NetXPTO - LinkPlanner

December 11, 2018

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.

3.4 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.

log.txt

```
2018-04-08 23:27:37
MQamTransmitter|S1|space=20
MQamTransmitter|S0|space=20
0.224

2018-04-08 23:27:37
LocalOscillator|S2|space=20
0.001

2018-04-08 23:27:37
BalancedBeamSplitter|S1|ready=20
BalancedBeamSplitter|S2|ready=20
BalancedBeamSplitter|S3|space=20
BalancedBeamSplitter|S4|space=20
0

2018-04-08 23:27:37
I_HomodyneReceiver|S3|ready=20
I_HomodyneReceiver|S4|ready=20
I_HomodyneReceiver|S5|space=20
0.004
```

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

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*

Chapter 4

Development Cycle

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

Chapter 5

Visualizer

visualizer

Chapter 6

Case Studies

6.1 BPSK Transmission System

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

Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying (PSK), in which binary information is encoded into a two state constellation with the states being separated by a phase shift of π (see Figure 6.1).

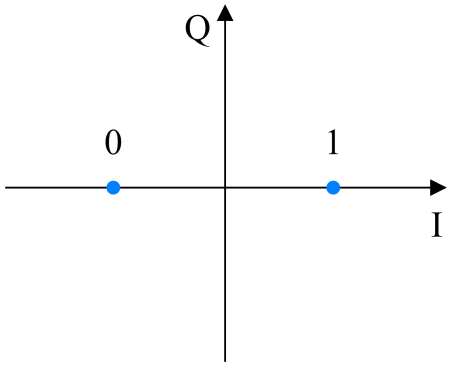


Figure 6.1: BPSK symbol constellation.

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

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

6.1.1 Theoretical Analysis

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

$$m_i = 2\sqrt{P_L P_S} G_{ele} \cos(\Delta\theta_i), \tag{6.1}$$

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

$\Delta\theta_i$ is the phase difference between the local oscillator and the signal, for BPSK this takes the values π and 0, in which case (6.1) can be reduced to,

$$m_i = (-1)^{i+1} 2\sqrt{P_L P_S} G_{ele},\ i = 0,\ 1. \tag{6.2}$$

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

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

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m_i)^2}{2\sigma^2}}. \tag{6.3}$$

The BER is calculated in the following manner,

$$BER = \frac{1}{2}\int_0^{+\infty} f(x|\Delta\theta = \pi)\mathrm{d}x + \frac{1}{2}\int_{-\infty}^{0} f(x|\Delta\theta = 0)\mathrm{d}x, \tag{6.4}$$

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

$$BER = \int_0^{+\infty} f(x|\Delta\theta = \pi)\mathrm{d}x = \frac{1}{2}\mathrm{erfc}\left(\frac{-m_0}{\sqrt{2}\sigma}\right) \tag{6.5}$$

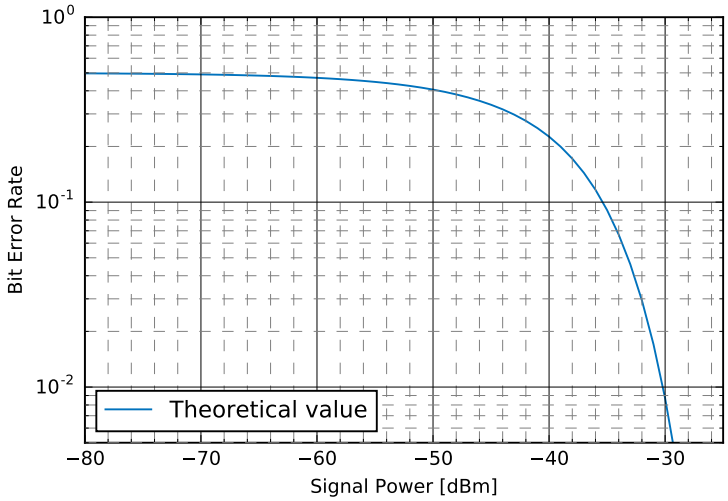


Figure 6.2: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

6.1.2 Simulation Analysis

A diagram of the system being simulated is presented in the Figure 6.3. A random binary sequence is generated and encoded in an optical signal using BPSK modulation. The decoding of the optical signal is accomplished by an homodyne receiver, which combines the

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

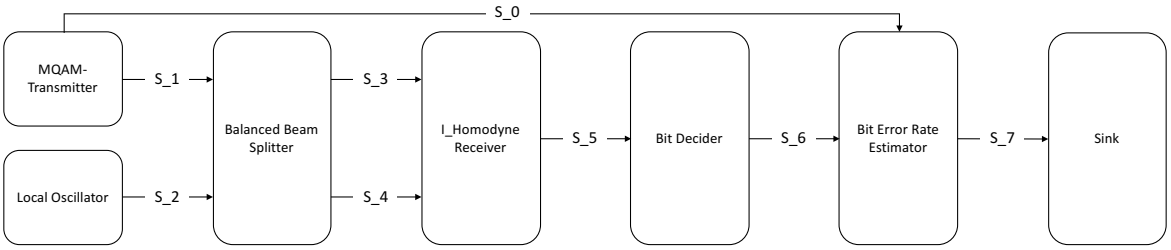


Figure 6.3: Overview of the BPSK system being simulated.

Required files

Header Files		
File	**Comments**	**Status**
add_20171116.h		✓
balanced_beam_splitter_20180124.h		✓
binary_source_20180118.h		✓
bit_decider_20170818.h		✓
bit_error_rate_20171810.h		✓
discrete_to_continuous_time_20180118.h		✓
i_homodyne_receiver_20180124.h		✓
ideal_amplifier_20180118.h		✓
iq_modulator_20180130.h		✓
local_oscillator_20180130.h		✓
m_qam_mapper_20180118.h		✓
m_qam_transmitter_20180118.h		✓
netxpto_20180118.h		✓
photodiode_old_20180118.h		✓
pulse_shaper_20180118.h		✓
sampler_20171116.h		✓
sink_20180118.h		✓
super_block_interface_20180118.h		✓
ti_amplifier_20180102.h		✓
white_noise_20180118.h		✓

Source Files		
File	**Comments**	**Status**
add_20171116.cpp		✓
balanced_beam_splitter_20180124.cpp		✓
binary_source_20180118.cpp		✓
bit_decider_20170818.cpp		✓
bit_error_rate_20171810.cpp		✓
discrete_to_continuous_time_20180118.cpp		✓
i_homodyne_receiver_20180124.cpp		✓
ideal_amplifier_20180118.cpp		✓
iq_modulator_20180130.cpp		✓
local_oscillator_20180130.cpp		✓
m_qam_mapper_20180118.cpp		✓
m_qam_transmitter_20180118.cpp		✓
netxpto_20180118.cpp		✓
photodiode_old_20180118.cpp		✓
pulse_shaper_20180118.cpp		✓
sampler_20171116.cpp		✓
sink_20180118.cpp		✓
super_block_interface_20180118.cpp		✓
ti_amplifier_20180102.cpp		✓
white_noise_20180118.cpp		✓

System Input Parameters

This system takes into account the following input parameters:

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

Inputs

This system takes no inputs.

Outputs

This system outputs the following objects:

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

Bit Error Rate - Simulation Results

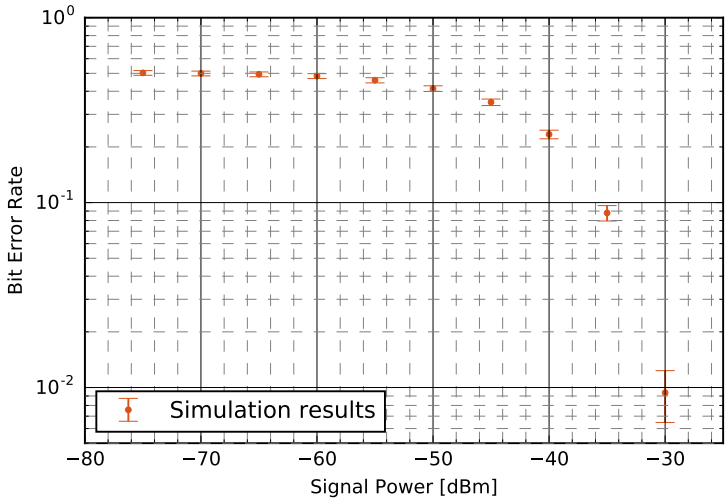


Figure 6.4: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

6.1.3 Comparative Analysis

The following results show the dependence of the error rate with the signal power assuming a constant Local Oscillator power of $0\ dBm$, the signal power was evaluated at levels between -70 and -25 dBm, in steps of 5 dBm between each. The simulation results are presented in orange with the computed lower and upper bounds, while the expected value, obtained from (6.5), is presented as a full blue line. A close agreement is observed between the simulation results and the expected value. The noise spectral density was set at $5 \times 10^{-4}\sqrt{2}\ \mathrm{V}^2$ [1].

