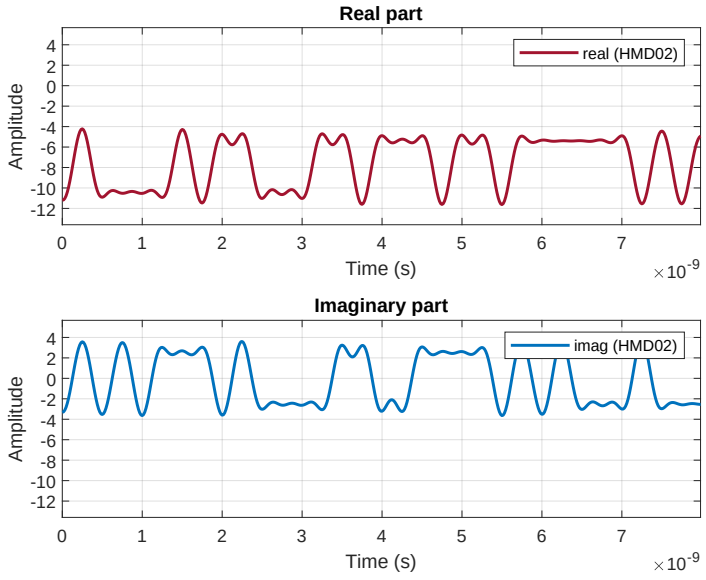


Figure 6.21: HMD02, output 2 of the optical hybrid.

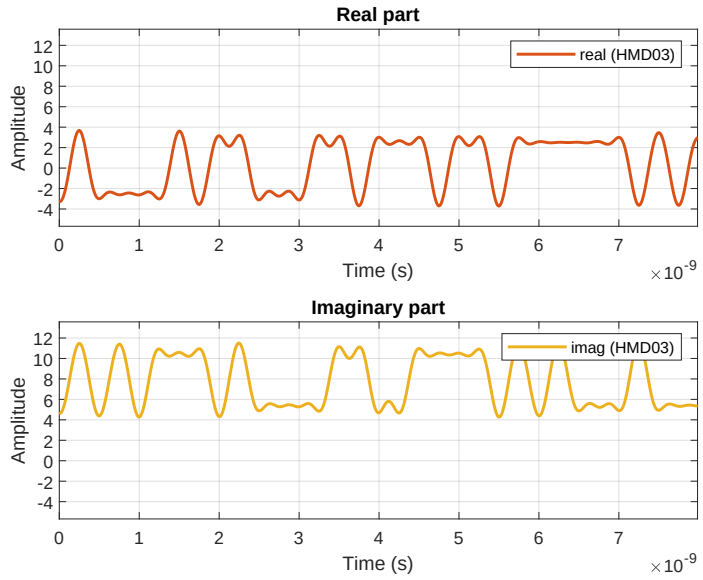


Figure 6.22: HMD03, output 3 of the optical hybrid.

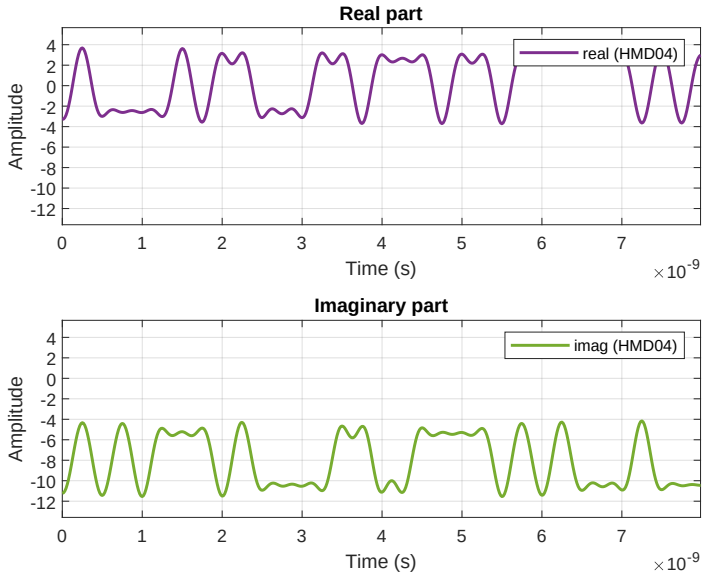
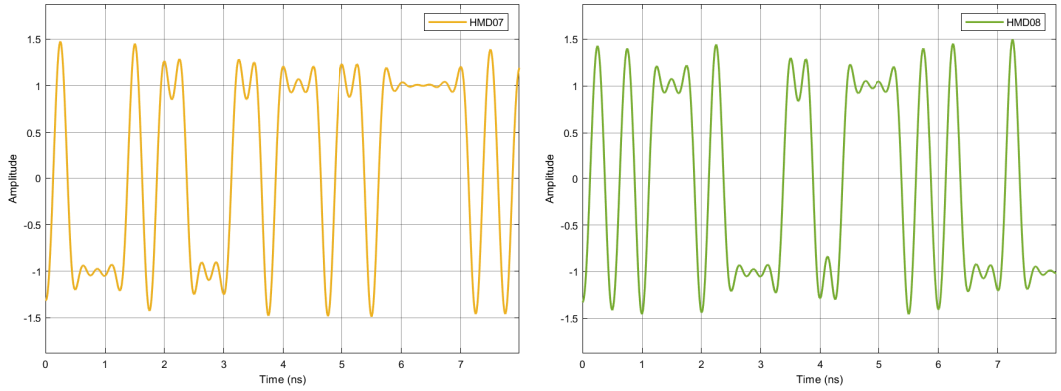


Figure 6.23: HMD04, output 4 of the optical hybrid.

These optical signals are then sent in pairs to two *photodiode* blocks, where they are detected (with a *responsivity* defined in the parameters) and subtracted. The output of the photodiode blocks is then directed to the *ti_amplifier* blocks, which generate the signals shown in Figure 6.24. The *ti_amplifier* usually does three things: it adds noise, amplifies the signal and noise, and lastly passes the signal and noise through a filter. However, as

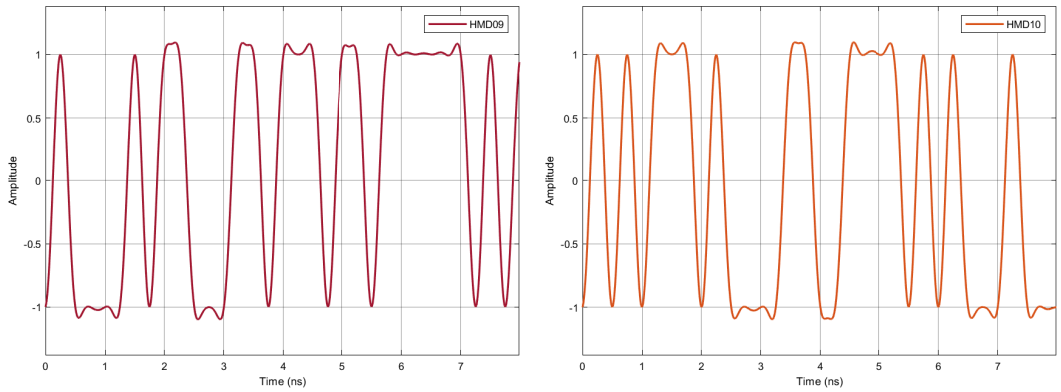
there is no noise here and the filter's bandwidth is much larger than the signal bandwidth, the only visible effect is the amplification.



(a) (b)

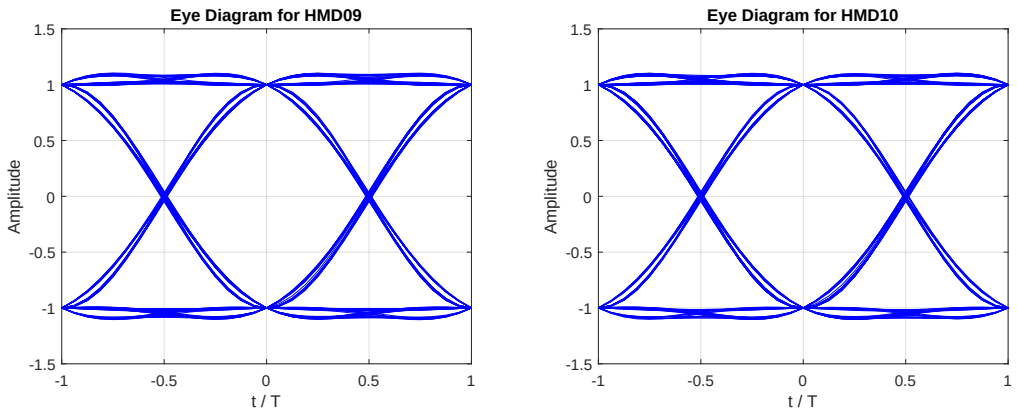
Figure 6.24: Signals HMD07 (a) and HMD08 (b), containing the amplified versions of the signals detected at the *photodiode* blocks

The next step is the matched filter. Again, as there is no noise, the only visible effect is the change in the signal shape. By using another root-raised-cosine filter, the signal now follows a raised-cosine shape.



(a) (b)

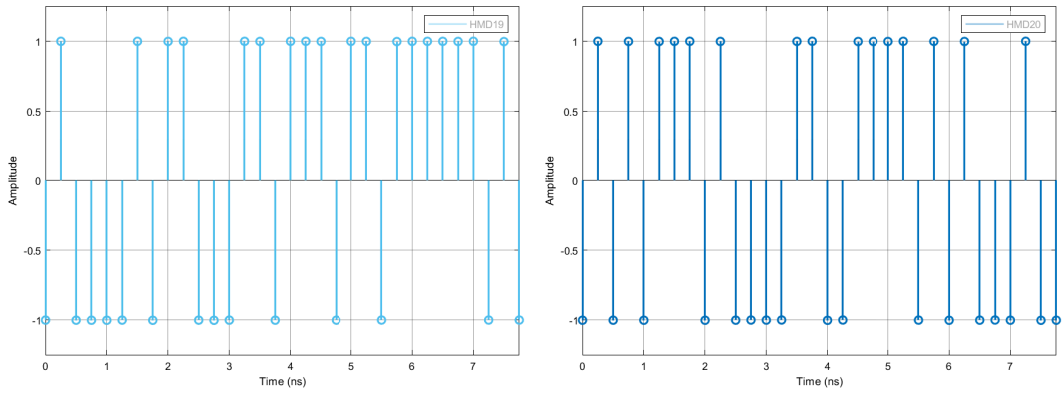
Figure 6.25: Signals HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter.



(a) (b)

Figure 6.26: Eye diagrams for HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter. They are now following a raised-cosine shape.

For the purpose of this simulation, no electrical noise is added at the receiver. So HMD09 and HMD10 are effectively the signals that will be sampled. As can be seen from their eye diagrams in Figure 6.26, they suffer from no intersymbol interference, having always the same exact value at sampling time. This is then sampled and then decoded, transforming the received signal into a bitstream. As the reception is perfect, the received bitstream is exactly equal to the transmitted one.



(a) (b)

Figure 6.27: Signals HMD19 (a) and HMD20 (b), sampled at instants $t = T_s$.

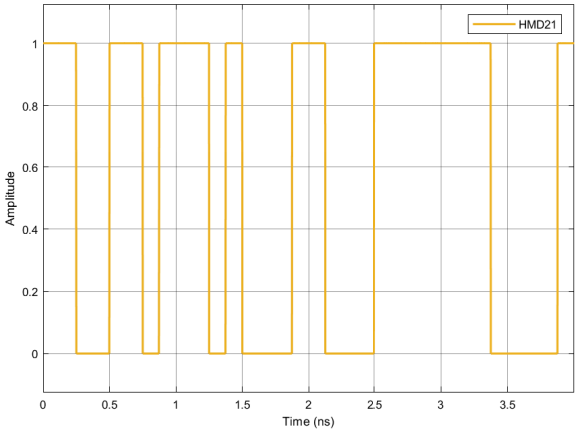


Figure 6.28: Signal HMD21, containing the decoded bitstream.

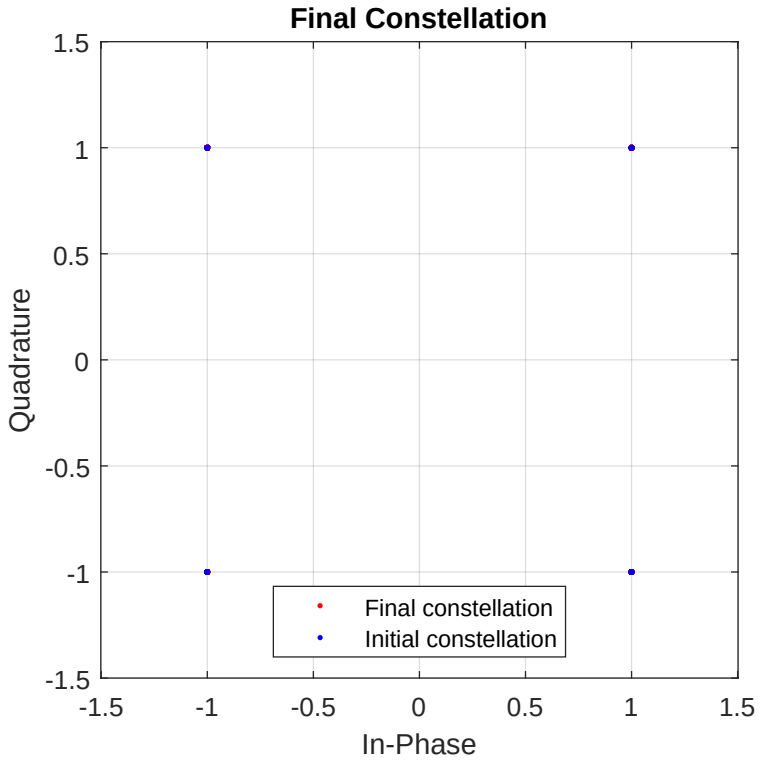
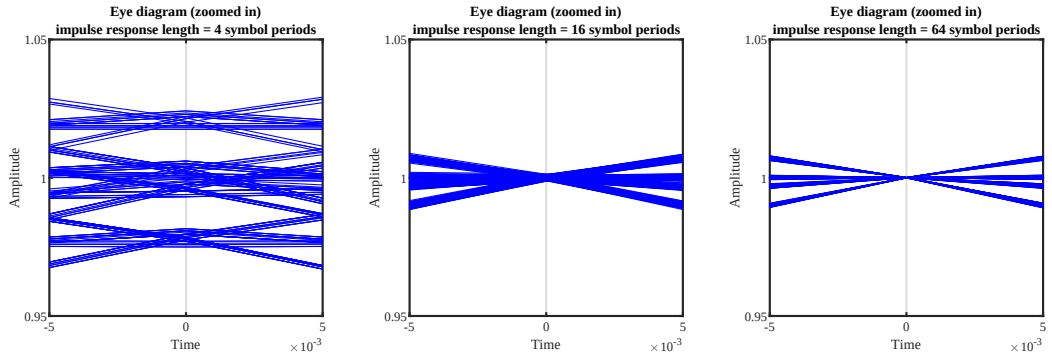


Figure 6.29: Constellation of the encoded and decoded signals. They are exactly equal due to lack of ISI and noise.

We can see that the received constellation is almost equal to the transmitted one, with all received bits having nearly no variation from their amplitude value. However, looking closely, we can still see a bit of the red points in figure 6.29. This a consequence of the finite impulse response of the used filters. The ideal filters, which provide zero ISI, have infinite

impulse response.



(a) (b) (c)

Figure 6.30: Zoom in on the eye diagram at sampling time. The figures used filter impulse responses with lengths of (a) 4, (b)16 and (c) 64 symbol periods. As the filter impulse response increases in size, the vestigial amount of ISI diminishes.

However, the practical implementation of the FIR filter implies that the impulse response is finite, leading to very small variations at sampling time. This effect can be diminished by using a larger impulse response. We have conducted the simulations using impulse responses with a size of 16 symbol periods. We can see in Figure 6.30b that this effect is not particularly significant for this situation, and thus we can consider the intersymbol interference to be negligible.

6.1.3.6 Simulation results - Thermal Noise

After showing that the implemented shaping process does not create any inter-symbol interference, we will now verify the effects of electrical noise. Electrical noise can have several origins. For now, we will only consider thermal noise.

Thermal noise in the simulation will be modeled with a additive white Gaussian noise source, added after immediately before the sampler. According to this, thermal noise will not be affected by either the amplifier or matched filter, directly affecting the sampled signal. In addition, it is considered that the noise power is constant for a given temperature, which we shall consider to be 290 K.

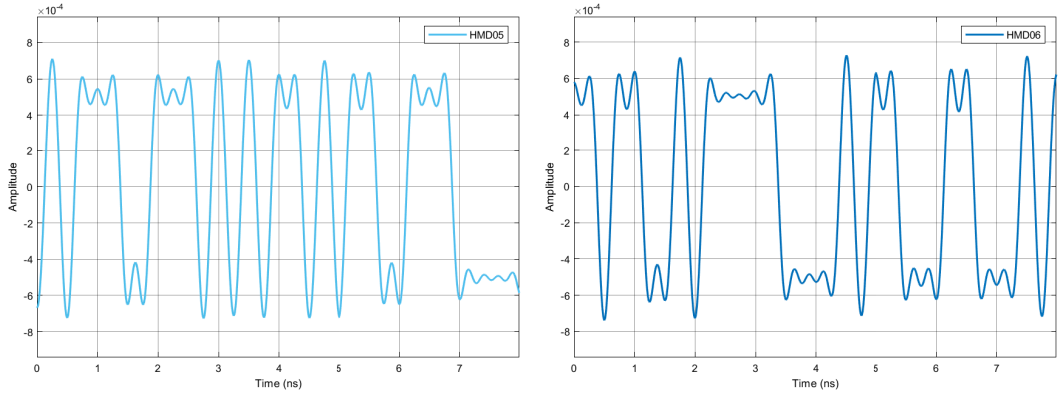
The effect of this noise source is to establish a limit to the detection performance. Without any noise, the detected constellation could always be replicated perfectly, even if scaled down with the optical output power. Having a constant noise source makes it so that signals below a given output power will not be properly detected, being indistinguishable from noise.

Table 6.6: Simulation parameters

Parameter	**Value**	**Units**
numberOfBitsGenerated	100×10^3	

samplingRate	64×10^9	Hz
symbolRate	4×10^9	Bd
samplesPerSymbol	16	
symbolPeriod	250×10^{-12}	s
bitPeriod	125×10^{-12}	s
signalOutputPower_dBm	-6	dBm
localOscillatorPower_dBm	0	dBm
localOscillatorPhase	0	rad
nBw	18×10^9	Hz
responsivity	1	A/W
amplification	0	
amplifierInputNoisePowerSpectralDensity	0	
thermalNoisePower	2.56248e-08	
rxResistance	50	Ω
temperatureKelvin	290	K
outputFilter	RootRaisedCosine	
shaperFilter	RootRaisedCosine	
rollOffFactor_out	0.9	
rollOffFactor_shp	0.9	
seedType	RandomDevice	
numberOfBitsReceived	-1	
elFilterType	Defined	
fiberLength_m	0	m
fiberAttenuation_m	4.6052e-05	m^{-1}
elFilterOrder	20	
opticalGain_dB	0	dB
noiseFigure	0	dB
bufferLength	512	
bitSourceMode	Random	
confidence	0.95	

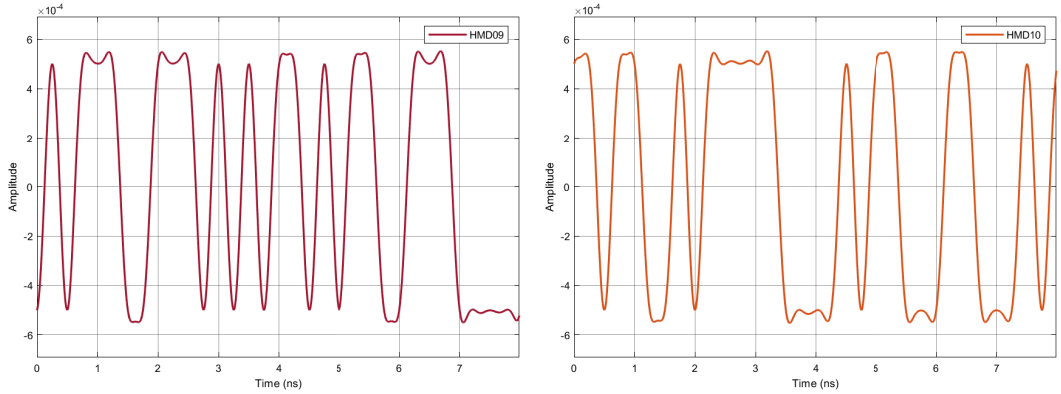
The transmitted signal is similar to the one shown in section 6.1.3.5, so it shall not be shown here. The starting constellation is also similar to the one in the previous section. Figure 6.31 shows the output of the photodiodes.



(a) (b)

Figure 6.31: Signals HMD05 (a) and HMD06 (b), outputs of the photodiodes.

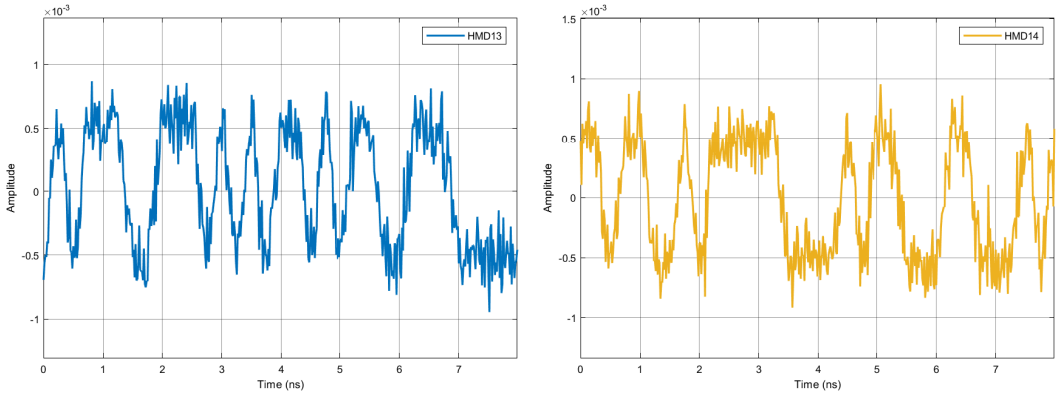
In this case no signal amplification is used, so there is no amplifier gain or noise. This means that the electrical signal will be much smaller than in the previous case. This can be verified by comparing Figures 6.32 and 6.25.



(a) (b)

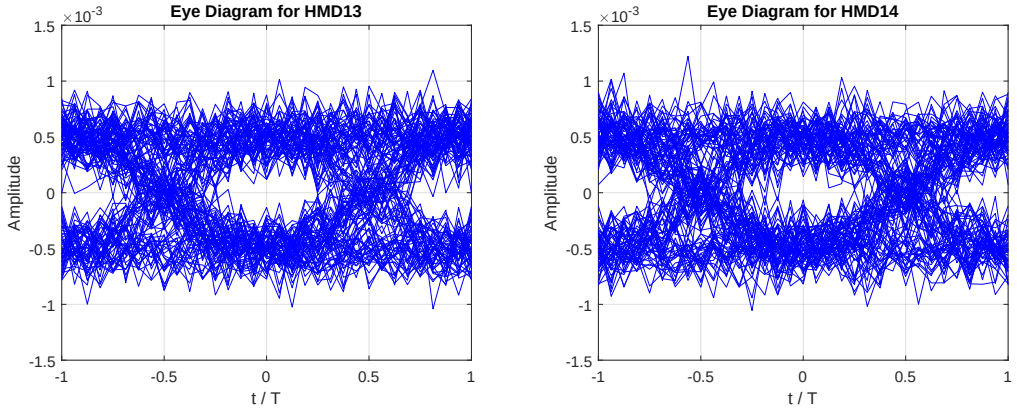
Figure 6.32: Signals HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter. They are now following a raised-cosine shape.

The thermal noise is then added after the matched filter. As it is only added at this point, it is not attenuated or filtered before the sampling process.



(a) (b)

Figure 6.33: Signals HMD13 (a) and HMD14 (b), after adding thermal noise with an RMS voltage amplitude of 1.6008×10^{-4} V.

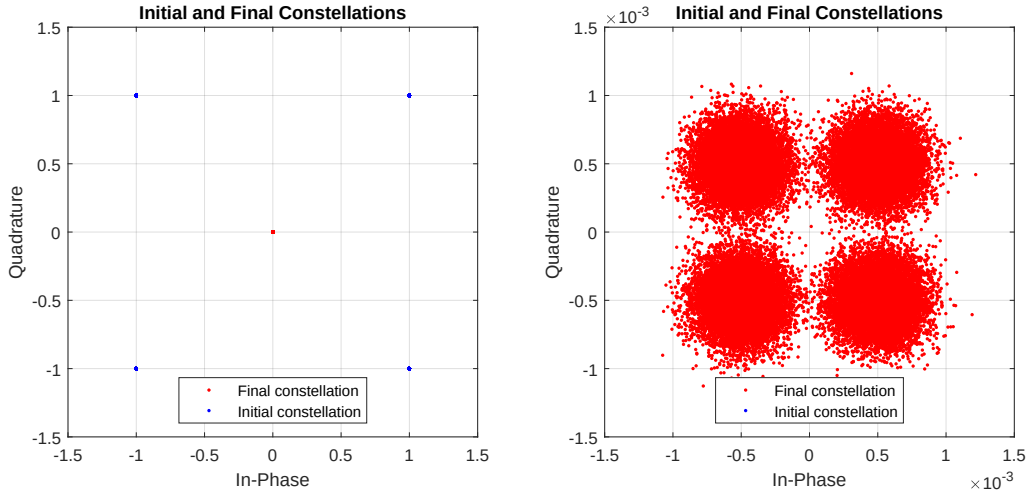


(a) (b)

Figure 6.34: Eye diagrams of signals HMD13 (a) and HMD14 (b), after adding thermal noise with an RMS voltage amplitude of 1.6008×10^{-4} V.

Figure 6.35 shows the final received constellations. Two things are worth noting:

- The amplitude of the received constellation points (1×10^{-3}) is much smaller than the initial constellation due to the lack of any amplification;
- The received signal and constellations are strongly affected by noise. The thermal noise is not a particularly strong source of noise: However, as the signal has not been amplified, they have comparable values.



(a) (b)

Figure 6.35: (a) Comparison of transmitted and received constellations. (b) Constellation of the decoded signals. Signal optical output power of 0 dBm and 30 km of fiber length (-6 dBm at receiver input).

The signals shown so far had an optical output power of -6 dBm. By attenuating using various fiber lengths, we plot a BER curve to study receiver sensitivity. Receiver sensitivity is usually defined as the minimum signal optical power required to achieve a certain BER performance [**hui09**]. The target BER used to measure the sensitivity may vary, and is usually chosen considering a desired minimum performance. For instance, in some cases a BER smaller than 10^{-12} may be desirable to ensure a very low error rate. In other cases, the requirement may be only 10^{-3} in order for the FEC to be effective.

In our case, there is no FEC, and the target BER value can be arbitrarily chosen to compare the different configurations. Therefore, we use choose to consider the minimum optical signal power required to reach a BER of 10^{-3}. This choice was made in order to allow reaching the values relatively quick, with a small confidence interval. This way, generating 10^5 bits we can get an average of 100 errors, which is important for the measurement precision. In order to achieve the same with a BER of 10^{-6} we would need to generate 10^8 bits, likely requiring several days to obtain a single datapoint from the simulation.

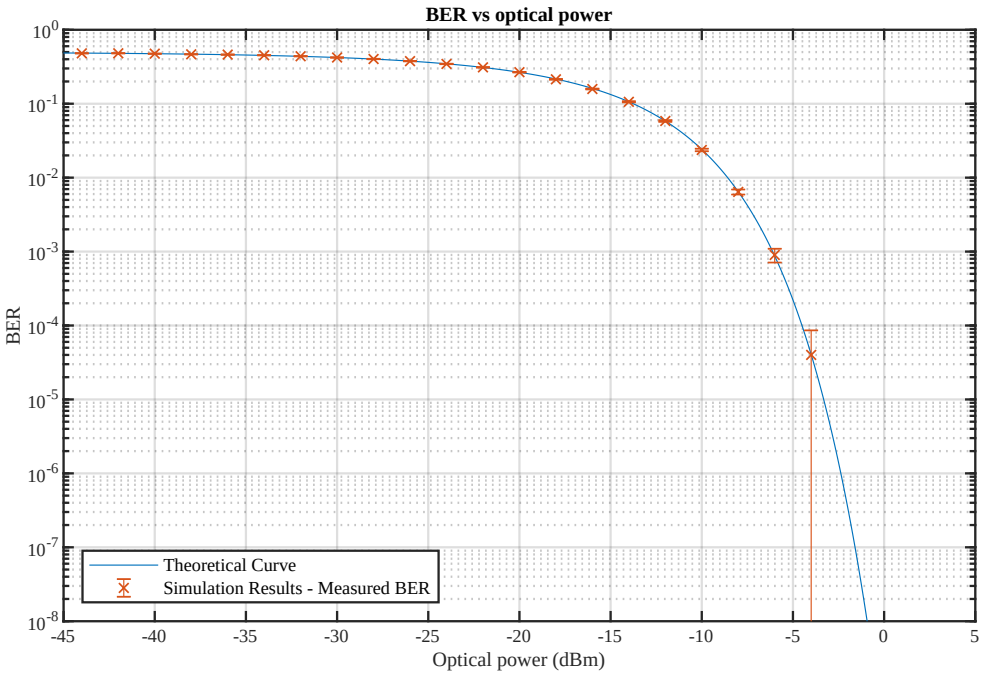


Figure 6.36: BER curve of the receiver with thermal noise.

The theoretical curve was obtained with

$$\text{BER} = \frac{1}{2}\text{erfc}\left(\frac{A}{\sqrt{2N}}\right) \tag{6.43}$$

with

$$A = \eta K \sqrt{P_{LO} P_s e^{-L\alpha}}$$
$$N = 4k_B TBR$$

where η is the responsivity, P_s is the optical output power, P_{LO} is the local oscillator optical output power, L is the fiber length, α is the attenuation constant and K is a constant related to the matched filter energy. The thermal noise power N is calculated as shown, where k_B is the Boltzmann constant, T is the absolute temperature in Kelvin, B is the bandwidth and R is the resistance.

For the sensitivity value to meaningful, in addition to the power and BER we should also specify the conditions in which those values were measured in the receiver, such as the data rate. We can therefore say the sensibility values presented here are obtained for a BER of $10^{-}3$, using a 4 GBd QPSK.

The sensitivity considering the thermal noise establishes the baseline sensitivity upon which to compare the rest of them. In the mentioned conditions, we have measured a sensibility of -6.1 dBm when using no amplification. We can use this value to compare

with the rest of the simulations, in order to quantify the increased performance provided by improvements on the system.

6.1.3.7 Simulation results - Electrical Noise

Keeping the thermal noise, we will now add a transimpedance amplifier after each photodiode. The amplifier has a certain bandwidth, gain and input referred noise. We assume that the noise gain is equal to the signal gain in this amplifier.

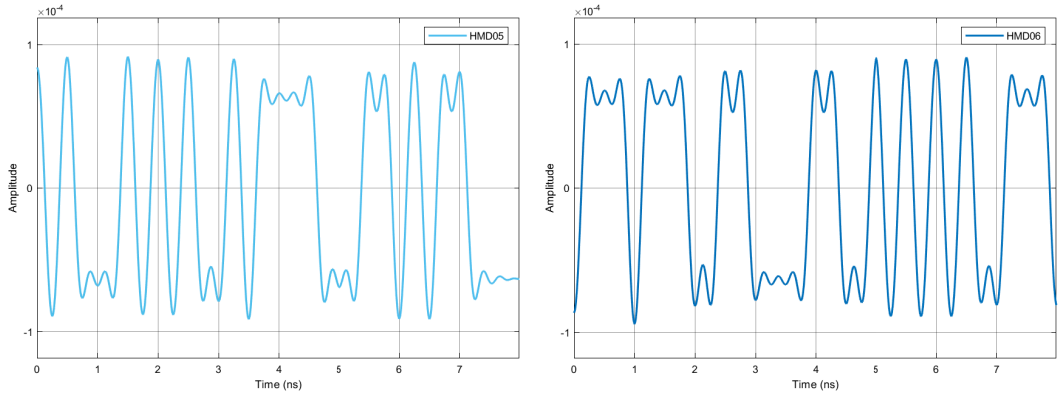
Table 6.7: Simulation parameters

Parameter	**Value**	**Units**
numberOfBitsGenerated	100×10^3	
samplingRate	64×10^9	Hz
symbolRate	4×10^9	Bd
samplesPerSymbol	16	
symbolPeriod	250×10^{-12}	s
bitPeriod	125×10^{-12}	s
signalOutputPower_dBm	-10	dBm
localOscillatorPower_dBm	0	dBm
localOscillatorPhase	0	rad
nBw	18×10^9	Hz
responsivity	1	A/W
amplification	300	
amplifierInputNoisePowerSpectralDensity	1.5657×10^{-19}	
thermalNoisePower	2.56248×10^{-8}	
rxResistance	50	Ω
temperatureKelvin	290	K
outputFilter	RootRaisedCosine	
shaperFilter	RootRaisedCosine	
rollOffFactor_out	0.9	
rollOffFactor_shp	0.9	
seedType	RandomDevice	
numberOfBitsReceived	-1	
elFilterType	Defined	
fiberLength_m	50000	m
fiberAttenuation_m	4.6052e-05	m^{-1}
elFilterOrder	20	
opticalGain_dB	0	dB
noiseFigure	0	dB
bufferLength	512	
bitSourceMode	Random	

confidence	0.95	

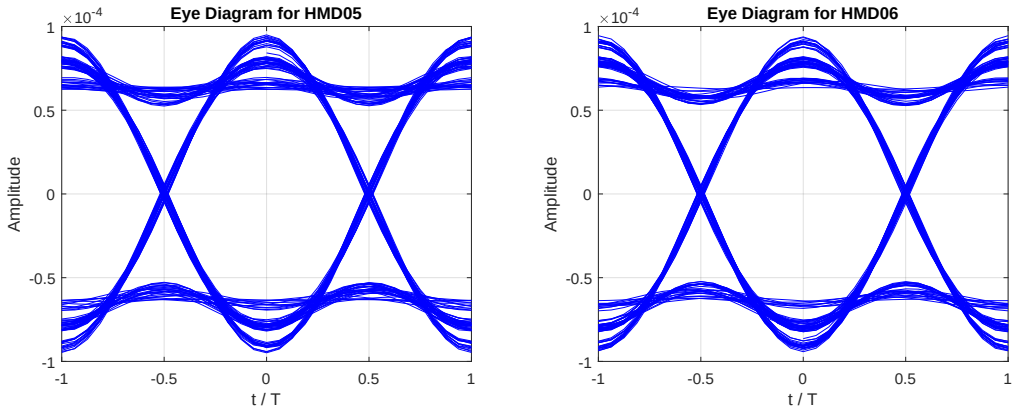
The addition of an amplifier helps the overcome the thermal noise. Nevertheless, it introduces another noise source which also limits the receiver performance. This noise source, however, is proportional to the amplifier gain, so increasing the gain to very high values does not improve the sensitivity of the receiver.

We shall now examine the differences to the previous case. The signals up to the photodiodes in the receiver are similar to the previous cases. Therefore, we will start by showing the signals at the photodiodes. They are similar to the previous cases, but are a point to start.



(a) (b)

Figure 6.37: Signals HMD05 (a) and HMD06 (b), after detection in the photodiodes.

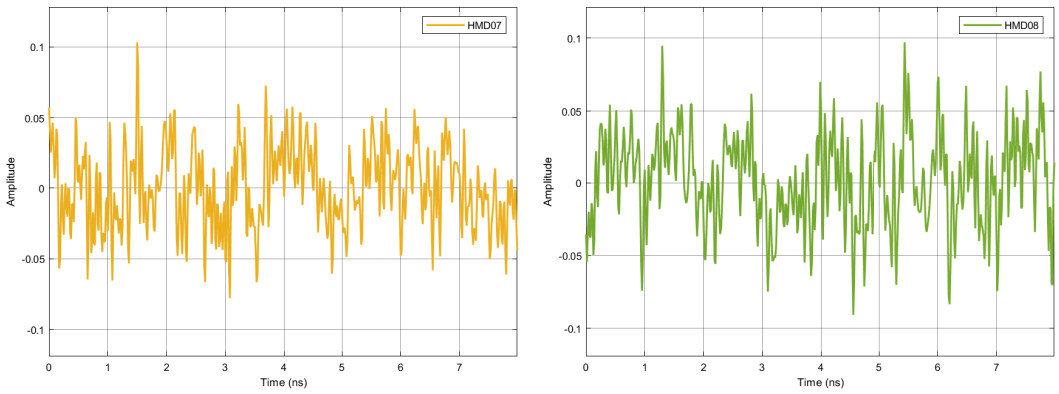


(a) (b)

Figure 6.38: Eye diagrams of signals HMD05 (a) and HMD06 (b), after detection in the photodiodes. Shaped with a root-raised cosine filter.

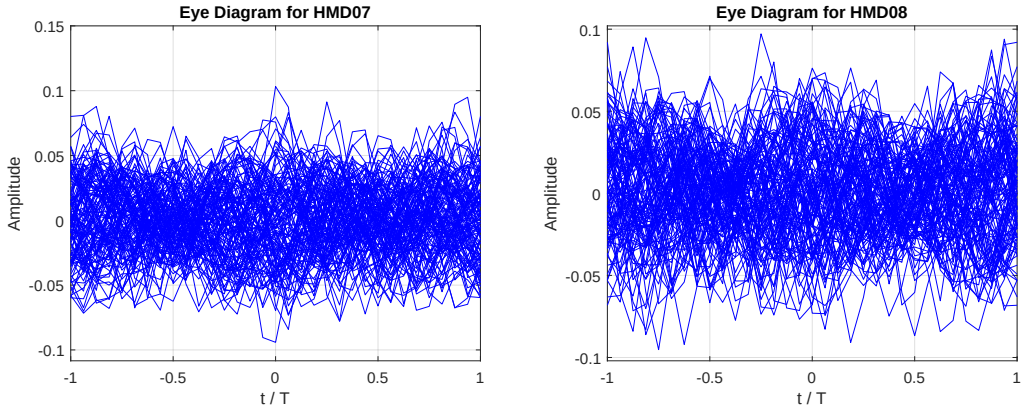
The main difference between previous cases and this one is the existence of an amplifier after the photodiodes. This amplifier has three properties which affect the signal: a gain,

an input referred noise source, and a bandwidth. The bandwidth is much larger than the signal bandwidth, and therefore does not have any appreciable effects other than limiting the noise bandwidth. The main effects of the amplifier, as shown in Figures 6.39 and 6.40, are the amplification of the signal and the introduction of noise. The noise standard deviation increases proportionally to the amplifier gain.



(a) (b)

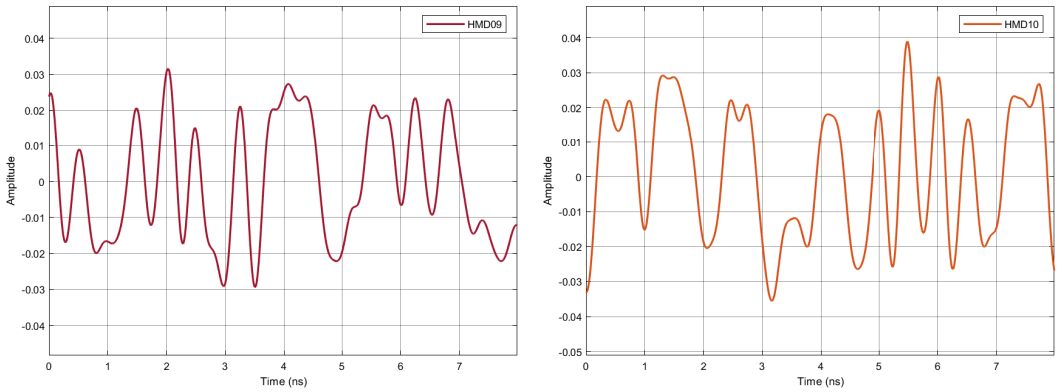
Figure 6.39: Signals HMD07 (a) and HMD08 (b), after detection in the photodiodes.



(a) (b)

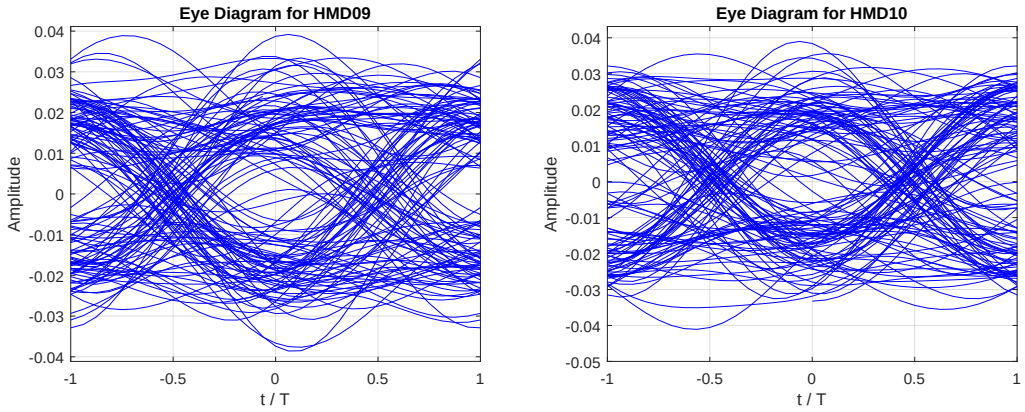
Figure 6.40: Eye diagrams of signals HMD07 (a) and HMD08 (b), after detection in the photodiodes.

The new noise source is placed prior to the matched filter. As such, the noise is strongly attenuated when the signal goes through the matched filter. This can be seen particularly well by comparing the eye diagrams of Figures 6.40 and 6.42. In the latter, the eye is widely open when compared to the former, and the variations in the signal due to noise are much smoother.



(a) (b)

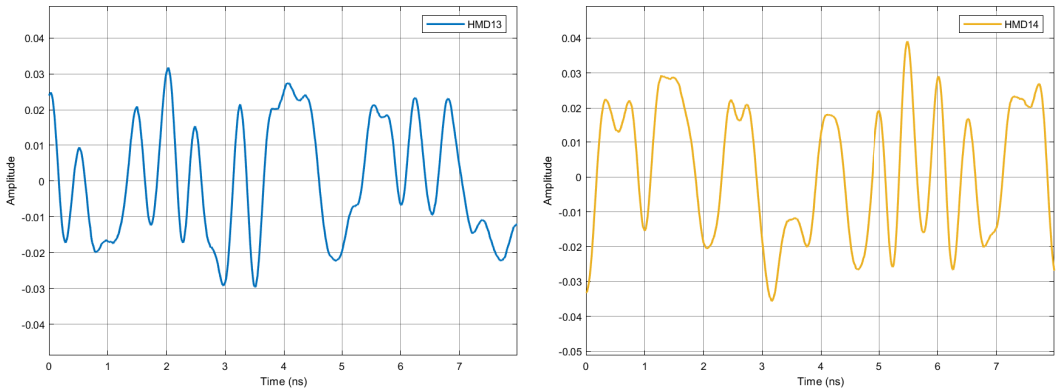
Figure 6.41: Signals HMD09 (a) and HMD10 (b), after detection in the photodiodes.