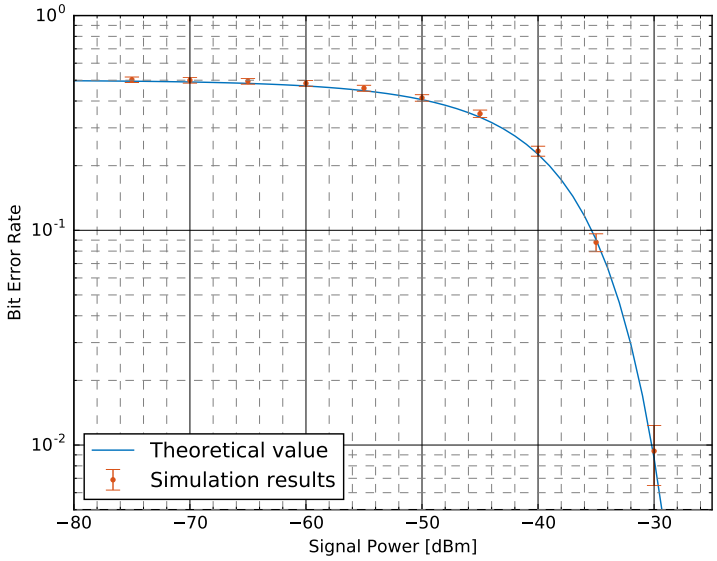


Figure 6.5: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 7.3

References

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

6.2 Kramers-Kronig Transceiver

Student Name	:	Romil Patel (16/08/2017 - Cont.)
Goal	:	Develop a simplified structure (low cost) for a coherent transceiver, that can be used in coherent PON, inter-data center connections, or metropolitan networks (optical path lengths < 100 km). We are going to explore a Kramers-Kronig transceiver with Stokes based PolDemux.
Directory	:	LinkPlanner\doc\tex\sdf\kramers_kronig_transceiver

Coherent optical transmission schemes are spectrally efficient since they allow the encoding of information in both quadratures of the sinusoid signal. However, the cost of coherent receiver becomes a major obstacle in the case of short-reach links applications like PON, inter-data-center communications and metropolitan network. In order to make the transceiver applicable in short-reach links, an architecture has been proposed which combines the advantages of coherent transmission and cost-effectiveness of direct detection. The working principle of the proposed transceiver is based on the Kramers-Kronig (KK) relationship. The KK transceiver scheme allows digital compensation of propagation impairment because both amplitude and phase of the electrical field can be retrieved at the receiver.

6.2.1 Theoretical Analysis

The Kramers-Kronig relations are bidirectional mathematical relations, connecting the real and imaginary parts of any complex function that is analytic in the upper half-plane. For instance, a signal $x(t) = x_r(t) + ix_i(t)$ satisfies the Kramers-Kronig relationship if [1, 2],

$$x_r(t) = -\frac{1}{\pi} p.v. \int_{-\infty}^{\infty} \frac{x_i(t')}{t - t'} dt'$$

$$x_i(t) = \frac{1}{\pi} p.v. \int_{-\infty}^{\infty} \frac{x_r(t')}{t - t'} dt'$$

where $p.v.$ stands for $principle\ value$ which is a standard method applied in mathematical applications by which an improper, and possibly divergent, integral is measured in a balanced way around singularities or at infinity [3]. A signal that satisfies the Kramers-Kronig relationship is also named as an analytical signal. This relationship imposes that the real and the imaginary parts of the signal are related to each other though Hilbert transform. Therefore, if we have the real part of the signal then the imaginary part can be calculated by its Hilbert transform.
For a signal that satisfies the Kramers-Kronig relationship, the real and imaginary part can be obtained only from the module. The following questions would give a comprehensive overview of Kramers-Kroning relation and the detailed mathematical calculation which depicts how phase can be extracted from the amplitude information.

1. What is the Hilbert transform?

If we consider a filter $H(\omega)$, described in Figure 6.6, that has a unity magnitude response for all frequencies and the phase response is $\pi/2$ for all positive frequencies and $-\pi/2$ for negative frequencies. The transfer function of this filter is given by

$$H(\omega) = -i\,\mathrm{sgn}(\omega) = \begin{cases} -i & \text{for } i > 0 \\ i & \text{for } i < 0 \\ 0 & \text{for } i = 0 \end{cases} \tag{6.6}$$

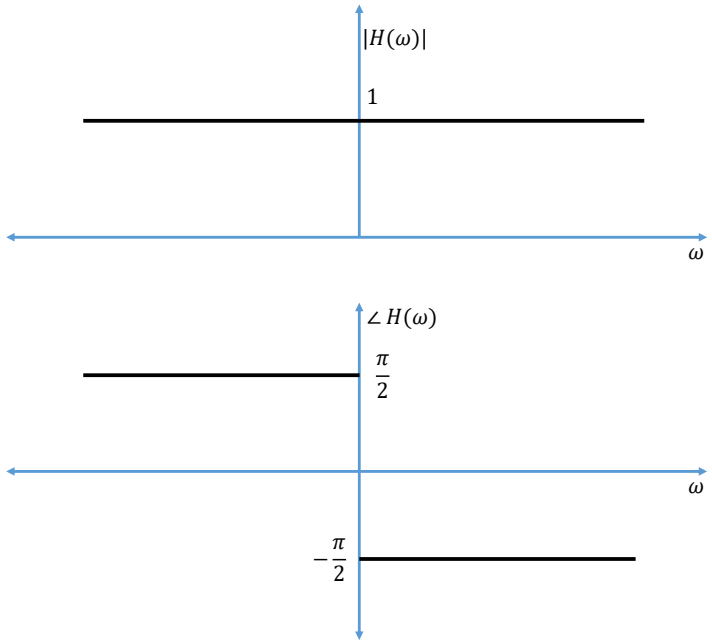


Figure 6.6: Magnitude and phase of Hilbert transform filter

The impulse response of this filter can be given as,

$$\begin{aligned} h(t) &= \mathcal{F}^{-1}[H(i\omega)] \\ &= -i\mathcal{F}^{-1}[\mathrm{sgn}(\omega)] \\ &= -i\left(\frac{i}{\pi t}\right) \\ &= \frac{1}{\pi t} \end{aligned} \tag{6.7}$$

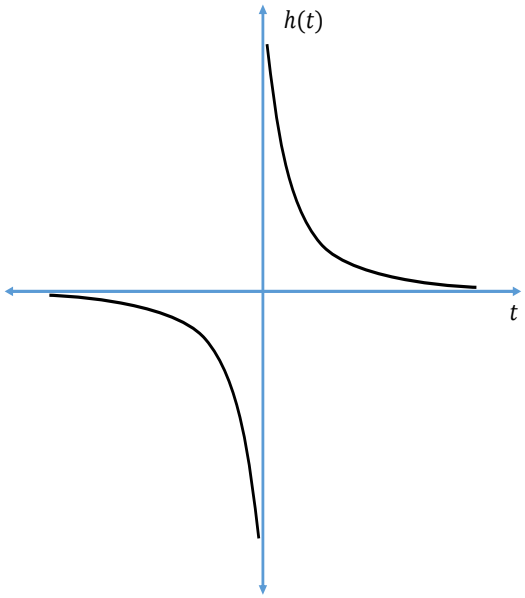


Figure 6.7: Impulse response $h(t)$ of Hilbert transform filter

where F^{-1} is the inverse transform. When this filter is driven by an arbitrary signal $s(t)$, the filter produces the output as,

$$\begin{aligned} \hat{s}(t) &= s(t) \circledast h(t) \\ &= \int_{-\infty}^{\infty} \frac{s(u)}{\pi(t-u)} du \end{aligned} \tag{6.8}$$

The time function $\hat{s}(t)$ is called the Hilbert transform of $s(t)$. Note that

$$\mathcal{F}[\hat{s}(t)] = H(\omega)S(\omega) = i\,\text{sgn}(\omega)S(\omega) \tag{6.9}$$

which means that the Fourier transform of $\hat{s}(t)$ equals the Fourier transform of s(t) multiplied by the factor $i\,\text{sgn}(\omega)$, where $sgn(\omega)$ is -1 for negative frequencies and 1 for positive frequencies. In conclusion, if we convolve any time domain signal with $\frac{1}{\pi t}$ then it will give us Hilbert transformed signal in time domain. Similarly, from the convolution property of the Fourier transform, if we multiply $i\ sgn(\omega)$ with any frequency domain signal $S(\omega)$ then it'll give us Hilbert transformed signal in frequency domain.

Example of rectangle function

Consider the rectangular signal $r(t)$ and its Hilbert transformed $\hat{r}(t)$ as shown in the Figure 6.8. The Fourier transformed version of the signal $r(t)$ and $\hat{r}(t)$ is described as $R(f)$ and $\hat{R}(f)$

and their magnitude and the phase spectrum are shown in Figure 6.9 and 6.10, respectively.

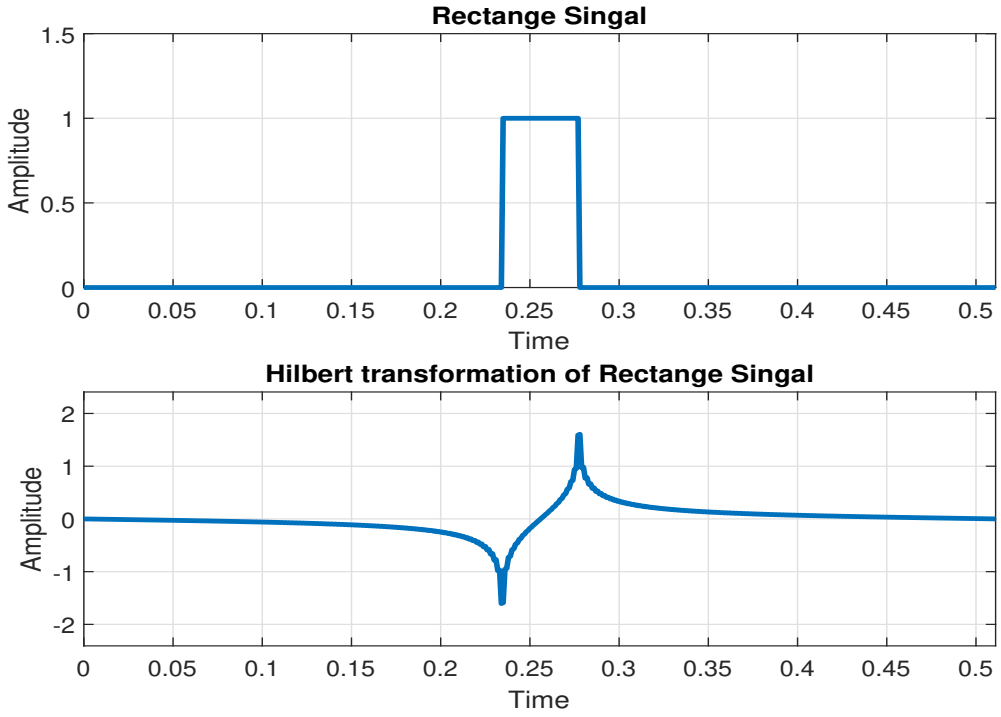


Figure 6.8: Rectangle signal $r(t)$ and its Hilbert transformed $\hat{r}(t)$

2. What is an analytical signal?

An analytic signal, i.e. a signal that satisfies the KK relationship, 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.

$$s_a(t) = s(t) + i\hat{s}(t) \tag{6.10}$$

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. If we denote an analytic signal as,

$$s_a(t) = s_{a,r}(t) + is_{a,i}(t) \tag{6.11}$$

then in the equation 6.11, the real and imaginary parts $s_{a,r}(t)$ and $s_{a,i}(t)$ are related through the Kramers-Kronig relation with each other. An intuitive way to analyze this relation is based on expressing its Fourier transform $S_a(\omega)$ as follows,

$$S_a(\omega) = \frac{1}{2}[1 + \text{sgn}(\omega)]S_a(\omega) \tag{6.12}$$

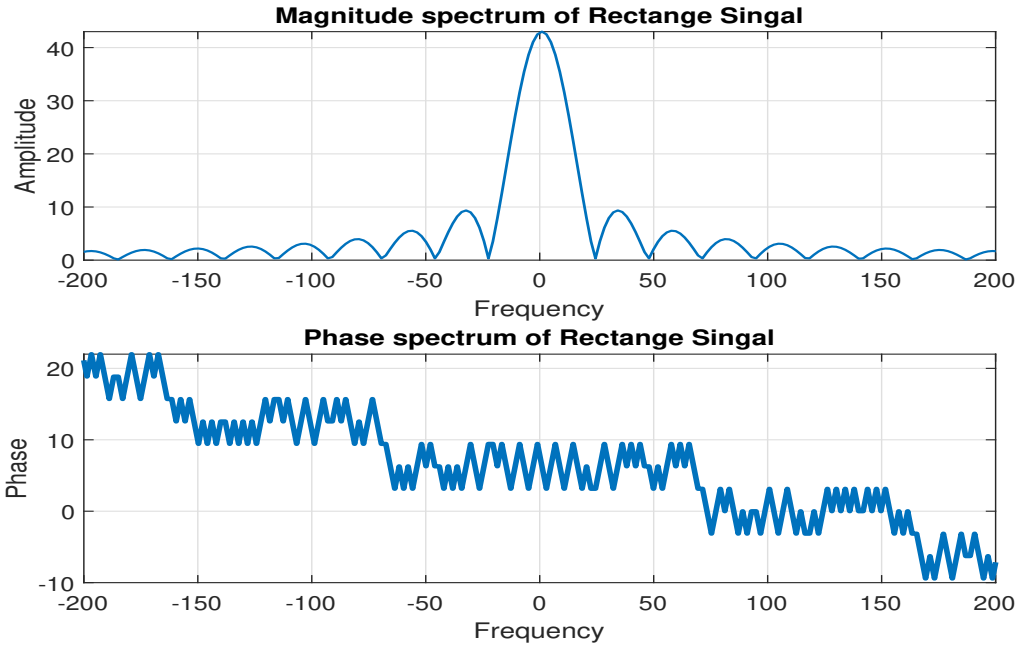


Figure 6.9: Magnitude and phase spectrum of $R(f)$

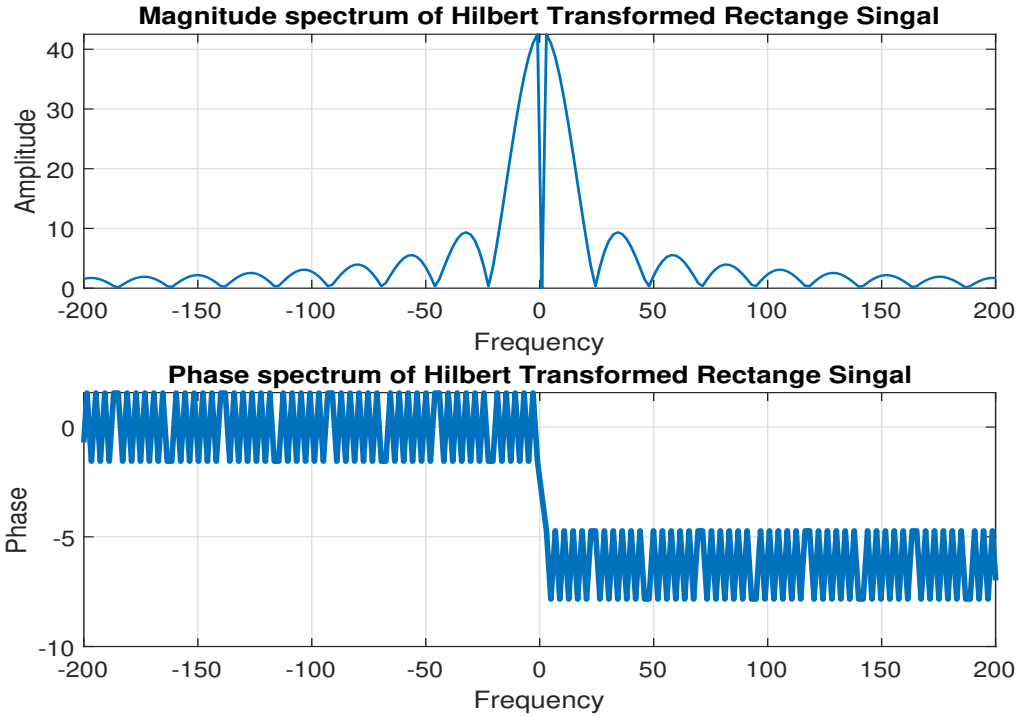


Figure 6.10: Magnitude and phase spectrum of $\hat{R}(f)$

The equation 6.12 follows the SSB signal condition $S_a(\omega) = 0$ for $\omega < 0$. Further, simplification of the signal can be summarized as follows:

$$\begin{aligned} S_a(\omega) &= \frac{1}{2}[1 + \text{sgn}(\omega)]S_a(\omega) \\ &= \frac{1}{2}S_a(\omega) + \frac{1}{2}\,\text{sgn}(\omega)S_a(\omega) \end{aligned} \tag{6.13}$$

Taking inverse Fourier transform of the equation 6.13,

$$\begin{aligned} s_a(t) &= \mathcal{F}^{-1}\{S_a(\omega)\} \\ &= \frac{1}{2}s_a(t) + \underline{\frac{1}{2}[\mathcal{F}^{-1}\{\,\text{sgn}(\omega)\} \circledast s_a(t)]} \end{aligned} \tag{6.14}$$

The underlined term in Equation 6.14 displays that multiplication in frequency domain converted into the convolution in the time domain. Further, IFT of the function $sgn(\omega)$ given as $(-i/\pi t)$. As a consequence, we can further simplify our equation as,

$$\begin{aligned} s_a(t) &= \frac{1}{2}s_a(t) + \frac{1}{2}\left[\frac{i}{\pi t} \circledast s_a(t)\right] \\ \frac{s_a(t)}{2} &= \frac{1}{2}\left[\frac{i}{\pi t} \circledast s_a(t)\right] \\ s_a(t) &= i\left[\frac{1}{\pi t} \circledast s_a(t)\right] \\ s_a(t) &= \frac{i}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_a(t')}{t-t'}dt' \end{aligned} \tag{6.15}$$

Using Equation 6.11 into Equation 6.15,

$$s_{s,r}(t) + i\,s_{a,i}(t) = \frac{i}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_a(t')}{t-t'}dt' \tag{6.16}$$

Therefore,

$$\begin{aligned} s_{s,r}(t) + i\,s_{a,i}(t) &= \frac{i}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_{a,r}(t') + i\,s_{a,r}(t')}{t-t'}dt' \\ s_{s,r}(t) + i\,s_{a,i}(t) &= -\frac{1}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_{a,i}(t')}{t-t'}dt' + \frac{i}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_{a,r}(t')}{t-t'}dt' \end{aligned} \tag{6.17}$$

which leads to,

$$\begin{aligned} s_{a,r}(t) &= -\frac{1}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_{a,i}(t')}{t-t'}dt' \\ s_{a,i}(t) &= \frac{1}{\pi}p.v.\int_{-\infty}^{\infty}\frac{s_{a,r}(t')}{t-t'}dt' \end{aligned} \tag{6.18}$$

In conclusion, the signal with its $S_a(\omega) = 0$ for $\omega < 0$ (i.e analytical in the upper half of the complex plane) satisfies the Kramers-Kronig relationship.