(a) (b)

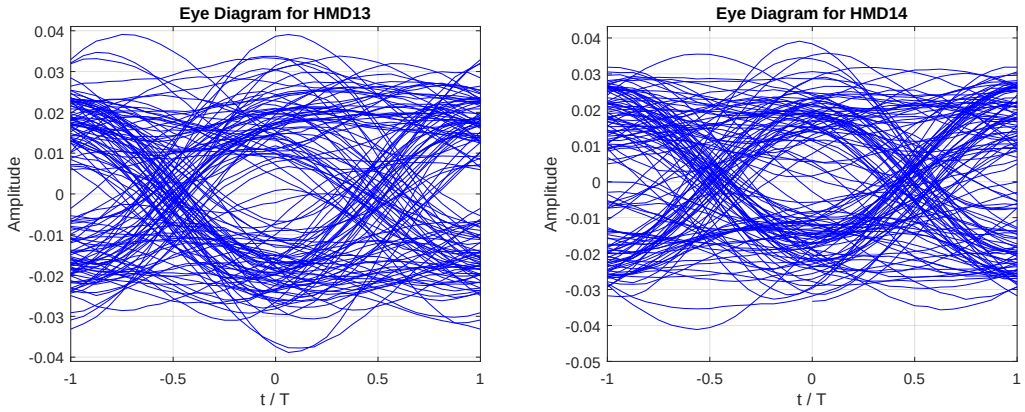
Figure 6.42: Eye diagrams of signals HMD09 (a) and HMD10 (b), after detection in the photodiodes.

Lastly, thermal noise is added prior to sampling. However, as can be seen in Figures 6.43 and 6.44, thermal noise is negligible in this case, as the signal has been amplified to several order of magnitude above the thermal noise level.



(a) (b)

Figure 6.43: Signals HMD13 (a) and HMD14 (b), after detection in the photodiodes.



(a) (b)

Figure 6.44: Eye diagrams of signals HMD13 (a) and HMD14 (b), after detection in the photodiodes.

We then get the constellation shown in Figure 6.45. Comparing with Figure 6.35 from the previous section, they appear to have a similar signal to noise ratio. While this might be true, it's worth noting that the constellation shown here has a much higher amplitude, equal to the constellation at the transmitter. In addition, it was obtained with a much weaker signal, -20 dBm, compared with 0 dBm in the previous example.

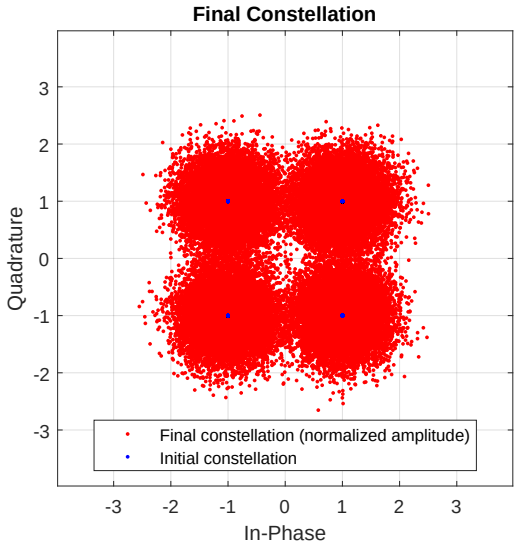


Figure 6.45: Final constellation. Signal optical output power of -20 dBm and 20 km of fiber length (-24 dBm at receiver input).

We can compare the results obtained here with the results from the previous section, where there was no amplification and only thermal noise was present. It is clear that the use of the amplifier leads to much better results.

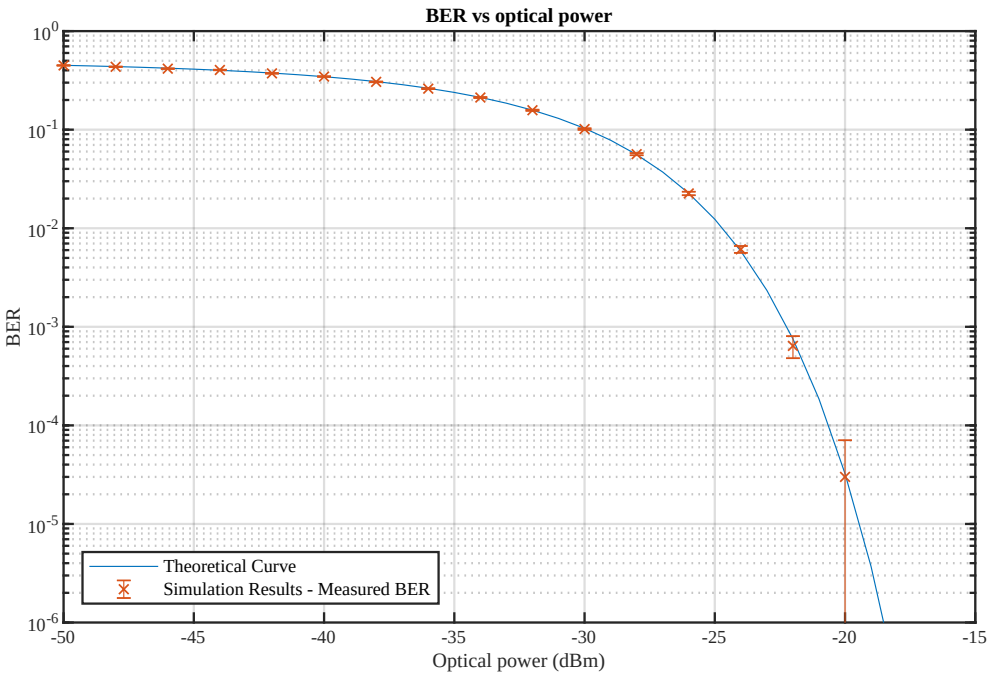


Figure 6.46: BER curve. LO power at 0 dBm, amplifier gain set at 300.

The sensitivity at BER = 10^{-3} in these conditions is -22.1 dBm, which is quite an improvement over the unamplified scenario.

It is worth noting that the results shown here do not use the same amplification for all data points. Instead, the amplification in each point is chosen so that the signal amplitude at sampling time is equal to 1, in order to replicate the initial constellation. Taking this into account, first we need to obtain the equation for the amplifier gain necessary to make the average of the constellation points have the desired value pmA_g (in this case, $A_g = 1$, as it is the case on the initial constellation).

$$G_e = \frac{A_g}{\eta K \sqrt{P_s e^{-L\alpha} P_{LO}}} \tag{6.44}$$

The curve can now be obtained with Equation 6.43, with

$$\begin{aligned} A &= \eta G_e K \sqrt{P_s P_{LO} e^{-L\alpha}} \\ N &= 4k_B T B R + \left(\sqrt{n_{in} \frac{1}{T_s}} G_e K \right)^2 \end{aligned} \tag{6.45}$$

All variables are as previously defined. In addition, T_s is the symbol period of the transmitted signal and n_{in} is the input referred noise spectral density of the transimpedance amplifier.

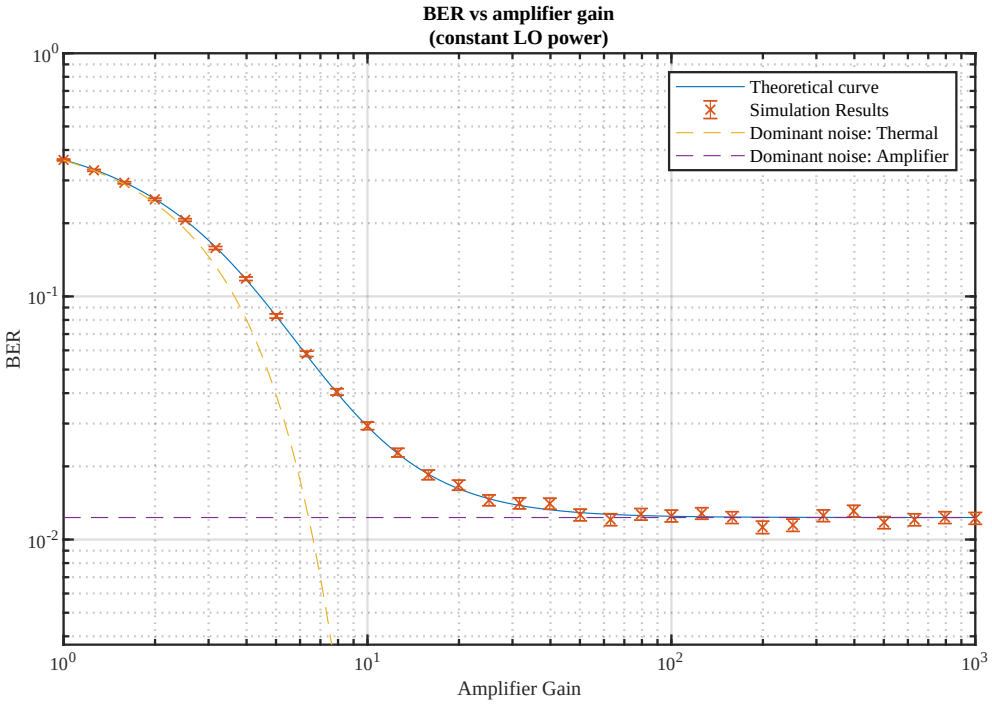


Figure 6.47: BER as a function of the transimpedance amplifier gain, along with the curves where the individual noise sources dominate. Signal output power of -25 dBm and local oscillator power of 0 dBm.

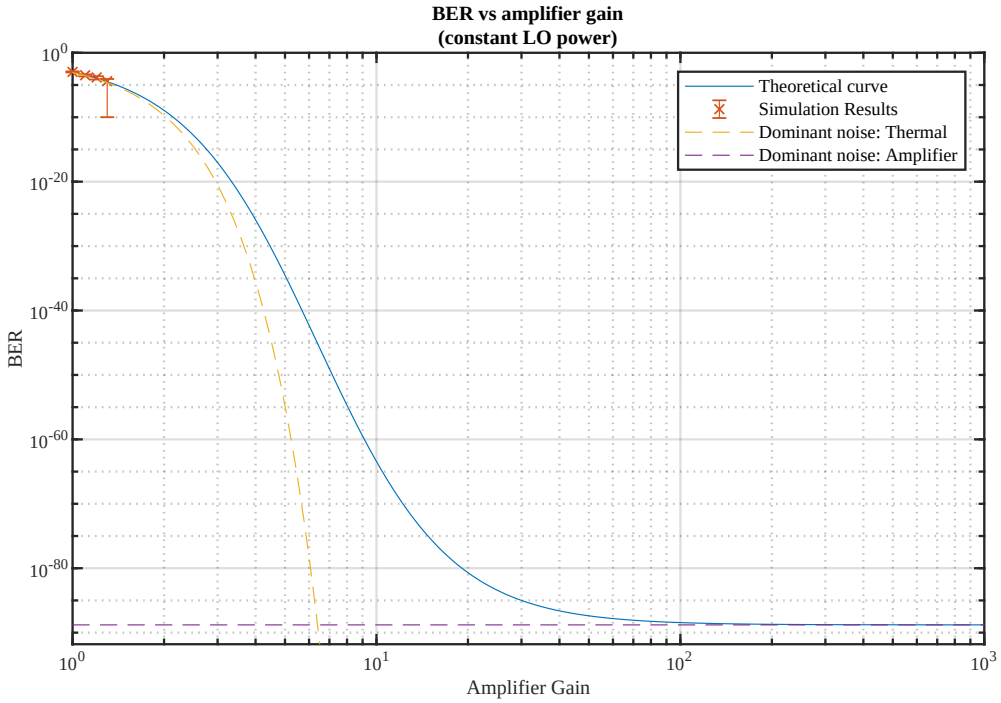


Figure 6.48: BER as a function of the transimpedance amplifier gain, along with the curves where the individual noise sources dominate. Signal output power of -6 dBm and local oscillator power of 0 dBm.

We can see the direct effect of the amplifier on the BER on Figure 6.47. The yellow curve is the expected behavior if thermal noise was the only noise source. It better demonstrates the system behavior at low gains, reflecting the case where the thermal noise overwhelms the amplifier-generated noise. As the gain increases, the thermal noise component becomes less significant when compared to the signal and to the amplifier noise. When the gain is high enough (10^2 in the plot), thermal noise is negligible when compared to the amplifier output, and system follows the purple line. This line shows the expected behavior if the only noise present was due to the amplifier. At this point, the performance limiting factor is the relation between the current generated at the photodiodes and the input referred noise of the amplifiers. This ratio is constant and independent from the gain.

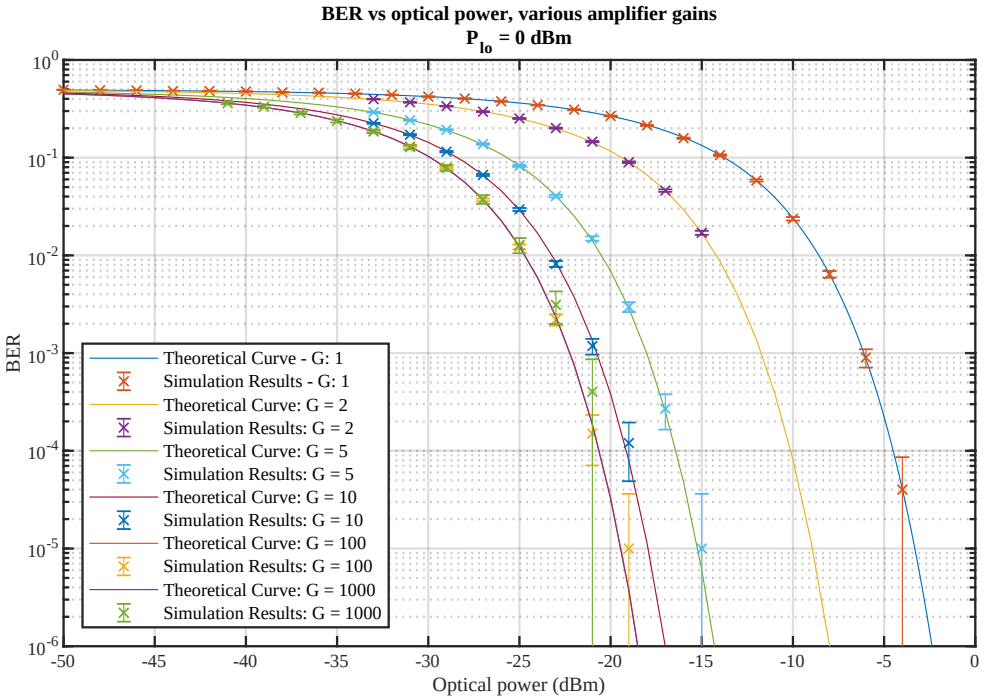
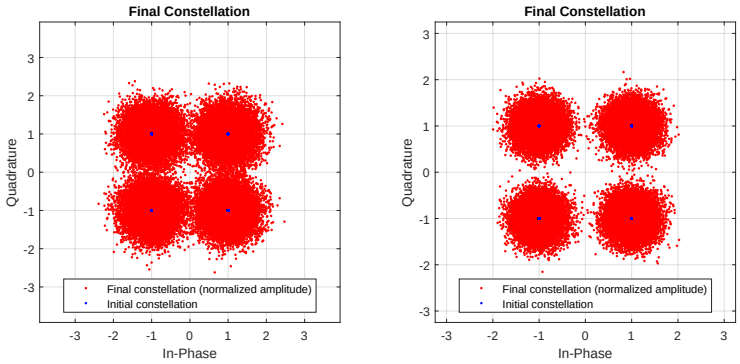
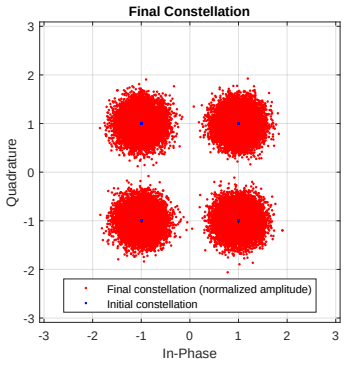


Figure 6.49: Comparison of BER curves with different gains. Local oscillator power of 0 dBm. We can see that when G=1, the curve is similar to the case without amplifier shown in Figure 6.36

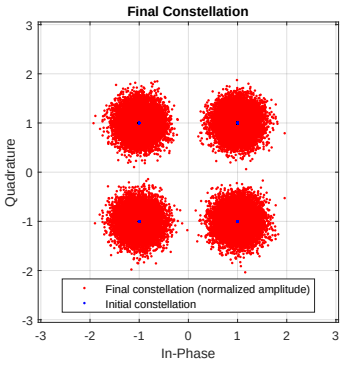


(a) $G = 5$

(b) $G = 10$



(c) $G = 100$



(d) $G = 1000$

Figure 6.50: Final constellations for various local oscillator optical powers. Signal optical power at is -15 dBm, with 20 km of fiber length (-19 dBm at receiver input). It is easy to see that there does not appear to be a significant improvement between the constellations with the gains of 100 and 1000.

6.1.3.8 Simulation results - Increased LO

As mentioned in the previous section, one way to further increase performance is to improve the relation between the current generated at the photodiodes and the input referred noise of the amplifiers. This can be done by using a higher optical power on the Local Oscillator. Thus, the current generated at the current generated at the photodiodes will be increased.

Table 6.8: Simulation parameters

Parameter	**Value**	**Units**
numberOfBitsGenerated	100×10^3	
samplingRate	64×10^9	Hz
symbolRate	4×10^9	Bd
samplesPerSymbol	16	
symbolPeriod	250×10^{-12}	s
bitPeriod	125×10^{-12}	s
signalOutputPower_dBm	-20	dBm
localOscillatorPower_dBm	5	dBm
localOscillatorPhase	0	rad
nBw	18×10^9	Hz
responsivity	1	A/W
amplification	300	
amplifierInputNoisePowerSpectralDensity	1.5657×10^{-19}	
thermalNoisePower	2.56248×10^{-8}	
rxResistance	50	Ω
temperatureKelvin	290	K
outputFilter	RootRaisedCosine	
shaperFilter	RootRaisedCosine	
rollOffFactor_out	0.9	
rollOffFactor_shp	0.9	
seedType	RandomDevice	
numberOfBitsReceived	-1	
elFilterType	Defined	
fiberLength_m	20000	m
fiberAttenuation_m	4.6052e-05	m^{-1}
elFilterOrder	20	
opticalGain_dB	0	dB
noiseFigure	0	dB
bufferLength	512	
bitSourceMode	Random	
confidence	0.95	

The increased current at the photodiodes' output means that the same amplitude can be

reached with a lower amplifier gain. A lower gain, by itself, implies that the input referred noise of the amplifier will stay at a lower value, thereby increasing the sensitivity of the system. In this example, the signals will not be shown here, as the only difference is that the amplitude of the signal after the photodiodes is higher.

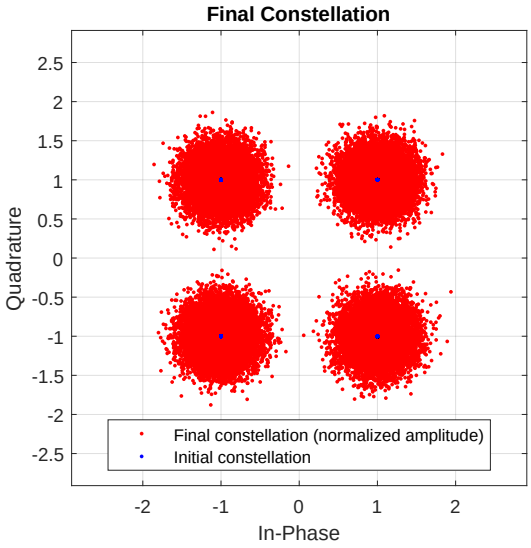


Figure 6.51: Final constellation (normalized). Signal optical output power of -20 dBm and 20 km of fiber length (-24 dBm at receiver input). Local oscillator at 5 dBm.

In Figure 6.51 we can see the final constellation obtained when choosing an output optical power of -20 dBm and setting the fiber length to 20 km. In conditions similar to Figure 6.45 it shows a much better constellation. This is all due to the higher local oscillator value.

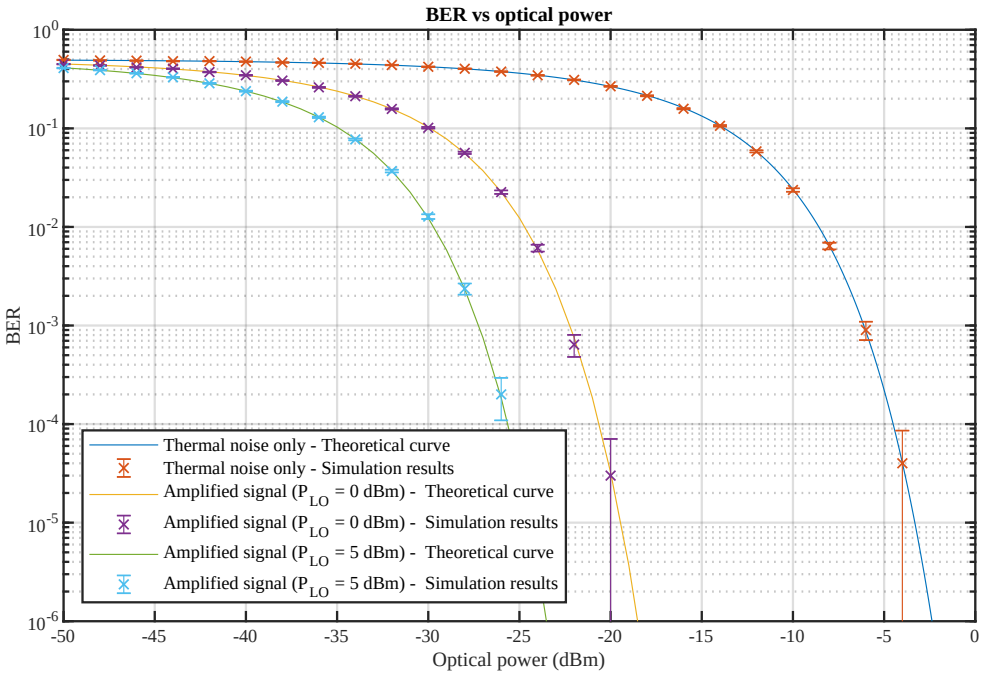


Figure 6.52: Comparison of BER curves: considering thermal noise only, using an electrical amplifier, and using a higher value of local oscillator together with the amplifier. Amplifier gain was set at 300.

The sensitivity of the new curve, at BER 10^{-3},with local oscillator power of 5 dBm, is -27.1 dBm, 5 dB lower than the curve with the LO at 0 dBm.

The theoretical curve in Figure 6.52 was obtained according to the same equation as the second curve in the previous section. Notice that the new curve is approximately 5 dB to the left of curve with the amplifier and the LO at 0 dBm. This is because in the equations 6.43 and 6.45, the power of the local oscillator is as important as the optical signal power, and the amplitude at sampling time grows at the same rate with both quantities.

This might lead us to believe that by increasing the LO, the performance can be improved as much as required. However, that would only be the case if the Local Oscillator was independent from any noise source. That is not the case, as we shall see in the next section.

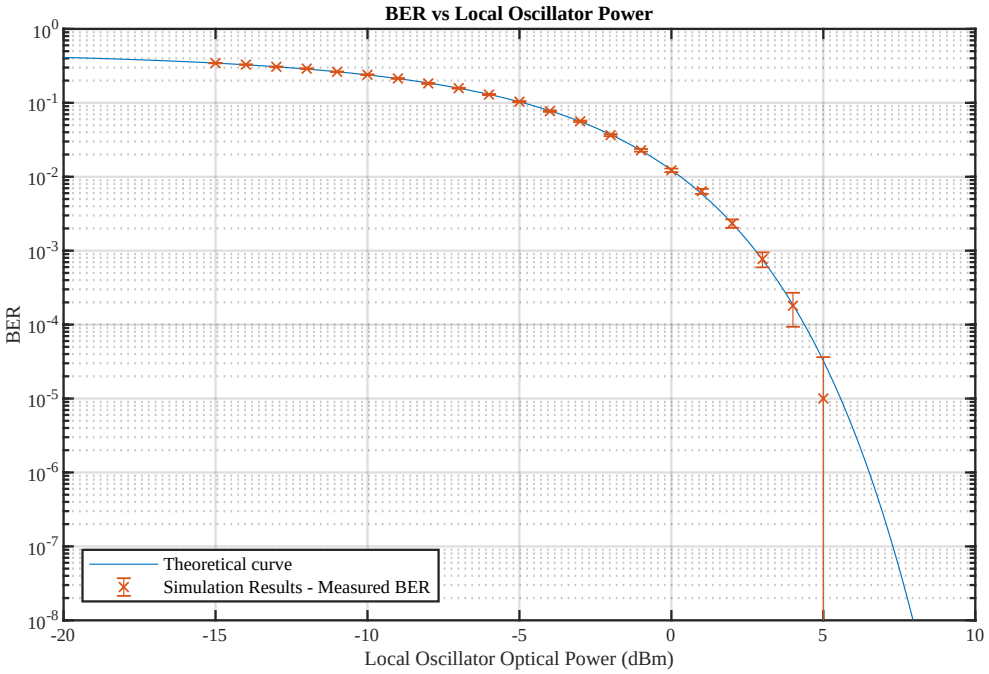
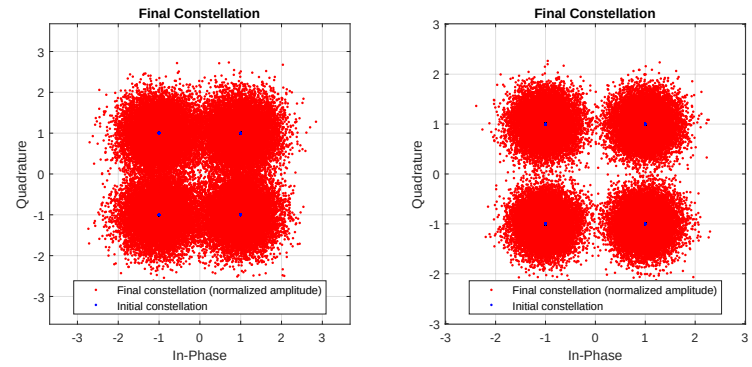
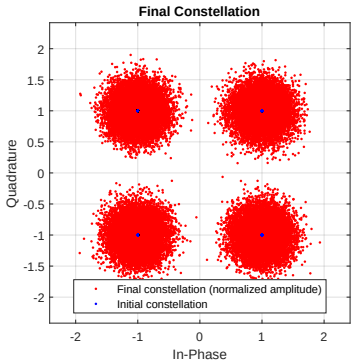


Figure 6.53: Variation of BER with the LO power. Signal optical power at receiver input is -25 dBm.

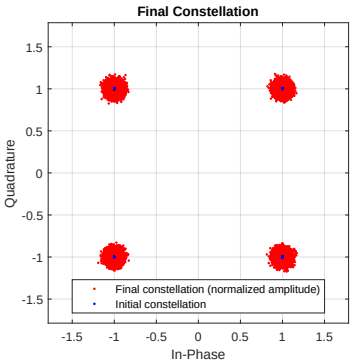


(a) $P_{lo} = 0$ dBm

(b) $P_{lo} = 3$ dBm



(c) $P_{lo} = 6$ dBm



(d) $P_{lo} = 20$ dBm

Figure 6.54: Final constellations for various local oscillator optical powers. Signal optical power at receiver input is -25 dBm.

6.1.3.9 Simulation results - Quantum Noise

We will now consider the more realistic case when there is quantum noise associated with the local oscillator. We can consider quantum noise to be similar to shot noise.

In this simulation, we will add noise at the photodiodes, where the noise current variance will be proportional to total generated photocurrent. Normally, shot noise follows a Poisson distribution. However, for the purpose of this section, we shall consider that the number of photons is high enough so that it can be considered to follow a normal distribution.

$$\sigma^2_{\text{shot}} = 2qIB = 2\eta P_{\text{opt}} B \tag{6.46}$$

As the local oscillator will be much higher than the signal, we can consider that the total optical power generating current is equal to the local oscillator power. Due to the optical hybrid, each photodiode receives $1/4$ of the local oscillator optical power.

Keeping in mind that the noise and signal will be affected by the same gains, the factor which control performance are the signal and noise ratio at the photodiodes output, and the matched filter. The other blocks of the receiver will have no practical effects, as the LO amplification will render the other noise sources negligible, and the matched filter ultimately limits the noise bandwidth.

We then have that the signal current amplitude at the photodiode's output is given by:

$$A = \eta\sqrt{P_{LO}P_s} \tag{6.47}$$

On the other hand, the shot noise variance $N_{\text{shot i}}$ in each photodiode will be given by

$$N_{\text{shot i}} = 2\eta h f \frac{P_{LO}}{4} B \tag{6.48}$$

with $2B = 1/\tau$, where τ is the sampling period.

Considering the shot noise to be approximated as a gaussian random variable, the noise at output of each pair of photodiodes can be obtained by the sum of their variances:

$$\begin{aligned} N_{\text{shot}} &= N_{\text{shot 1}} + N_{\text{shot 2}} \\ &\approxeq 2N_{\text{shot i}} = \\ &= 4\eta h f \frac{P_{LO}}{4} B \end{aligned} \tag{6.49}$$

We can now calculate A and N after the matched filter output:

$$\begin{aligned} A &= \eta G_e K \sqrt{P_s P_{LO} e^{-L\alpha}} \\ N &= 4k_B TBR + \left(\sqrt{n_{in}\frac{1}{T_s}} G_e K\right)^2 + \left(\sqrt{\frac{N_{\text{shot}}}{B}\frac{1}{T_s}} G_e K\right)^2 \end{aligned} \tag{6.50}$$

In addition, we can compare these results with the quantum limit. The quantum limit establishes the absolute minimum sensitivity of the receiver, and varies according to the

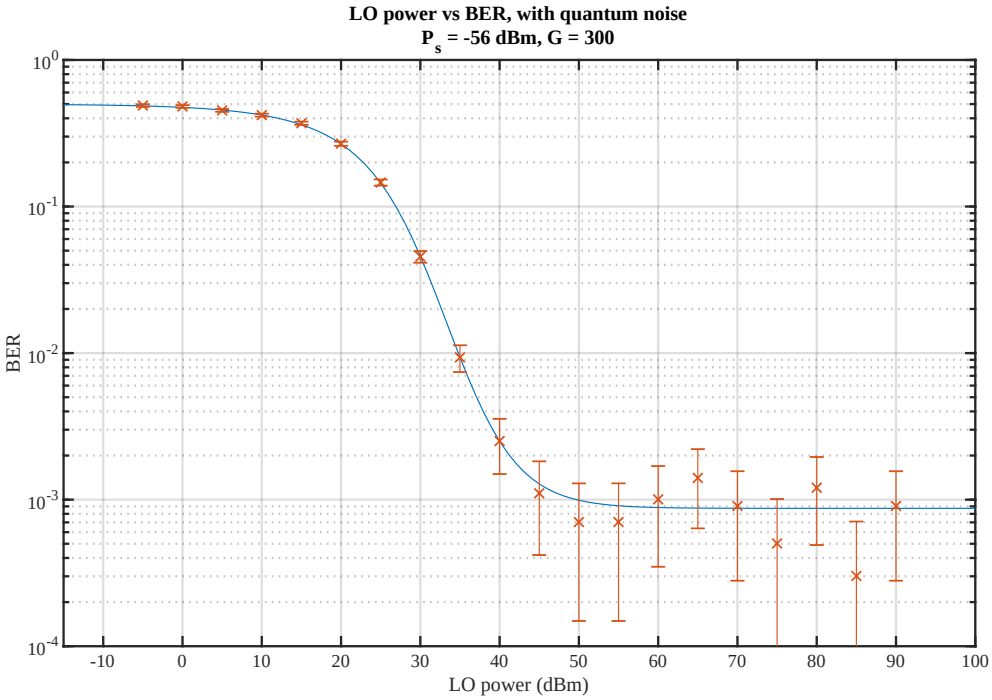


Figure 6.55: Variation of BER with the LO power, with shot noise. Signal optical power at receiver input is -56 dBm.

modulation scheme and type of detector. In the case of QPSK with homodyne detection this limit is given by [**senior09**, **agrawal12**]:

$$\text{BER} = \frac{1}{2}\text{erfc}\left(\sqrt{2\eta_q N_p}\right) \tag{6.51}$$

where eta_q is the quantum efficiency of the receiver, and N_p is the number of photons per bit. We can convert the average number of photons per bit to dBm or vice versa:

$$P_{\text{dBm}} = 10\log_{10}\left(\frac{hf}{T_b}N_p \times 10^3\right) \tag{6.52}$$

$$N_p = \frac{10^{(P_{\text{dBm}}/10)}T_b \times 10^{-3}}{hf} \tag{6.53}$$

It is worth remembering that we are neglecting most effects which arise due to working with a small number of photons, and just assuming that the system behaves similarly as for larger numbers of photons.

We can see that in this case the sensitivity at a BER 10^{-3} is -56 dBm, which is at the quantum limit. This corresponds to an average of 2.4 photons per bit.

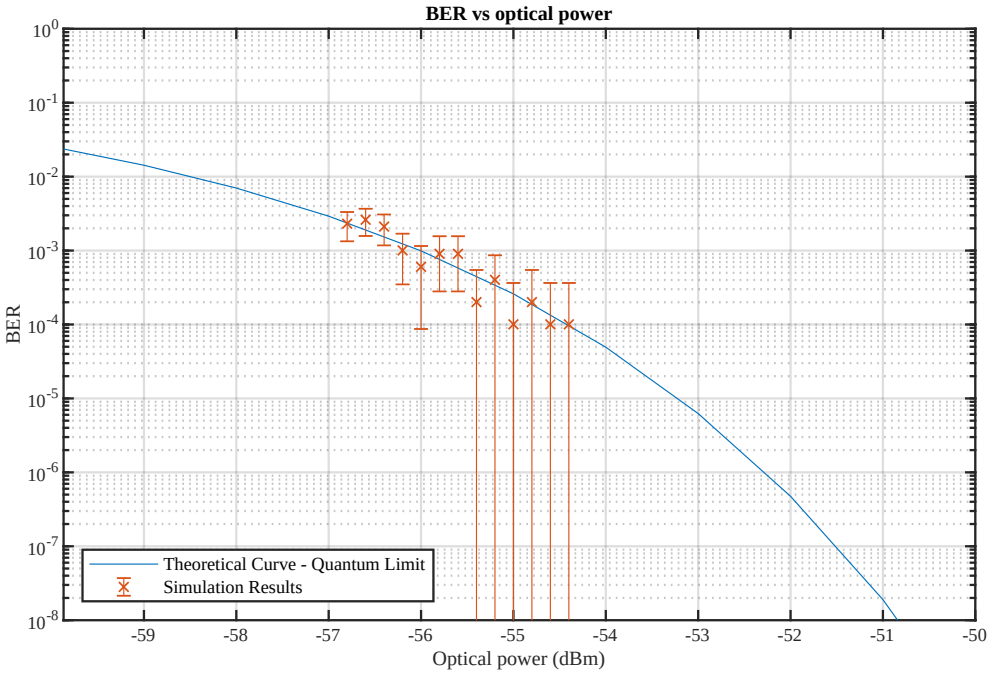


Figure 6.56: BER variation with the optical power. $P_{lo} = 50$ dBm .

6.1.4 Experimental Setup

The setup shown in Figure 6.57 was used to obtain experimental results to compare with the theory and validate the simulation. The list of devices used for the setup is available in Table 6.9. Tables 6.10 to 6.14 show the devices' specifications.

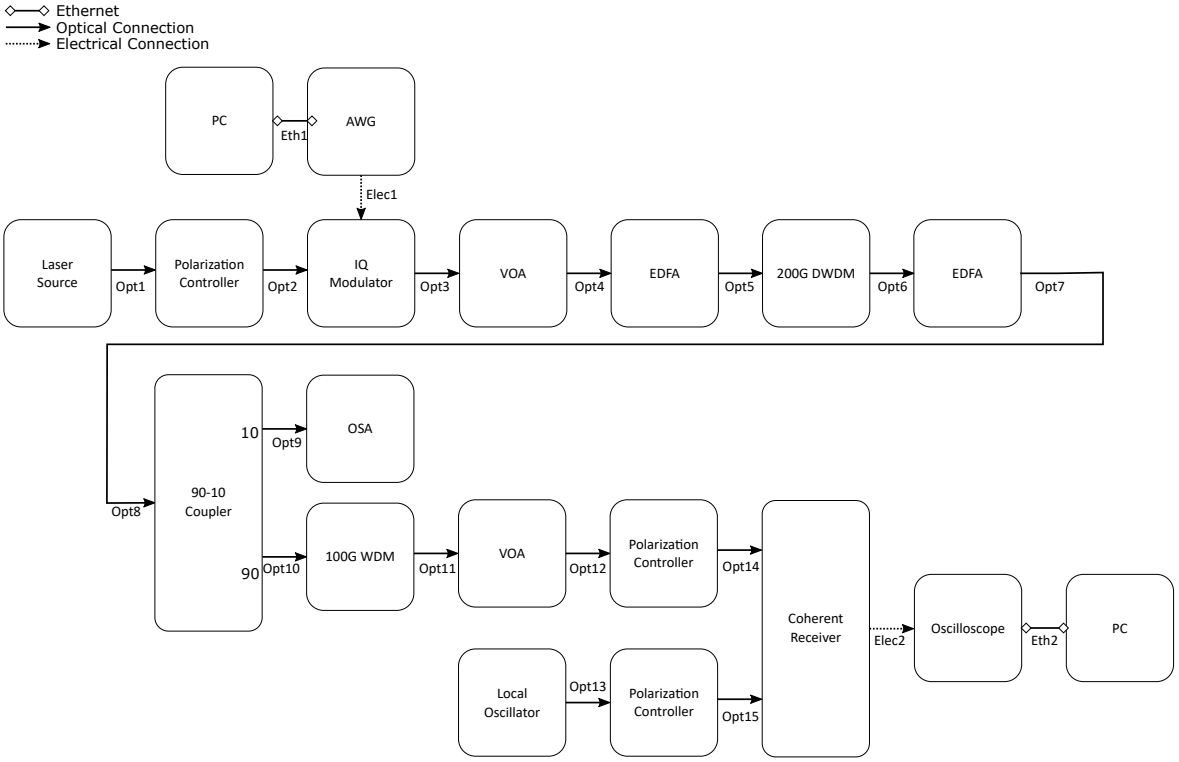


Figure 6.57: Experimental setup

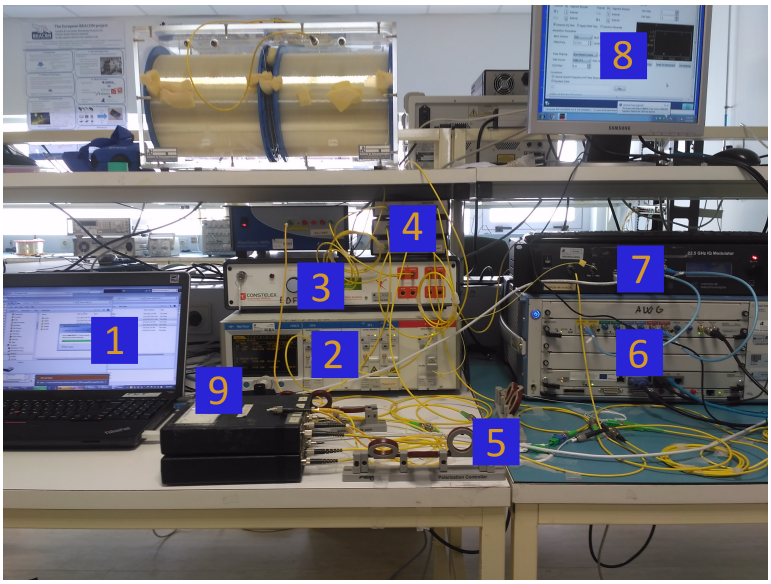


Figure 6.58: Photo of the experimental setup. 1-Computer; 2-TX Laser source; 3-EDFA; 4-Couplers and filters; 5-Polarization controllers; 6-AWG; 7-IQ Modulator; 8-AWG monitor; 9-USB controlled variable optical attenuator.

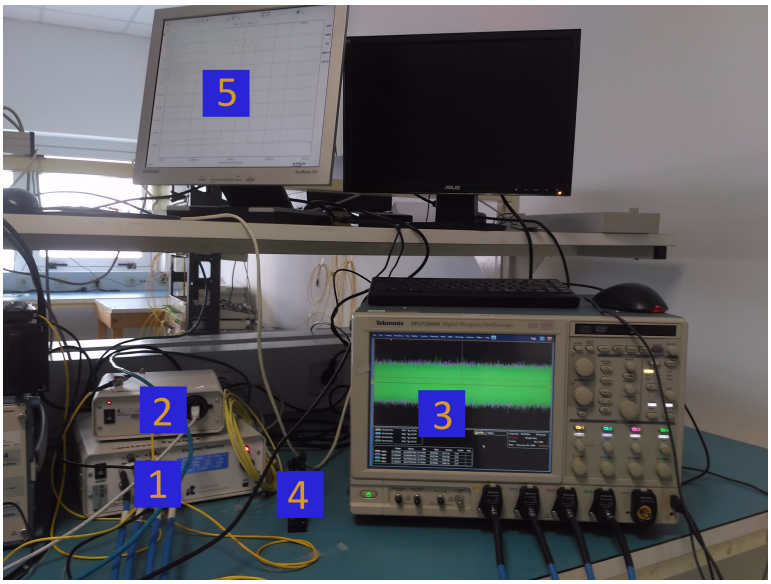


Figure 6.59: Photo of the experimental setup. 1-Coherent Receiver; 2-Local Oscillator laser source; 3-Oscilloscope; 4-Polarization controller; 5-OSA monitor.

Device	Model	Description
Laser Source	Yenista OSICS Band C/AG TLS Laser	Optical laser source for modulate the signal
IQ Modulator	22.5GHz IQ Modulator with automatic Bias Controller	
AWG	Keysight M8195A	
VOA		USB-Controlled Variable Optical Attenuator
EDFA	Constelex Hydra-C-17-17 EDFA	
200G DWDM		
EDFA	Constelex Hydra-C-17-17 EDFA	
90/10 Coupler		
OSA	Apex Technologies AP-2043B	
100G WDM		
VOA		
Local Oscillator	Emcore CRTND3U02D ECL Laser	
Coherent Receiver	Picometrix CR-100D	
Oscilloscope	Tektronix DP720004B	

Table 6.9: Devices in experimental setup.

Tables 6.10 to 6.14 list the relevant specifications for the devices used in the experimental setup.