3. What is an SSB signal and how it can be generated?

This section will represent the brief idea of generating SSB signal using Hilbert transform method. To understand this, we may express signal $s(t)$ as a summation of the two complex-valued functions.

$$s(t) = \frac{1}{2}[s(t) + i\hat{s}(t)] + \frac{1}{2}[s(t) - i\hat{s}(t)] \tag{6.19}$$

From Equation 9.15,

$$s(t) = s_a(t) + is_a^*(t) \tag{6.20}$$

where $s_a^*(t)$ is the complex conjugate of $s_a(t)$. Such representation of $s_a(t)$ and $s_a^*(t)$ divide the signal into non-negative frequency component and non-positive frequency component respectively. Considering only non-negative frequency $s_a(t)$ part, we can write it as

$$\frac{1}{2}S_a(f) = \begin{cases} S(f) & \text{for } f > 0 \\ 0 & \text{for } f < 0 \end{cases} \tag{6.21}$$

where $S_a(f)$ and $S(f)$ are the Fourier transform of $s_a(t)$ and $s(t)$ respectively. The frequency translated version of $S_a(f - f_0)$ contains only one side (positive) of $S(f)$ and hence it is called single sideband signal $s_{ssb}(t)$,

$$F^{-1}\{S_a(f - f_0)\} = s_a(t)e^{i2\pi f_0 t} \tag{6.22}$$

Therefore, from the Euler's formula,

$$\begin{aligned} s_{ssb}(t) &= Re\{s_a(t)e^{i2\pi f_0 t}\} \\ &= Re\{[s(t) + i\hat{s}(t)][cos(2\pi f_0 t) + isin(2\pi f_0 t)]\} \\ &= s(t)cos(2\pi f_0 t) - \hat{s}(t)sin(2\pi f_0 t) \end{aligned} \tag{6.23}$$

This Equation 6.23 displays the mathematical modeling of the upper sideband SSB signal. Similarly, we can generate lower sideband SSB signal by,

$$s_{ssb}(t) = s(t)cos(2\pi f_0 t) + \hat{s}(t)sin(2\pi f_0 t) \tag{6.24}$$

Discrete Hilbert transform

The equation 6.6 is extended from $-\infty$ to ∞ in the frequency domain. In the time domain the number of points must be limited (say, to $2M+1$), which is equivalent to adding rectangular window to the input signal. The Z-transform of windowed $h(nt_s)$ is given as,

$$H(z) = \sum_{n=-M}^{M} h(nt_s)z^{-n} \tag{6.25}$$

By definition, the equation is not causal because summation starts from a negative value. In order to implement it into the practice, the equation 6.25 must be made causal. The FIR

design scheme can be used to achieve the discrete Hilbert transform.
1: Write the z transform of the function $h(nt_s)$ as,

$$\begin{aligned} H(z) &= \sum_{n=-\infty}^{\infty} h(nt_s)z^{-n} \\ &= \sum_{n=\infty}^{-} 1h(nt_s)z^{-n} + h(0) + \sum_{n=1}^{\infty} h(nt_s)z^{-n} \\ &= h(0) + \sum_{n=1}^{\infty}[h(-nt_s)z^{n} + h(nt_s)z^{-n}] \end{aligned} \tag{6.26}$$

Substituting $z = exp(i2\pi ft_s)$, the result can be written as,

$$\begin{aligned} H(e^{i2\pi ft_s}) &= H_r(e^{i2\pi ft_s}) + iH_i(e^{i2\pi ft_s}) \\ &= \sum_{n=-\infty}^{\infty} h(nt_s)e^{i2\pi nft_s} \\ &= h(0) + \sum_{n=1}^{\infty}[h(-nt_s)cos(2\pi nft_s) + ih(-nt_s)sin(2\pi nft_s) + \\ & h(nt_s)cos(2\pi nft_s) + ih(nt_s)sin(2\pi nft_s)] \end{aligned} \tag{6.27}$$

where,

$$\begin{aligned} H_r(e^{i2\pi ft_s}) &= h(0) + \sum_{n=1}^{\infty}[h(-nt_s) + h(nt_s)]cos(2\pi nft_s) \\ H_i(e^{i2\pi ft_s}) &= \sum_{n=1}^{\infty}[h(-nt_s) + h(nt_s)]sin(2\pi nft_s) \end{aligned} \tag{6.28}$$

2: When the sampling frequency is f_s, the transfer function $H(f)$ is limited to the bandwidth $\frac{f_s}{2}$. Due to the periodic property of sampling, the Hilbert transfer function is actually as shown in Figure 7.9. This can be represented by Fourier series as,

$$H_i(e^{i2\pi ft_s}) = \sum_{n=1}^{\infty} b_n sin(2\pi nft_s) \tag{6.29}$$

$$\begin{aligned} b_n &= \frac{2}{f_s}\int_{-f_s/2}^{f_s/2} H(e^{i2\pi nft_s})sin(2\pi nft_s)df \\ &= \frac{2}{f_s}\left[\int_{-f_s/2}^{0} sin(2\pi nft_s)df + \int_{0}^{f_s/2} sin(2\pi nft_s)df\right] \\ &= \frac{1}{n\pi}[-2 + 2cos(n\pi)] \\ &= \begin{cases} 0 & \text{n = even} \\ \frac{-4}{n\pi} & \text{n = odd} \end{cases} \end{aligned} \tag{6.30}$$

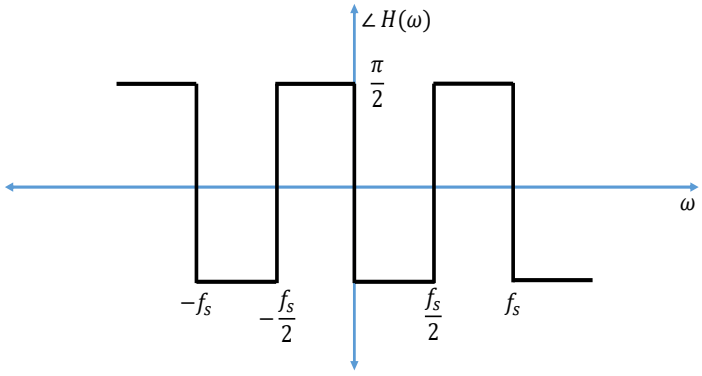


Figure 6.11: Periodic representation of $H(f)$

In the above equation, the relation of $f_s t_s = 1$ is used.
3: The equation 6.6 has only the imaginary part, therefore, $H_r(f) = 0$ and $H_i(f) \neq 0$. This condition can be fulfilled if,

$$h(n) = 0 \text{and} h(-nt_s) = -h(nt_s) \tag{6.31}$$

Using this relationship, H_i in equation 6.28 can be written as

$$H_i(e^{j2\pi nft_s}) = -2\sum_{n=1}^{\infty} h(nt_s)sin(2\pi nft_s) \tag{6.32}$$

Comparing equations 6.32 and 6.29, we can obtain

$$h(nt_s) = -\frac{b_n}{2} \tag{6.33}$$

Finally, we can compute the impulse response of the discrete Hilbert transform filter as,

$$\begin{aligned} h(nt_s) &= \begin{cases} 0 & \text{n = even} \\ \dfrac{2}{n\pi} & \text{n = odd} \end{cases} \\ h(-nt_s) &= \begin{cases} 0 & \text{n = even} \\ \dfrac{-2}{n\pi} & \text{n = odd} \end{cases} \end{aligned} \tag{6.34}$$

4: As mentioned before, the results obtained in the equations 6.34 are not causal. In order to make them causal, a simple shift in time domain can be used. The value of n in $h(nt_s)$ is windowed from $-M$ to M as shown in equation 6.25. The shift in time domain is equivalent to multiplying the result of equation 6.25 by Z^{-M} and substituting with $k = n + M$, the new results is,

$$H(z) = \sum_{n=-M}^{M} h(nt_s)Z^{-(n+M)} = \sum_{k=0}^{2k} h(kt_s - Mt_s)Z^{-k} \tag{6.35}$$

Graphical explanation SSB signal generation

This section describes the generation of SSB signal using Hilbert transformation method (Phase Shift Method). Consider a message signal $m(t)$ with its frequency domain spectrum $|M(F)|$ as shown in Figure6.12. From the Figure 6.12, we can see that both the side are scaled by factor '1' which means it represents the original signal.

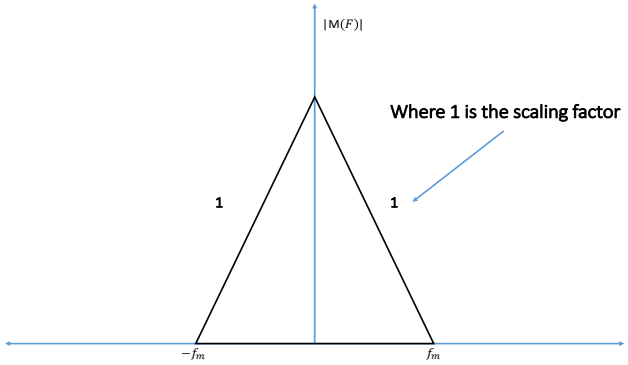


Figure 6.12: An original baseband signal

Now let's consider the modulated signal $x(t)$ given as,

$$x(t) = m(t)cos(2\pi f_c t) \tag{6.36}$$

Frequency domain representation (see Figure 6.13) of the equation 6.36 can be given as,

$$X(F) = \frac{1}{2}M(f - f_c) + \frac{1}{2}M(f + f_c) \tag{6.37}$$

Here in equation 6.37, we can observe that each sideband is scaled by $\frac{1}{2}$ on the frequency spectrum. The figure displays the frequency domain representation of the modulated signal $X(F)$.

Next, we will discuss something more interesting which is called as a Hilbert transform of the original message signal $m(t)$. As we discussed earlier, in the frequency domain, the Hilbert transformed signal $\hat{M}(f)$ can be achieved by multiplying the Fourier transformed signal $M(F)$ with $[-i\text{sgn}(F)]$ (see Figure 6.14). Suppose we modulate the Hilbert transformed message signal $\hat{m}(t)$ with the $sin(2\pi f_c t)$ (quadrature phase carrier), then we get the following results:

$$\begin{aligned}
\hat{m}(t)sin(2\pi f_c t) &= \hat{m}(t)\frac{e^{i2\pi f_c t} - e^{-i2\pi f_c t}}{2} \\
&= \hat{m}(t)\frac{e^{i2\pi f_c t}}{2} - \hat{m}(t)\frac{e^{-i2\pi f_c t}}{2} \\
&= \frac{\hat{M}(f - f_c)}{2i} - \frac{\hat{M}(f + f_c)}{2i} \\
&= \frac{-i}{2}\hat{M}(f - f_c) + \frac{i}{2}\hat{M}(f + f_c)
\end{aligned} \tag{6.38}$$

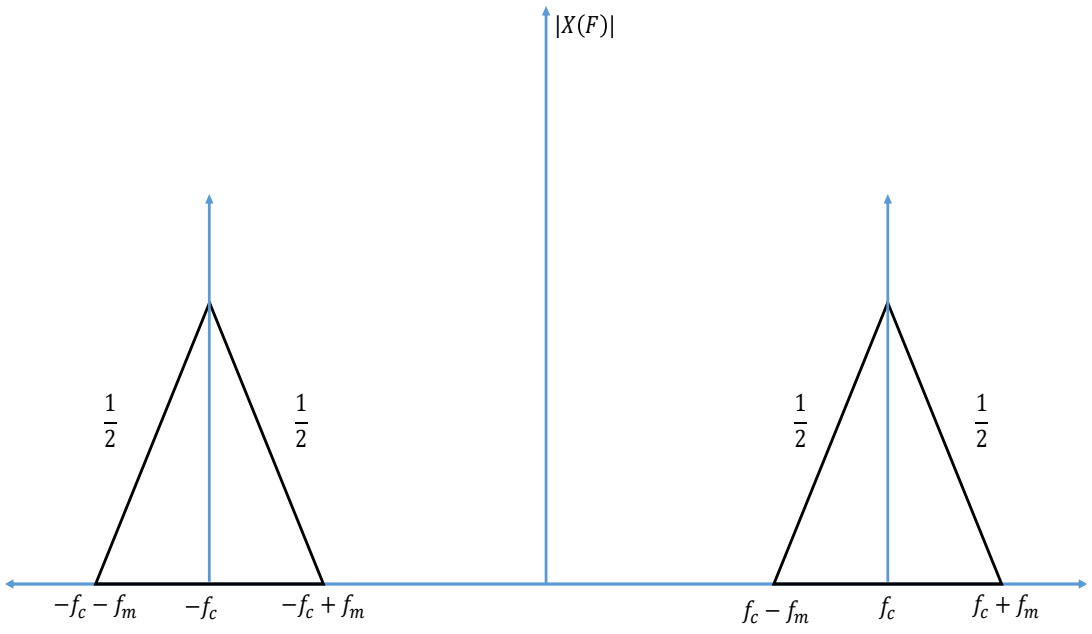


Figure 6.13: Frequency spectrum of modulated signal $X(F)$

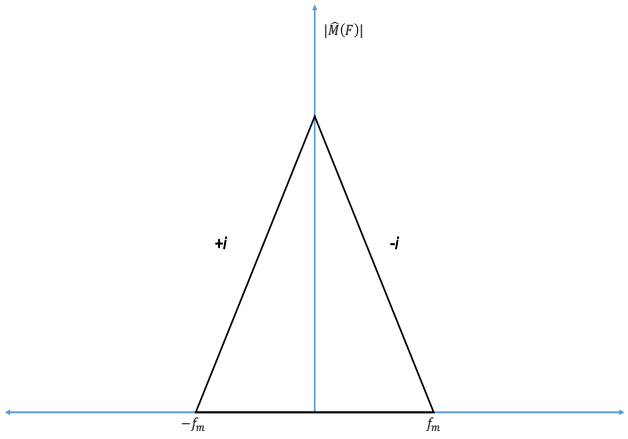


Figure 6.14: Spectrum of Hilbert transformed message signal $\hat{M}(F)$

The detailed explanation of the equation 6.38 has been given in the Figure 6.15 and 6.16. Figure 6.15 displays the spectrum of the $\hat{M}(f + f_c)$ and $\hat{M}(f - f_c)$ for the positive and negative frequencies respectively. The final equation resolution of equation displays that both positive and negative side of the spectrum multiplied with $\frac{i}{2}$ and $\frac{-i}{2}$ respectively. Finally, the spectrum of the signal $\hat{m}(t)sin(2\pi f_c t)$ can be given as Figure 6.16.

Further, summation of the two signals $m(t)cos(2\pi f_c t)$ and $\hat{m}(t)sin(2\pi f_c t)$ will generate

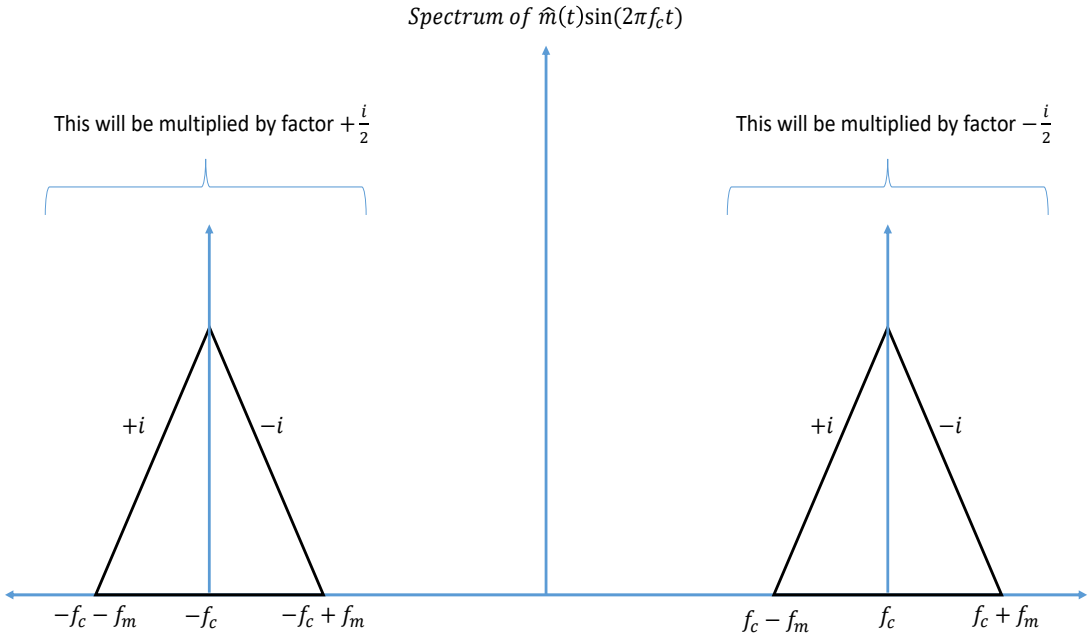


Figure 6.15: Hilbert transformed modulated signal

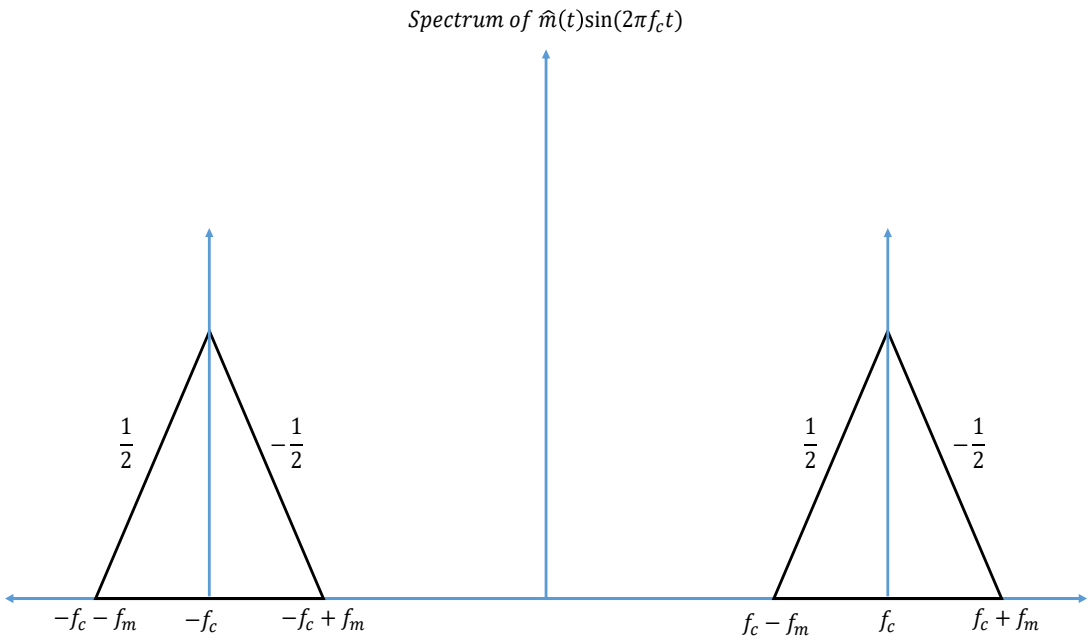


Figure 6.16: Hilbert transformed modulated signal

the upper sideband SSB signal as follows,

$$u(t) = m(t)cos(2\pi f_c t) - \hat{m}(t)sin(2\pi f_c t) \tag{6.39}$$

From the above discussion, the spectrum of the Equation 6.39 can be given by the Figure 6.17. Similarly, for the lower sideband SSB can be generated by Equation,