6.1.4.1 Device Specifications

Tektronix DP720004B Oscilloscope, TekConnect channels	
Analog channels bandwidth	16 GHz
Sample rate per channel	50 GS/s
Rise time (typical) 10% to 90% 20% to 80%	 18 ps 14 ps
Vertical noise, bandwidth filter on, max sample rate 50 mV/div(typical)	0.56% of full scale @ 0 V offset (500 mV_{FS})
Timing resolution	10 ps
Sensitivity range	200 mV_{FS} to 5 V_{FS}
Passband Flatness	± 0.5 dB to 50% of nominal bandwidth at 25^o
Vertical resolution	8 bits (11 bits with averaging)
Effective number of bits	5.5 bits @ 50 mV/div, 100 GS/s

Table 6.10

Yenista OSICS TLS-AG Wide C-band	
Frequency Range (Wavelength Range)	196.275 - 191.125 THz (1527.41 - 1568.57 nm)
Output Power	20 mW (+13 dBm)
Power Range (typ.)	+6 to +13.6 dBm
Relative Frequency (Wavelength) Accuracy (typ.)	± 0.5 GHz ($\pm$ 4 pm)
Absolute Frequency (Wavelength) Accuracy (typ.)	± 1.5 GHz ($\pm$ 12 pm)
Frequency Setting Resolution	Down to 1 MHz
Instantaneous Linewidth (FWHM)	< 100 kHz
Power Stability	± 0.03 dB
Absolute Output Power Deviation Accross Tuning Range	± 0.2 dB
Side Mode Suppression Ratio	60 dB
Relative Intensity Noise	-145 dB/Hz

Table 6.11

Local Oscillator Laser	
Optical Output Power Adjustment Range	+7 - +13.5 dBm
Optical Output Power Adjustment Range - high power variant	7 - 15.5 dBm
Short term power variation	0.05 dB
Optical Output Power Step Size	0.01 dB
Operating Frequency (Wavelength) Range	191.5 - 196.25 THz (1527.6 - 1565.5 nm)
Fine Tune Frequency Resolution (typ.)	1 MHz
Fine Tune Frequency Range	± 6 GHz
Frequency Accuracy EOL	± 2.5 GHz
Frequency Accuracy EOL (25 GHz spacing variant)	± 1.5 GHz
Instantaneous Linewidth (FWHM)	100 kHz
Relative Intensity Noise (+13dBm output)	-145 dB/Hz
Relative Intensity Noise (+7dBm output)	-140 dB/Hz
Side Mode Suppression Ratio (typ.)	55 dB
Back reflection	-14 dB
Optical Isolation	30 dB
SSER (typ.)	55 dB
Polarization Extinction Ratio	20 dB

Table 6.12

Balanced Receiver	
Wavelength Range	1525 - 1570 nm
Bit Rate (max)	32 Gb/s
Signal Input Power	-6 dBm
Polarization Extinction Ratio	18 dB
LO input Power	16 dBm
I/Q relative phase in Mixer	
minimum	85 deg
typical	90 deg
maximum	95 deg
I/Q phase stability in mixer (over lifetime)	-2 - +2 deg
PBS mixer excess loss	3.1 dB
Optical input return loss	-27 dB
Bandwidth (-3 dB elec.)	18.5 - 28 GHz
Low frequency cutoff	100 kHz
Group delay variation	
0.1 - 15 GHz	-3 - +3 ps
15 - 25 GHz	-9 - +9 ps
CMMR (signal input)	
DC	-17 dB
22GHz	-16 dB
CMMR (LO input)	
DC	-12 dB
22GHz	-10 dB
Linearity	5%
Temporal Skew	
P/N outputs	3 ps
Across all four channels	10 ps
Photodiode dark current (25^oC)	100 nA
Photodiode responsivity @1550 nm	0.64 A/W
Effective responsivity (both inputs)	0.05 A/W

Table 6.13

Constelex Hydra-C EDFA	
Input wavelength range	1530-1565 nm
Saturated output power	13 -21 dBm
Input power	-20 - +3 dBm
Small signal gain (Pin = -20dBm)	>28 dB
Noise Figure (Pin = -10dBm @1555 nm)	<4 dB

Table 6.14

Keysight M8195A AWG	
Sample Rate	65 GSa/s
Analog Bandiwidth (typ.)	25 GHz
Vertical Resolution	8 bits
Amplitude (single ended)	75 mV$_{PP}$ to 1 V$_{PP}$
Amplitude Resolution	200 μV
Intrinsic Random Jitter	< 200 fs
Rise time (typical) 20% to 80%	 18 ps

Table 6.15

22.5 GHz IQ Modulator with automatic Bias Controller	
Wavelength Range	1520 - 1580 nm
Electro Optical Bandwidth (min.)	22 GHz
Electro Optical Bandwidth (typ.)	25 GHz
Optical Return Loss	30 dB
Electrical Input High Frequency 3 dB point (typ.)	40 GHz
Electrical Input High Frequency 3 dB point (max.)	65 kHz
Electrical input Gain Ripple (typ.)	$\pm$1 dB
Gain delay ripple (max.)	$\pm$50 ps

Table 6.16

6.1.4.2 Considerations on signal quality and noise

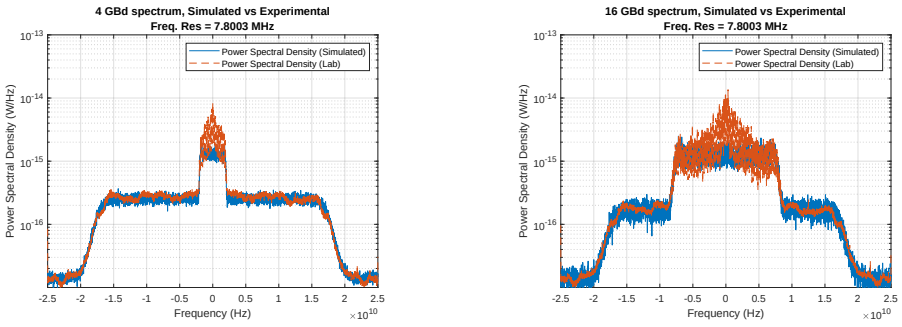
The noise can come from several sources. Assuming that any nonlinearities can be corrected in post processing, the noise present in the signal has several sources: beatings from the different signal components (signal, local oscillator and ASE), shot noise and thermal noise. Some of the optical components can be ignored in the case of ideal balanced detection, as they cancel out. However, if the beamsplitters are not exactly balanced, these components still have some effect.

The E_b/n_0 will be used to evaluate the BER curves. It the ratio will be calculated in two different points

- on the optical domain through analysis of trace obtained at the optical spectrum analyzer. Using an EDFA it is possible to obtain approximately uniform optical noise. Using this does not take into account the electrical filter, electrical noise or other effects, instead quantifying only the realtion between the optical noise generated at the EDFA and the signal optical power. Therefore, it is not expected that the data points are coincident with the theoretical curve,
- In a case without the use of the EDFA, the E_b/n_0 of the electrical signal can be calculated by measuring the noise at the receiver. The power spectral density of frequencies lower than the symbol rate is used to get an estimate on n_0, obtained through spectral analysis.The signal power is considered to be the total power measured at the receiver, minus the power when there is no input. This ignores the existence of any optical noise.

We will now see some differences between the simulation and the lab data. Specifically, we will study the electrical noise and the spectrum of the optical signal.

It can be seen that there are more differences between the existing simulation and the lab data. By watching Figures 6.60 some of those differences can be seen. It was already mentioned that the noise is not white. A difference not yet mentioned is on the signals themselves. As can be seen, the signal from the lab, shaped with a root raised cosine filter with a roll-off of 0.05, is very well defined and has nearly constant spectral density in the spectrum. In the experimental case, however, the signal shape in the spectrum appears distorted.

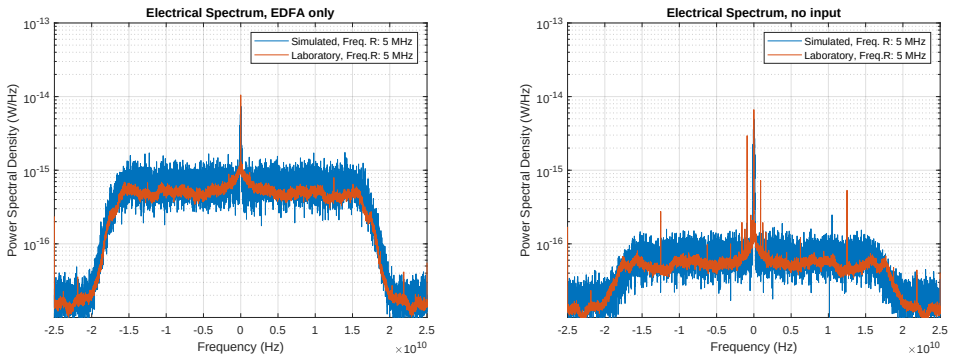


(a) (b)

Figure 6.60: Comparison of spectra from simulated and experimental signals. 4 GBd signals on the left and 16 GBd on the right. The simulated and experimental signals produce the same BER.

Figure 6.61 shows the power spectrum of noise only signals, and their simulated counterparts, using the current simulation structure (optical white noise, electrical filter and

electrical white noise after the electrical filter). The use filter is a contained least squares linear phase low pass filter, designed in MATLAB with the function `fircls1`. An IIR butterworth filter was previously tested but failed to keep the signal usable. It can be seen that some peaks are present in the lower frequencies. The same does not happen with the current simulation configuration. In addition to the peaks, the noise spectral density seems to increase as it approaches DC.

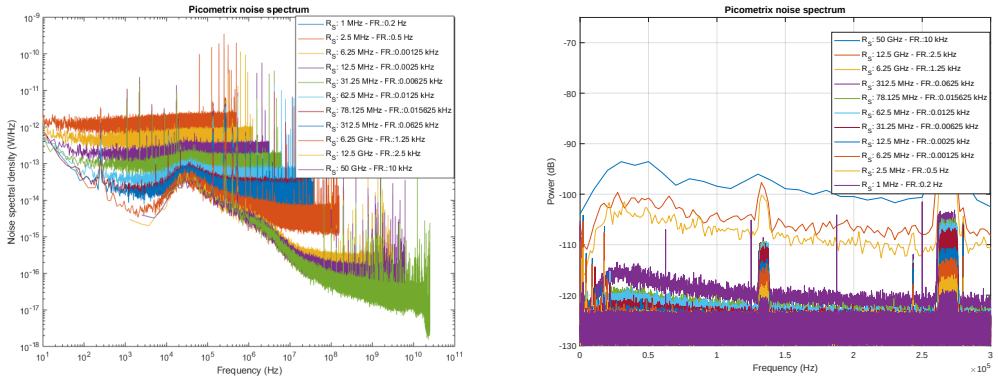


(a) (b)

Figure 6.61: Power spectrum of the signal obtained when (a) the input signal is non-existing, and the local oscillator is turned off; (b) the input signal is made only of noise generated by the EDFA.

In order to study the effects visible in the lower frequencies of the experimental spectra, several noise-only signals were acquired, with different sample rates. This allowed a better analysis of the situation, with greater emphasis on lower frequencies. There is a DC component always present, in the order of hundreds of microvolts. It can also immediately be seen that the measured noise power is independent of the sampling rate of the oscilloscope. Therefore, the noise spectral density is higher for lower sampling rates.

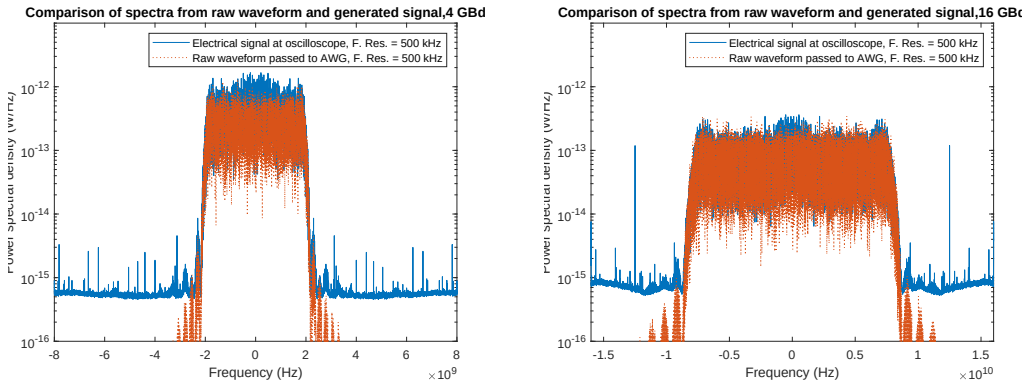
At low enough sampling rates (below 6.25 MHz) no slope is noticed in the noise power spectral density ca be noticed. Its is possible that the vertical noise of the oscilloscope at ends up hiding the effect. At higher frequencies the effect is visible, but the noise spectral density does not keep continuously increasing in the lower frequencies. It actually only increases up to the range of 10-100 KHz and the starts decreasing again. This does not appear to be a malfunction, but rather an intrinsic characteristic of the receiver.



(a) (b)

Figure 6.62: Noise spectra at various sample rates (a) Logarithmic plot, normalized to frequency resolution (b) Linear frequency scale, not normalized.

Two more aspects should be considered in the transmission process. The signal appears to be different from the expected starting early on the transmission process. The electrical waveform generated by the AWG is slightly different from the waveform it uses as a source for signal generation, particularly for frequencies lower than 1 GHz. On the other hand, most of the fault seems to be at the modulator, as the optical spectrum after modulation is the first point where the spectrum is really distorted.



(a) (b)

Figure 6.63: Spectra of electrical signals generated by the AWG compared to the original wabveforms

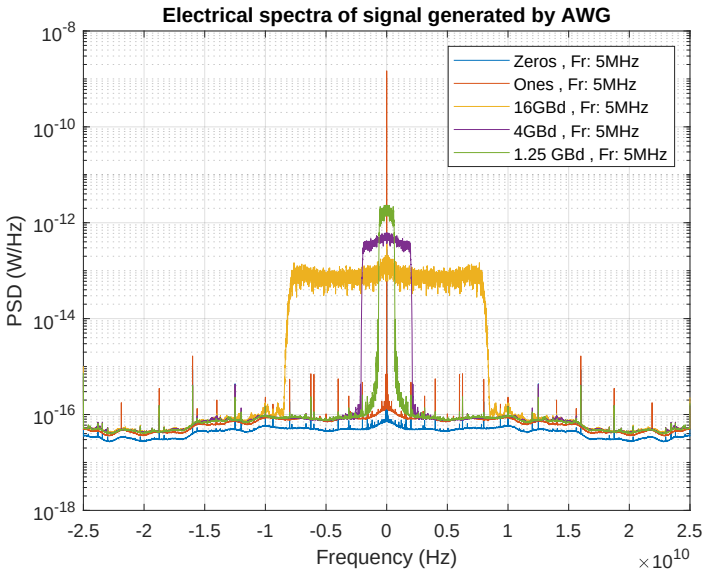
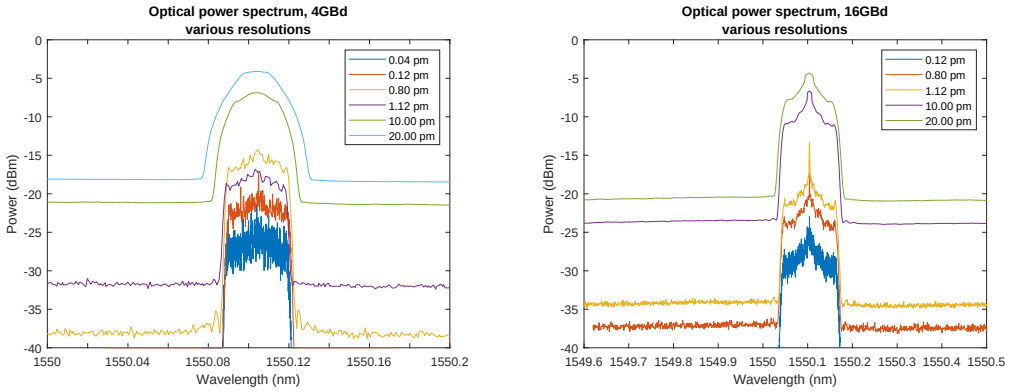


Figure 6.64: Spectra of the electrical signals generated by the AWG.



(a) (b)

Figure 6.65: Optical spectra of the modulated signals at different resolutions.

These phenomena can help partially explain the deviation from the ideal curves that can be seen in the following sections.

6.1.4.3 Considerations about parameters and results

. Similarly to what was discussed in section **??**, we need to establish the origin of each value used to analyze the perfomance of the system. Using that section as a reference, we can say from the start that the estimated values and their origin are quite similar to the ones in the simulation. However, some differences are worth noting, due to differences between the

simulation and the real world. In the following descriptions, keep in mind that the overall architecture of the setup is described in Figure 6.57, and the specifications of each particular device are in the tables following the figure. With this in mind, the possible origins for the values to use are:

- Value defined in the equipment settings, such as transmitter power, or intrinsic to the used devices, such as bandwidth;
- Value obtained through direct measurement or calculations made using the optical signal at the Optical Spectrum Analyzer. This optical signal contains a conjunction of the modulated signal and noise generated by the EDFA, if available;
- Value measured or calculated offline using the waveform acquired with the oscilloscope connected to the coherent receiver; this is a digital representation of the electrical signal generated at the receiver, and is also the one used to apply the DSP and ultimately obtain the results. The sampling rate is 50 GHz, higher than any symbol rate.

Notice that there is a certain parallelism between the experimental measurements and the signals mentioned in section **??**. In particular, they were chosen in way that they can be studied as similarly as possible. For this reason, it is possible to measure the relevant parameters in a similar way. For instance, measuring the Q-Factor instead of the BER brings benefits similar to the ones described in that section. In addition, the E_b/n_0 or SNR values can be estimated in the same way as in the simulation, where it's easier to keep track of the source of each noise or imperfection source. Therefore, the E_b/n_0 and SNR values used are measured in the same points: at the OSA or using the waveform acquired at the oscilloscope, using the same processes described in section **??**. However, unlike in the simulation, it is hard to estimate the SNR of the electrical signal using Matzner's method, as nonlinearities would need to be compensated previously. Nevertheless, methods based on spectral analysis work well in the simulation or in the lab setup, and so they are used in both. Further information can be found in Section **??**.

Overall, on the results presented here, only E_b/n_0 is used in the BER plots. This E_b/n_0 can be estimated on any of the two cases mentioned above, and the source will be identified in the caption or legend. If the EDFA is used to generate noise, the E_b/n_0 is measured in the OSA, and the value is referred to as "Optical E_b/n_0". Otherwise, if no EDFA is present, it cannot be measured in the optical domain, so it measured through the spectrum of the electrical signal, as described in section **??**, and referred to as "Electrical E_b/n_0". This way, it is possible to accurately study both cases.

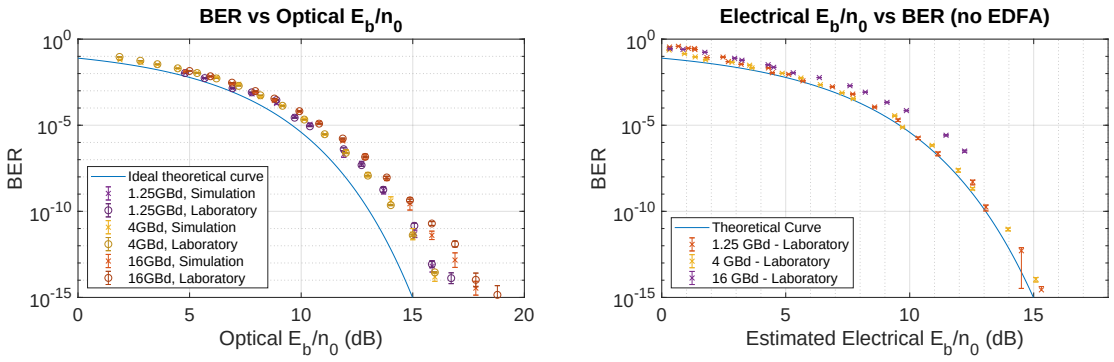
It is worth noting that, when using the "Optical E_b/n_0", it does not encompass all noise sources in the signal. This value only considers the optical noise generated by the EDFA, ignoring all electrical noise sources. Also, it is important to remember that this analysis considers that all nonlinearities in the signal can be neglected or compensated by DSP (a reasonable assumption used in [**crognale14**]). It is also assumed that the receiver is perfectly balanced. While this is a bit more unrealistic, it is the case used in the simulation.

6.1.5 Homodyne Detection

Using the same laser source for transmitting the signal and acting as local oscillator, the frequencies of the carrier and local oscillator are always synchronized. Therefore, using this configuration there should be no need for frequency estimation or phase corrections. However, some DSP is still required.

The final constellations were obtained resorting to the OptDSP libraries for signal processing. The post-processing is done offline and in several stages. It starts by removing the existing skew between the in-phase and quadrature components.These components are acquired in different channels of the oscilloscope with a given timing skew that requires correction. The DC component of the signal is also removed. Afterwards, Gram-Schmidt Orthogonalization is done to compensate possible quadrature imbalances A matched filter is then used to improve the signal's SNR, as discussed previously. A constant modulus algorithm MIMO 2x2 adaptive equalizer is used to compensate for the channel distortion of the transmitted signal. Due to the homodyne configuration, there was no need to perform frequency or phase corrections. Lastly, a Least-Mean-Square MIMO 4x4 is used for further adaptive equalization.

Figure 6.66 shows the BER curves obtained for 16, 4 and 1.25 GBd signals.The theoretical curves were obtained resorting to equation 6.38.



(a) (b)

Figure 6.66: Comparison of theoretical BER curves against obtained results. Optical E_b/n_0 calculated through the optical spectrum, electrical E_b/n_0 estimated from the raw waveforms obtained at the oscilloscope, through spectral analysis of the acquired signal. Locally generated local oscillator .

The E_b/n_0 used to plot the experimental data points in Figure 6.66 is estimated offline. Some further comments on the results follow in the sections of the respective symbol rate.

6.1.5.1 16 GBd Signal

Figures 6.67 to 6.71 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the intradyne configuration, but without phase or frequency corrections.

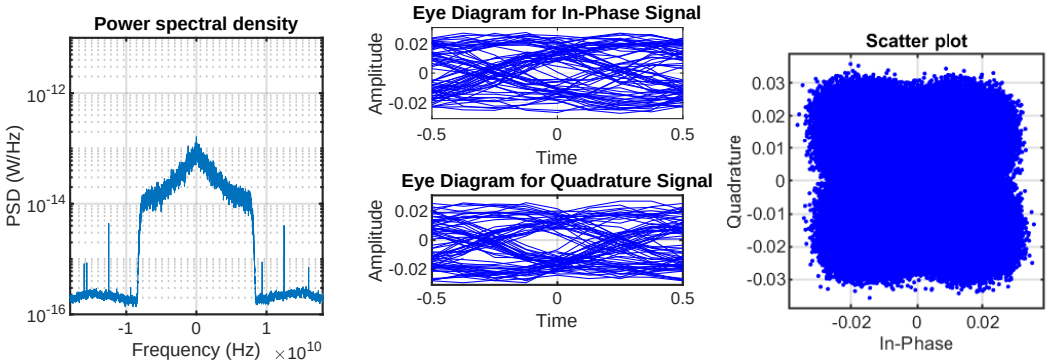


Figure 6.67: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -16.6 dBm.

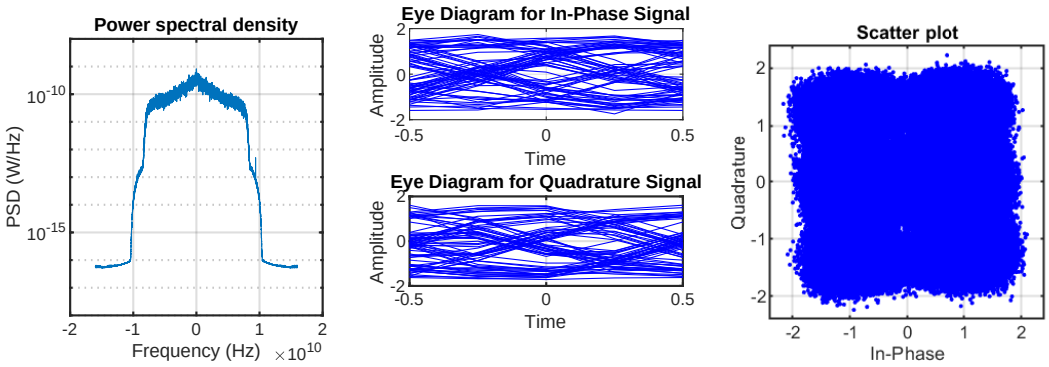


Figure 6.68: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to $2S_R$.

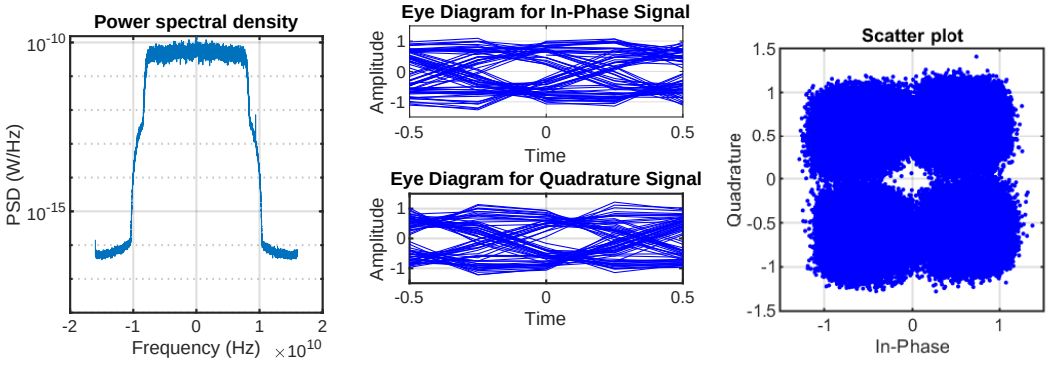


Figure 6.69: Spectrum, eye diagram and constellation after using a contant modulus algorithm.

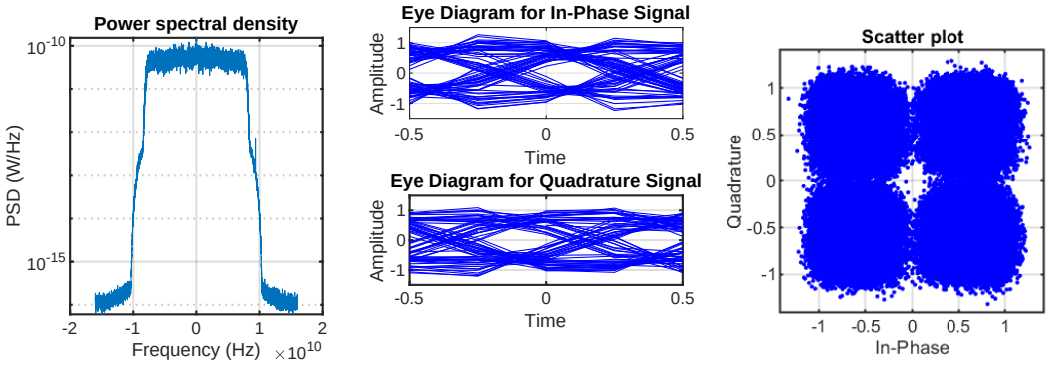


Figure 6.70: Spectrum, eye diagram and constellation after carrier-phase compensation.

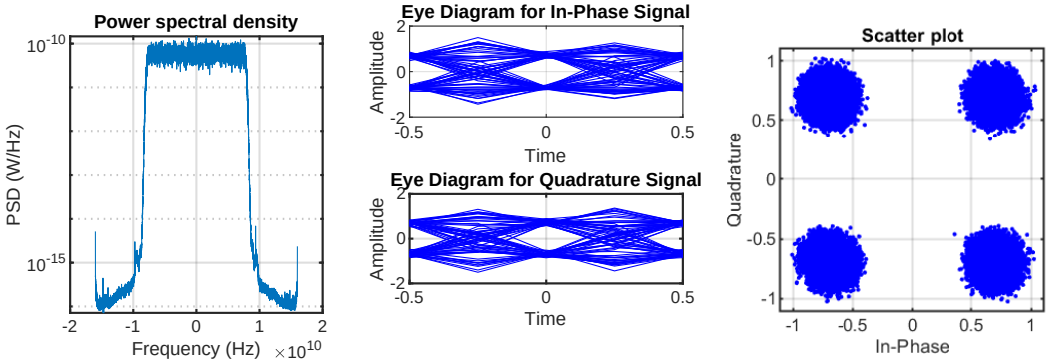
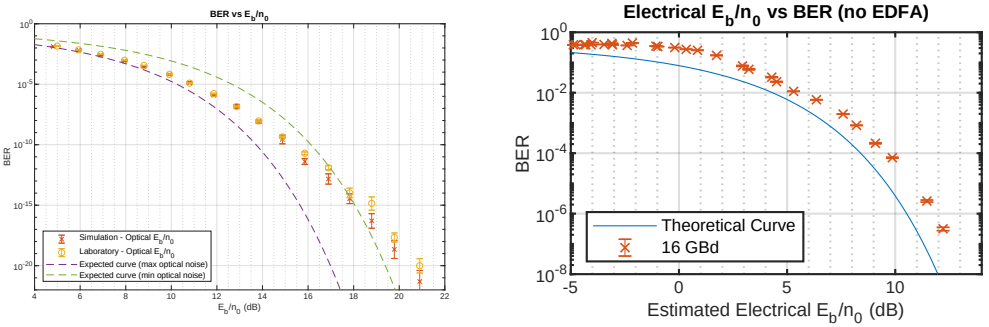


Figure 6.71: Spectrum, eye diagram and constellation after an adaptive equalizer.



(a) (b)

Figure 6.72: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the E_b/n_0 measured in the optical domain, as described in Sections **??** and 6.1.4.3.

Two things can be noticed right away from the plot above: the results from the simulation are very close to the results from the experimental setup; and there are points outside the

curves. The curves shown were the ones described in Section **??**, and should coincide with the two simulation points with the highest and lowest E_b/n_0. Through further testing in the simulation, it was found that the divergence from the curves, at leas of the simulated points, is due to the roll-off values used. Using a higher roll-off (0.9 instead of 0.05) in the simulation made the points follow the curves exactly. However, as the simulations were done trying to replicate the conditions of the experimental setup, the values used were the same, which led to these results. The same effect can be observed for the other symbol rates, with a lower shift from the theoretical curves.

6.1.5.2 4 GBd Signal

This section is fairly similar as the one for the 16 GBd signal with homodyne receiver.

Figures 6.73 to 6.77 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the 16 GBd signal with homodyne receiver. However, some differences can be seen, particularly on the signal spectrum before the corrections. However, it can be seen that the distribution of points in the final constellation is not optimal, unlike in the 16 GBd case.

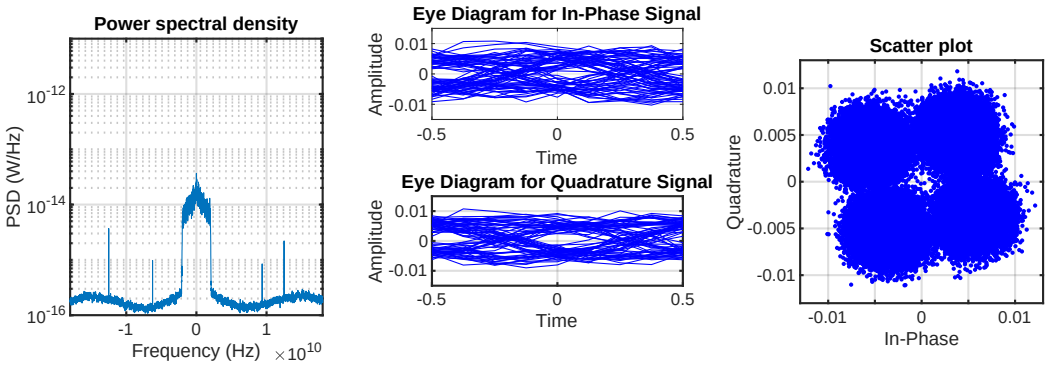


Figure 6.73: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -28.4 dBm.

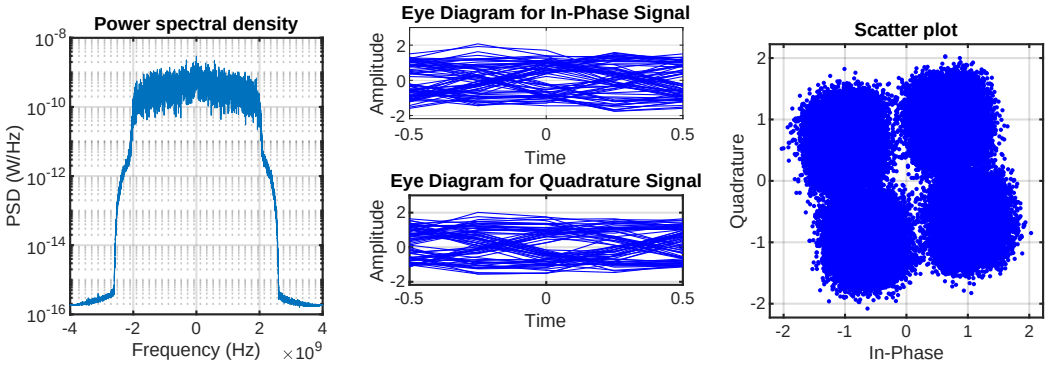


Figure 6.74: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to $2S_R$.

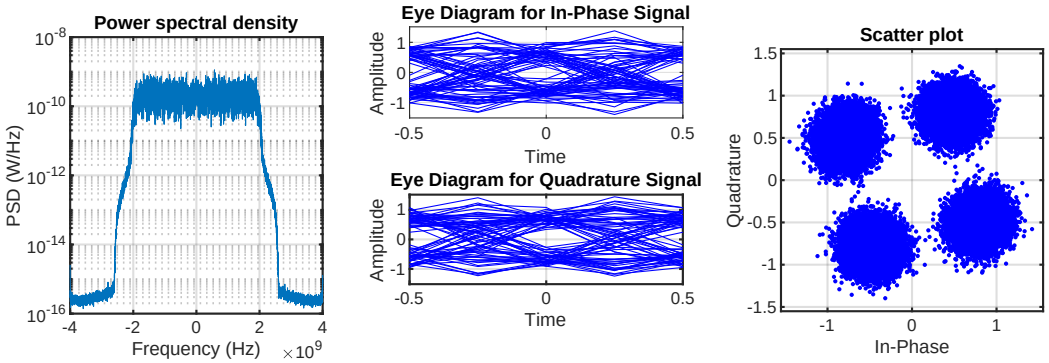


Figure 6.75: Spectrum, eye diagram and constellation after using a contant modulus algorithm.

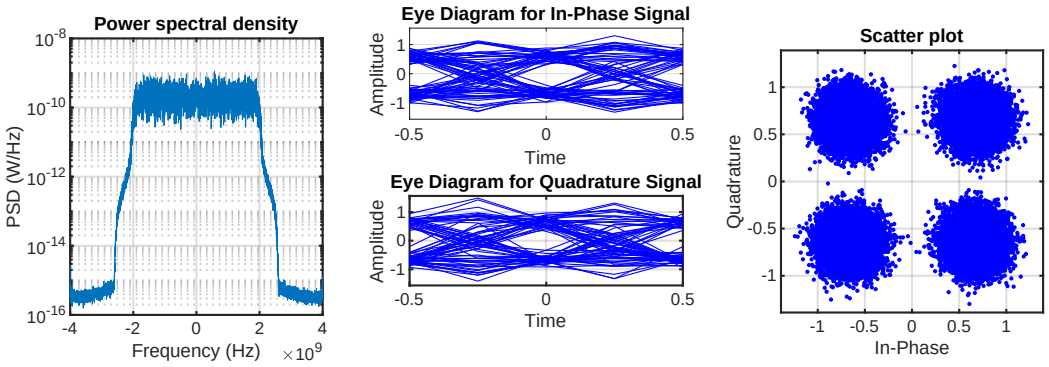


Figure 6.76: Spectrum, eye diagram and constellation after carrier-phase compensation.

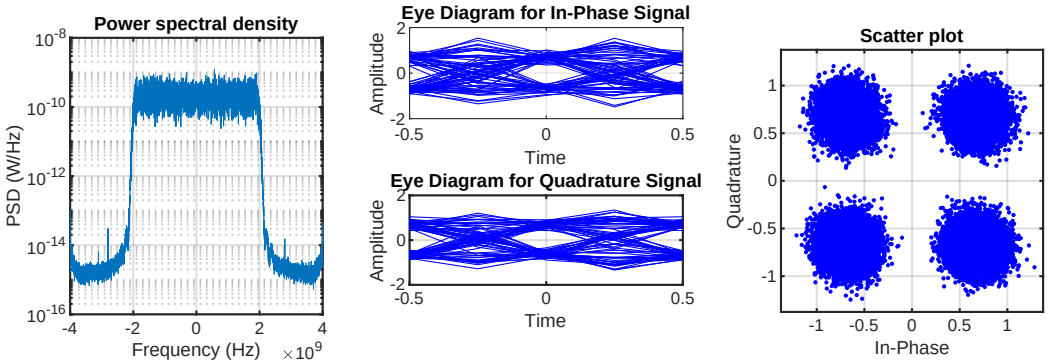
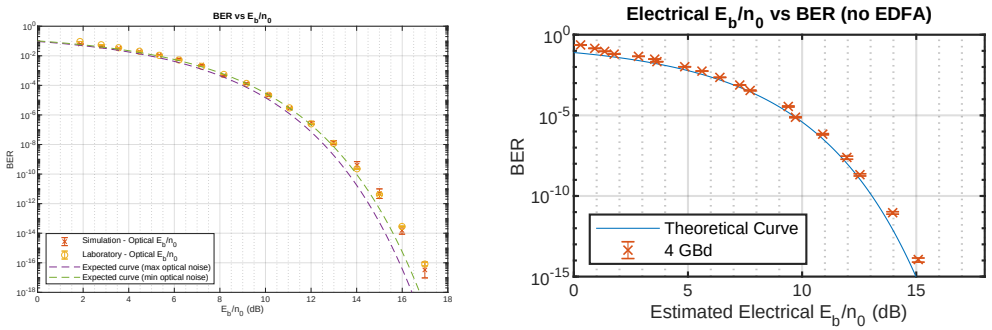


Figure 6.77: Spectrum, eye diagram and constellation after an adaptive equalizer.



(a) (b)

Figure 6.78: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the E_b/n_0 measured in the optical domain, as described in Sections **??** and 6.1.4.3.

This plot is pretty similar to the 16 GBd. The major difference here is that the two curves appear to be much closer together. As the curves depend on the relation between optical and electrical noise, this is to be expected. In this case, the transmitter power was typically much lower than in the 16 GBd plot, and it was spread through a smaller range. This means that the variation on the optical noise produced by the EDFA wasn't as noticeable, and therefore the curves should be closer together.

6.1.5.3 1.25 GBd signal

This section is similar as the one for the 16GBd signal with homodyne receiver.

Figures 6.79 to 6.83 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the 16 GBd signal with homodyne receiver. However, some differences can be seen, particularly on the signal spectrum before the corrections. However, it can be seen that the distribution of points in the final constellation is not optimal, unlike in the 16 GBd case.

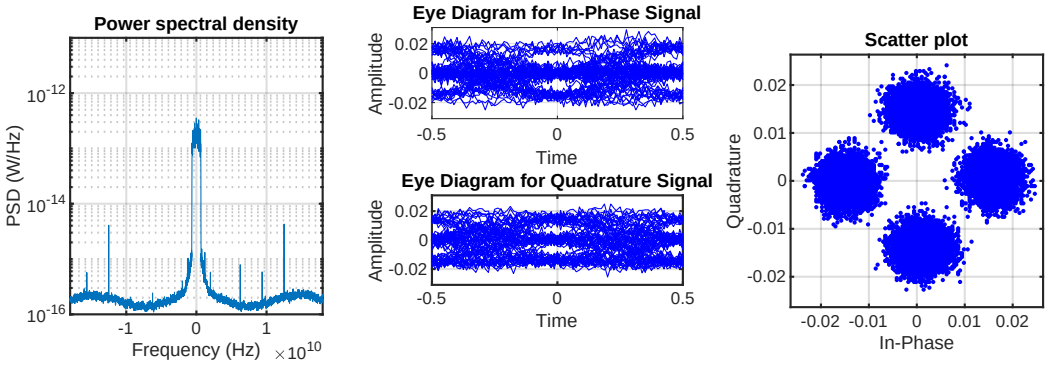


Figure 6.79: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -22.2 dBm.

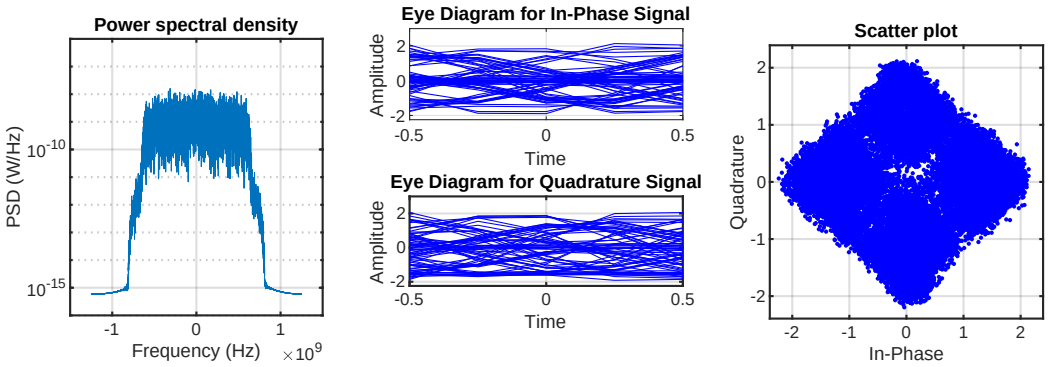


Figure 6.80: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to $2S_R$.

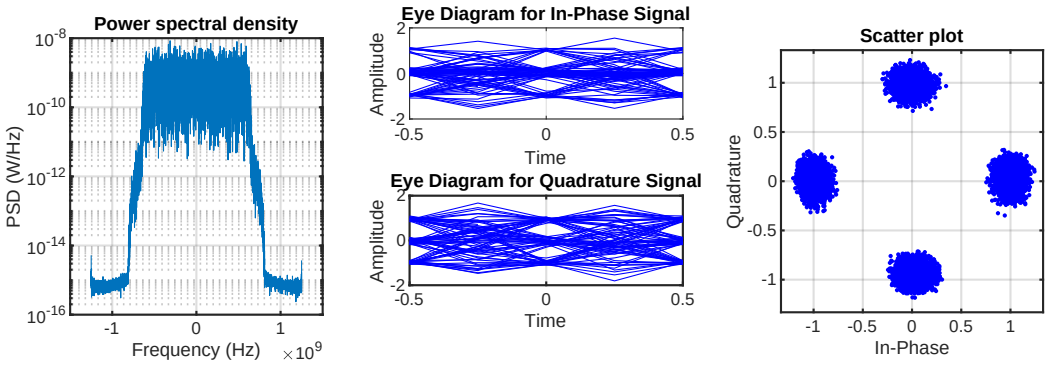


Figure 6.81: Spectrum, eye diagram and constellation after using a constant modulus algorithm.