$$u(t) = m(t)cos(2\pi f_c t) + \hat{m}(t)sin(2\pi f_c t) \tag{6.40}$$

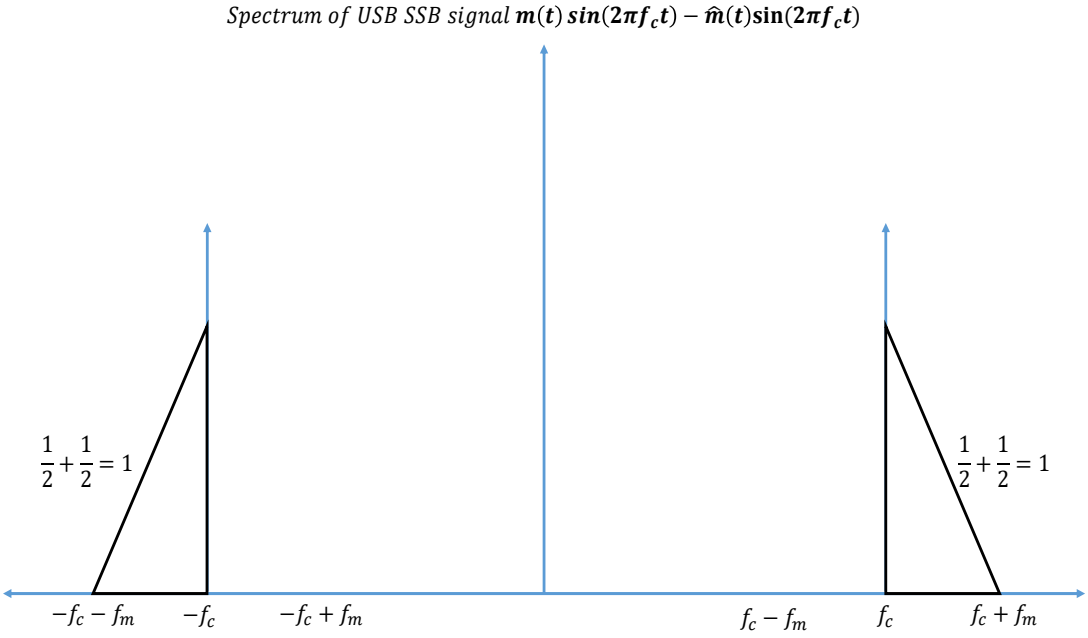


Figure 6.17: SSB signal spectrum

4. What is minimum phase signal how we can profit from it?

A necessary and sufficient condition for a complex signal $E(t)$ to be minimum phase is that the curve described in a complex plane by $A(t)$ when $t \to -\infty$ to $t \to \infty$ does not encircle the origin [4] . A minimum-phase signal has a useful property that the natural logarithm of the magnitude of the frequency response is related to the phase angle of the frequency response by the Hilbert transform.
For instance, if we consider a complex data-carrying signal $E_s(t)$ with its optical bandwidth B and the LO is assumed to be a continuous-wave (CW) signal with its amplitude of E_o and the frequency which coincides with the left edge of the information-carrying signal spectrum and the constructed signal $E(t)$ as,

$$E(t) = E_o exp(i\pi Bt) + E_s(t) \tag{6.41}$$

where E_o is a constant; Here, $E(t)$ is a minimum phase if and only if the winding number of its trajectory into the complex plane is zero i.e. the representation of the signal in the complex plane do not circle the origin. The condition $|E_o| > |E_s(t)|$ is sufficient for guaranteeing minimum phase property [3]. Therefore, when $E(t)$ is minimum phase signal, its phase $\phi(t)$ can be reconstructed from its magnitude $|E(t)|$ by means of using Kramers-Kronig relationship. In other words, E_o should be large enough to ensure that the signal presented by Equation 6.42 becomes a minimum phase signal.

$$E(t)exp(-i\pi Bt) = E_o + E_s(t)exp(i\pi Bt) \tag{6.42}$$

Once the minimum phase condition is satisfied for the received optical signal then its phase can be constructed using its intensity and the original signal can be reconstructed in the following manner.

$$E_s(t) = \left[\sqrt{I}\ exp[i\phi_E(t)] - E_o\right] exp(i\pi Bt) \tag{6.43}$$

$$\phi_E(t) = \frac{1}{2\pi}\ p.v.\ \int_{-\infty}^{\infty} dt' \frac{log[I(t')]}{t - t'} \tag{6.44}$$

6.2.2 Numerical Validations

This section contains the numerical analysis of the Kramers-Kronig transceiver for both the complex and real valued signal. In the case I, the complex-valued signal is analyzed numerically as shown in Figure 6.18. It shows that first the generated QAM signal passed through the RRC (Root-Raised Cosine) filter and added a CW tone. This signal is then up-converted such that the CW tone coincide with the DC component, this up-converted signal called as minimum phase SSB signal. At the receiver end, the minimum phase signal is direct-detected using a single photo detector and the full complex signal recovered using the KK (Kramers-Kronig) algorithm.

Case I : 16-QAM signal

Consider a 16-QAM complex signal $E_s(t) = I(t) + jQ(t)$ generated using the simulator as shown in Figure 6.19. The detail of the generated signal $I(t)$ and $Q(t)$ are given in the table.

Parameter	**Value**
bitPeriod	1.0 / 30e9
numberOfBits	1000
numberOfSamplesPerSymbol	16
rollOffFactor	0.3

The constellation diagram and the magnitude spectrum of the signal are shown in the Figure 6.20a and 6.20b respectively. The signal $E_s(t)$ is confined within the bandwidth of $\sim$10.5 GHz. A continuous wave (CW) tone is required at the edge (either lower or higher) of the information bandwidth to make the signal to be a minimum phase signal. In this example, a

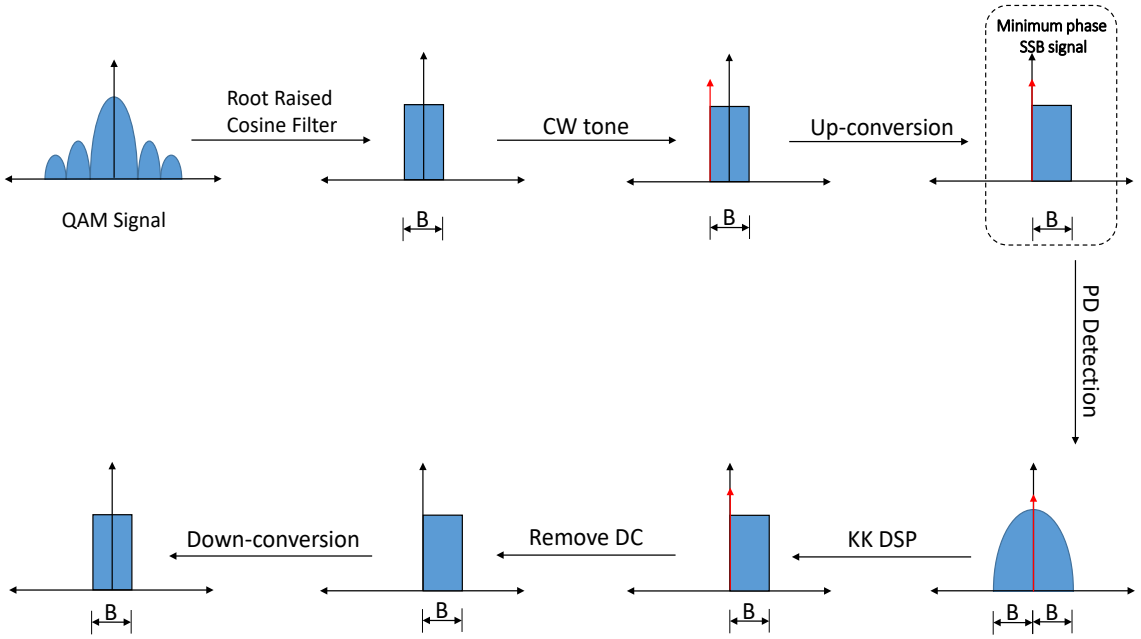


Figure 6.18: Generation and detection minimum phase SSB signal for QAM

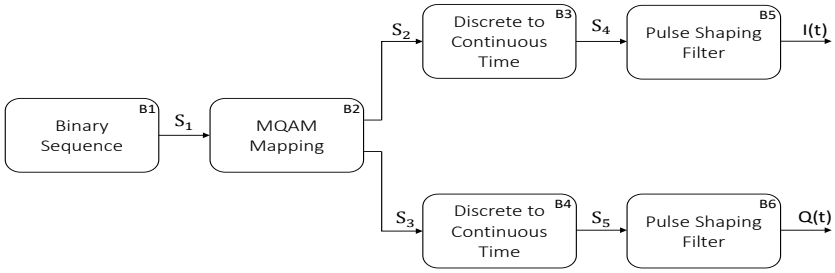


Figure 6.19: set-up to generate a 16-QAM modulation signal

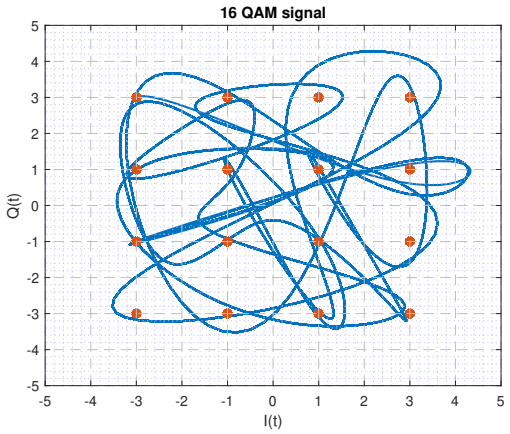
CW tone has been added at the lower edge of the information bandwidth as shown In Figure 6.21b. The mathematical model of the CW tone added signal can be written as,

$$E(t) = E_s(t) + E_o e^{-i\pi Bt} \tag{6.45}$$

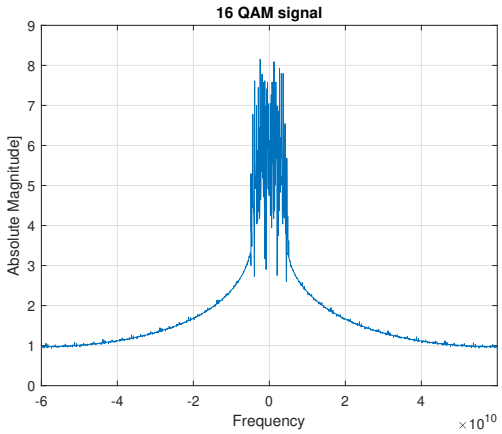
where the frequency of the CW is -5.5 GHz (half of the information bandwidth) and amplitude $E_o \geq |E_s(t)|$ (It should be greater than 6dB of the information spectrum).
The frequency shifted version (frequency up-converted signal) of the minimum phase signal can be written in the following form.

$$E(t)\, e^{i\pi Bt} = E_s(t)\, e^{i\pi Bt} + E_o \tag{6.46}$$

Figure 6.22 displays that the real part of the frequency shifted signal is non-negative. As long as this non-negativity of the real part is preserved, the signal can be fully recovered from its

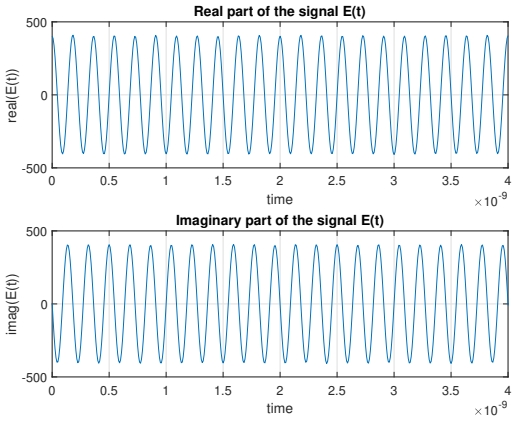


(a) Constellation of 16-QAM signal $E_s(t)$

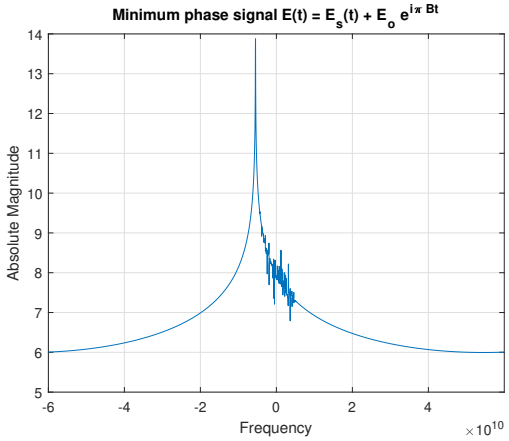


(b) Magnitude spectrum of $E_s(t)$

Figure 6.20: 16-QAM signal $E_s(t) = I(t) + iQ(t)$



(a) Real and imaginary part of the signal $E(t)$

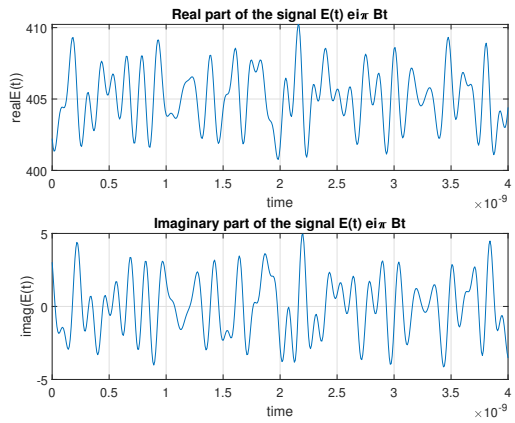


(b) Magnitude spectrum of $E(t)$

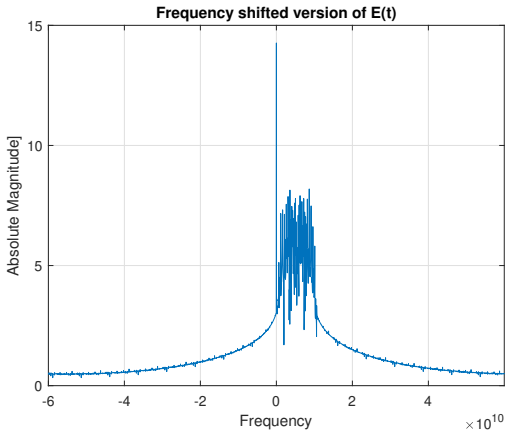
Figure 6.21: 16-QAM minimum phase signal $E(t) = E_s(t) + E_o\, e^{-i\pi Bt}$

intensity. Therefore, if we sample the optical signal with the sampling frequency equals to the twice of the information signal bandwidth then we can achieve the resulting spectrum similar to 6.22b and we can recover the full spectrum using its magnitude value.

Next, if we take the square of the modulus of the signal as $|real\{E(t)\, e^{i\pi Bt}\}|^2$ then the resultant output is as shown in Figure 6.23. By employing Kramers-Kronig algorithm on the $|real\{E(t)\, e^{i\pi Bt}\}|^2$, we can recover the full $E(t)e^{i\pi Bt}$ as shown in Figure 6.24. Next, the signal is down-converted (see Figure 6.25) and then the CW tone jas been removed (see Figure 6.26) to recover the original complex signal $E_s(t) = I(t) + iQ(t)$.

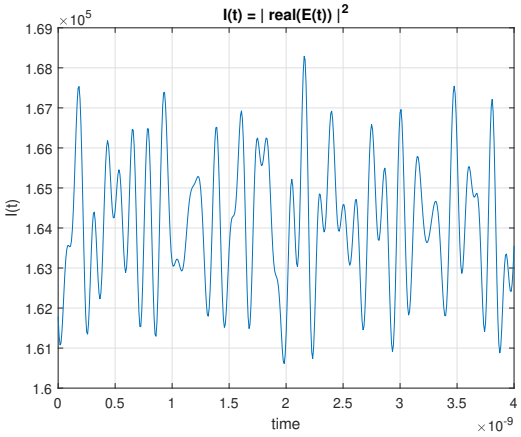


(a) Real and imaginary part of the signal $E(t)\, e^{i\pi Bt}$

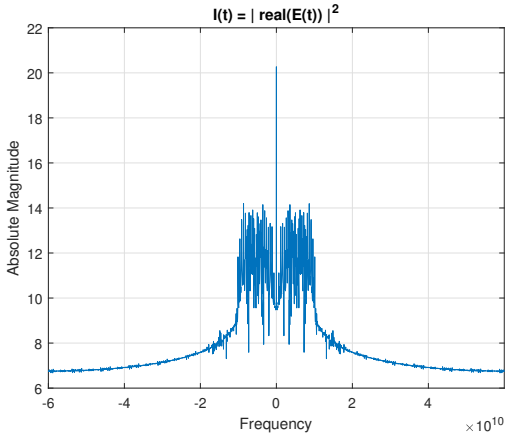


(b) Magnitude spectrum of $E(t)\, e^{i\pi Bt}$

Figure 6.22: Frequency shifted version of $E(t)$ as $E(t)\, e^{i\pi Bt} = E_s(t)\, e^{i\pi Bt} + E_o$



(a) absolute value $|real\{E(t)\, e^{i\pi Bt}\}|^2$

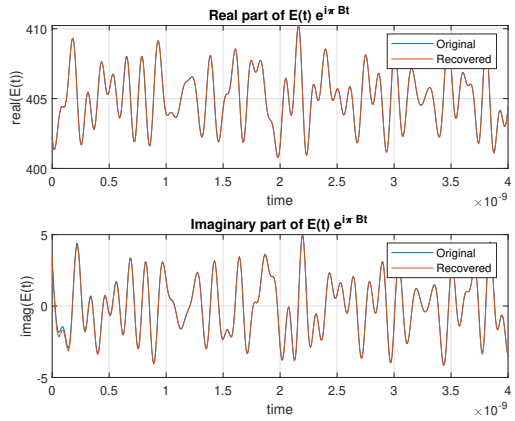


(b) Magnitude spectrum of $|real\{E(t)\, e^{i\pi Bt}\}|^2$

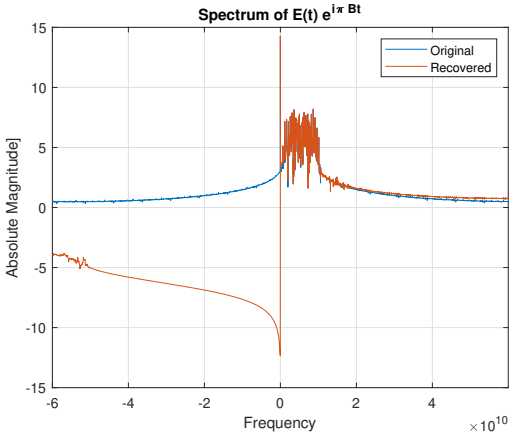
Figure 6.23: Frequency shifted minimum phase signal $E(t)\, e^{i\pi Bt}$

Case II : PAM signal

In case of PAM signal, we can use the idea displayed in the Figure 6.27 to generate minimum phase SSB signal. The figure displays the idea of generating SSB signal using the Hilbert transformation method. The same thing can be achieved by using the SSBOF (Single Side Band Optical Filter) which makes use of the simplicity of amplitude modulation instead of using IQ modulator in the Hilbert filter method. Therefore, in the real-valued signal, a less complex configuration has been recently proposed [5] which facilitates to generating minimum phase SSB signal (see Figure 6.35).
Consider a 4-PAM real signal $E_s(t) = I(t)$ generated using the simulator as shown in Figure 6.28. The detail of the signal $I(t)$ is given in the following table.