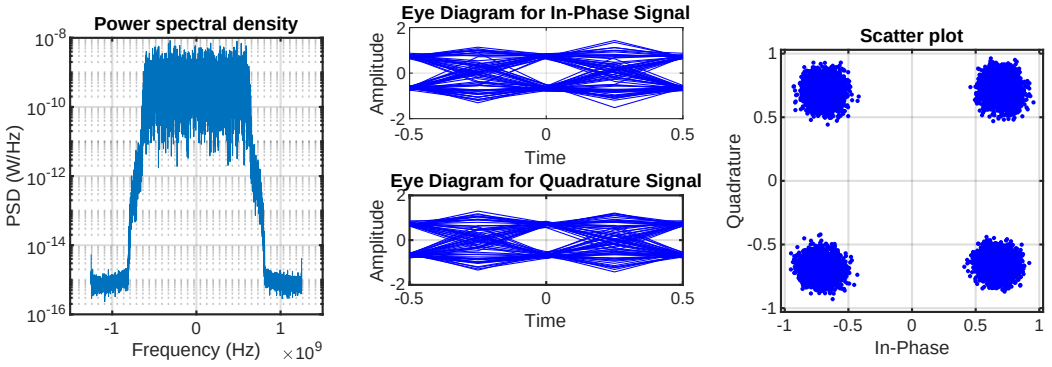


Figure 6.82: Spectrum, eye diagram and constellation after carrier-phase compensation.

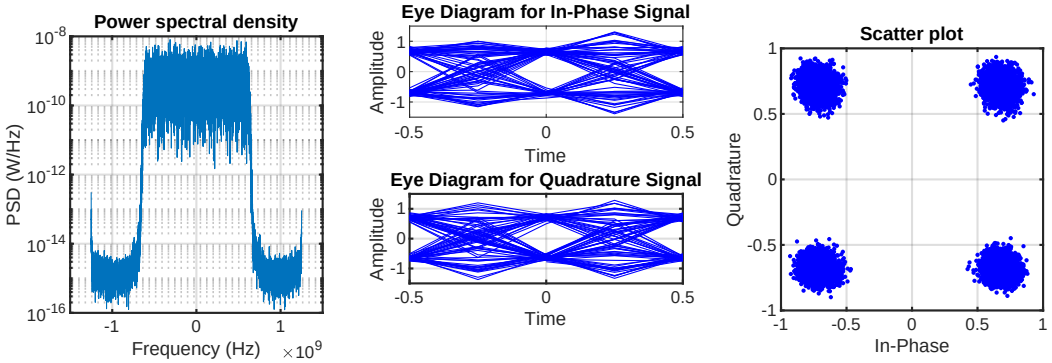
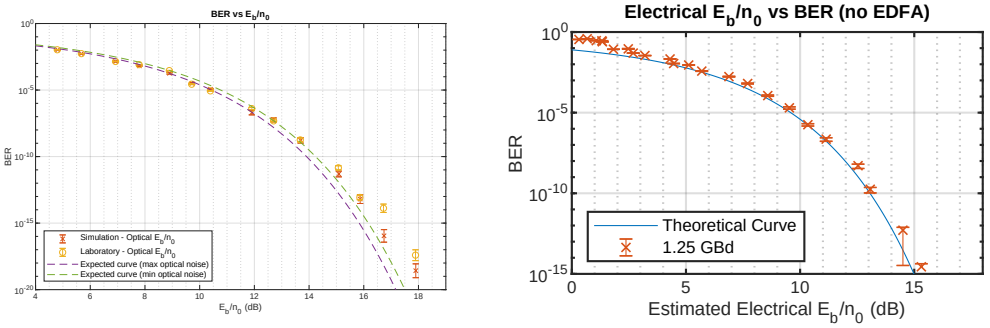


Figure 6.83: Spectrum, eye diagram and final constellation after an adaptive equalizer. The BER estimated from the Q-factor is 9.2598×10^{-29}



(a) (b)

Figure 6.84: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the E_b/n_0 measured in the optical domain, as described in Sections **??** and 6.1.4.3.

This plot is not very different from the 4 GBd case, and the simulation continues to show

good agreement with the experimental results.

6.1.6 Digital Signal Post-Processing

The stages of the DSP process applied to the experimental results of this setup are presented in Figure 6.85. After conversion to digital and combination in a complex vector, the signal is

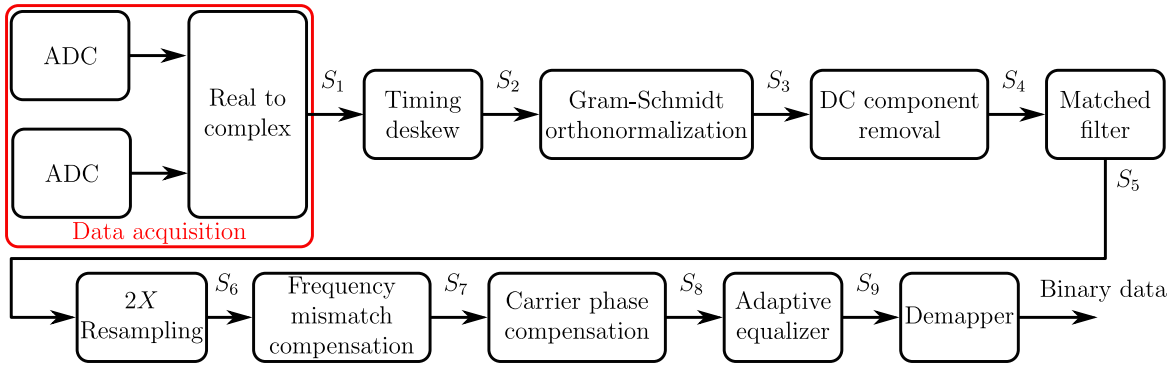


Figure 6.85: Block diagram of the DSP applied offline in the netxpto environment.

first passed through a timing-deskew process, in which IN PROGRESS

The constellation on which the DSP will be implemented is presented in Figure 6.86. This data was obtained from the .mat file **SNR=38_trial8_.mat**, included in the folder *\sdf\m_qam_system_dsp\Data*. A short matlab code present in same folder, named **mat2txt.m**, reads the **.mat** file and outputs the received signal to a text file named **Input.txt**. This **.txt** file can be read into the netxpto environment by using the *LoadAscii* block, the code necessary for this is included below.

```
TimeContinuousAmplitudeContinuousComplex SIn{ "SIn.sgn" };

LoadAscii BI{ {}, { &SIn } };
BI.setAsciiFileName("Input.txt");
BI.setDataType(ComplexValue);
BI.setDelimiterType(delimiter_type::CommaSeperatedValues);
BI.setSamplingPeriod(1 / 50e9);
BI.setSymbolPeriod(1 / 1.25e9);
```

After loading the signal to the netxpto environment for a first time, it is saved in the **.sgn** format, this version of the signal can then be loaded into the netxpto environment with the *LoadSignal* block, the code necessary for this is included below.

```
TimeContinuousAmplitudeContinuousComplex S1{ "S1.sgn" };

LoadSignal BI{ {}, { &S1 } };
BI.setSgnFileName("SIn.sgn");
```

This constellation was obtained by downsampling the signal obtained from the oscilloscope down to 1 sample per symbol and roughly centering the sampling point to its peaks. All DSP steps applied to this constellation are now described one by one.

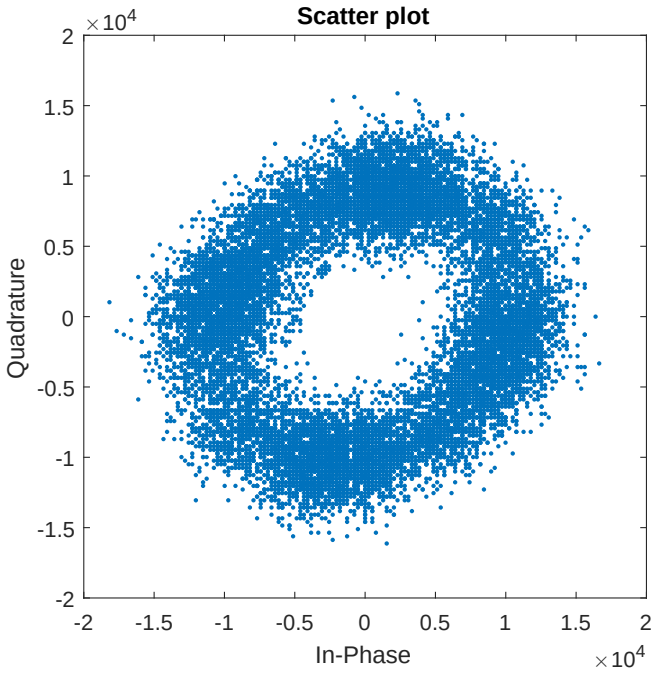


Figure 6.86: Constellation obtained at the oscilloscope from a signal with a baud rate of 1.25 Gbd and a SNR of 38 dB.

6.1.6.1 Timing deskew

This DSP step takes a complex input signal and removes a given amount of timing skew between its real (in-phase) and imaginary (quadrature) parts. This DSP step follows the topology presented in Figure 6.87 The input signal $S_{\text{in}}(n)$ is separated into its in-phase and

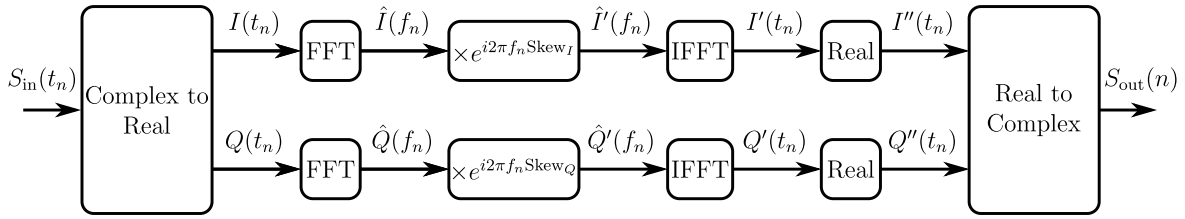


Figure 6.87: Block diagram representation of the deskew procedure.

in-quadrature components, $I(n)$ and $Q(n)$, both components are transformed to Fourier space, where they are multiplied by a deskew element

$$\hat{I}'(f_n) = \hat{I}(f_n)e^{i2\pi f_n \text{Skew}_I}, \tag{6.54}$$

$$\hat{Q}'(f_n) = \hat{Q}(f_n)e^{i2\pi f_n \text{Skew}_Q}. \tag{6.55}$$

These two components are then transformed back to time domain, the imaginary part of each component is discarded and the resulting components are combined to form the output

signal

$$S_{\text{out}}(t_n) = \text{real}(I'(t_n)) + i\text{real}(Q'(t_n)). \tag{6.56}$$

The effect of this DSP step is presented in Figure 6.88.

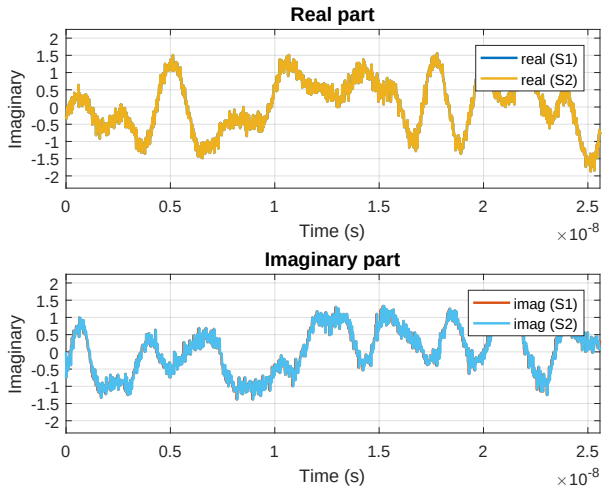


Figure 6.88: Signals before and after deskew DSP step.

6.1.6.2 Gram-Schmidt orthonormalization

This step orthonormalizes the data by implementing a Gram-Schmidt algorithm. This is implementation follows the topology presented in Figure 6.89. The input signal $S_{\text{in}}(t_n)$ is

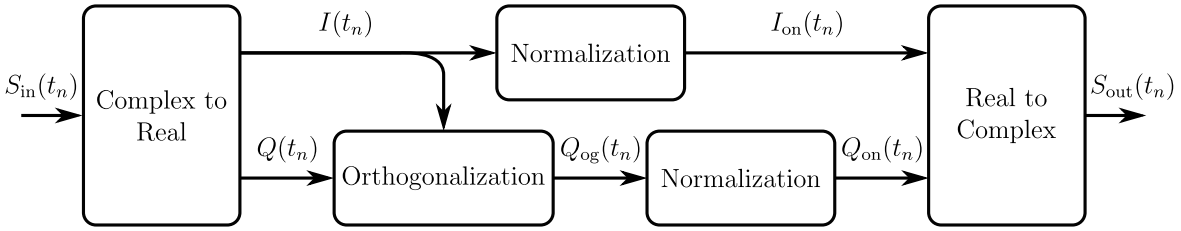


Figure 6.89: Block diagram representation of the Gram-Schmidt orthonormalization procedure.

separated into its in-phase and in-quadrature components, $I(t_n)$ and $Q(t_n)$, the amplitude of the in-phase component, P_I, is estimated as the average of its square, ie.:

$$P_I = \text{mean}(I^2(t_n)), \tag{6.57}$$

the in-phase signal is then normalized, ie.:

$$I_{\text{on}}(t_n) = \frac{I(t_n)}{\sqrt{P_I}}. \tag{6.58}$$

The in-quadrature component is then orthogonalized in relation to the in-phase component

$$Q_{\text{og}}(t_n) = Q(t_n) - \frac{I(t_n)\text{mean}(I(t_n)Q(t_n))}{P_I}, \tag{6.59}$$

which is then normalized in a manner similar to what was done for the in-phase component

$$P_Q = \text{mean}(Q_{\text{og}}^2(t_n)), \tag{6.60}$$

$$Q_{\text{on}}(t_n) = \frac{Q_{\text{og}}(t_n)}{\sqrt{P_Q}}. \tag{6.61}$$

Finally, the two orthonormalized components are combined to form the output signal

$$S_{\text{out}}(t_n) = I_{\text{on}}(t_n) + iQ_{\text{on}}(t_n) \tag{6.62}$$

The effect of this DSP step is presented in Figure 6.90.

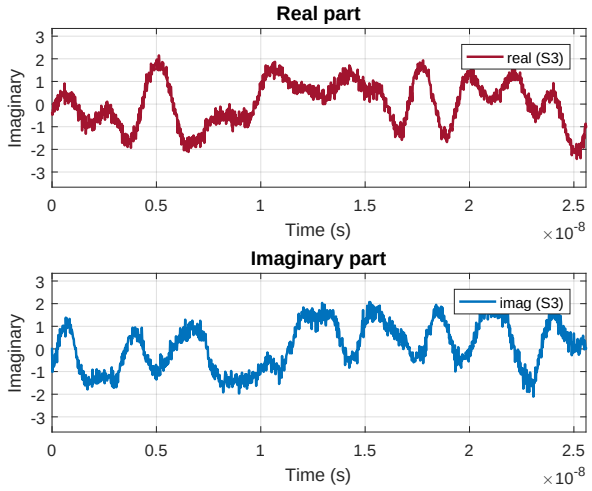


Figure 6.90: Signal after orthonormalization DSP step.

6.1.6.3 DC component removal

This DSP step removes the DC contribution on the input signal ,$S_{\text{in}}(t_n)$. This DC contribution can either be estimated by taking the average of the full input signal or from the average of a moving window. This contribution is then subtracted from the input

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n) - \text{mean}(S_{\text{in}}(t_n)) \tag{6.63}$$

The effect of this DSP step is presented in Figure 6.91.

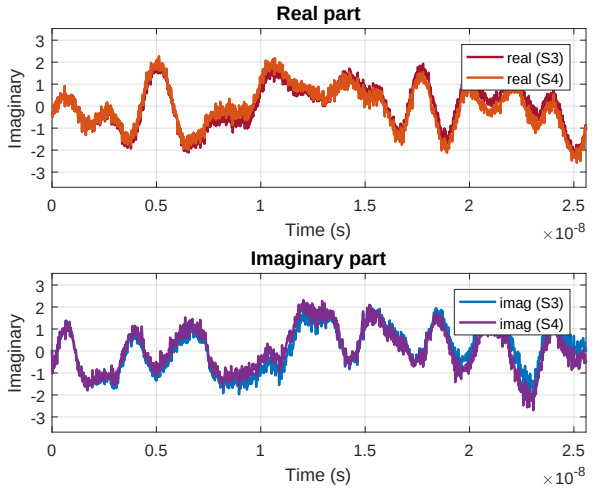


Figure 6.91: Signals before and after DC component removal DSP step.

6.1.6.4 Matched filtering

This step implements some form of matched filtering. For the modulation of the signal corresponding to the constellation in Figure 6.86, the matched filter consists of a root raised cosine function. The filter can be applied either in time-domain, through convolution of the signal with the filtering function, or in frequency domain, through a simple multiplication.

The effect of this DSP step is presented in Figure 6.92.

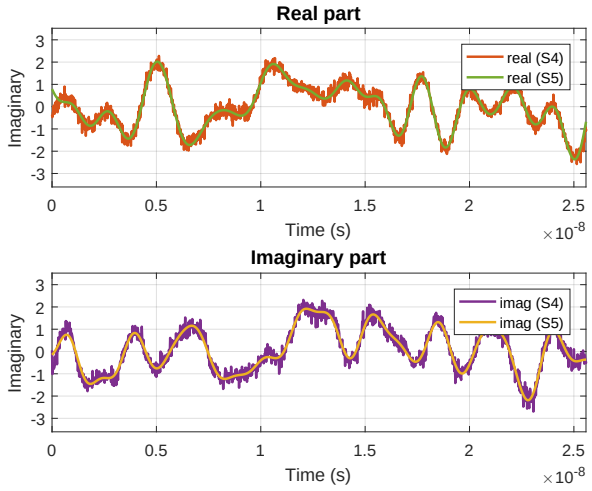


Figure 6.92: Signals before and after matched filtering.

6.1.6.5 2× resampling

This step consists of a digital resampling that effectively doubles the sampling rate of the signal, with the values of the new points filled using a polyphase anti-aliasing filter.

6.1.6.6 Frequency mismatch compensation

This step compensates for the frequency mismatch between the local oscillators employed at the transmission and reception stages. Multiple methods for this are available, the one employed for the results presented here is the spectral method, a topological representation of which is presented in Figure 6.93. The input signal, which can be described as

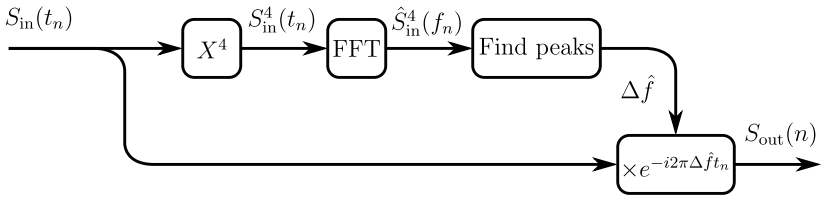


Figure 6.93: Block diagram representation of the spectral frequency mismatch estimation and compensation procedure.

$$S_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi\Delta f t_n + \theta(t_n) + \Delta\phi)}, \tag{6.64}$$

is powered by 4, yielding

$$S^4_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi 4\Delta f t_n + 4\theta(t_n) + 4\Delta\phi)}, \tag{6.65}$$

since $\theta(t_n) \in \left\{\frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4}\right\}$, we get

$$S^4_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi 4\Delta f t_n + \pi + 4\Delta\phi)}, \tag{6.66}$$

so this process effectively removes the QPSK modulation of the signal. The $S^4_{\text{in}}(t_n)$ signal is then converted to the frequency domain, because of the removal of the phase modulation, its spectrum is expected to display a maximum at a frequency of $4\Delta f$, which is determined by some *find peaks* function, yielding the estimate $\Delta\hat{f}$. The input signal S_{in} is then multiplied by $e^{-i2\pi\Delta\hat{f}t_n}$. Assuming the estimate is a good one, this process will remove the effect of the frequency mismatch, yielding the following output signal

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n)e^{-i2\pi\Delta\hat{f}t_n} = |S_{\text{in}}(t_n)|e^{i(\theta(t_n)+\Delta\phi)} \tag{6.67}$$

6.1.6.7 Carrier phase compensation

This step compensates for the phase mismatch between the local oscillators employed at the transmission and reception stages. Multiple methods for this are available, the one employed for the results presented here is the blind method, a topological representation

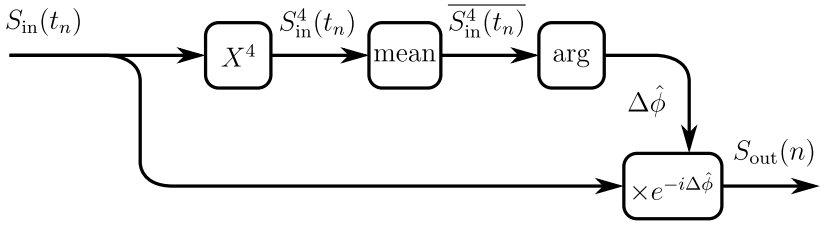


Figure 6.94: Block diagram representation of the blind phase mismatch estimation and compensation procedure.

of which is presented in Figure 6.94. The input signal, which at this stage can be described as

$$S_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(\theta(t_n)+\Delta\phi)}, \tag{6.68}$$

is powered by 4, yielding

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(4\theta(t_n)+4\Delta\phi(t_n))}, \tag{6.69}$$

since $\theta(t_n) \in \left\{\frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4}\right\}$, we get

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(\pi+4\Delta\phi(t_n))}, \tag{6.70}$$

so this process effectively removes the QPSK modulation of the signal. The average of the $S_{\text{in}}^4(t_n)$ signal is then computed and its phase evaluated, yielding the estimate $4\Delta\hat{\phi}$. The input signal S_{in} is then multiplied by $e^{-i\Delta\hat{\phi}}$. Assuming the estimate is a good one, this process will remove the effect of the phase mismatch, yielding the following output signal

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n)e^{-i\Delta\hat{\phi}} = |S_{\text{in}}(t_n)|e^{i\theta(t_n)} \tag{6.71}$$

6.1.6.8 Adaptive equalizer

6.1.7 Open Issues

1. The DSP used to process the lab data needs to be implemented in the NetXPTO platform, as it is currently implemented in MATLAB.
    - Currently under development.
2. It is necessary to implement a way to open external signals from the oscilloscope on the netXPTO platform.
3. There are still differences between the data produced in the simulation and the data measured in the Lab.

6.1.8 Future work

Extend this block to include other values of M.

Chapter 7

Library

7.1 Add

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

Input Parameters

This block takes no parameters.

Functional Description

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

Input Signals

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

Output Signals

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

7.2 Bit Error Rate

Header File	:	bit_error_rate_*.h
Source File	:	bit_error_rate_*.cpp
Version	:	20171810 (Daniel Pereira)
	:	20181424 (Mariana Ramos)

Input Parameters

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

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; }

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. In version **20181113** instead of the previous binary output string, this block outputs a 0 if the two input samples are equal and 1 if not.

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:

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \tag{7.1}$$

The upper and lower bounds, BER_{UB} and BER_{LB} respectively, are calculated using the Clopper-Pearson confidence interval, which returns the following simplified expression for $N_T > 40$ **[Almeida16]**:

$$\text{BER}_{\text{UB}} = \widehat{\text{BER}} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 + (2 - \widehat{\text{BER}})\right] \tag{7.2}$$

$$\text{BER}_{\text{LB}} = \widehat{\text{BER}} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1-\widehat{\text{BER}})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \widehat{\text{BER}}\right) z_{\alpha/2}^2 - (1 + \widehat{\text{BER}})\right], \tag{7.3}$$

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

Version 20181424

Version 20181424 allows the user to choose the type of middle reports he wants. So, the input parameter mideRepType can has the value *Cumulative*, where the BER estimation is done by taking into account all samples acquired in a cumulative way, or *Reset*, where the BER estimation is done by taking into account only the number of samples set as *m* in each middle report.

- **Input Parameters**

Name	Type	Default Value
midRepType	MidReportType	Cumulative

- **Methods**
  - void setMidReportType(MidReportType mrt) { midRepType = mrt; };

7.3 Binary Source

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

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

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

Signals

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

Table 7.1: Binary source signals

Input Parameters

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

Table 7.2: 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)

Illustrative Examples

Random Mode

PseudoRandom Mode Consider a pseudorandom sequence with *patternLength*=3 which contains a total of 7 ($2^3 - 1$) bits. In this sequence it is possible to find every combination of 0's and 1's that compose a 3 bit long subsequence with the exception of 000. For this example the possible subsequences are 010, 110, 101, 100, 111, 001 and 100 (they appear in figure 7.1 numbered in this order). Some of these require wrap.

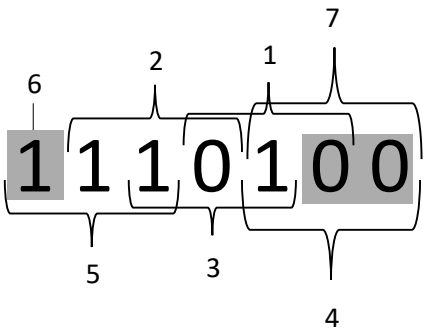


Figure 7.1: Example of a pseudorandom sequence with a pattern length equal to 3.

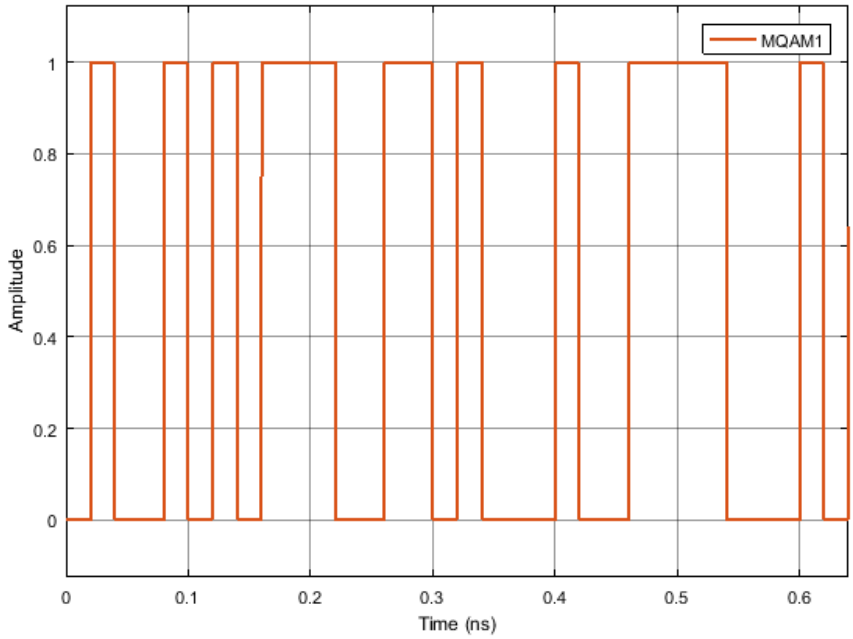


Figure 7.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

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

DeterministicAppendZeros Mode Take the *bit stream* '0100011101010101'. The generated binary signal is displayed in 7.2.

Suggestions for future improvement

Implement an input signal that can work as trigger.

7.4 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 7.3: Binary source input parameters

Methods

Decoder()

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

void initialize(void)

bool runBlock(void)

void setM(int mValue)

void getM()

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues>getIqAmplitudes()

Functional description

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

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

Input Signals

Number: 1

Type: Electrical complex (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Binary

Examples

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

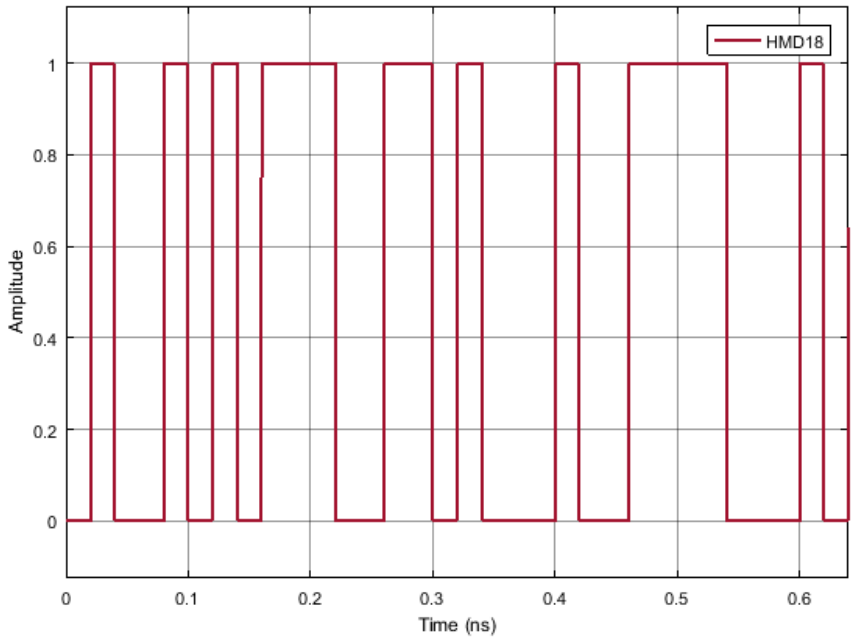


Figure 7.3: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

7.5 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 7.4: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)

int const getNumberOfSamplesPerSymbol(void)

Functional Description

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

Input Signals

Number : 1

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

Output Signals

Number : 1

Type : Sequence of Dirac delta functions (ContinuousTimeDiscreteAmplitude)

Example

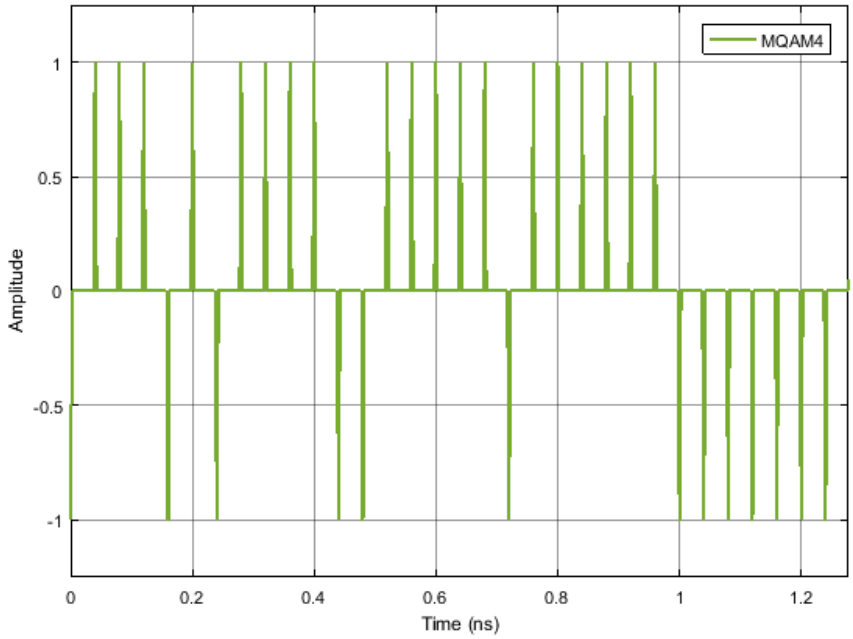


Figure 7.4: Example of the type of signal generated by this block for a binary sequence 0100...

7.6 M-QAM Receiver

Header File	:	m_qam_receiver.h
Source File	:	m_qam_receiver.cpp
Version	:	20180815 (*Pedro Loureiro*)

This block simulates the reception and demodulation of an optical signal (which is the input signal of the system) and outputs a binary signal corresponding to the reconstructed transmitted bitstream, which will later be used to calculate bit error rate (BER). A simplified schematic representation of this block is shown in figure 7.5.

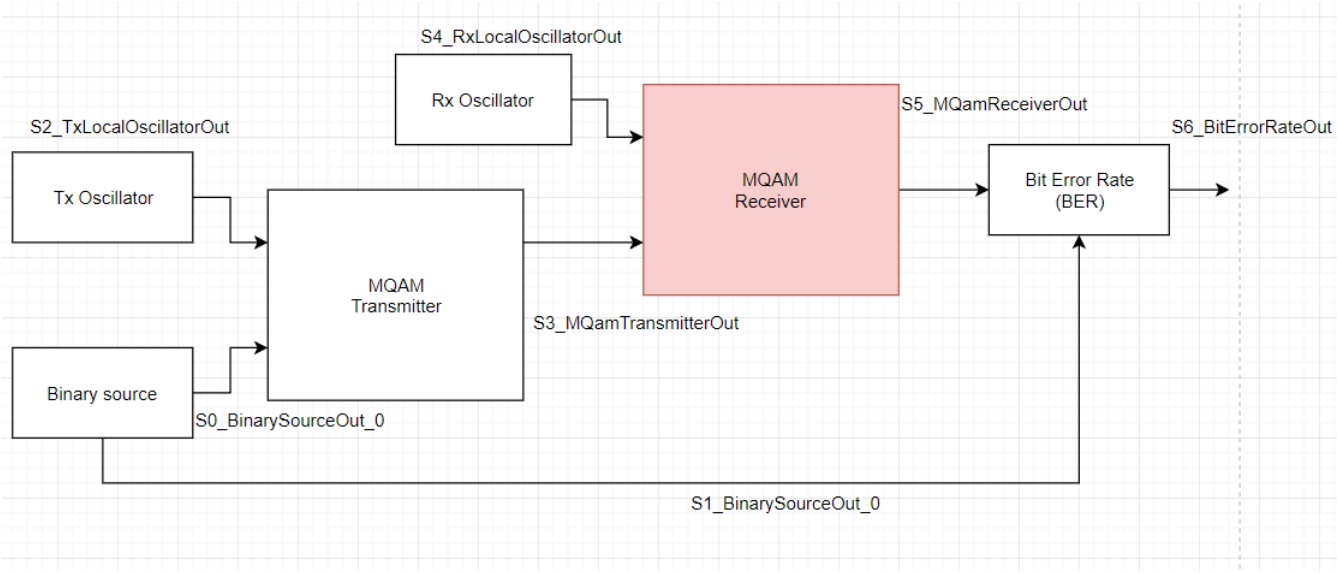


Figure 7.5: Simplified model of the MQAM receiver

Functional description

This block accepts two optical inputs signals and outputs one binary signal that corresponds to the decoded information transmitted in the input signal. It is a complex block (as it can be seen from figure 7.6) made up of several simpler blocks whose description can be found in the *lib* repository. Unlike the block in 7.6, this block does not include the local oscillator.

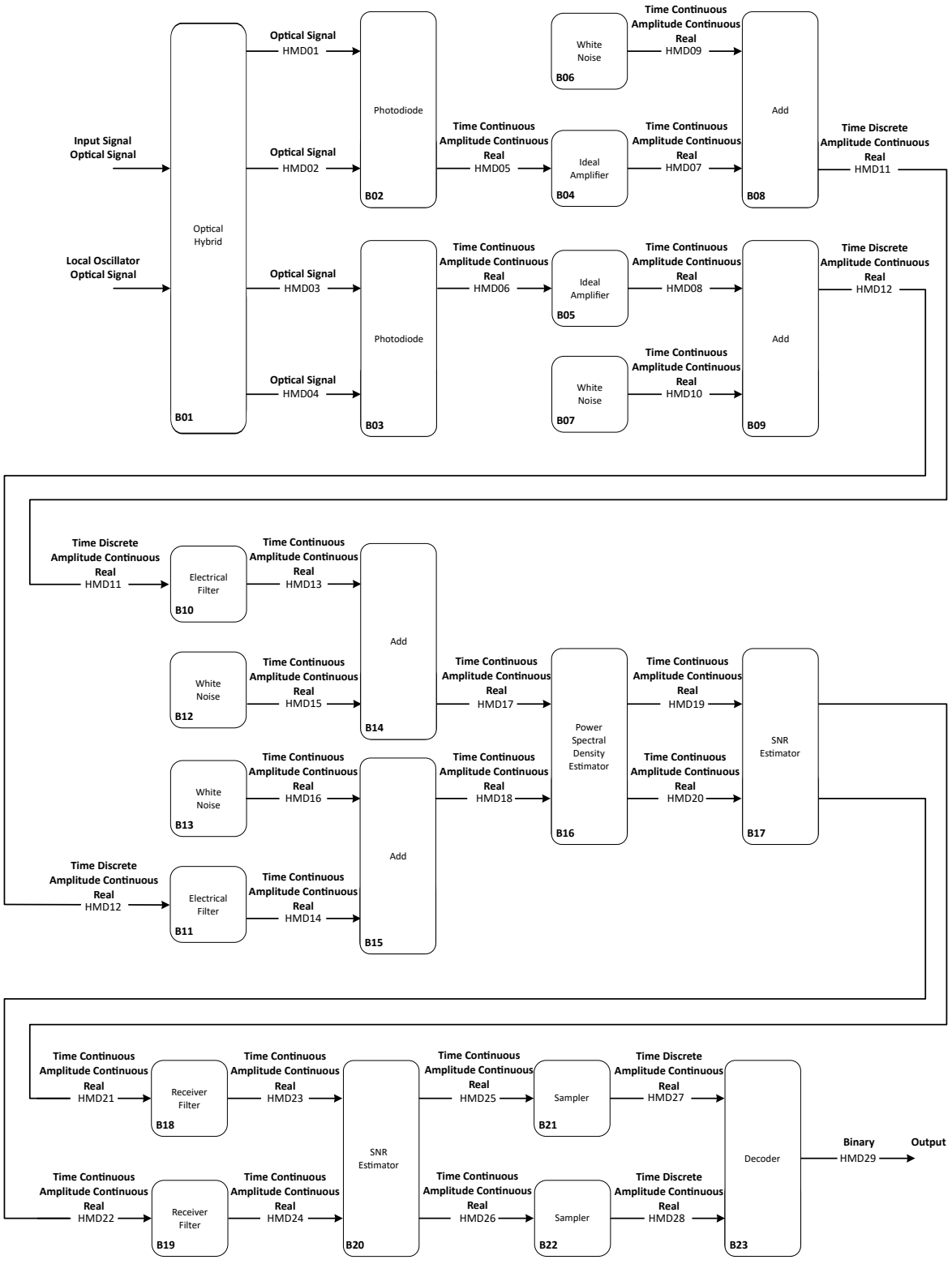


Figure 7.6: Schematic representation of the block homodyne receiver.

Input parameters

This block input parameters that can be manipulated by the user in order to change the configuration of the receiver. Each parameter is changed by calling a particular function. In the following table (Table 7.11) the input parameters and corresponding functions are summarized.

Parameter	Type	Values	Default
signalsFolderName	string	any	signals/ SuperBlock_MQamReceiver
logValue	bool	any	true
logFileName	string	any	SuperBlock_ _MQamReceiver.txt
Photodiodes			
Responsivity	t_real	any	1
TI Amplifier			
NoiseSpectralDensity	t_real	any	$1.5e^{-17}$
gain	t_real	any	$1e^{4}$
fType	Filter	any[1]	LowPass
ctfFreq	double	any	5
irl	int	any	128
ir	vector<t_real>	any	0
fName	string	any	impulse_response.imp
sBeginningOf ImpulseResponse	bool	any	true
General Noise			
SamplingPeriod	t_real	any	1.0
nSymbolPeriod	t_real	any	1.0
Thermal noise			
NoiseSpectralDensity	t_real	any	$1.5e^{-17}$
cp	bool	any	0
noiseSeeds	array<int, 2>	any	1
seedType	SeedType	any[2]	RandomDevice
Pulse shaper			
impResponseTimeLength	int	any	16
fType	pulse_shapper_filter_type	any[3]	RaisedCosine
rOffFactor	double	$\in [0, 1]$	0.9
ne	bool	any	false
pFilterMode	bool	any	false
sBeginningOf ImpulseResponse	bool	any	true
Sampler			
sToSkip	int	any	0
Decoder			
iqAmplitudesValues	vector<t_iqValues>	any	{ { 1, 1 },{ -1, 1 }, { -1, -1 },{ 1, -1 } }

Table 7.5: Binary source input parameters

[1] LowPass, Defined, Unitary

[2] RandomDevice, DefaultDeterministic, SingleSelected
[3] RaisedCosine, Gaussian, Square, RootRaisedCosine

Methods

1. Block Declaration and Initialization

   - MQamReceiver(initializer_list<Signal *> &inputSig, initializer_list<Signal *> &outputSig)
   - void initialize(void)
   - bool runBlock(void)

2. Functions to set parameters

   - void setPhotodiodesResponsivity(t_real Responsivity)
   - void setGain(t_real gain)
   - void setAmplifierInputNoisePowerSpectralDensity(t_real NoiseSpectralDensity)
   - void setTiAmplifierFilterType(Filter fType)
   - void setTiAmplifierCutoffFrequency(double ctfFreq)
   - void setTiAmplifierImpulseResponseTimeLength_symbolPeriods(int irl)
   - void setElectricalFilterImpulseResponse(vector<t_real> ir)
   - void setElectricalImpulseResponseFilename(string fName)
   - void setElectricalSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)
   - void setNoiseSamplingPeriod(t_real SamplingPeriod)
   - void setNoiseSymbolPeriod(t_real nSymbolPeriod)
   - void setThermalNoiseSpectralDensity(t_real NoiseSpectralDensity)
   - void setThermalNoisePower(t_real NoiseSpectralDensity)
   - void setThermalConstantPower(bool cp)
   - void setSeeds(array<int, 2> noiseSeeds)
   - void setSeedType(SeedType seedType)
   - void setThermalConstantPower(bool cp)
   - void setImpulseResponseTimeLength(int impResponseTimeLength)
   - void setFilterType(pulse_shapper_filter_type fType)
   - void setRollOffFactor(double rOffFactor)
   - void usePassiveFilterMode(bool pFilterMode)
   - void setRrcNormalizeEnergy(bool ne)
   - void setMFImpulseResponseFilename(string fName)

- void setMFSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)
- void setSamplesToSkip(int sToSkip)
- void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

3. Functions to get parameters

- t_real getGain(void)
- t_real getAmplifierInputNoisePowerSpectralDensity(void)
- double const getElectricalSeeBeginningOfImpulseResponse(void)S
- double const getMFSeeBeginningOfImpulseResponse(void)
- vector<t_iqValues> const getIqAmplitudes(void)

Input Signals

Number: 2

Type: 1 optical signal from the local oscillator and 1 optical signal from the transmitter

Output Signals

Number: 1

Type: Binary signal

Example

Receiver sensitivity

In order to analyze the sensitivity of the receiver were tested some thermal noise power values to obtain to obtain the respective BER. The graph is shown in figure 7.7.

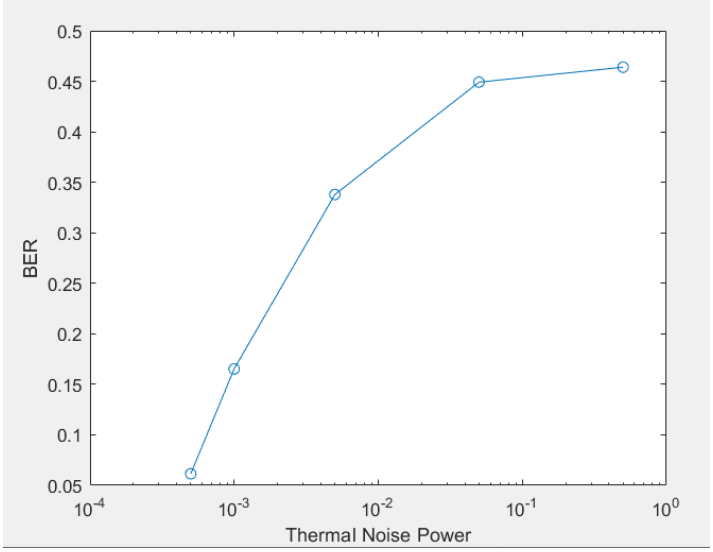
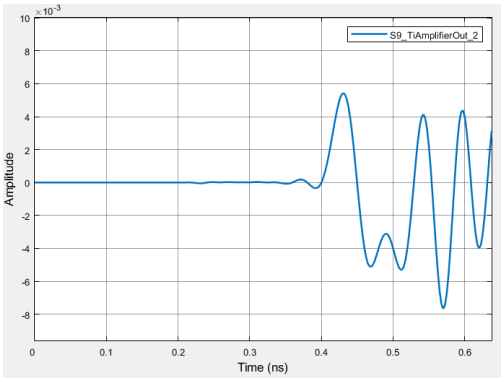


Figure 7.7: Example of the BER in function of Thermal noise power

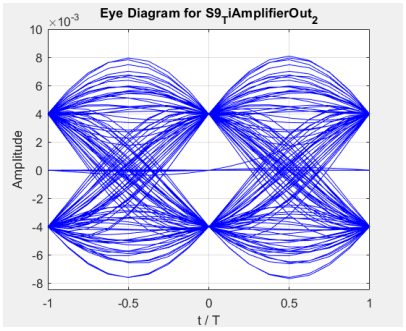
As expected the increase of the thermal noise power increase also the BER.

Impact of the electrical amplifier

To evaluate the impact of the electrical amplifier we tried to change the bandwidth of the filter. The signals at the output of the electrical amplifier was compared when the bandwidth was 50Ghz and 5GHz.

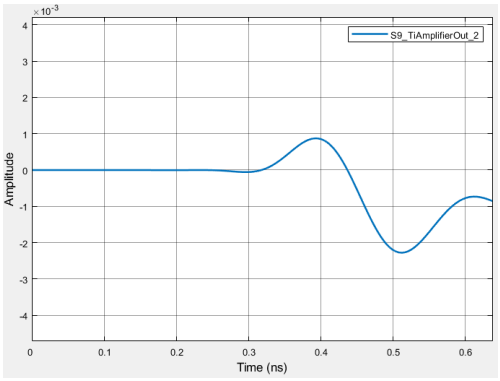


(a) Time domain when the bandwidth was 5GHz

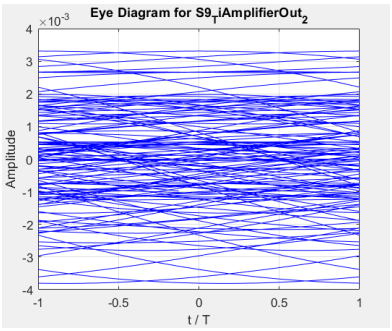


(b) Eye diagram when the bandwidth was 50GHz

Figure 7.8



(a) Time domain when the bandwidth was 5GHz



(b) Eye diagram when the bandwidth was 5GHz

Figure 7.9

By the figures 7.9 and 7.8 when the bandwidth was reduced from 50GHz to 5GHz the signal, the signal was distorted and ber increased.

Impact of the electrical filter

NÃ£o Ã© o mesmo que se falou no pulse shaper?

Sugestions for future improvement

7.7 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 7.6: 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

7.8 IQ Modulator

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

Version 20180130

This blocks accepts one 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 7.7: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setOutputOpticalPower(double outOpticalPower)

void setOutputOpticalPower_dBm(double outOpticalPower_dBm)

void setOutputOpticalWavelength(double outOpticalWavelength)

void setOutputOpticalFrequency(double outOpticalFrequency)

Functional Description

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

Input Signals

Number : 2

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

Output Signals

Number : 1 or 2

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

Example

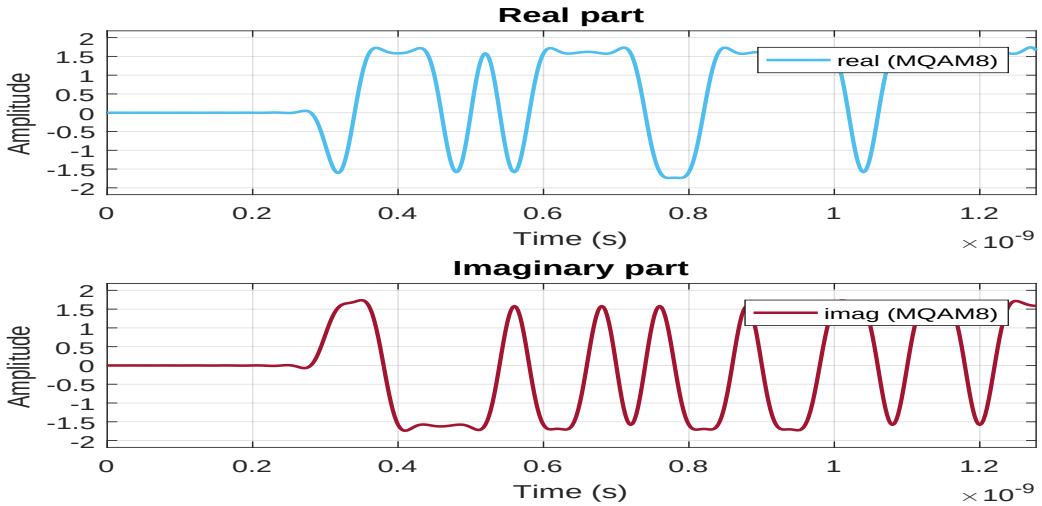


Figure 7.10: Example of a signal generated by this block for the initial binary signal 0100...

Version 20180828

Input Parameters:

—NA—

Input Signals:

Number: 1, 2,3
Type: RealValue

Output Signals:

Number: 4
Type: RealValue

Functional Description

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

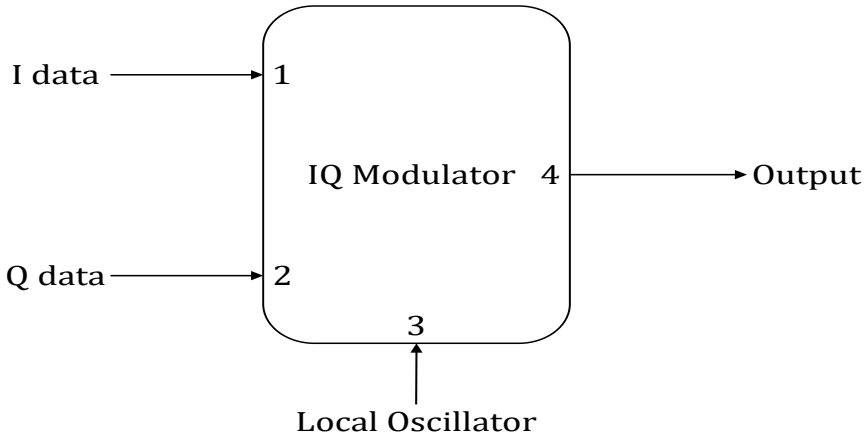


Figure 7.11: IQ Modulator block

IQ MZM Description

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

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

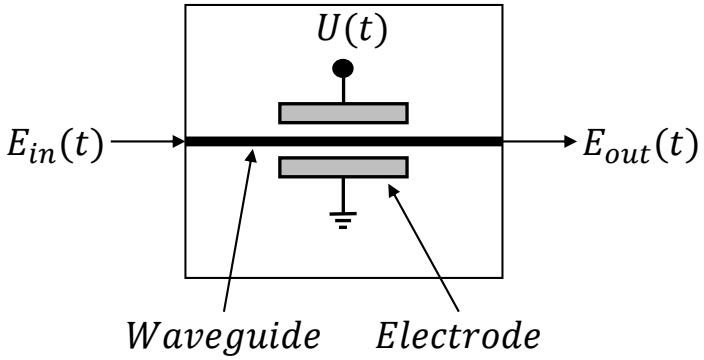


Figure 7.12: Phase Modulator

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

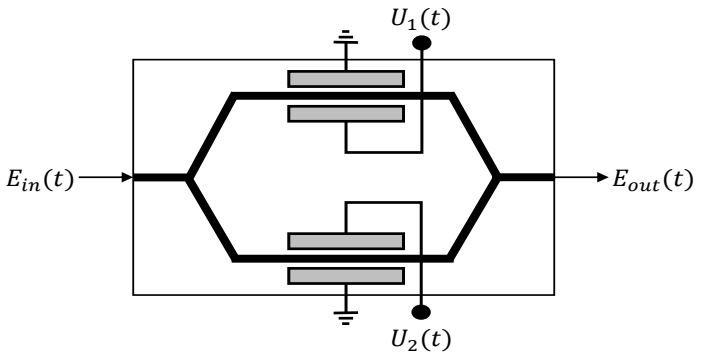


Figure 7.13: Mach-Zehnder Modulator

$$\frac{E_{out}(t)}{E_{in}(t)} = \frac{1}{2} \cdot (e^{j\phi_1(t)} + e^{j\phi_2(t)}) \tag{7.4}$$

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

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

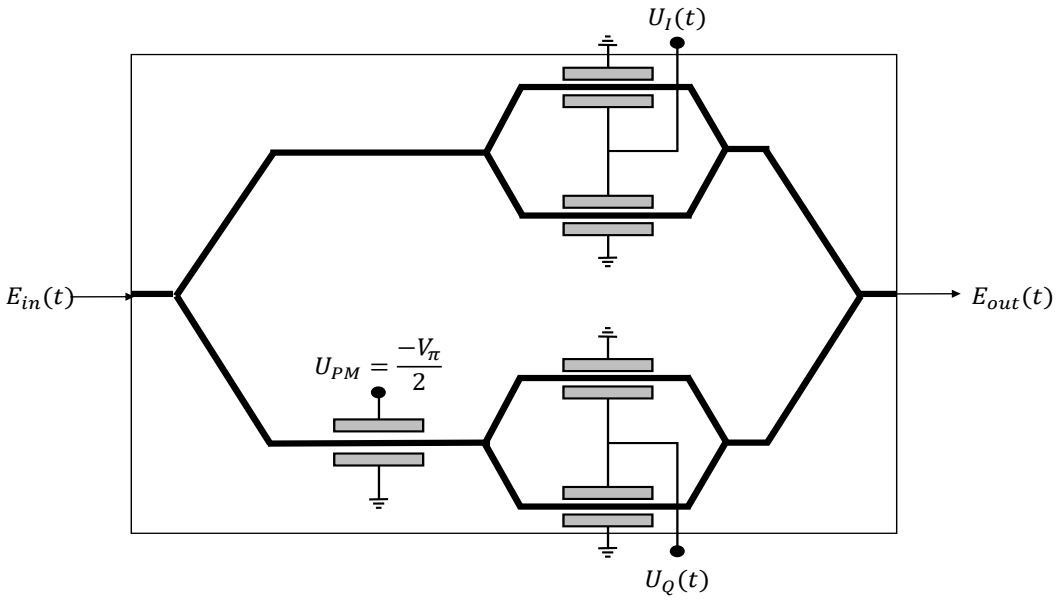


Figure 7.14: IQ Mach-Zehnder Modulator

function of the IQ MZM can be written as,

$$E_{out}(t) = \frac{1}{2}E_{in}(t)\left[cos(\frac{\pi U_I(t)}{2V_\pi}) + j \cdot cos\left(\frac{\pi U_Q(t)}{2V_\pi}\right)\right] \tag{7.5}$$

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

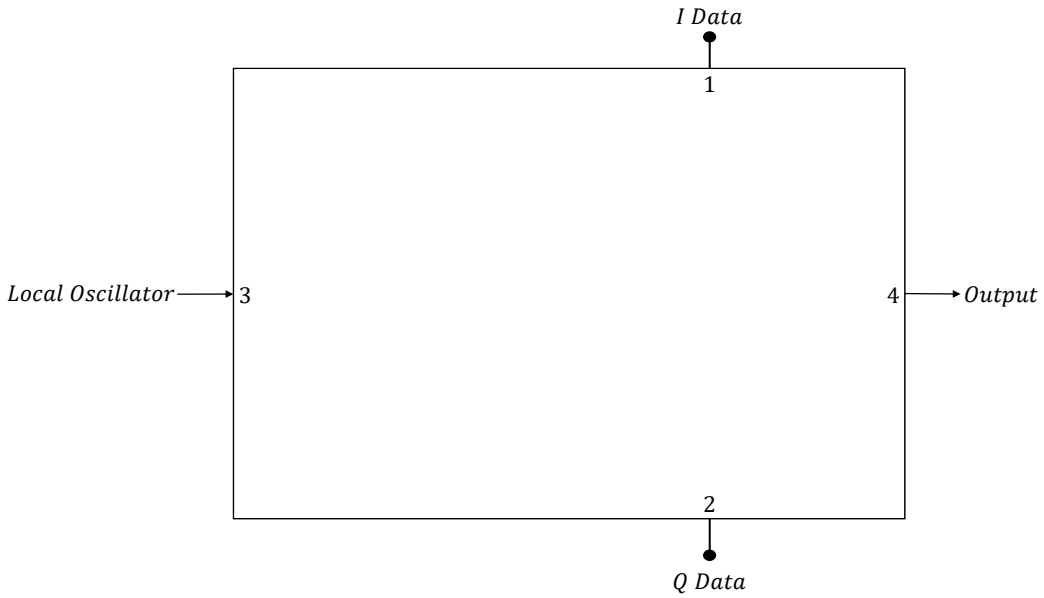


Figure 7.15: Simulation model of the IQ Mach-Zehnder Modulator

7.9 IIR Filter

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

Input Parameters

Name	Type	Default Value

Methods

IIR_Filter()

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

void initialize(void)

bool runBlock(void)

void setBCoeff(vector<double> newBCoeff)

void setACoeff(vector<double> newACoeff)

int getFilterOrder(void)

Input Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

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

Theoretical Description

Known Issues

7.10 Local Oscillator

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

Version 20180130

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

Input Parameters

Parameter	Type	Values	Default
opticalPower	double	any	$1e-3$
outputOpticalWavelength	double	any	$1550e-9$
outputOpticalFrequency	double	any	SPEED_OF_LIGHT / outputOpticalWavelength
phase	double	$\in \left[0, \frac{\pi}{2}\right]$	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 7.8: Binary source input parameters

Methods

LocalOscillator()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setSymbolPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setFrequency(double freq);

void setPhase(double lOscillatorPhase);

void setSignaltoNoiseRatio(double sNoiseRatio);

void setLaserLinewidth(double laserLinewidth);

void setLaserRIN(double laserRIN);

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1
Type: Optical signal

Version 20180828

7.11 Local Oscillator

Header File	:	local_oscillator.h
Source File	:	local_oscillator.cpp
Version	:	20180815 (Pedro Loureiro)

This block simulates a local oscillator, this means an electronic oscillator used to generate a signal with a constant power, initial phase and Wavelength, however we can simulate a non-ideal local oscillator by putting a non-zero value for laserLineWidth, frequencyMismatch or laserRIN. It will produce one output complex signal and it doesn't have input signals. By the figure 7.16 its possible to see the two local oscillators in the Block diagram.

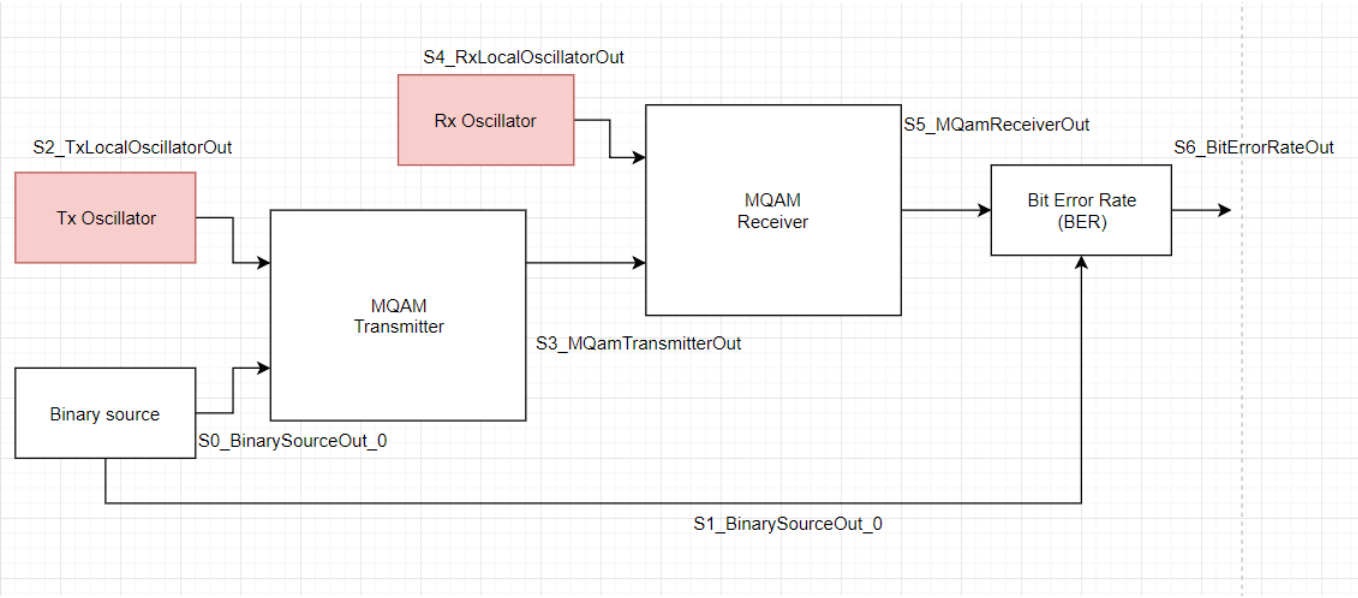


Figure 7.16: Block diagram of a transmission system

Input Signals

Number: 0

Output Signals

Number: 1

Type: The Laser emits an optical signal.

Input Parameters

Parameter	Type	Values	Default
opticalPower	double	any	$1e-3$
outputOpticalWavelength	double	any	$1550e-9$
outputOpticalFrequency	double	any	$\frac{SPEED\_OF\_LIGHT}{outputOpticalWavelength}$
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[1]	double	any	0.0
laserRIN[2]	double	any	0.0
frequencyMismatch	double	any	0.0

Table 7.9: Binary source input parameters

[1] Laser linewidth is the spectral linewidth of a laser beam.

[2] The relative intensity noise (RIN)is the power noise normalized to the average power level

Methods

1. Block (Laser) Declaration and Initialization
    - Laser(initializer-list<Signal *> InputSig, initializer-list<Signal *> OutputSig) : Block(InputSig, OutputSig)
    - void initialize(void)
    - bool runBlock(void)
2. Functions to set parameters
    - 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 setFrequencyMismatch(double fMismatch)
    - void setPhase(double lOscillatorPhase)
    - void setSignaltoNoiseRatio(double sNoiseRatio)
    - void setLaserLinewidth(double laserLinewidth)

- void setLaserRIN(double lRIN)

3. Functions to get parameters

- double getWavelength()
- double getFrequency()
- double getFrequenyMismatch()
- double const getPhase()
- double const getSignaltoNoiseRatio()
- double getLaserLinewidth()
- double getLaserRIN()

Examples

The first example is with the default values of the input parameters, therefore without noise, and 6dBm of power.

$$Power\_dB = 6dBm = -24dBW <=> Power = 10^{-2.4} = 0.0040$$

$$Out = \sqrt{\frac{Power + noise}{2}} e^{i*phase} = 0.044$$

, as we can see in the figure 7.17.

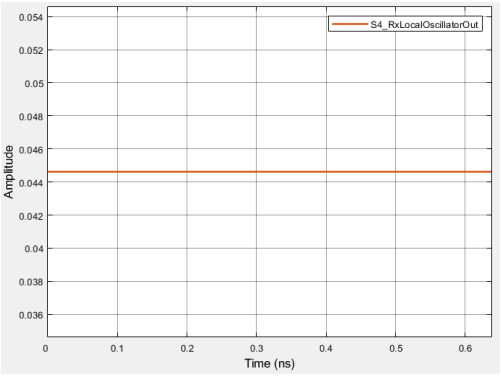
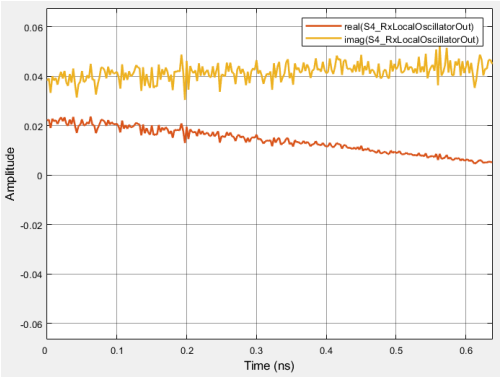
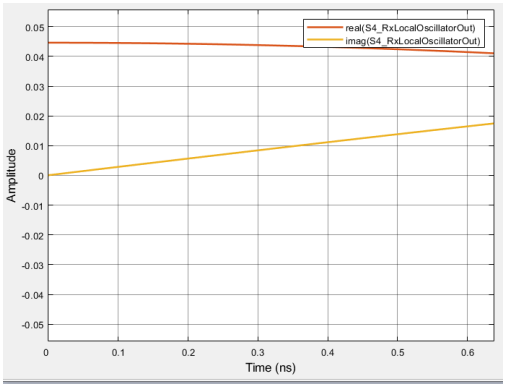


Figure 7.17: Example of the generated signal with no noise

In the following examples the frequencyMismatch, laserRIN, laserLineWidth, opticalPower and phase are changed.

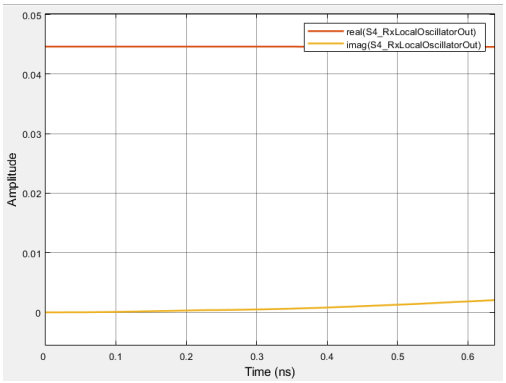


(a) Example of the generated signal with opticalPower=6dBm, phase=$\frac{pi}{3}$, laserRIN=5e^{-14} and laserLineWidth=10Hz

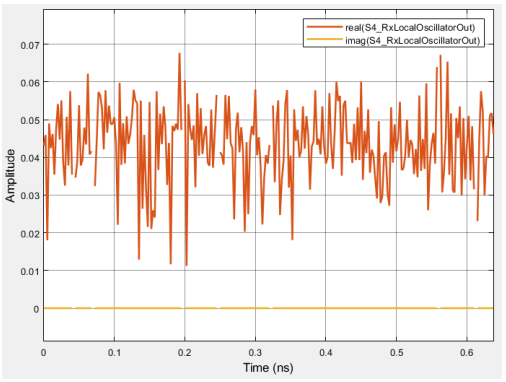


(b) Example of the generated signal with frequencyMismatch=100MHz

Figure 7.18

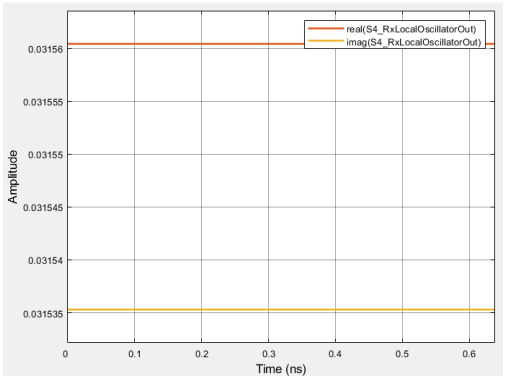


(a) Example of the generated signal with laserLineWidth=10Hz

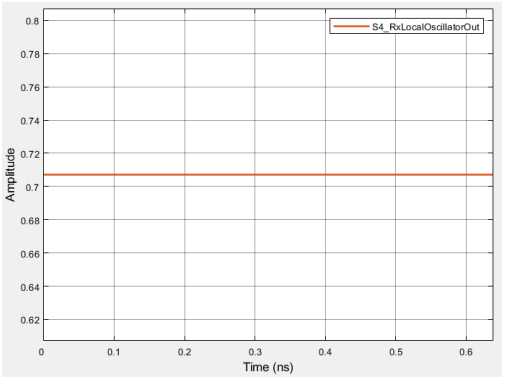


(b) Example of the generated signal with laserRIN=5e^{-13}

Figure 7.19



(a) Example of the generated signal with phase=$\frac{pi}{4}$



(b) Example of the generated signal with opticalPower=30dBm

Figure 7.20

Functional description

This block hasn't inputs and one output. It generates a complex signal:

$$Out = \sqrt{\frac{Power + Intnoise}{2}} e^{i*phase}$$

The phase depends on the initial phase (input parameter), on the phase Noise (depends on the laserLineWidth) and on the frequencyMismatch. The Intnoise depends on laserRIN. In the case of no noise the output signal will only be:

$$Out = \sqrt{\frac{Power}{2}} e^{i*Inphase}$$

Open Issues

Can't be possible to choose frequencyMismatch or laserLineWidth higher than outputOpticalFrequency.

Sugestions for future improvement

Be possible to see in frequency domain the signals in the visualizer.

7.12 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 7.10: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setM(int mValue);

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues);

Functional Description

In the case of m=4 this block atributes to each pair of bits a point in the I-Q space. The constellation used is defined by the *iqAmplitudes* vector. The constellation used in this case is ilustrated in figure 7.21.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

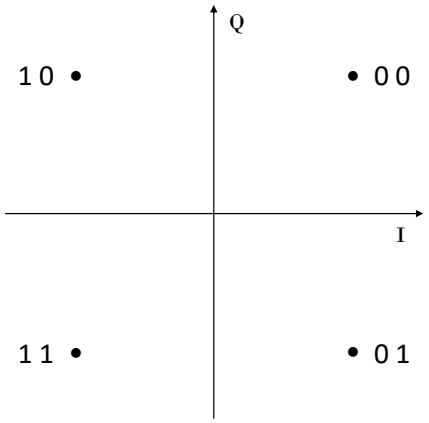


Figure 7.21: Constellation used to map the signal for m=4

Output Signals

Number : 2

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

Example

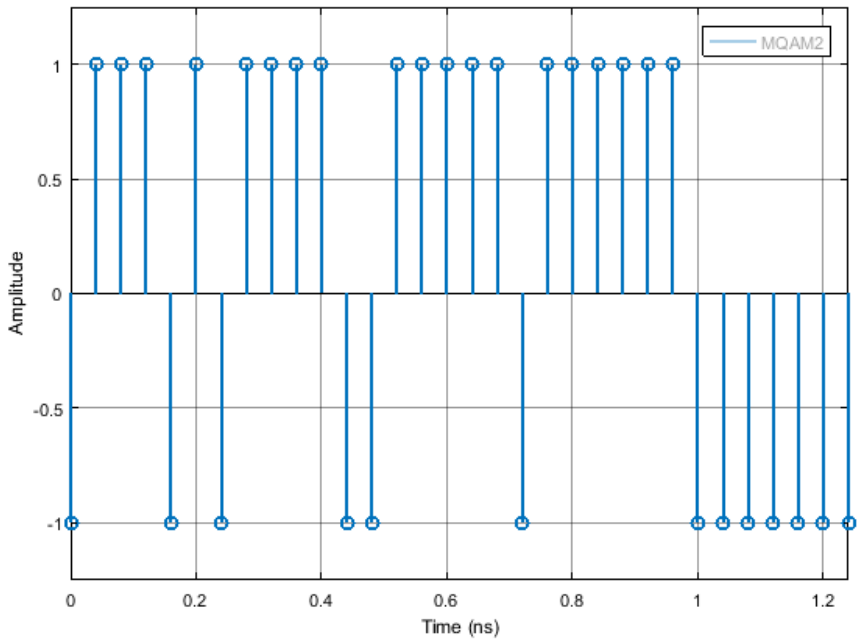


Figure 7.22: Example of the type of signal generated by this block for the initial binary signal 0100...

7.13 M-QAM Transmitter

Header File	:	m_qam_transmitter.h
Source File	:	m_qam_transmitter.cpp
Version	:	20180815 (*Pedro Loureiro*)

This block generates a MQAM optical signal. It also have two inputs the binary sequence and the optical signal from the laser. The binary signal will be used later to calculate the Bit Error Rate (BER). A schematic representation of this block is shown in figure 7.23.

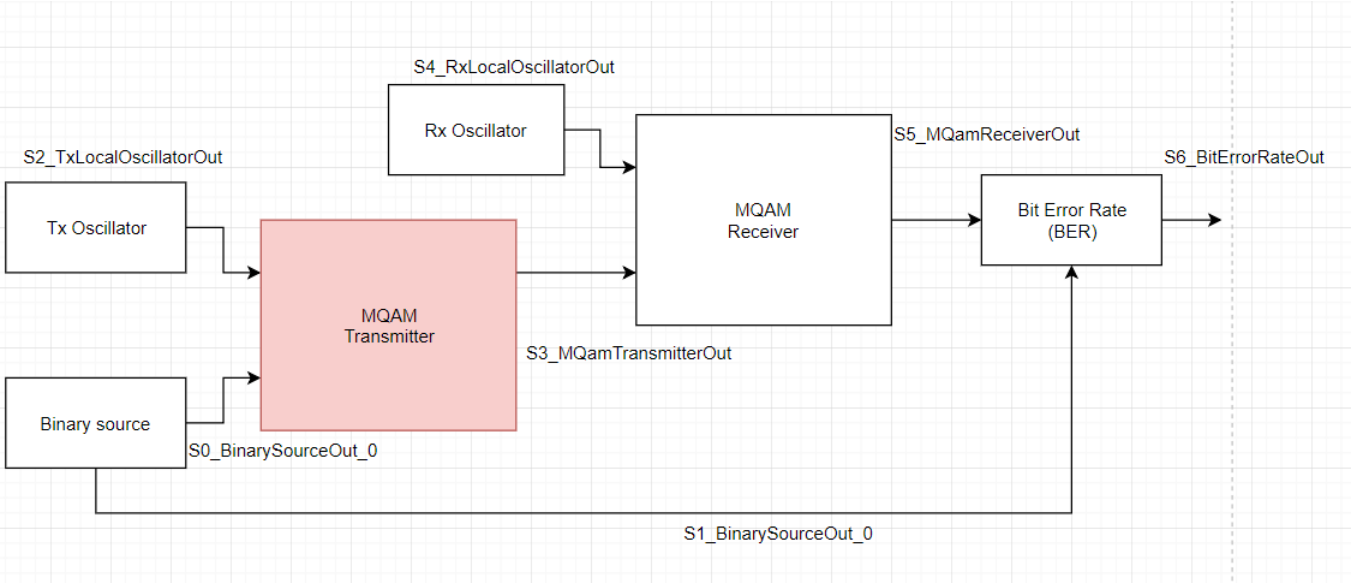


Figure 7.23: Basic configuration of the MQAM transmitter

Functional description

This block starts with a MQAM Mapper. In case of QPSK (M=4) the MQAM Mapper gives a point in the I-Q space (the constellation coding is gray) per 2 bits received. The next two blocks are Discrete to Continuous Time and Pulse Shaper. The Pulse Shaper block applies an electrical filter to the signal, the type of filter could be Raised Cosine, Gaussian, Square, Root Raised Cosine. This block ends with the IQ Modulator, which takes the amplitude and phase of the signal and generates a complex optical signal. Figure 7.24

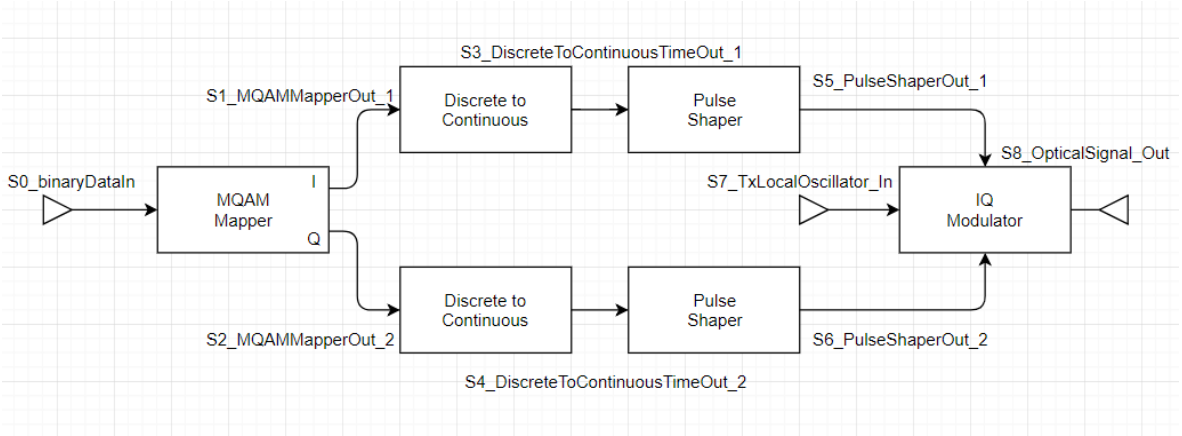


Figure 7.24: Schematic representation of the block MQAM transmitter.

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 7.11.

Parameter	Type	Values	Default
signalsFolderName	string	any	signals/SuperBlock _MQamTransmitter
logValue	bool	any	true
logFileName	string	any	SuperBlock_ _MQamTransmitter.txt
MQAM Mapper			
mValue	int	2^n	4
iqAmplitudesValues	vector<t_complex>	any	{ { 1, 1 },{ -1, 1 }, { -1, -1 },{ 1, -1 } }
fTime	bool	any	true
Discrete To Continuous Time			
nSamples PerSymbol	int	any	8
Pulse Shaper			
impResponse TimeLength	int	any	16
fType	pulse_shapper_filter_type	any[1]	RaisedCosine
rOffFactor	double	$\in [0, 1]$	0.9
pWidth	double	any	$5e^{-10}$
pFilterMode	bool	any	false

Table 7.11: Binary source input parameters

[1] Raised Cosine, Gaussian, Square, Root Raised Cosine

Methods

1. Block Declaration and Initialization
    - MQamTransmitter(initializer_list<Signal *> &inputSig, initializer_list<Signal *> &outputSig)
    - void initialize(void)
    - bool runBlock(void)
2. Functions to set parameters
    - void setIqAmplitudes(vector<vector<t_real» iqAmplitudesValues)
    - void setM(int mValue)
    - void setFirstTime(bool fTime)
    - void setNumberOfSamplesPerSymbol(int nSamplesPerSymbol)
    - void setImpulseResponseTimeLength_symbolPeriods(int impResponseTimeLength)
    - void setFilterType(pulse_shapper_filter_type fType)
    - void setRollOffFactor(double rOffFactor)
    - void setPulseWidth(double pWidth)
    - void setPassiveFilterMode(bool pFilterMode)
3. Functions to get parameters
    - bool getFirstTime()
    - int const getNumberOfSamplesPerSymbol(void)
    - int const getImpulseResponseTimeLength_symbolPeriods(void)
    - pulse_shapper_filter_type const getFilterType(void)
    - double const getRollOffFactor()
    - double const getPulseWidth()
    - bool const getPassiveFilterMode()

Input Signals

Number: 2

Type: 1 optical signal from the local oscillator and 1 binary from the binary source

Output Signals

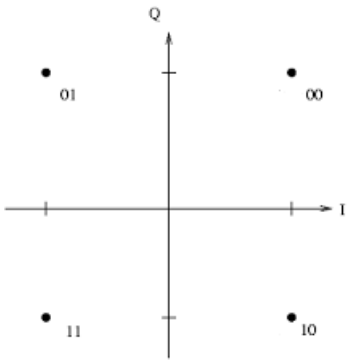
Number: 1 optical

Type: Optical signal

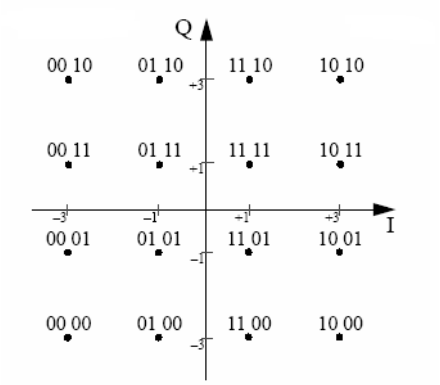
Example

Cardinality of the constellation

In order to test the impact of the Cardinality of the constellation will be analysed the BER's of QPSK (4QAM) and of 16QAM, the constellations are in the figures **??** and 7.25b.



(a) Constellation of 4QAM



(b) Constellation of 16QAM

Figure 7.25

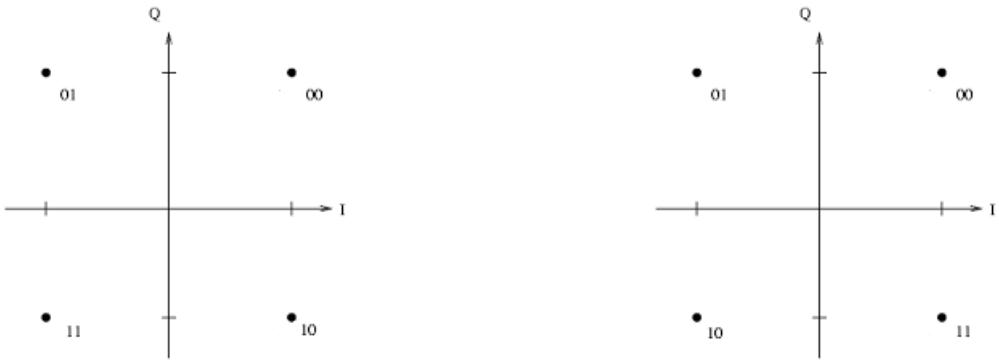
$$BER4QAM = 0$$

$$BER16QAM = 0.497487$$

We observe that the 16QAM has 4 different amplitudes and the 4QAM only 2 amplitudes per symbol. Also the BER from the 16QAM constellation is greater than the 4QAM. The reason to the BER of 16QAM to be higher is the 16QAM constellation has many more symbols that are surrounded by neighboring symbols than the 4QAM constellation, so they are much more subject to error, although equal noise.

Constellation coding

With the objective to analyze the impact of the constellation coding will be tested the BER's with gray and without gray, the constellations are in the figures **??** and 7.26b.



(a) Constellation of 4QAM with gray coding (b) Constellation of 4QAM without gray coding

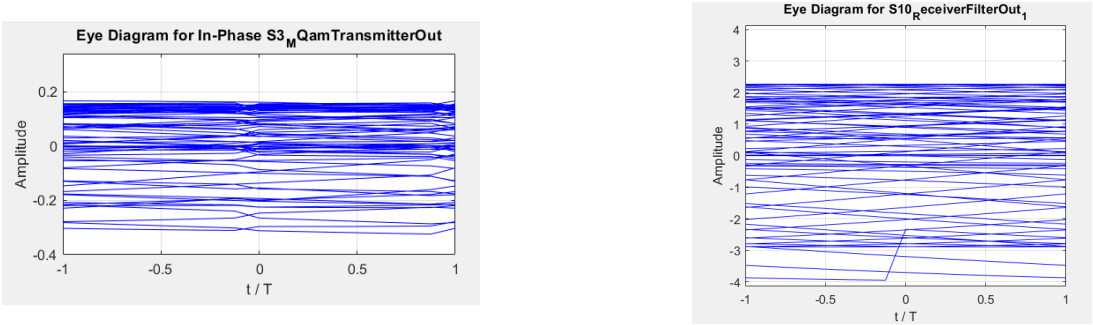
Figure 7.26

$$BER4QAMgray = 0$$

$$BER4QAMwithoutgray = 0.247881$$

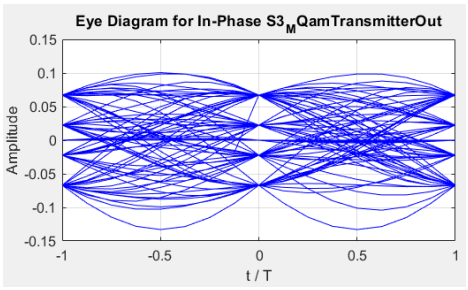
The BER of codification with gray is lower than without gray because in gray codification the neighboring symbols only have 1 bit different while the codification without gray has neighboring symbols with 2 different bits. So, in the case of a wrong symbol, could have two wrong bits.

Pulse shape

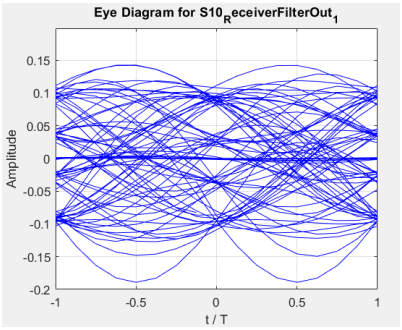


(a) Eye diagram 16QAM transmission with fType=Gaussian (b) Eye diagram 16QAM reception with fType=Gaussian

Figure 7.27

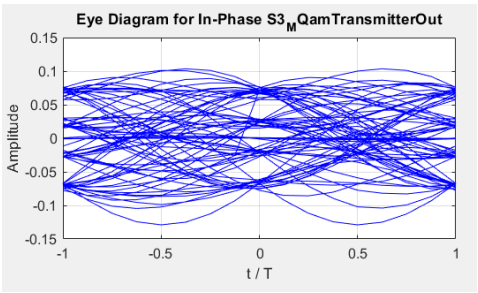


(a) Eye diagram 16QAM transmission with fType=RaisedCosine

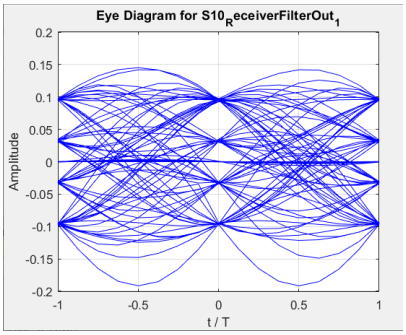


(b) Eye diagram 16QAM reception with fType=RaisedCosine

Figure 7.28



(a) Eye diagram 16QAM transmission with fType=RootRaisedCosine



(b) Eye diagram 16QAM reception with fType=RootRaisedCosine

Figure 7.29

By the figures 7.27, 7.28 and 7.29 the worst pulse shape was the Gaussian. The differences between the Root Raise Cosine and the Raise Cosine are, the Raise Cosine in the transmitter was Inter Symbol Interference (ISI) equal to 0, but in the receiver the ISI wasn't 0. The Root Raise Cosine has ISI equal to 0 in the receiver and different to 0 in the transmitter. The Square canÂ´t be simulate because it gives an error whenever it compiles.

7.13.1 Open Issues

In the pulse shaper block if the filter type is Square it gives an error whenever it compiles.

If we try to change the cardinality of the constellation to 16 doesn't happen nothing the only way is to add a line for when the cardinality is 16.

7.14 Netxpto

Header File	:	netxpto.h
	:	netxpto_20180118.h
	:	netxpto_20180418.h
Source File	:	netxpto.cpp
	:	netxpto_20180118.cpp
	:	netxpto_20180418.cpp

The netxpto files define and implement the major entities of the simulator.
Namely the signal value possible types

Signal Value Type	Data Range
BinaryValue	{0,1}
IntegerValue	$\mathbb{Z}$
RealValue	$\mathbb{R}$
ComplexValue	$\mathbb{C}$
ComplexValueXY	$(\mathbb{C}, \mathbb{C})$
PhotonValue	
PhotonValueMP	
PhotonValueMPXY	
Message	

7.14.1 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.

7.14.2 Version 20180418

Adds the possibility to include the parameters from an external file.

Sugestions for future improvement

7.15 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 7.30 shows a schematic representation of this block.

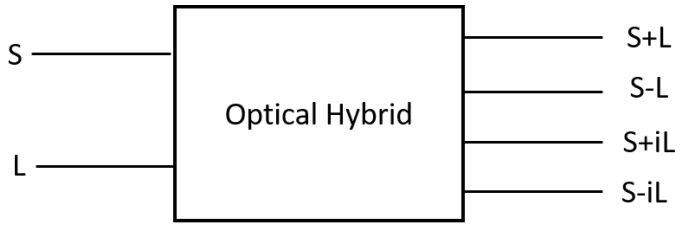


Figure 7.30: Schematic representation of an optical hybrid.

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 7.12: 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

7.16 Photodiode pair

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

This block simulates a block of two photodiodes assembled like in figure 7.33. 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.

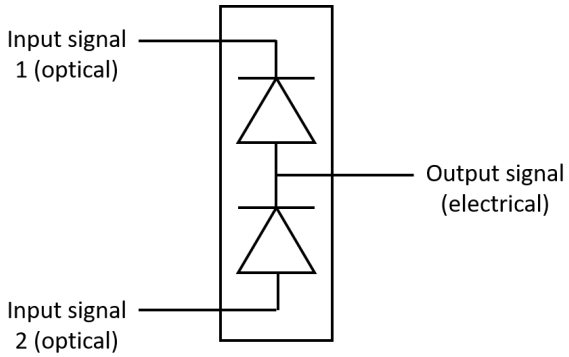


Figure 7.31: Schematic representation of the physical equivalent of the photodiode code block.

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

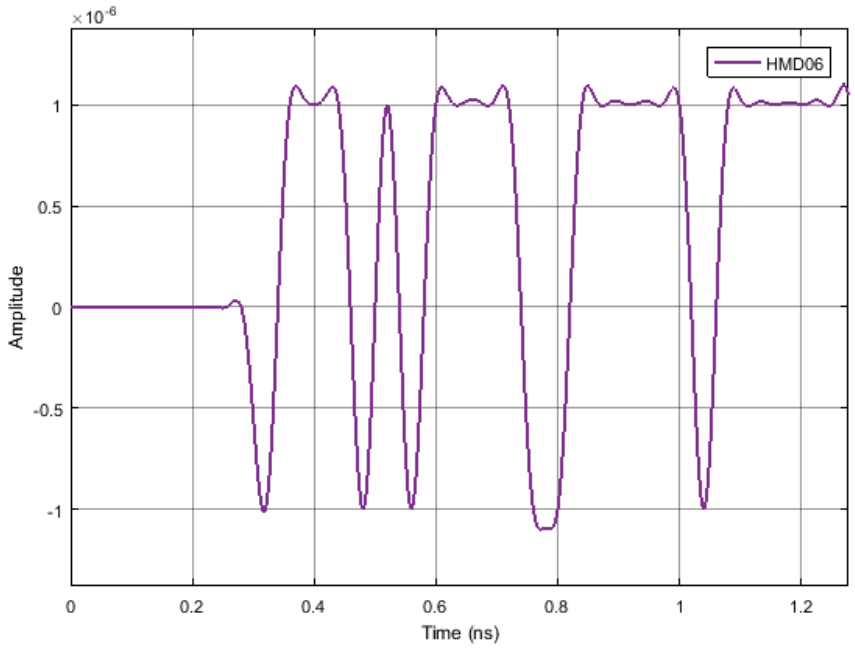


Figure 7.32: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

7.17 Photodiode pair

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

This block simulates a block of two photodiodes assembled like in figure 7.33. 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.

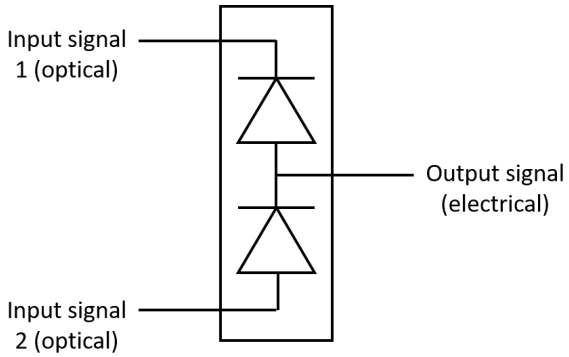


Figure 7.33: Schematic representation of the physical equivalent of the photodiode code block.

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

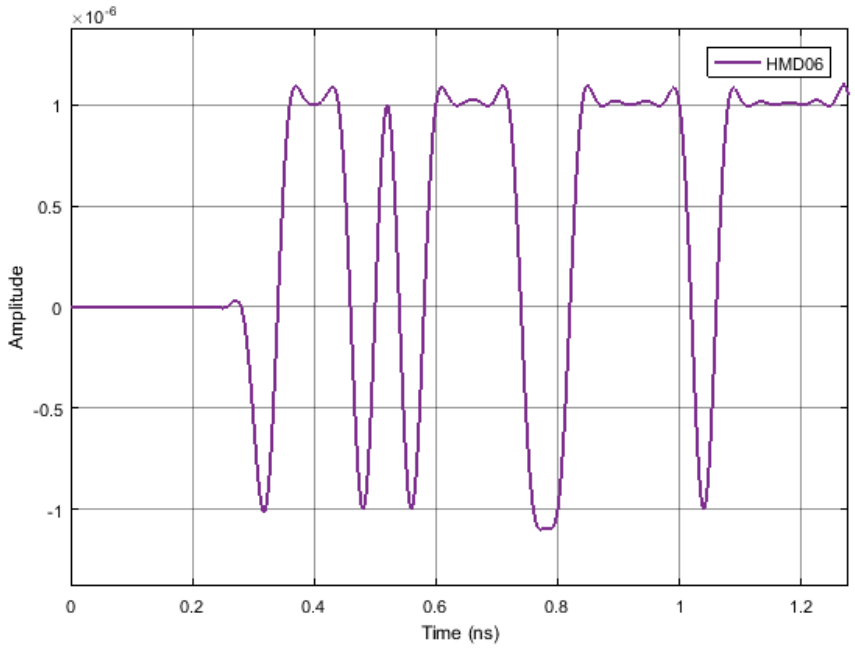


Figure 7.34: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

7.18 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 7.13: 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

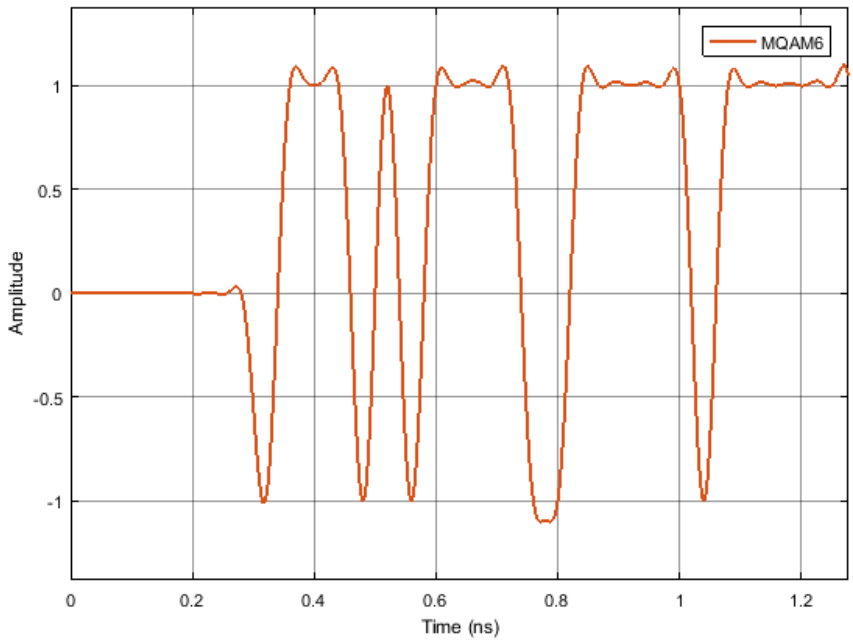


Figure 7.35: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

7.19 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 7.14: 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

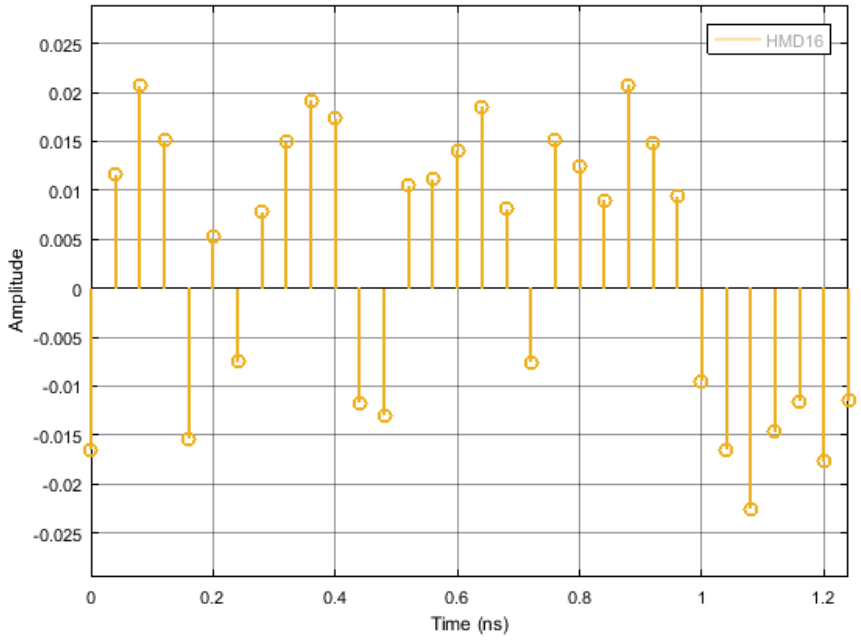


Figure 7.36: Example of the output signal of the sampler

Sugestions for future improvement

7.20 Sink

Header File	:	sink_*.h
Source File	:	sink_*.cpp
Version	:	20180523 (André Mourato)

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

Input Parameters

Parameter	Type	Values	Default
numberOfSamples	long int	any	-1
displayNumberOfSamples	bool	true/false	true

Table 7.15: Sink input parameters

Methods

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

bool runBlock(void)

void setAsciiFilePath(string newName)

string getAsciiFilePath()

void setNumberOfBitsToSkipBeforeSave(long int newValue)

long int getNumberOfBitsToSkipBeforeSave()

void setNumberOfBytesToSaveInFile(long int newValue)

long int getNumberOfBytesToSaveInFile()

void setNumberOfSamples(long int nOfSamples)

long int getNumberOfSamples const()

void setDisplayNumberOfSamples(bool opt)

bool getDisplayNumberOfSamples const()

Functional Description

The Sink block discards all elements contained in the signal passed as input. After being executed the input signal's buffer will be empty.

7.21 Super Block

Input Parameters

- bool saveInternalSignals{ false };

Methods

1. **SuperBlock(vector<Signal *> &inputSignal, vector<Signal *> &outputSignal) :Block(inputSignal, outputSignal){ setSaveInternalSignals(false); }**
2. **void initialize(void)** It starts by initializing each block of the super block. Then, it writes the header of each block using the information (ie sampling period, symbol period and first value to be saved) related with the information contained in the outputSignals of each block. Finally, it sets the symbol and sampling period of each output of the super block based on the information contained in the output of the last block of the super block with the same index.
3. **bool runBlock(void)** It starts to run each block od the super block. Then, for each outputSignal of the super block it tests the number of samples in the output signal of the last block in the super block (*ready()* and also the space in the output signal of the super block (*space()*). It gets the value type if the output signal of the last block in the super block and according with that it gets the value and put the value from the output signal of the last block into an output signal of the super block.
4. **void terminate(void)** It terminates each block of the super block and closes the output signal of the last block.

7.21.0.1 Set Methods

- void setModuleBlocks(vector<Block*> mBlocks){ moduleBlocks = mBlocks; };
- void setSaveInternalSignals(bool sSignals);
- bool const getSaveInternalSignals(void){ return saveInternalSignals; };

7.22 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 7.16: 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

7.23 TI Amplifier

This block has one input signal and one output signal both corresponding to electrical signals. The output signal corresponds to the amplification of the input signal with added noise.

Input Parameters

- amplification{1e6}
- noiseamp{ 1e-4 }

Methods

TIAmplifier()

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

void initialize(void)

bool runBlock(void)

void setAmpplification(`t_real` Amplification)

void setNoiseAmpplitude(`t_real` NoiseAmplitude)

Functional description

The output signal is the product of the input signal with the parameter *amplification* plus a component that corresponds to the noise introduced by the amplification of the signal.

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

7.24 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 7.17: White noise input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNoiseSpectralDensity(double SpectralDensity) spectralDensity = SpectralDensity;

double const getNoiseSpectralDensity(void) return spectralDensity;

void setSeedType(SeedType sType) seedType = sType; ;

SeedType const getSeedType(void) return seedType; ;

void setSeed(int newSeed) seed = newSeed;

int getSeed(void) return seed;

Functional description

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

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

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

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

The noise is 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.

$$N = n_0 B = n_0 \frac{2}{T} \tag{7.6}$$

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

Chapter 8

Mathlab Tools

8.1 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.

8.1.1 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.

8.1.2 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:

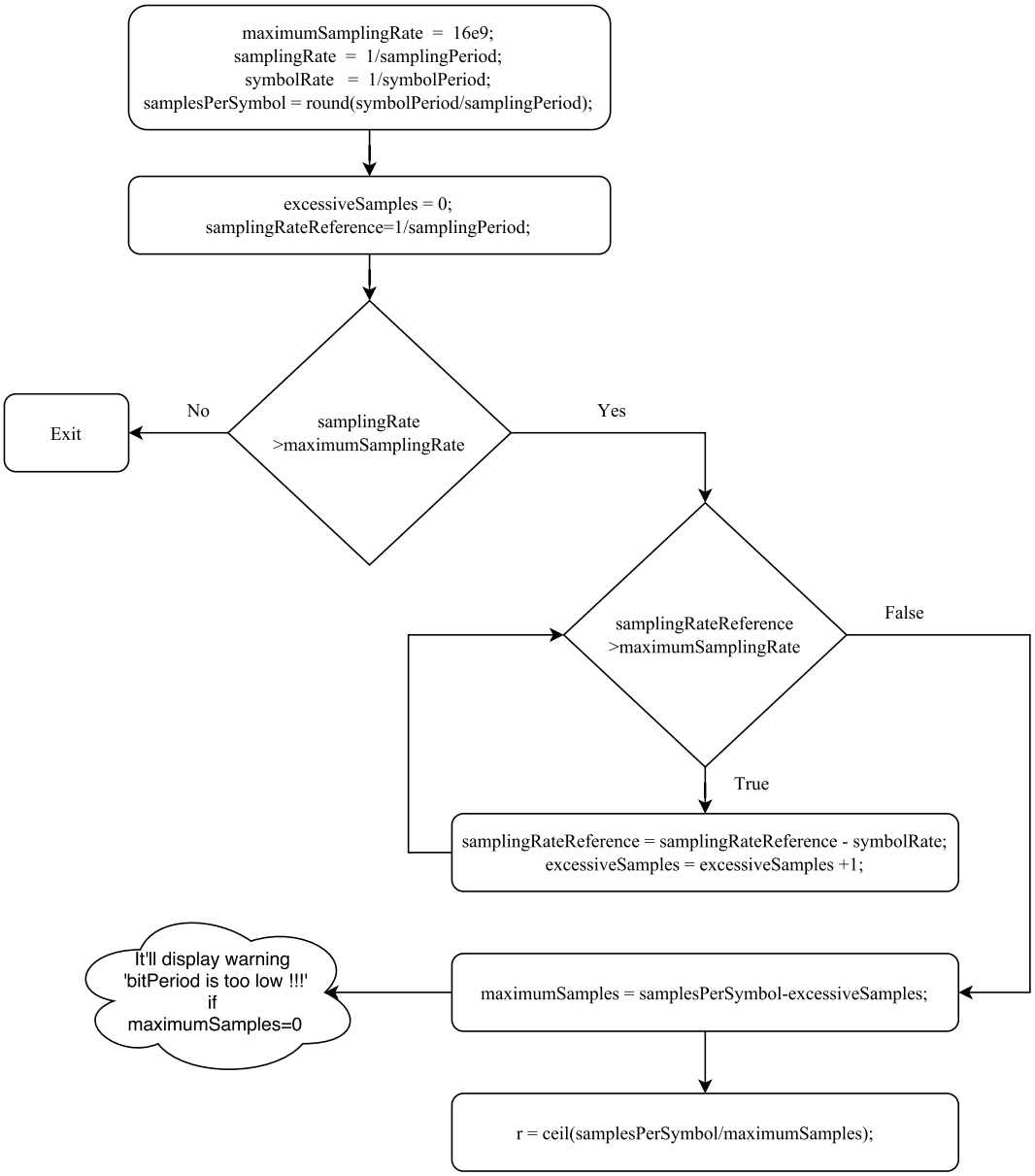


Figure 8.1: Flowchart to calculate decimation factor

»»»»> Develop.Romil

8.1.3 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).

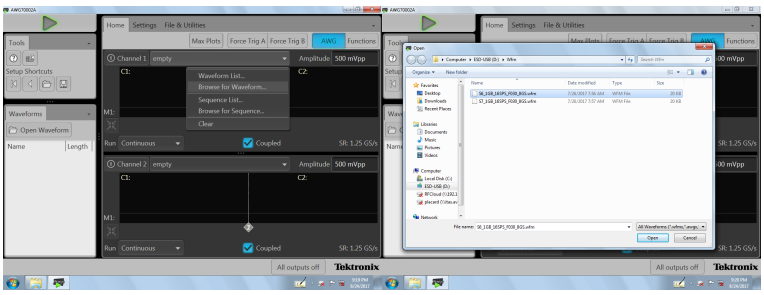


Figure 8.2: Selecting your waveform in the AWG

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).

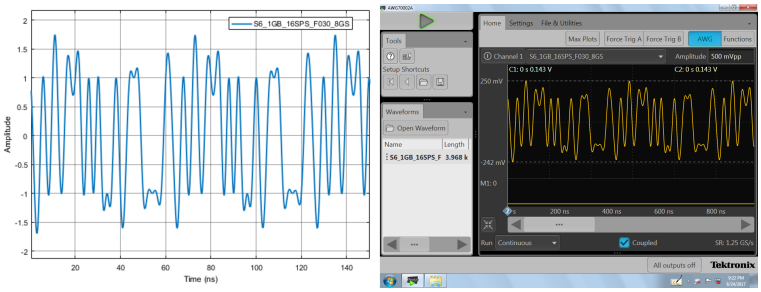


Figure 8.3: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

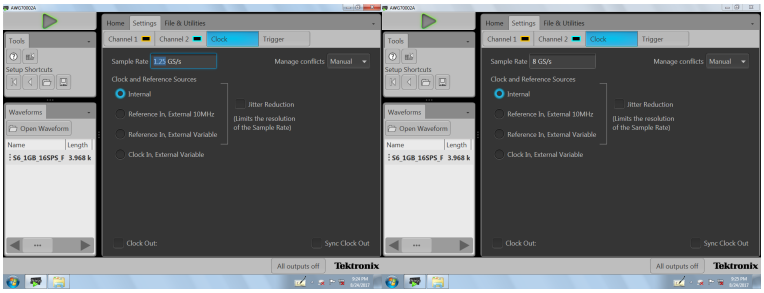


Figure 8.4: Configuring the right sampling rate

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

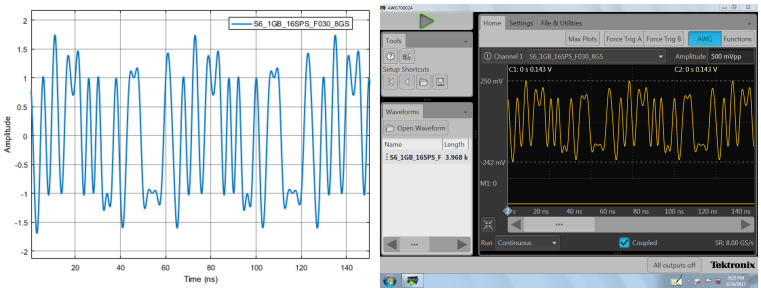


Figure 8.5: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

8.2 Polarization Analysis Signals

Students Name	:	Mariana Ramos
Goal	:	Analyse simulation Photon Stream signals into Stokes space using plot in Poincare sphere and Autocorrelation calculation and plot.
Version	:	jonesToStokes_20180614.m

This section shows some matlab functions to analyse photon stream signals from simulation.

8.2.1 jonesToStokes.m

Structure of a function

[] = jonesToStokes(fname,deltaP, filename,NumberOfSamples);

Inputs

fname: Input filename of the signal (*.sgn) you want to convert. It must be a photon stream signal (Type: PhotonStreamXY).

deltaP: Value of delta P (polarization linewidth which is a parameter that depends on optical fibre installation) used to perform the simulation.

filename: Input filename which contains the stokes parameters resulted from simulation (*.txt).

NumberOfSamples: Number of Samples to plot in Poincare sphere.

Outputs

Two graphics are plotted from the photon stream input signal:

Histogram of parameters $\vec{\alpha}$: Three plots of each α_i which were used as input parameters of the random matrix used to model the SOP drift block.

Poincare sphere: A plot of the time evolution of the photon stream into the Poincare sphere.

Functional Description

This matlab function converts the input signal (*.sgn) which is represented in Jones space in a signal represented in Stokes Space which allows us to plot the signal time evolution in Poincare sphere. Furthermore, the $\vec{\alpha}$ parameters obtained from simulation are also plotted in order to be sure that they were generated correctly by following a normal distribution with mean 0 and a standard deviation depending on these parameters values.

Students Name	:	Mariana Ramos
Version	:	ACF_20180614.m

8.2.2 ACF.m

Structure of a function

[] = ACF(fname,deltaP,N)

Inputs

fname: Input filename of the signal (*.sgn) you want to convert. It must be a photon stream signal (Type: PhotonStreamXY).

deltaP: Value of delta P (polarization linewidth which is a parameter that depends on optical fibre installation) used to perform the simulation.

N: Number of Samples used to plot the function of autocorrelation.

Outputs

This matlab function has two outputs:

.txt file with values of the autocorrelation calculated with data from simulation: A .txt file with the autocorrelation of the photon stream signal which inputs the function for N samples which is also an input of the function-.

ACF plot: A plot with numerical ACF calculated from simulation data and with the theoretical ACF calculated based on the value of Δp inserted as an input of the function.

Functional Description

This matlab function calculates the autocorrelation in time domain of the photon stream input signal as well as the theoretical autocorrelation based on the value of Δp inserted as an input of the function which must be the same used in simulation. This function outputs a txt file with the numerical ACF and plots a graphic with both theoretical and numerical autocorrelation.

Students Name	:	Mariana Ramos
Version	:	plotPhotonStream_20180102.m

8.2.3 plotPhotonStream_20180102.m

Structure of a function

[mag_x, mag_y, phase_dif, h] = plotPhotonStream_20180102(fname, opt, h)

Inputs

fname: Input filename of the signal (*.sgn) you want to convert. It must be a photon stream signal (Type: PhotonStreamXY).

opt: If opt equals 0, or no opt, we plot the absolute value of X and Y and the phase difference.If opt equals 1, we plot the amplitude of X and Y, this is only possible when X and Y are real values.

h: Number of the figure to plot.

Outputs

This matlab function has four outputs:

mag_x: Magnitude of the complex number X.

mag_y: Magnitude of the complex number Y.

phase_dif: Difference phase between the two complex numbers X and Y.

h: A plot of magnitude and phase difference of the two complex numbers depending on the choice of the input parameter **opt**.

Functional Description

This matlab function plots the magnitude of the two components of the input photon stream signal (*.sgn) and the phase difference between both components. This function allows us to visualize the photon stream signal over time.

Students Name	:	Mariana Ramos
Version	:	plot_sphere.m

8.2.4 plot_sphere.m

This function accepts the three stokes parameters S_1, S_2 and S_3. This function supports the other functions in this section allowing the plot of these parameters in Poincare sphere.

Chapter 9

Algorithms

9.1 Fast Fourier Transform

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

9.1.0.1 Algorithm

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

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \ e^{i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{9.1}$$

Similarly, for IFFT,

$$x_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X_k \ e^{-i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{9.2}$$

From equations 9.1 and 9.2, 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,

$$y = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x \ e^{m\ i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{9.3}$$

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.

9.1.0.2 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

$$x = ifft(y)$$

9.1.0.3 Flowchart

The figure 9.1 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 [**Rao2010a**]. 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 [**Chu2000**].

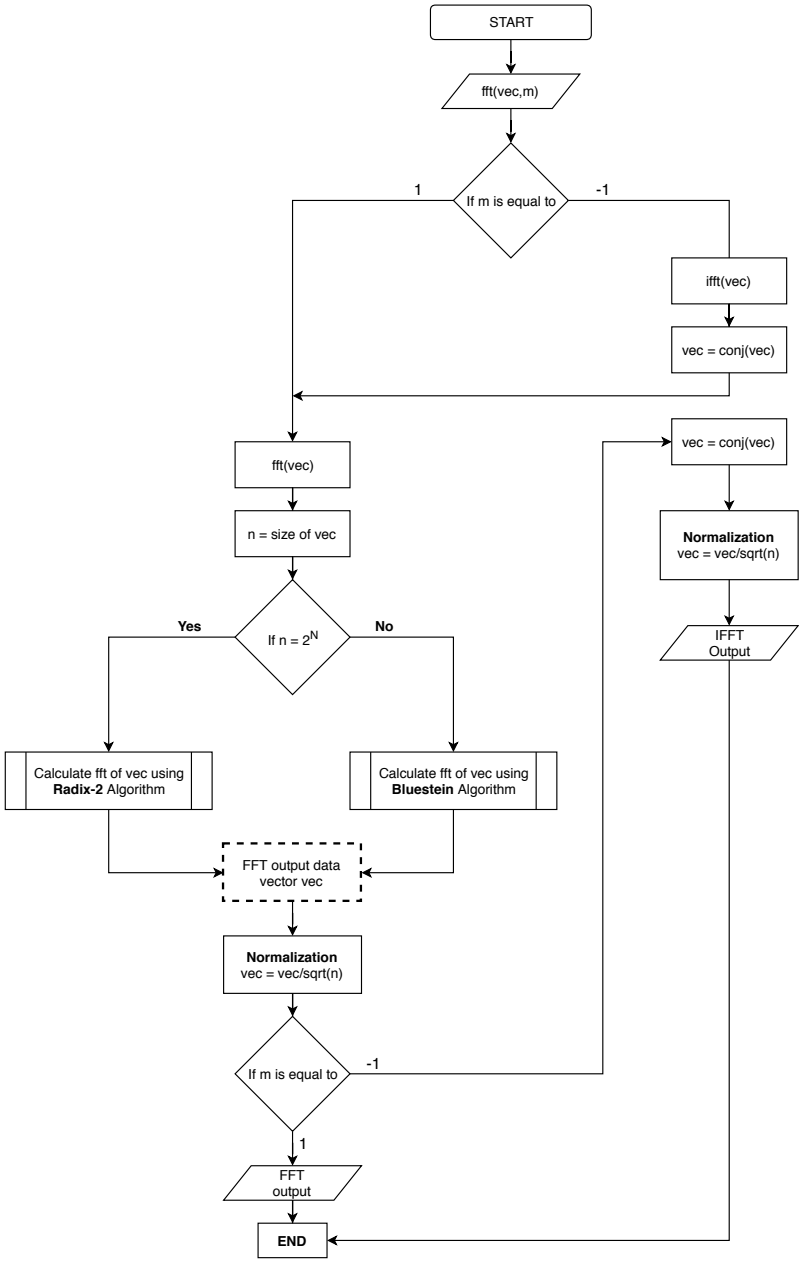


Figure 9.1: Top level architecture of FFT algorithm

9.1.0.4 Radix-2 algorithm

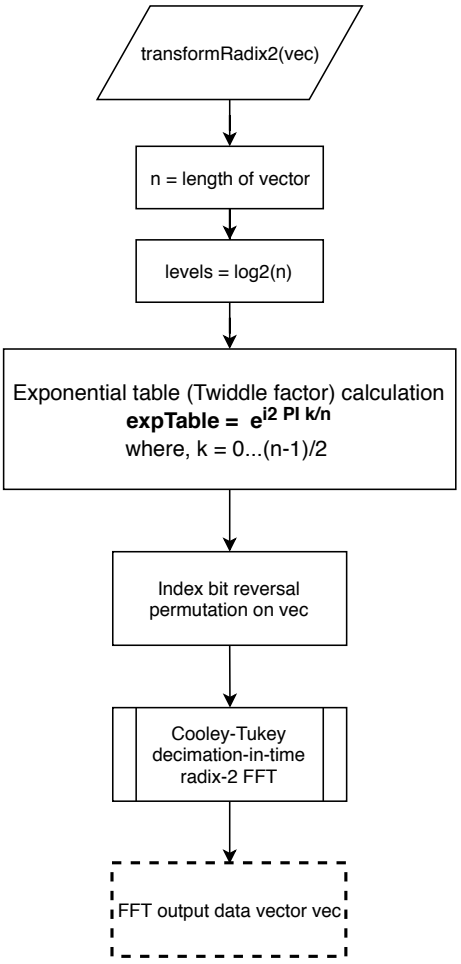


Figure 9.2: Radix-2 algorithm

9.1.0.5 Cooley-Tukey algorithm

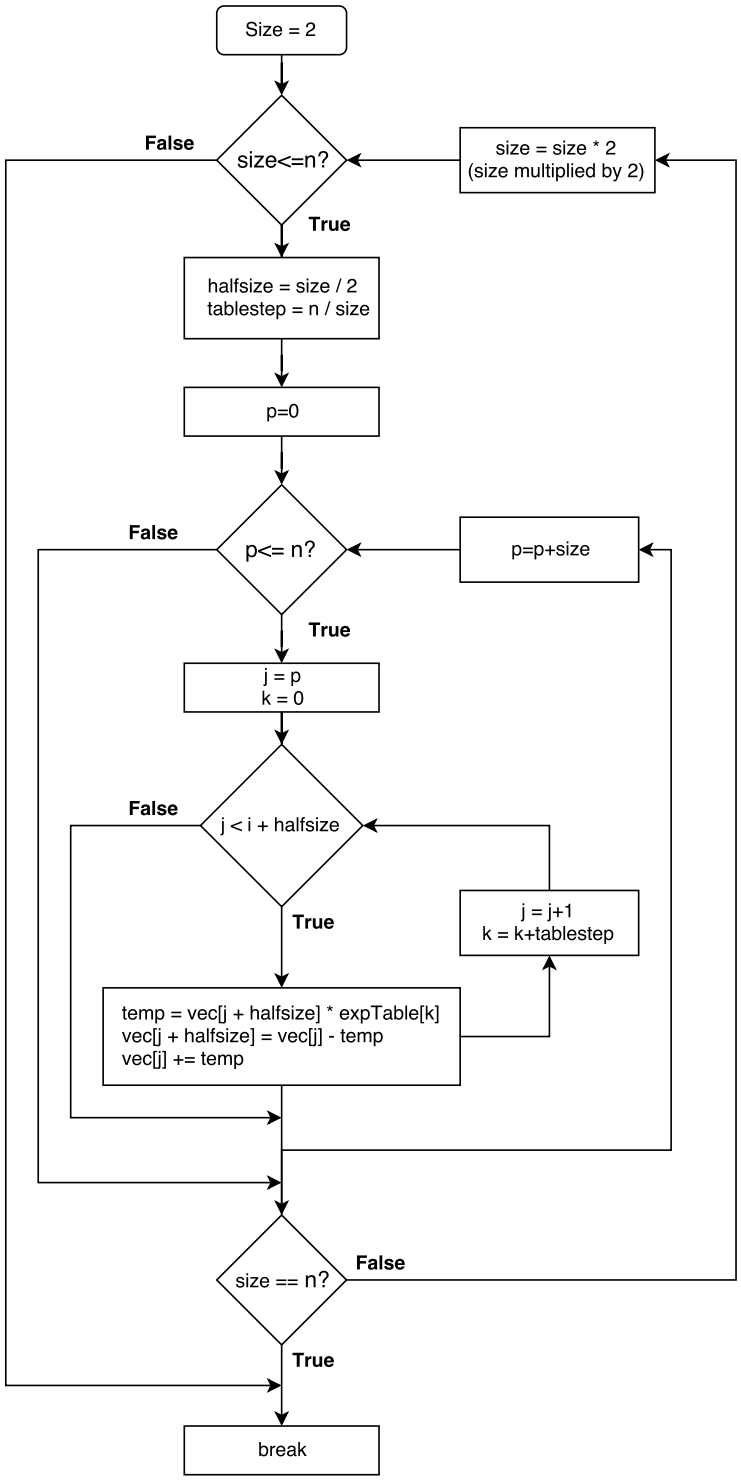


Figure 9.3: Cooley-Tukey algorithm

9.1.0.6 Bluestein algorithm

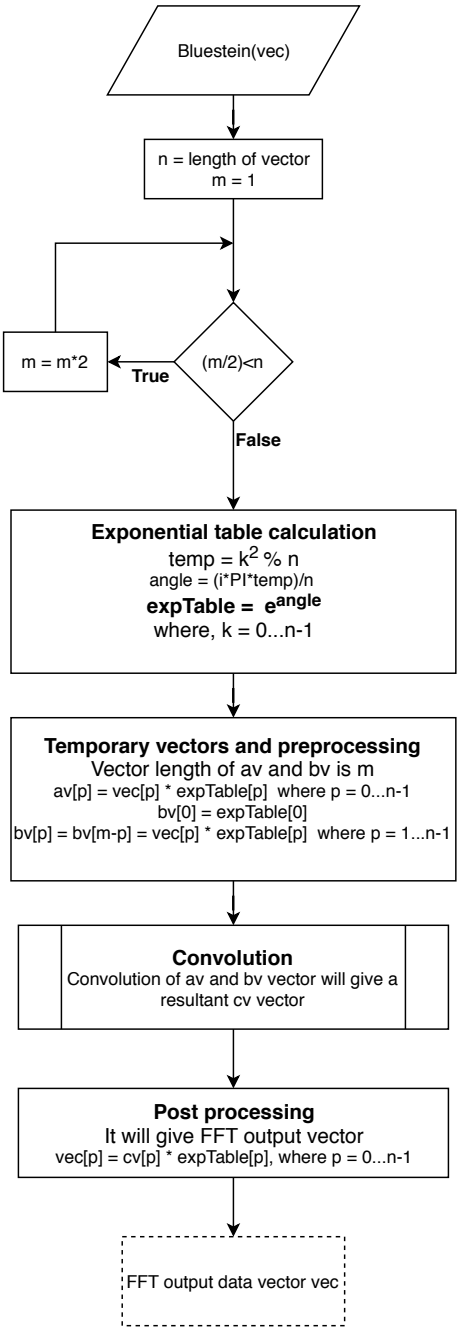


Figure 9.4: Bluestein algorithm

9.1.0.7 Convolution algorithm

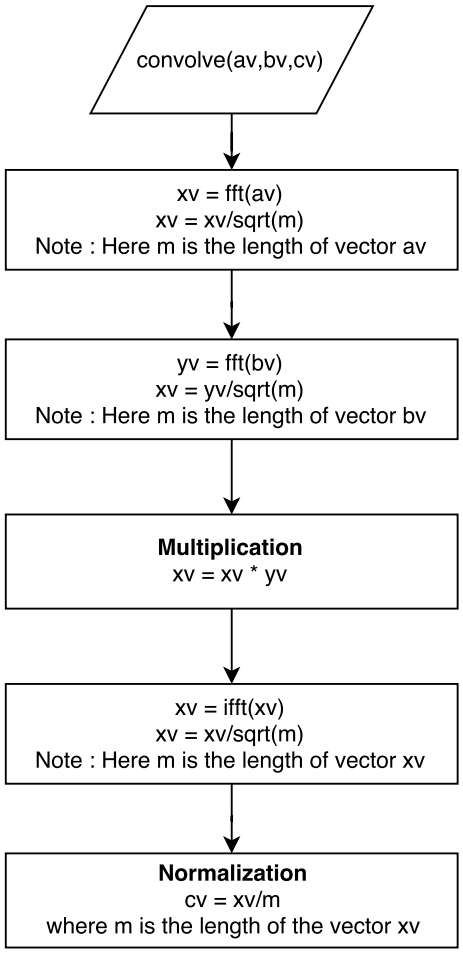


Figure 9.5: Circular convolution algorithm

9.1.0.8 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 9.1: 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.
```

```
44
    ////////////////////////////////// Section 2 ////////////////////////////////////
46  ////////////////////////////////// Calculate FFT ////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
48
    vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
50  vector <complex<double>> fourierTransformed;
    vector <double> re(inTimeDomain.size());
52  vector <double> im(inTimeDomain.size());

54  for (unsigned int i = 0; i < inTimeDomain.size(); i++)
    {
56    re[i] = inTimeDomain[i]; // Real data of the signal
    }
58
    // Next, Real and Imaginary vector to complex vector conversion
60  inTimeDomainComplex = reImVect2ComplexVector(re, im);

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

68  ////////////////////////////////// Section 3 ////////////////////////////////////
    //////// Save FFT calculated data (export "frequency_function.txt" ) /////////
70  ////////////////////////////////////////////////////////////////////////////////
    ofstream outFile;
72  complex<double> outFileData;
    outFile.open("frequency_function.txt");
74  outFile.precision(15);
    for (unsigned int i = 0; i <fourierTransformed.size(); i++){
76    outFile << fourierTransformed[i].real() << endl;
      outFile << fourierTransformed[i].imag() << endl;
78  }
    outFile.close();
80
    ////////////////////////////////// Section 4 ////////////////////////////////////
82  ////////////////////////////////// Display Section //////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
84  for (unsigned int i = 0; i < fourierTransformed.size(); i++){
      cout << fourierTransformed[i] << endl; // Display all FFT calculated data
86  }
    cout << "\n\nTime elapsed to calculate FFT : " << elapsed_secs << " seconds"<< endl;
88  cout <<"\nTotal length of of data :"<< count << endl;
    getchar();
90  return 0;
}
```

Listing 9.2: 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.

9.1.0.9 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.

9.1.0.10 1. Signal with two sinusoids and random noise

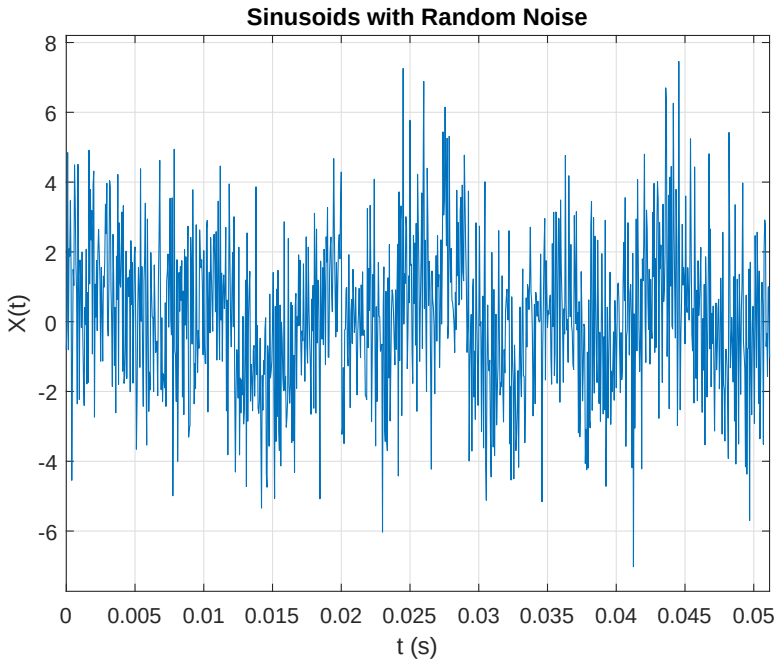


Figure 9.6: Random noise and two sinusoids

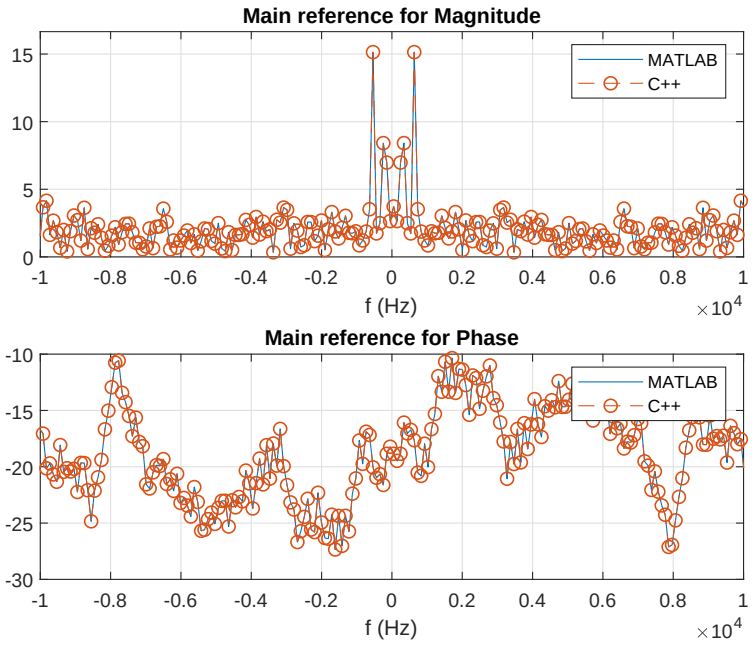


Figure 9.7: MATLAB and C++ comparison

9.1.0.11 2. Sinusoid with an exponent

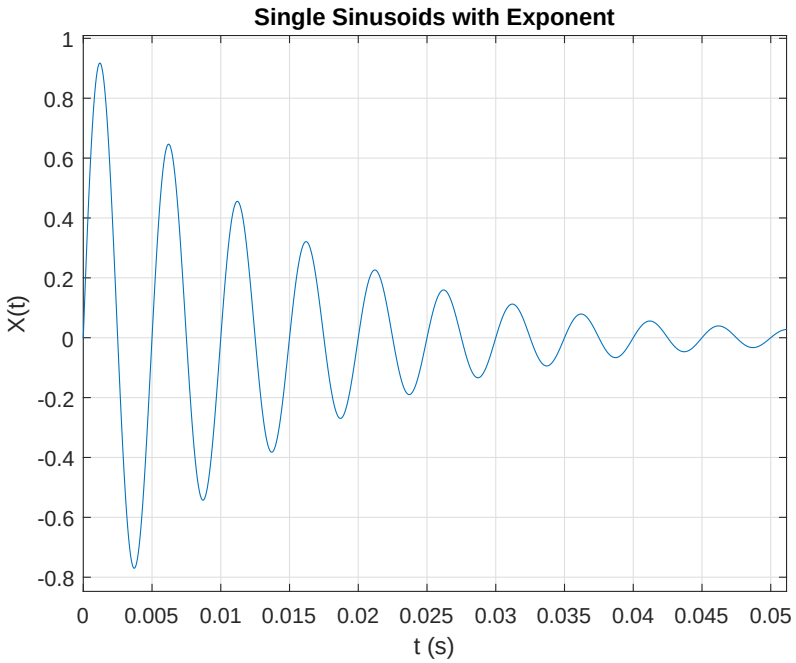


Figure 9.8: Sinusoids with exponent

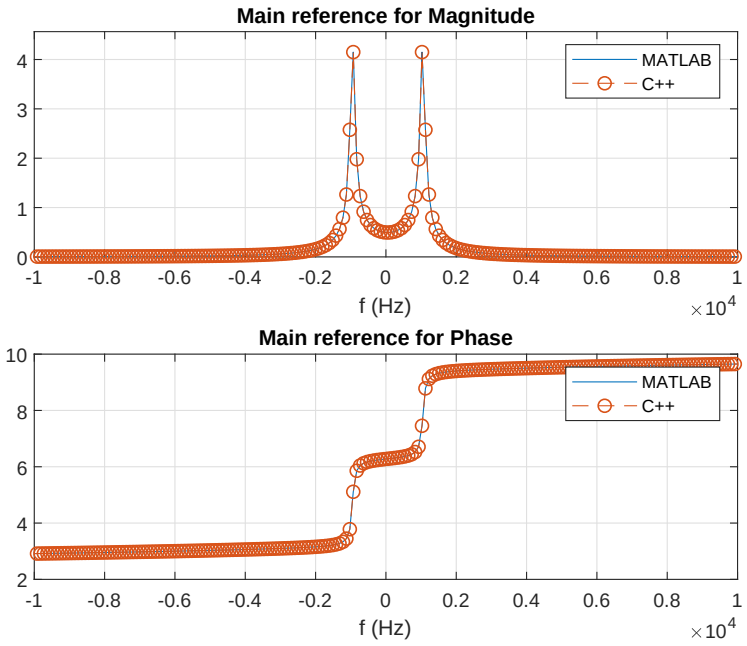


Figure 9.9: MATLAB and C++ comparison

9.1.0.12 3. Mixed signal

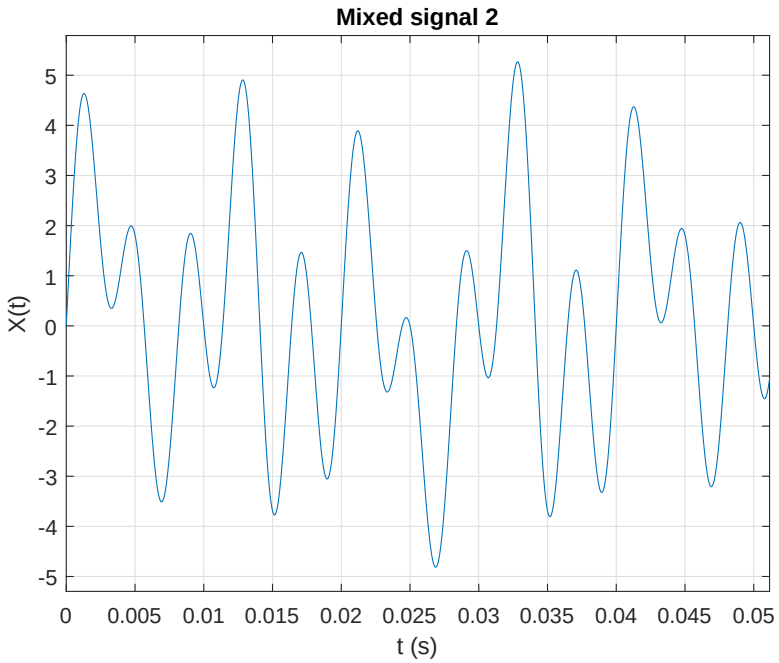


Figure 9.10: mixed signal

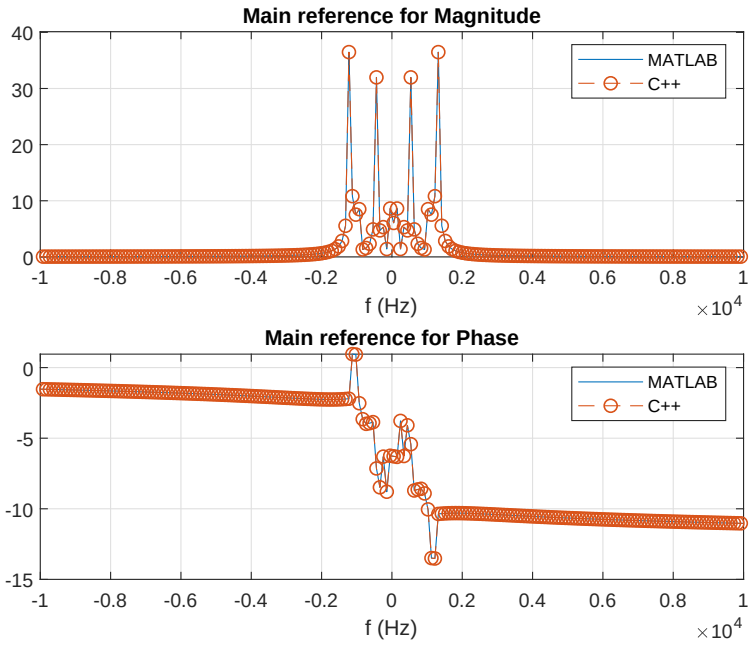


Figure 9.11: MATLAB and C++ comparison

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

9.1.0.13 Algorithm

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

$$X_k = \sum_{n=0}^{N-1} x_n \ e^{m\ i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{9.4}$$

Similarly, for IFFT,

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \ e^{m\ i2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{9.5}$$

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

9.1.0.14 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 9.1 and 9.2 respectively.

9.1.0.15 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

9.2 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 **[Smith1999, Blahut1985]** :
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,

$$y_i(n) = x_i(n) \circledast h(n). \tag{9.6}$$

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

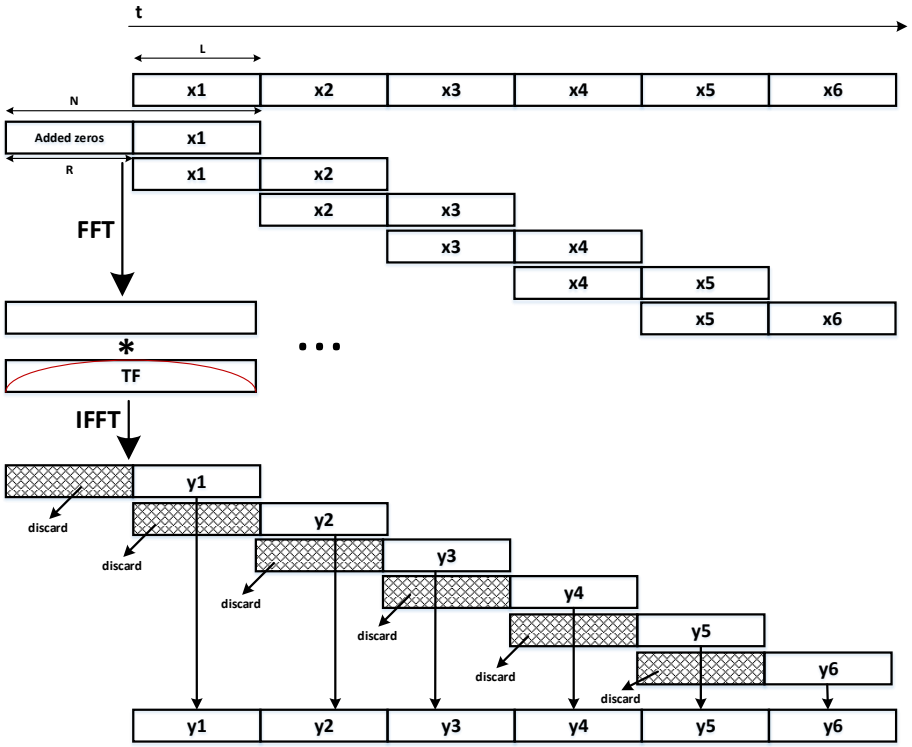


Figure 9.12: Illustration of Overlap-save method.

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

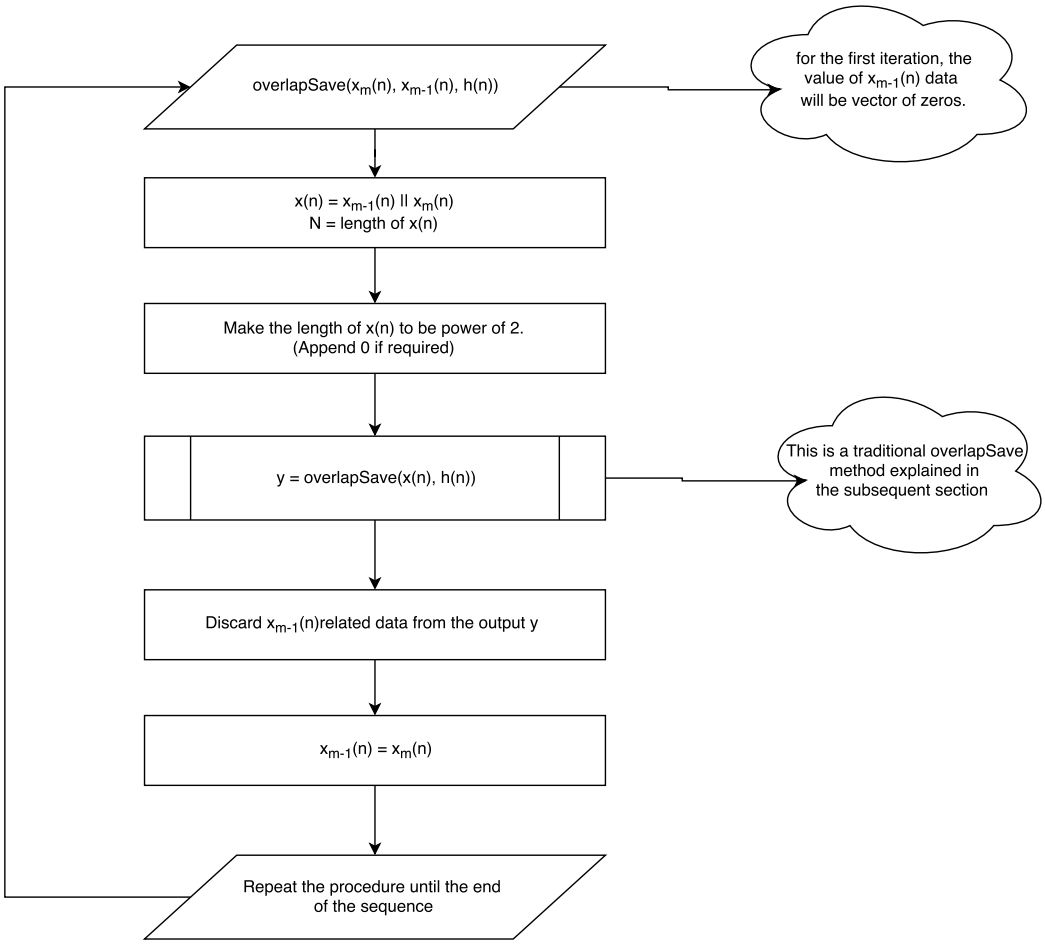


Figure 9.13: Flowchart for real-time overlap-save method

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

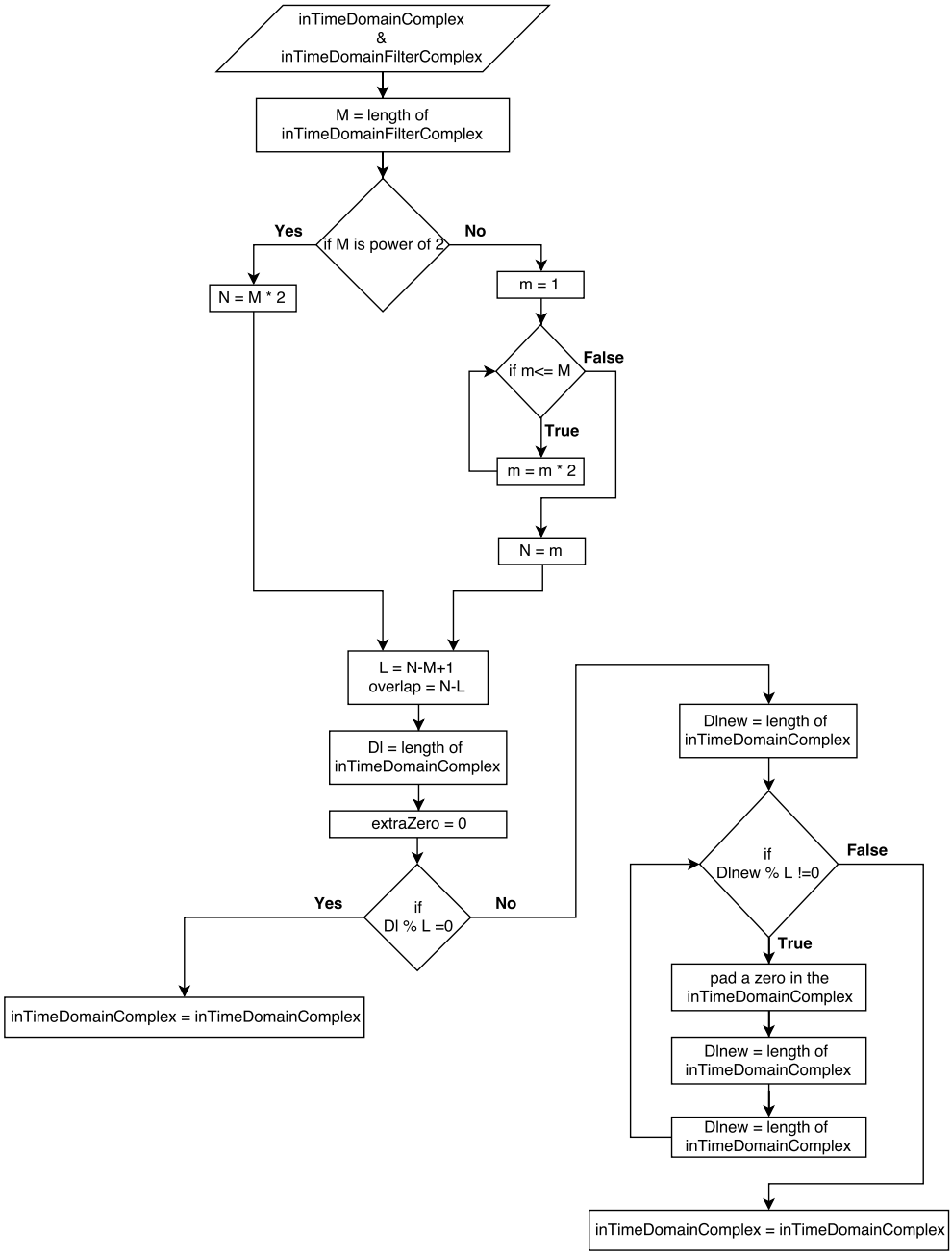


Figure 9.14: Flowchart for calculating length of FFT, data block and filter

2. Create matrix with overlap

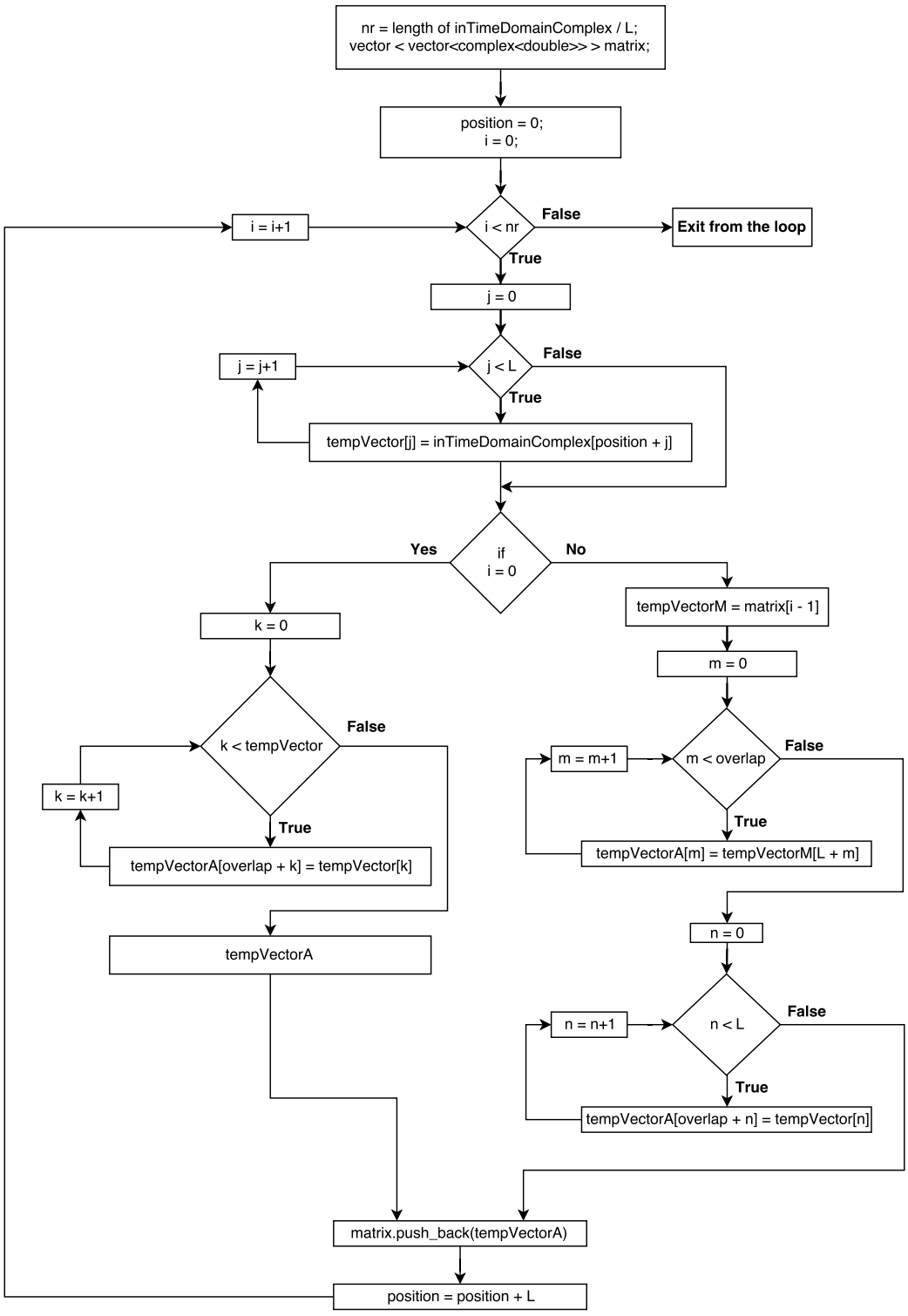


Figure 9.15: Flowchart of creating matrix with overlap

3. Convolution between filter and data blocks

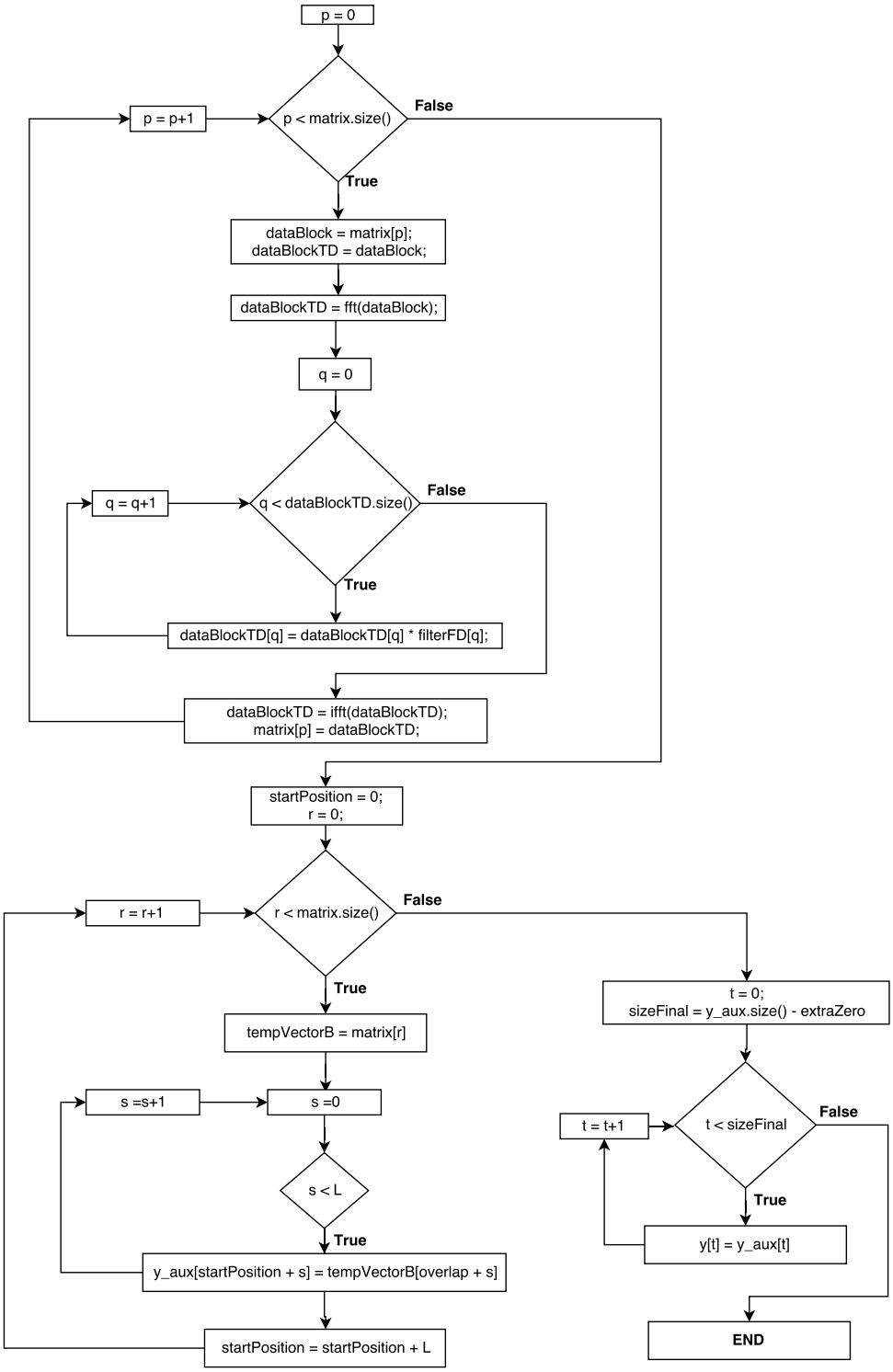


Figure 9.16: Flowchart of the convolution

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);
```

```
21          X = S + 2*randn(size(t));
    case 2
23          signal_title = 'Sinusoids with Random Noise';
            S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
25          X = S + 2*randn(size(t));
    case 3
27          signal_title = 'Single sinusoids';
            X = sin(2*pi*t);
29  case 4
            signal_title = 'Summation of two sinusoids';
31          X = sin(2*pi*1205*t) + cos(2*pi*1750*t);
    case 5
33          signal_title = 'Single Sinusoids with Exponent';
            X = sin(2*pi*250*t).*exp(-70*abs(t));
35  case 6
            signal_title = 'Mixed signal 1';
37          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
39          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);
41  end

43  Xref = X;
    % dlmwrite will generate text file which represents the time domain signal.
45  %dlmwrite('time_domain_data.txt', X, 'delimiter','\t');
    fid=fopen('time_domain_data.txt','w');
47  fprintf(fid,'%.20f\n',X); % 12-Digit accuracy
    fclose(fid);
49

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

65  %dlmwrite('time_domain_filter.txt', h, 'delimiter','\t');
    fid=fopen('time_domain_filter.txt','w');
67  fprintf(fid,'%.20f\n',h); % 20-Digit accuracy
    fclose(fid);
69

71  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 9.3: 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;

int main()
{
  ////////////////////////////////// Section 1 ////////////////////////////////////
  ////// Read the time_domain_data.txt and time_domain_filter.txt files ///////
  ////////////////////////////////////////////////////////////////////////////////
  ifstream inFile;
  double ch;
  vector <double> inTimeDomain;
  inFile.open("time_domain_data.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.s
    inFile >> ch;
    count++;
  }

  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 9.4: 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.

9.2.0.1 Resultant analysis of various test signals

9.2.0.2 1. Signal with two sinusoids and random noise

9.2.0.3 2. Mixed signal2

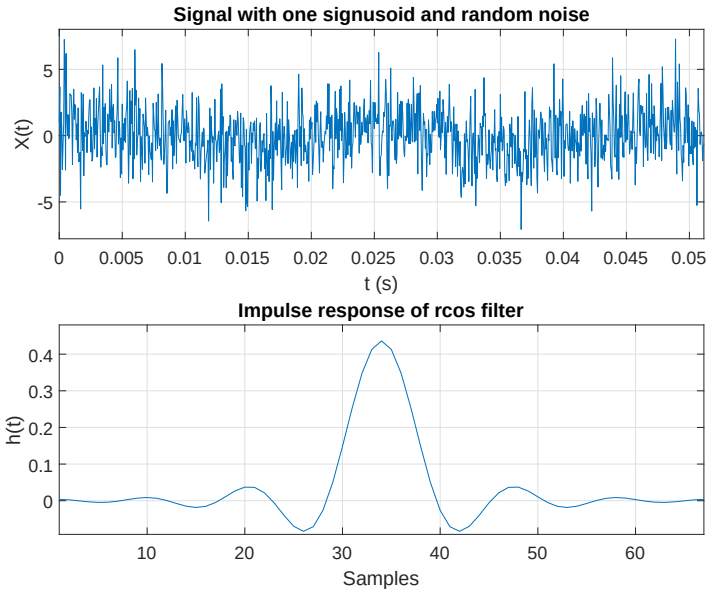


Figure 9.17: Random noise and two sinusoids signal & Impulse response of rcos filter

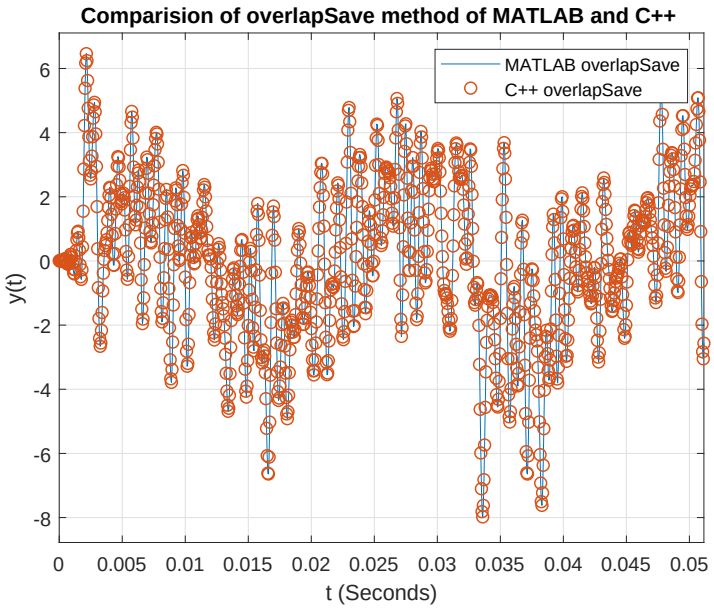


Figure 9.18: MATLAB and C++ comparison

9.2.0.4 3. Sinusoid with exponent

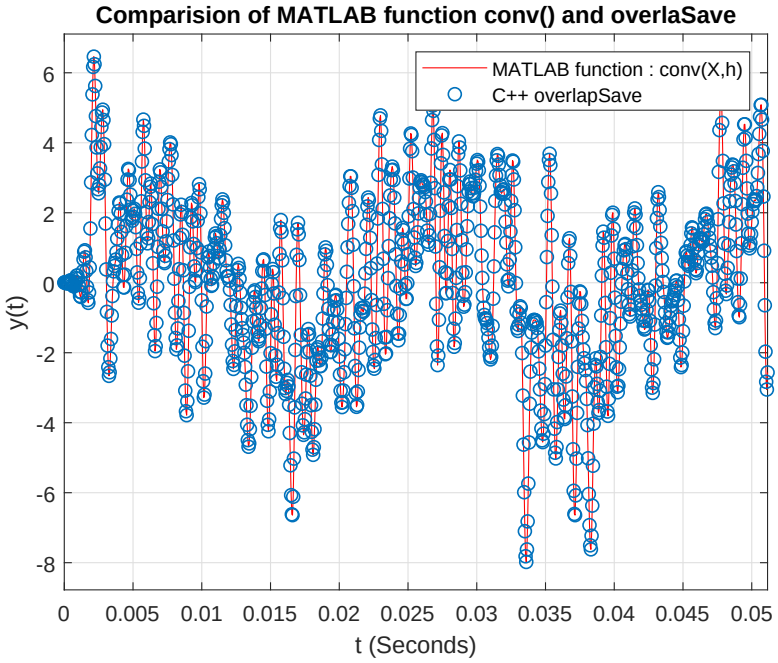


Figure 9.19: MATLAB function conv() and C++ overlapSave comparison

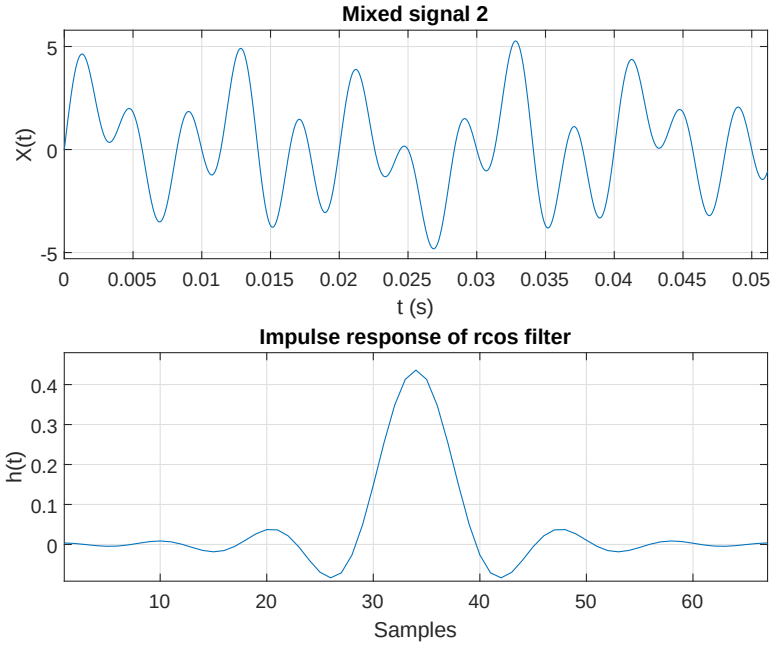


Figure 9.20: Mixed signal2 & Impulse response of rcos filter

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

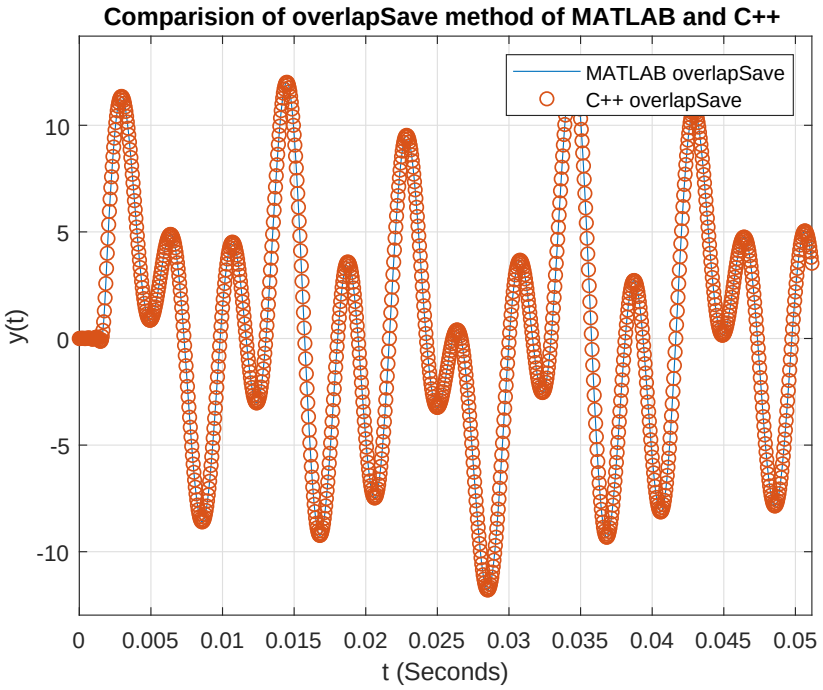


Figure 9.21: MATLAB and C++ comparison

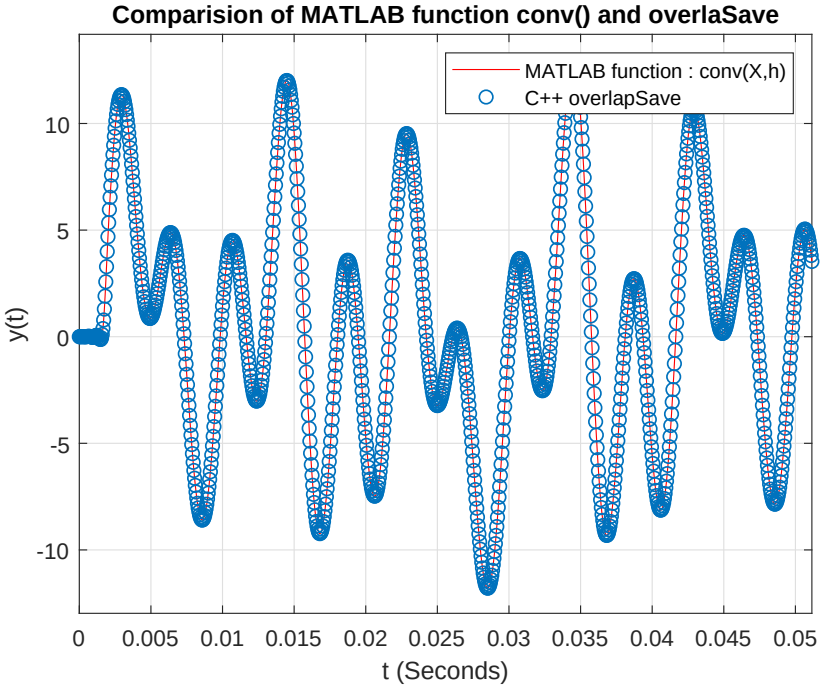


Figure 9.22: MATLAB function conv() and C++ overlapSave comparison

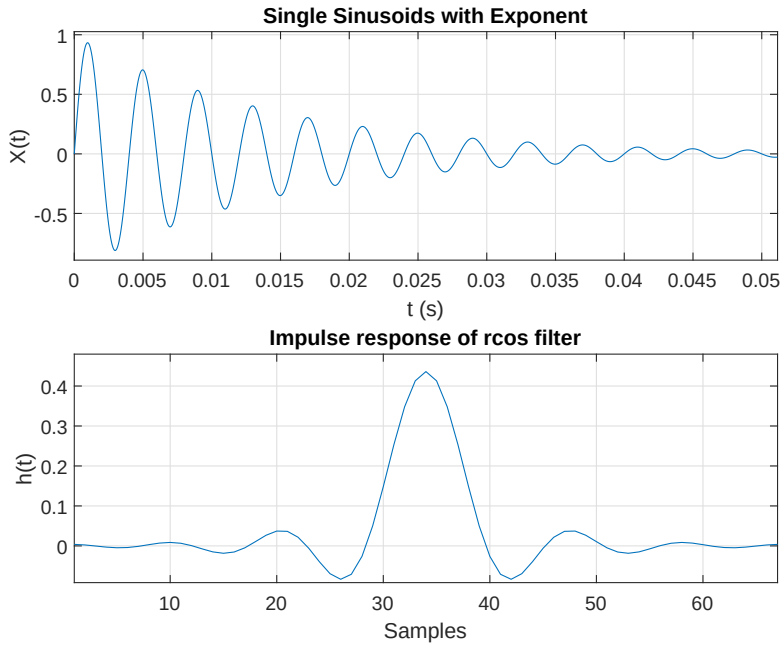


Figure 9.23: Sinusoid with exponent & Impulse response of Gaussian filter

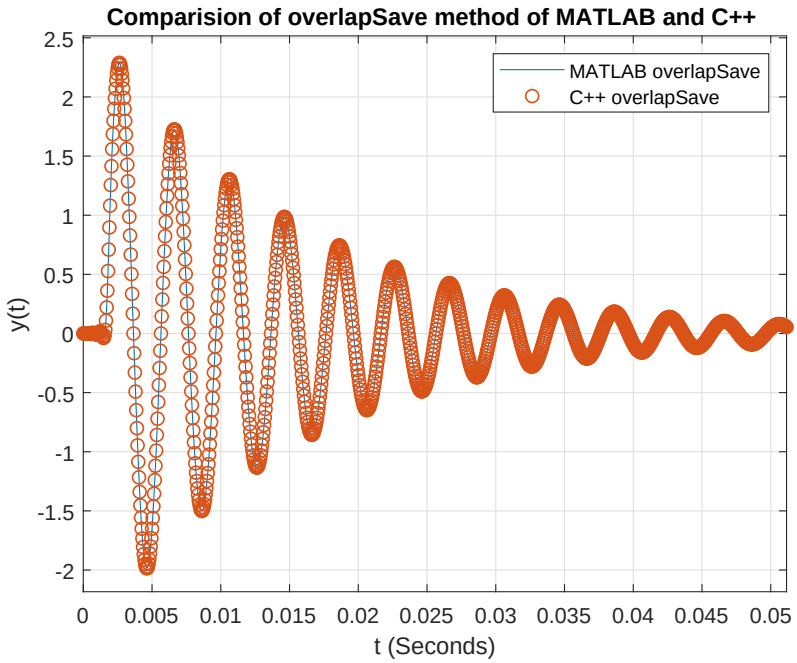


Figure 9.24: MATLAB and C++ comparison

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

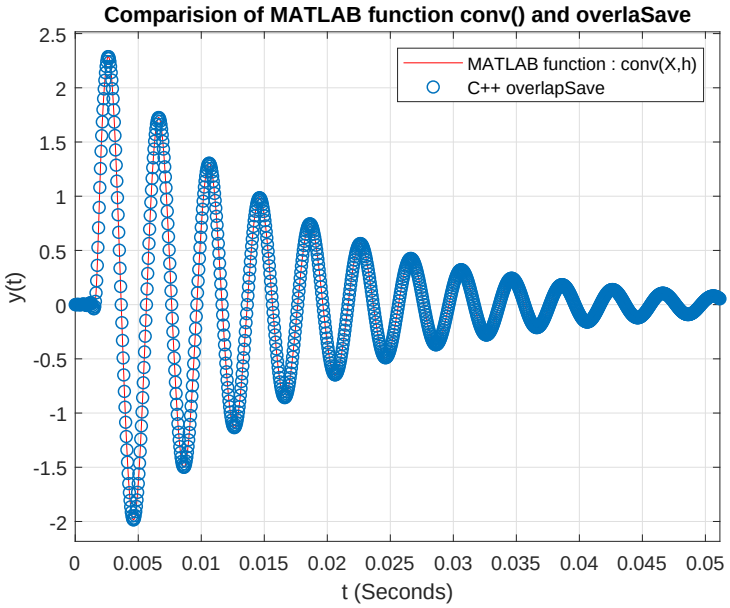


Figure 9.25: MATLAB function conv() and C++ overlapSave comparison

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"

////////////////////////////////////////////////////////////
////////////////////////// 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
```

```
62 ////////////////////////////////////////////////////////////////
   void FD_Filter::initializeFD_Filter (void)
64 {
     outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
66   outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
     outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;
68
     if (!getSeeBeginningOfTransferFunction()) {
70     int aux = (int) (((double)transferFunctionLength) / 2) + 1;
       outputSignals[0]->setFirstValueToBeSaved(aux);
72   }

74   if (saveTransferFunction)
     {
76     ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
       fileHandler << "// ### HEADER TERMINATOR ###\n";
78
       double samplingPeriod = inputSignals[0]->samplingPeriod;
80     t_real fWindow = 1 / samplingPeriod;
       t_real df = fWindow / transferFunction.size();
82
       t_real f;
84     for (int k = 0; k < transferFunction.size(); k++)
       {
86       f = -fWindow / 2 + k * df;
         fileHandler << f << " " << transferFunction[k] << "\n";
88     }
       fileHandler.close();
90   }
   }
92
   bool FD_Filter::runBlock(void)
94 {
     bool alive{ false };
96
     int ready = inputSignals[0]->ready();
98   int space = outputSignals[0]->space();
     int process = min(ready, space);
100  if (process == 0) return false;

102  ////////////////////// previousCopy & currentCopy ////////////////////
     ////////////////////////////////////////////////////////////////////
104  vector<double> re(process); // Get the Input signal
     t_real input;
106  for (int i = 0; i < process; i++){
       inputSignals[0]->bufferGet(&input);
108    re.at(i) = input;
     }
110
     vector<t_real> im(process);
112  vector<t_complex> currentCopyAux = reImVect2ComplexVector(re, im);
```

```
114   vector<t_complex> pcInitialize(process);   // For the  first  data block only
      if  (K == 0){  previousCopy = pcInitialize;  }
116
      // size  modification of  currentCopyAux to currentCopy.
118   vector<t_complex> currentCopy(previousCopy.size());
      for  (unsigned int i  = 0;  i  < currentCopyAux.size(); i++){
120     currentCopy[i] = currentCopyAux[i];
      }
122
      ////////////////////////// Filter Data "hn" ////////////////////////
124   ////////////////////////////////////////////////////////////////////
      vector<t_complex> impulseResponse;
126   impulseResponse = transferFunctionToImpulseResponse(transferFunction);
      vector<t_complex> hn = impulseResponse;
128
      ////////////////////// OverlapSave in Realtime ////////////////////
130   ////////////////////////////////////////////////////////////////////
      vector<t_complex> OUTaux = overlapSave(currentCopy, previousCopy, hn);
132
      previousCopy = currentCopy;
134   K = K + 1;

136   // Remove the size modified data (opppsite to "currentCopyAux to currentCopy")
      vector<t_complex> OUT;
138   for  (int  i  = 0;  i  < process;  i++){
        OUT.push_back(OUTaux[previousCopy.size() + i]);
140   }

142   // Bufferput
      for  (int  i  = 0;  i  < process;  i++){
144     t_real  val;
        val  = OUT[i].real();
146     outputSignals[0]->bufferPut((t_real)(val));
      }
148
      return  true;
150 }
```

Listing 9.5: 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 9.26.

Step 4 : Open the MATLAB visualizer and compare the signal **S6.sgn** and **S7.sgn** as shown in Figure 9.27.

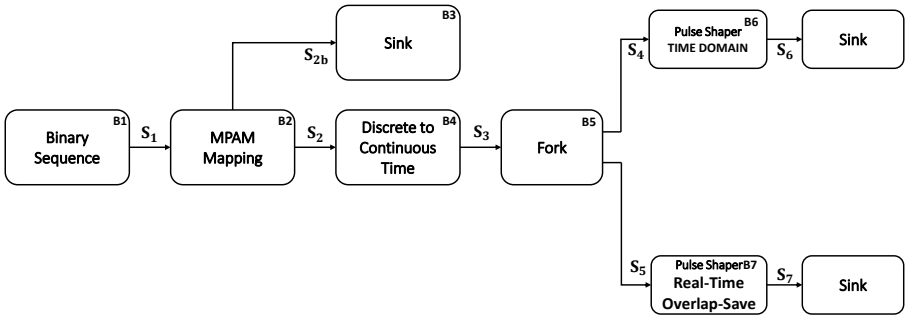


Figure 9.26: Real-time overlap-save example setup

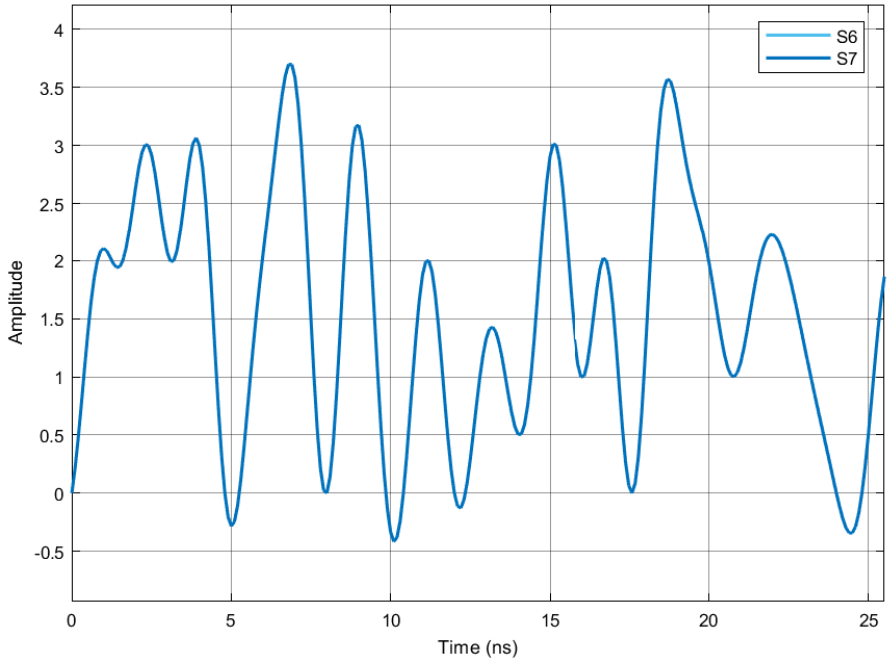


Figure 9.27: Comparison of signal S6 and S7

9.3 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 9.28). The $filter\_*.cpp$ file contains the definitions of all the functions declared in the **FIR_Filter** and **FD_Filter**.
In the Figure 9.28, the function **bool** $runblock(void)$ in the transfer function based **FD_Filter**

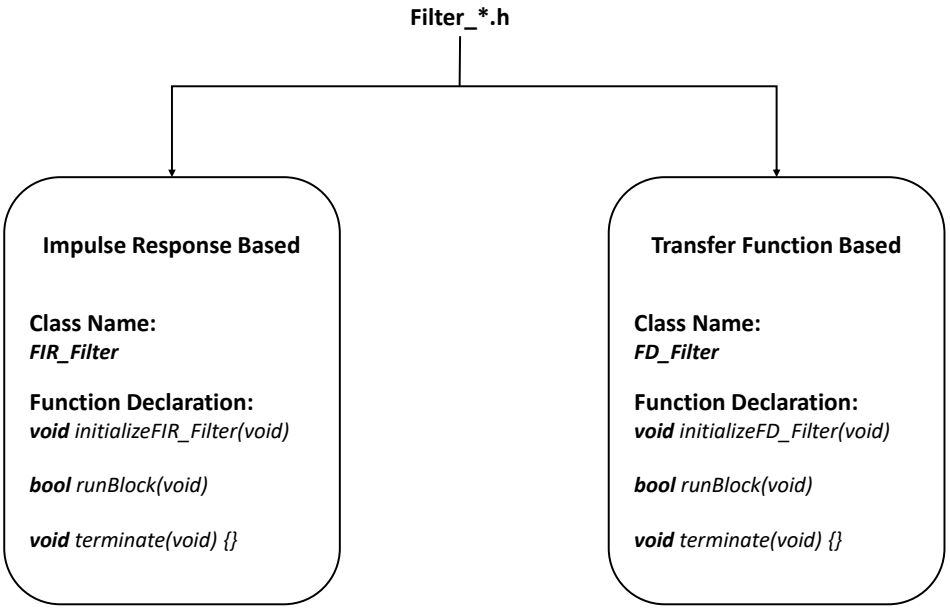


Figure 9.28: Filter class

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 [**Kuo**, **Rappaport2002**] . 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 9.6: 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 9.7: 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 9.29.

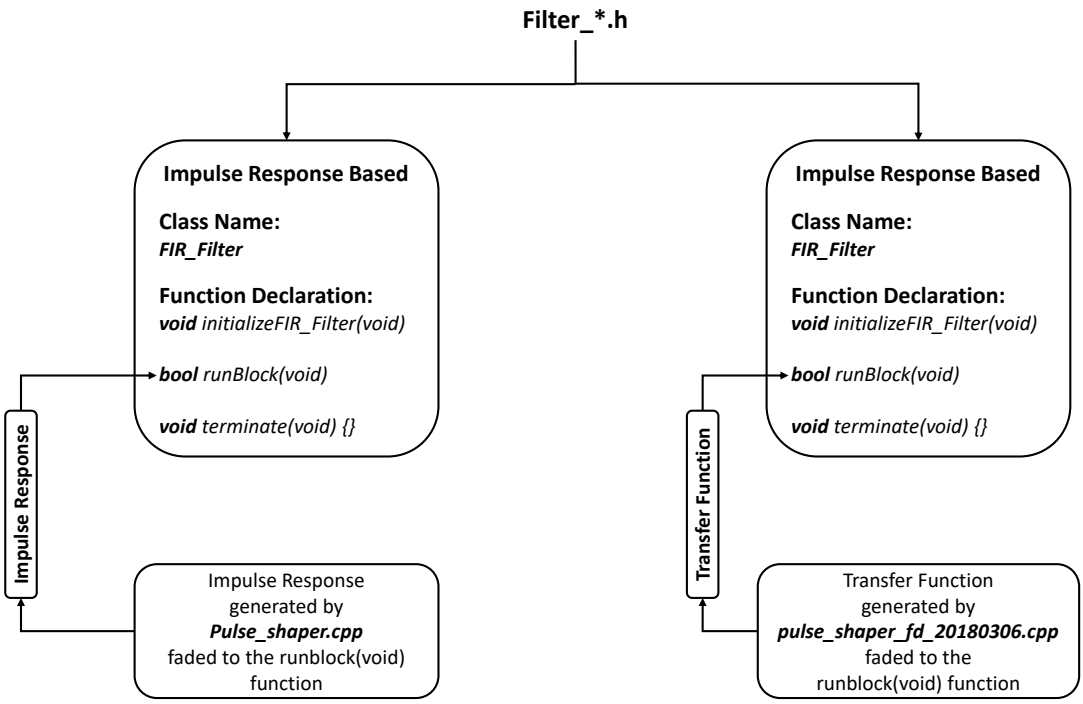


Figure 9.29: Pulse shaping using **filter_*.h**

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 9.30.

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 9.31, 9.32 and 9.33 display

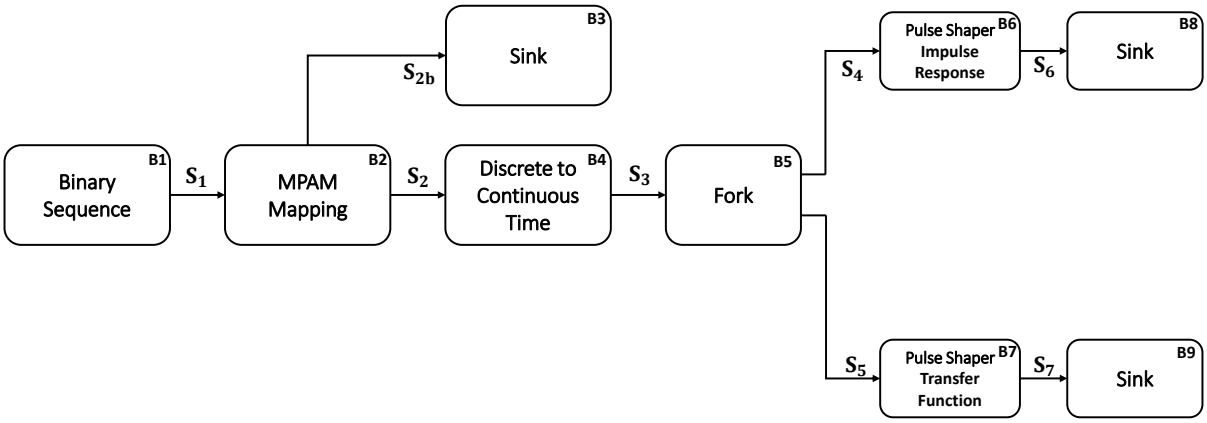


Figure 9.30: Filter test setup

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

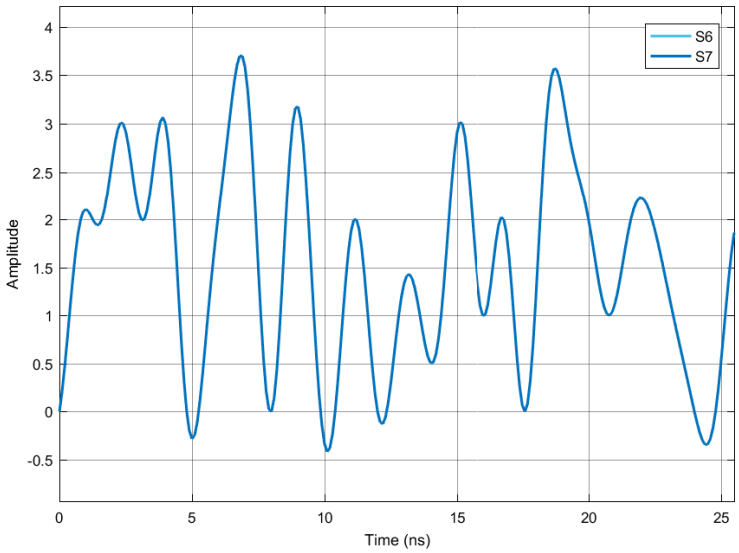


Figure 9.31: Raised cosine pulse shaping results comparison

Case 2 : Root raised cosine

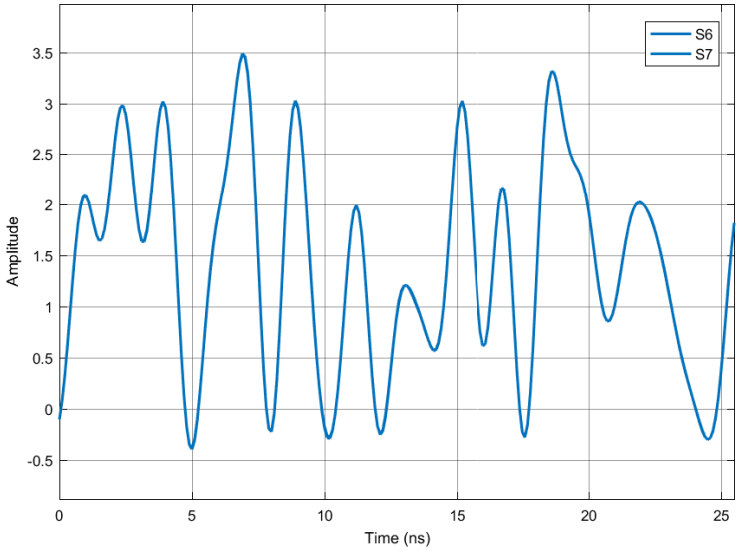


Figure 9.32: Root raised cosine pulse shaping result

Case 3 : Gaussian

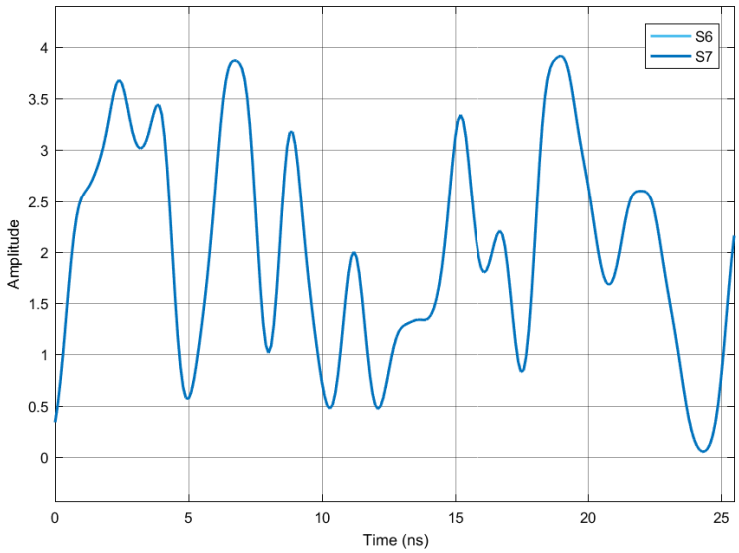


Figure 9.33: Gaussian pulse shaping results comparison

APPENDICES

A. Raised cosine pulse shaper

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

$$H_{RC}(f) = \begin{cases} 1 & \text{for } |f| \leq \dfrac{1-\beta}{2T_s} \\ \dfrac{1}{2}\left[1 + cos\left(\dfrac{\pi Ts}{\beta}\left[|f| - \dfrac{1-\beta}{2T_s}\right]\right)\right] & \text{for } \dfrac{1-\beta}{2T_s} < |f| \leq \dfrac{1+\beta}{2T_s} \\ 0 & \text{otherwise} \end{cases} \tag{9.7}$$

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

$$h_{RC}(t) = \frac{sin(\pi t/T_s)}{\pi t/Ts}\frac{cos(\pi\beta t/Ts)}{1 - 4\beta^2 t^2/T_s^2} \tag{9.8}$$

B. Root raised cosine pulse shaper

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

$$H_{RC}(f) = \begin{cases} 1 & \text{for } |f| \leq \dfrac{1-\beta}{2T_s} \\ \sqrt{\dfrac{1}{2}\left[1 + cos\left(\dfrac{\pi Ts}{\beta}\left[|f| - \dfrac{1-\beta}{2T_s}\right]\right)\right]} & \text{for } \dfrac{1-\beta}{2T_s} < |f| \leq \dfrac{1+\beta}{2T_s} \\ 0 & \text{otherwise} \end{cases} \tag{9.9}$$

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,

$$h_{RRC}(t) = \begin{cases} \dfrac{1}{T_s}\left(1 + \beta(\dfrac{4}{\pi} - 1)\right) & \text{for } t = 0 \\ \dfrac{\beta}{T_s\sqrt{2}}\left[\left(1 + \dfrac{2}{\pi}\right)sin\left(\dfrac{\pi}{4\beta}\right) + \left(1 - \dfrac{2}{\pi}\right)cos\left(\dfrac{\pi}{4\beta}\right)\right] & \text{for } t = \dfrac{T_s}{4\beta} \\ \dfrac{1}{Ts}\dfrac{sin\left[\pi\dfrac{t}{T_s}(1-\beta)\right] + 4\beta\dfrac{t}{Ts}cos\left[\pi\dfrac{t}{T_s}(1+\beta)\right]}{\pi\dfrac{t}{T_s}\left[1 - \left(4\beta\dfrac{t}{T_s}\right)^2\right]} & \text{otherwise} \end{cases} \tag{9.10}$$

C. Gaussian pulse shaper

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

$$H_G(f) = exp(-\alpha^2 f^2) \tag{9.11}$$

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

$$\alpha = \frac{\sqrt{ln2}}{\sqrt{2}B} = \frac{0.5887}{B} \tag{9.12}$$

From the equation 9.12, as α increases, the spectral occupancy of the Gaussian filter decreases. The impulse response of the Gaussian filter can be given by,

$$h_G(t) = \frac{\sqrt{\pi}}{\alpha} exp(-\frac{\pi^2}{\alpha^2} t^2) \tag{9.13}$$

From the equation 9.12, we can also write that,

$$\alpha = \frac{0.5887}{BT_s} T_s \tag{9.14}$$

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.

9.4 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.

$$s_a(t) = s(t) + i\hat{s}(t) \tag{9.15}$$

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 **[Marple]** (This method requires MATLAB Hilbert transform definition)

$$S_a(f) = \begin{cases} 2S(f) & \text{for } f > 0 \\ S(f) & \text{for } f = 0 \\ 0 & \text{for } f < 0 \end{cases} \tag{9.16}$$

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 **[Oppenheim1999]**

$$\hat{S}(f) = \begin{cases} i\,S(f) & \text{for } f > 0 \\ 0 & \text{for } f = 0 \\ -i\,S(f) & \text{for } f < 0 \end{cases} \tag{9.17}$$

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 9.15.

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 9.34. The filtering process will start only after acquiring first

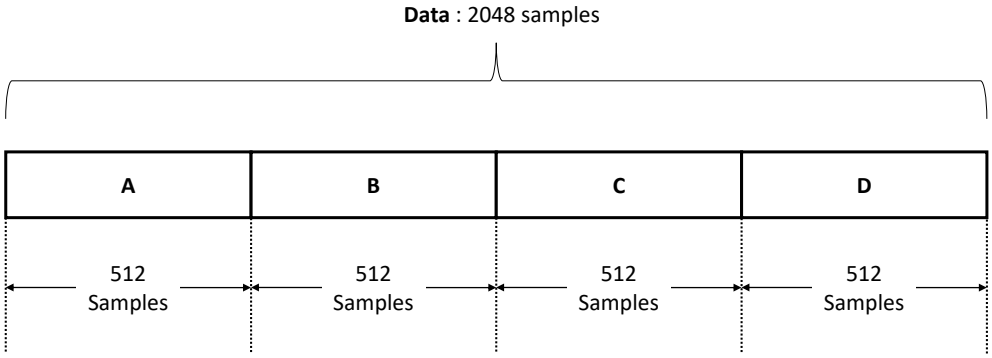


Figure 9.34: Logical flow

two blocks A and B (see iteration 1 in Figure 9.35), 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 $512 \text{ x } 3 = 1536$ 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"

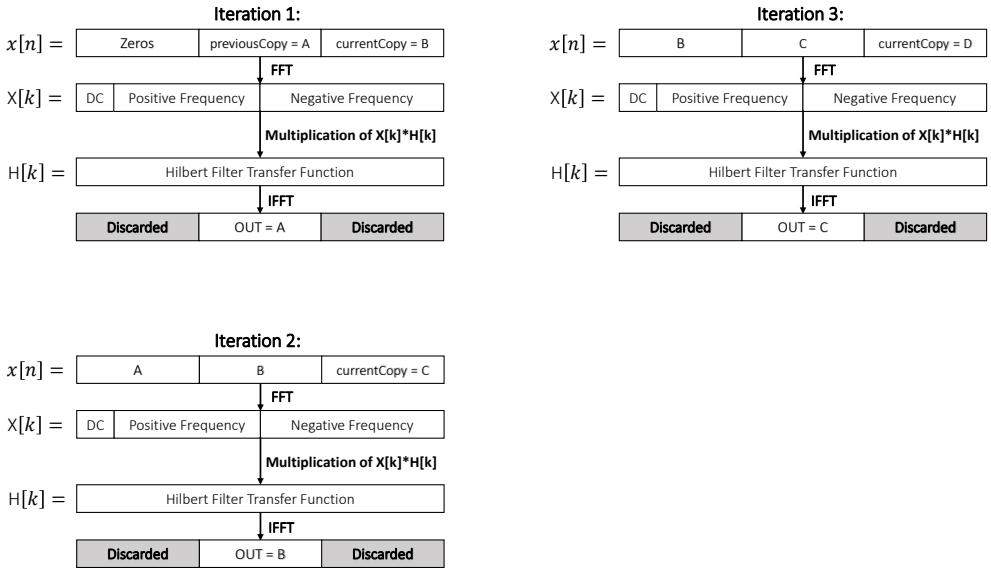


Figure 9.35: Logical flow of real-time Hilbert transform

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

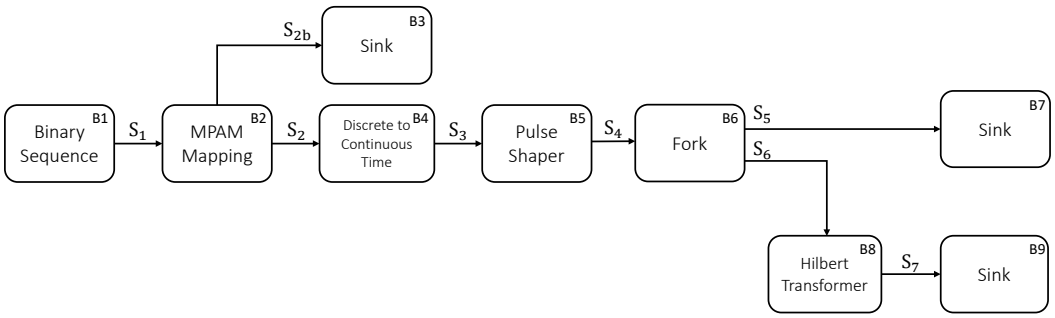


Figure 9.36: Test setup for the real time Hilbert transform

Real-time Hilbert transform : Results

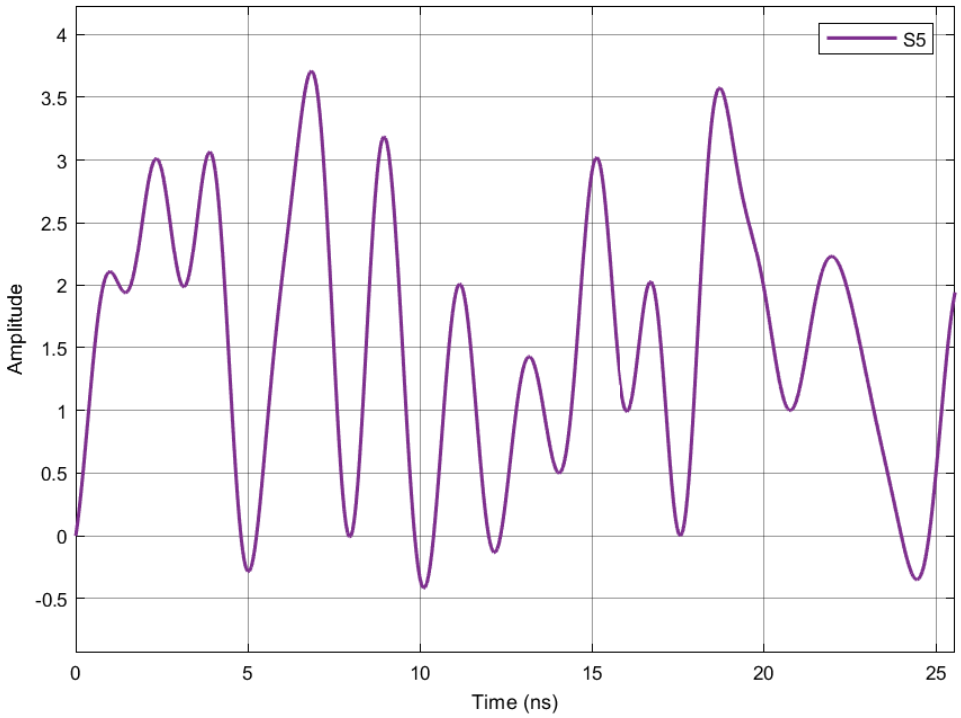


Figure 9.37: $S5$ signal

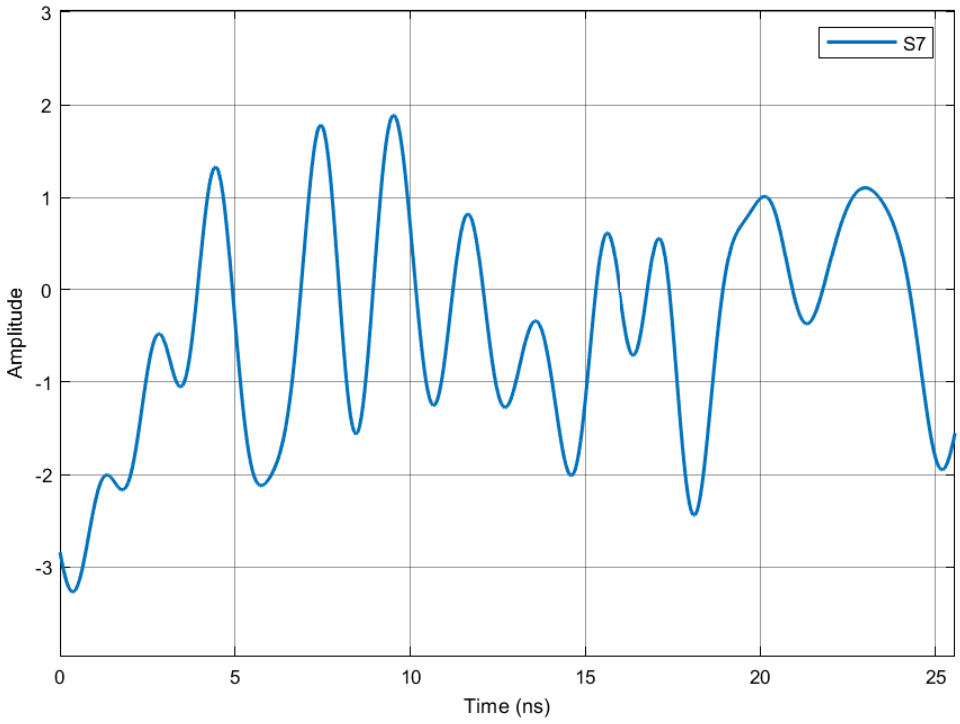


Figure 9.38: $S7$ signal

Remark : Here, we have used method 2 to generate analytical signal using Hilbert transform. If you want to use method 1 then you should use $ifft$ in place of fft and vice-versa.

Chapter 10

Code Development Guidelines

github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

10.0.1 Integrated Development Environment

The recommended IDE is the Visual Studio 2017. To install the Visual Studio 2017 first instal the Microsoft Visual Installer and after proceed to the Visual Studio 2017 installation.

Visual Studio Community 2017 - version 15.7.6

10.0.2 Compiler Switches

Disable Language Extensions	No
Conformance mode	No
C++ Language Standard	ISO C++14 Standard (std:c++ 14)

Chapter 11

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.

11.1 Installing Microsoft Visual C++ Build Tools

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

11.2 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.

11.3 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.

11.4 Known Issues

11.4.1 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 12

Git Helper

12.1 Starting with Git

Git is a free and open source distributed version control system [**Chacon14**]. Git creates and maintains a database that records all changes that occur in a folder. The Git database is named a repository. It also allows to merge repositories that shared a common state in the past. These can be local repositories, i.e. stored in the same machine, or can be remote repositores, i.e. stored anywhere.

To create this database for a specific folder the Git application must be installed on the computer. The Git application and directions for its installation in all majors platforms can be obtained in the Git website (http://git-scm.com). You can access to the Git commands through the console or through a GUI interface. Here, we assume that you are going to use the console.

After the installation, to create the Git initial database open the console program and go to a folder and execute the following command:

git init

The Git database is created and stored in the folder *.git* in the root of your folder. The *.git* folder is your repository.

The Git commands allow you to manipulate this database, i.e. this repository.

12.2 Data Model

To understand Git is fundamental to understand the 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;
- tags - text files that store information about commits.

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. The SHA1 is just an algorithm

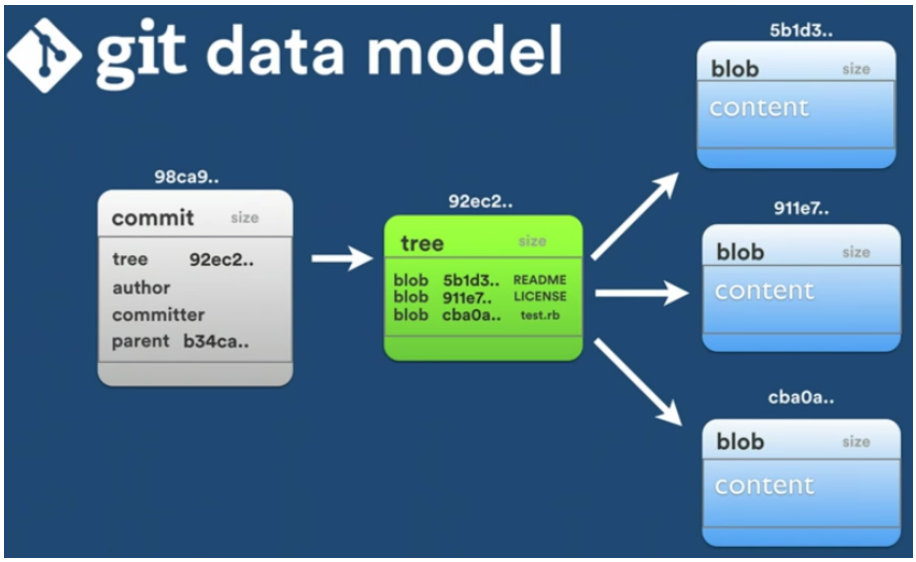


Figure 12.1: Git data model.

that accepts some binary information and generates 20 bytes which are ideally unique for that information. The probability of collisions is extremely low, i.e. the probability that different information generates the same hash value is extremely low. 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. Lets look more carefully in each stored object.

A **commit** object is identified by a SHA1 hash value, and has the following information: a pointer for a tree (the root), a pointer for the previous commit, the author, 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. A tree object identifies a folder and its contend. 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 object type, a space character, the file size in bytes and the \NUL caracter, for instance "blob 13\NUL", the tailor is just the \n caracter. The compressed blob (header+file contend+tailer) is stored as a binary file.

There are two types of **tags**, lightweight and annotated tags. Lightweight tags are only a ref to a commit, see section below. Annotated tags are objects stored as text files, which as has information about the commit, to each the tag point, the tagger (name and e-mail), tag date and a tag message.

12.2.1 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, blobs and annotated tags). 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 space.

12.3 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, used by the computer. 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. Tags and branches are example of refs. Tags are static references, i.e. tags are never updated, and branches are dynamic references, i.e. branches are always automatically updated.

12.3.1 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.

12.3.2 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 atualize so that it always points for the most recent commit of that branch.

12.3.3 Heads

Heads is a pointer for the branch where we are. If we are in a commit that is not pointed by a branch we are in a detached HEAD situation.

12.4 Git Logical Areas

Git uses several spaces.

- Working tree - is your directories where you are working;
- Staging area or index - temporary area used to specify which files are going to be committed in the next commit;
- History - recorded commits;

All information related with the staging area and the history is stored in the *.git* folder.

12.5 Merge

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

12.5.1 Fast-Forward Merge

It is used when there is a direct path between the two branches, the older branch is just updated.

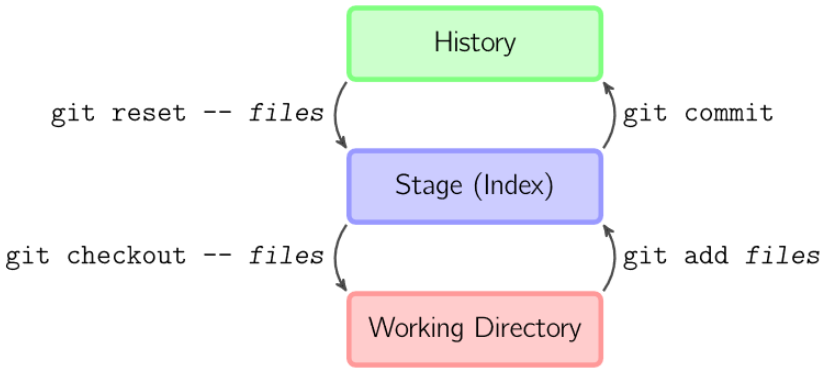


Figure 12.2: Git logical areas. Figure adapted from [**Lodato19**].

12.5.2 Three-Way Merge

It is used when there is no direct path between the two branches. In this case a common commit is found considering the two branches and the merge is performed using a recursive strategy. In this process conflicts can occur. The recursive strategy is applied to each modified file and line by line. If the line was not modified in both branches the line is not modified in the merged file. If the line was modified only in one branch the line is going to be modified in the merged file. If the line is modified in both branches Git cannot make a decision and a conflict occur. So, conflicts occur when branches that change the same file in the same line are being merged.

12.6 Remotes

A remote is a repository in another location. The remote location can be the http://github.com or the location of any other Git server.

12.6.1 GitHub

GitHub is a Git server that stores public repositories. A GitHub user will have a user name and password and inside his or her account creates public repositories. This repositories can be transferred to a local machine using the command *git clone <repository url>*. The local and remote repository will be linked and the local repository can be updated using the *git fetch* or *git pull* command. The remote repository can be updated using the *git push* command. When the remote repository is cloned an alias, named origin, is created that points to that remote repository. Another way to create a GitHub repository is by forking another existente repository. Forked repositories are linked and they can be syncronized using pull requests.

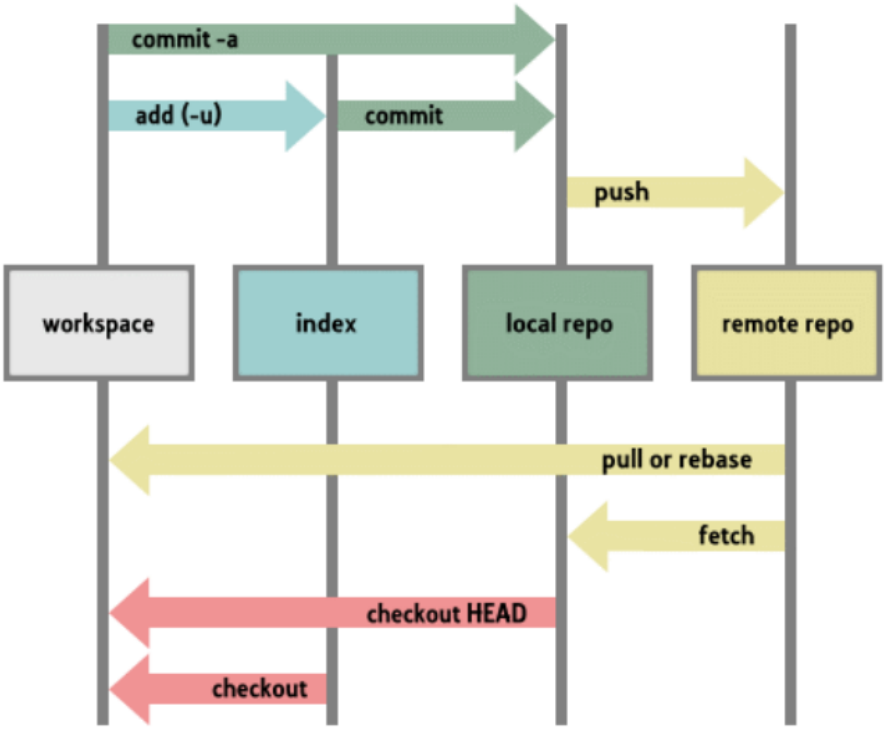


Figure 12.3: Git remotes.

12.7 Commands

12.7.1 Porcelain Commands

Porcelain commends are high-level commands.

git add
git add, adds a a new or modified file (or files) to the staging area.

git branch
git branch, lists local branches.
git branch -r, lists remote branches.
git branch -a, lists all branches.
git branch -u <remote>/<branch>, links the local branch in which you are in with a remote branch.
git branch –set-upstream-to=<remote>/<branch>, longer version of the previous command.
git branch -u <remote>/<branch> <local_branch>, links the local branch with a remote branch.
git branch –set-upstream-to=<remote>/<branch> <local_branch>, longer version of the previous command.

git cat-file

git cat-file -t <hash>, shows the type of the object identified by the hash value.
git cat-file -p <hash>, shows the contend of the file associated with the object identified by the hash value.

git checkout
git checkout -b <new_branch_name>, creates a new branch in the same position as the current branch and move to it.
git checkout -b <new_branch_name> <branch_name>, create a new branch in the position of <branch_name> and move to it.

git clean
git clean -f -d, removes from the working directory all untracked directories (d) and files (f).

git clone
git clone <url>, downloads the contend of the <url> repository, for instance http://www.github.com/netxpto/linkplanner.git, and creates a local repo.

git config
git config –global user.name "netxpto", sets the user name globally.
git config –global user.email "netxpto.gmail.com", sets the user e-mail globally.
git config –global user.emmail "netxpto.gmail.com", sets the user e-mail globally.
git config –global alias.<alias name> <commands>, creates a global alias for the commands.

git diff
git diff, shows the changes between the working space and the staging area.
git diff –name-only, shows the changes between the working space and the staging area, in the specified files.
git diff –cached, shows the changes between the staging area and the current branch history. Note that –cached or –staged are synonymous.
git diff –cached –name-only, shows the changes between the staging area and the current branch history, in the specified files.

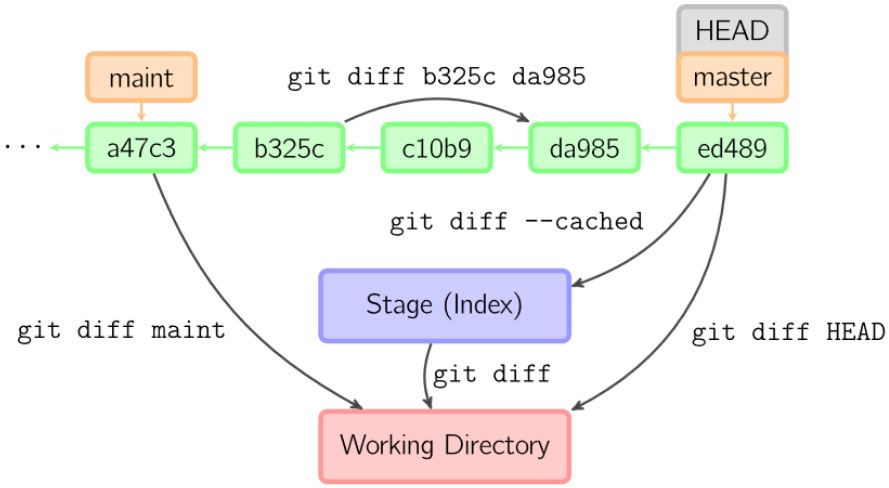


Figure 12.4: Git diff command. Figure adapted from [**Lodato19**].

git fetch
git fetch –all, downloads all history from all remote repositories.
git fetch <repository>, downloads all history from the remote repository.

git init
git init, inicializes a git repository. It creates the *.git* folder and all its subfolders and files.

git log
git log, shows a list of commits from reverse time order.
git log –graph –decorate –oneline <commit1>..<commit1>, shows the history of the repository in a colourful way.
git log –graph, shows a graphical representation of the repository commits history.
git log –stat, shows the name of the files that were changed in each commit
git log –follow <file>, lists version history for a file, including rename.

git ls-files

git ls-remote

git merge
git merge <branch>, combines the specified branchâ€™s history into the current branch.

git pull

git pull, downloads remote branch history and incorporates changes.

git push
git push, uploads local branch commits to remote repository branch.

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 remote
git remote, shows a list of existing remotes.
git remote -v, shows the full location of existing remotes.
git remote add <remote name> <remote repository url>, adds a remote.
git remote remove <remote name>, removes a remote.

git reset
git reset –soft <commit_hash_value>, moves to the commit identified by <commit_hash_value> but leaves in the staging area all modified files.
git reset –hard <commit_hash_value>, moves to the commit identified by <commit_hash_value> and cleans all modified tracked files.
git reset <commit_hash_value>, this is a mix reset (is the default reset), puts all modified files in the working area.
git reset <file>, unstages the file, but preserve its contents.

git reflog
git reflog, shows all commands from the last 90 days. Git only perform garbaged collection after 30 days.

git rm
git rm <file>, deletes the file from the working directory and stages the deletion.
git rm –cached <file>, removes the file from version control but preserves the file locally.

git show
git show, shows what is new in the last commit.

git stash
git stash, temporarily stores all modified tracked files.
git stash –list, shows what is in the stash.
git stash pop, restores the most recently stashed files.

git stash drop, discards the most recently stashed changeset.

git status
git status, lists all new or modified files to be committed.

12.7.2 Pluming Commands

Pluming commands are low-level commands.

git cat-files
git cat-files -p <sha1>, shows the contend of a file in a pretty (-p) readable format.
git cat-files -t <sha1>, shows the type of a object, i.e. blob, tree or commit.

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

git gc
git gc, garbage collector, eliminates all objects that has no reference associated with.
git gc –prune=all

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

git merge-base
git merge-base <branch1> <brach2>, finds the base commit for the three-way merge between <branch1> and <branch2>.

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 rev-parse
git rev-parse <ref>, return the hash value of <ref>.

git rev-parse <short_hash_value>, return the full hash value associated with <short_hash_value>.

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

12.8 Navigation Helpers

<*ref*>^, one commit before <ref>.
<*ref*>^^, two commits before <ref>.
<*ref*>~*5*, five commits before <ref>.
<*ref1*>..<*ref2*>, between commit <ref1> and <ref2>.
<*branch*>^*tree*, identifies the tree pointed by the commit pointed by <branch>.
<*commit*>:<*file_name*>, identifies the version of a file in a given commit.

12.9 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

12.10 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.

12.11 Applications

12.11.1 Meld

12.11.2 GitKraken

12.12 Error Messages

12.12.1 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.

12.13 Git with Overleaf

You can use git with overleaf. For that you have to create a project on overleaf. Associate with that overleaf project it is also create a git repository. The address of that git repository is almost the same as the overleaf project that you can obtain going to the overleaf Menu/Share. Let's assume that the overleaf project address is:

https://www.overleaf.com/12925162jkwbhrdkwrfm

In this case the repository address is https://git.overleaf.com/12925162jkwbhrdkwrfm
The only change was the replacement of **www** by **git**.
Now you can just do
git clone https://git.overleaf.com/12925162jkwbhrdkwrfm
and clone your repository.
You can also do
git push https://git.overleaf.com/12925162jkwbhrdkwrfm

Chapter 13

Simulating VHDL Programs with GHDL

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

13.1 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\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\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\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\ghdl\bin**.
10. Press **Ok** and you're done.

13.2 Using GHDL To Simulate VHDL Programs

This guide is meant to explain how to execute the testbench for module CPE BPS.
This simulation will take two .sgn files as input and produce two .sgn files.

13.2.1 Simulation Input

Make sure that files S19.sgn and S20.sgn exist in directory **\LinkPlanner\sdf\dsp_laser_phase_compensation\signals**. The content of these files will be used as input of the CPE BPS module.

13.2.2 Executing Testbench

Execute the batch file **simulation.bat**, located in the directory **\LinkPlanner\sdf\dsp_laser_phase_compensation\VHDL\Simulator** of this repository.

13.2.3 Simulation Output

The simulation will produce two files: **sim_out_1.sgn** and **sim_out_2.sgn** which will contain the output of the CPE BPS module.