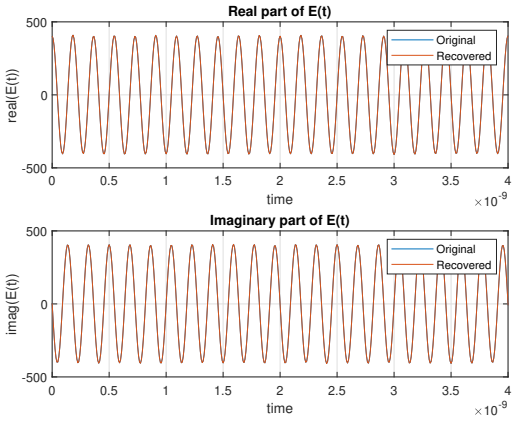


(a) Comparison of real and imaginary part

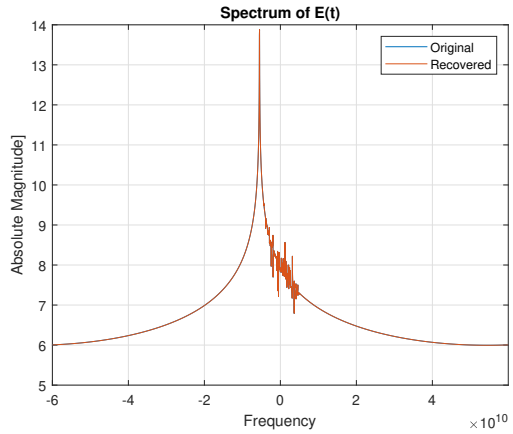


(b) Comparison of the magnitude spectrum

Figure 6.24: Comparison between original and recovered $E(t)\,e^{i\pi Bt}$



(a) Comparison of real and imaginary part



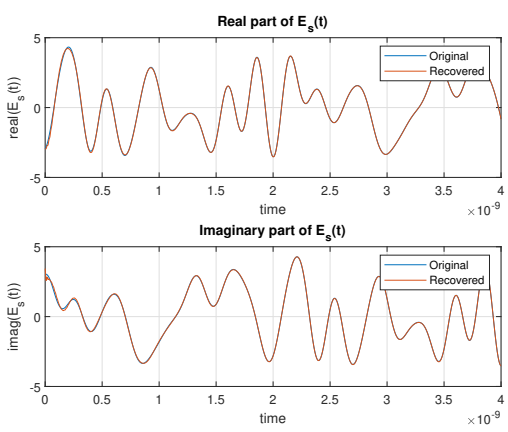
(b) Comparison of magnitude spectrum

Figure 6.25: Comparison between original and recovered $E(t)\,e^{i\pi Bt}$

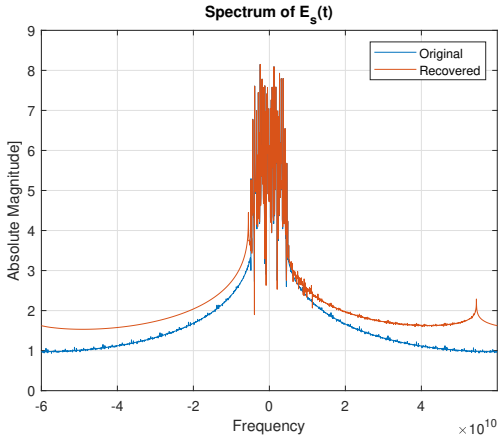
Parameter	**Value**
bitPeriod	1.0 / 30e9
numberOfBits	1000
numberOfSamplesPerSymbol	16
rollOffFactor	0.3

The baseband signal $I(t)$ and its magnitude spectrum are displayed as shown in Figure 6.29a and 6.29b respectively. The signal is confined within the bandwidth of approximately ∼22 GHz. A DC bias is applied to the signal $E_s(t)$ to ensure the signal becomes non-negative. The mathematical model of the signal can be written as,

$$E_{nn}(t) = I(t) + E_o \tag{6.47}$$



(a) Comparison of real and imaginary part



(b) Comparison of magnitude spectrum

Figure 6.26: Comparison between original and recovered $E_s(t)$

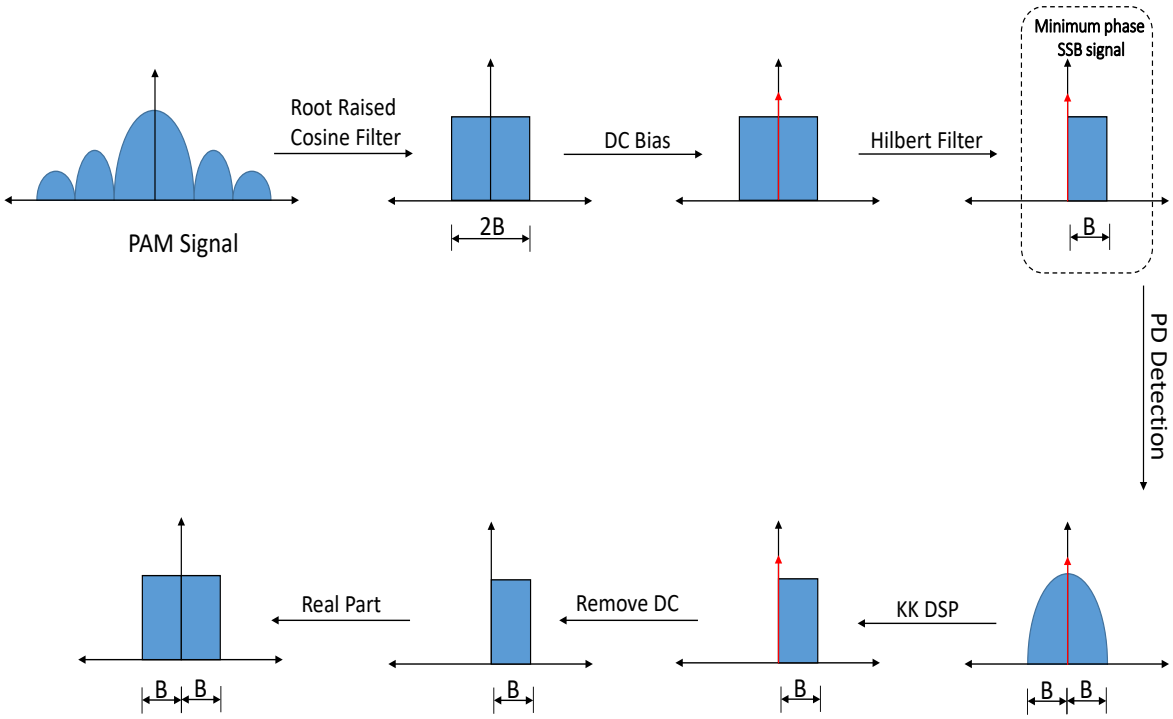


Figure 6.27: Generate minimum phase SSB of PAM signal

where E_o is the DC bias which should be large enough to ensure that the signal should become non-negative as shown in Figure 6.30. Next, this signal has been passed through the Hilbert filter which generates the minimum phase analytical SSB signal (Practically same thing can be achieved by using SSBOF) as shown in Figure 6.31. The mathematical model of

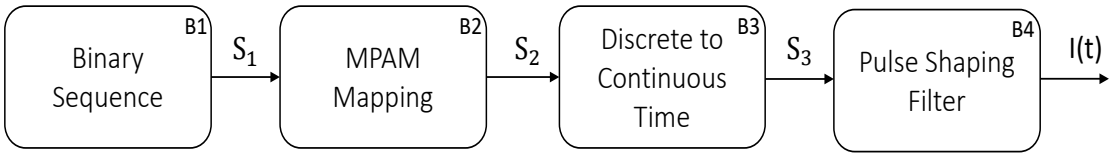
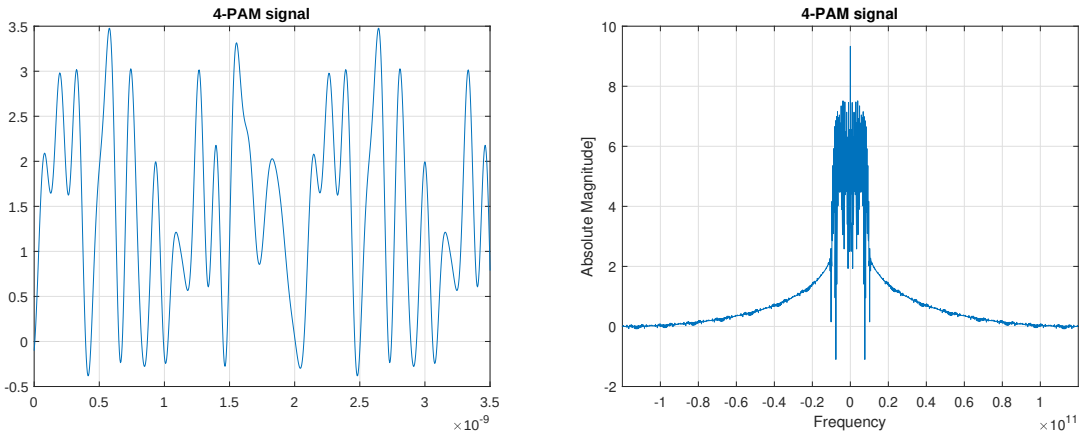


Figure 6.28: set-up to generate 4-PAM modulation signal

the minimum phase SSB signal can be written as,

$$E(t) = E_s(t) + E_o \tag{6.48}$$

where the complex signal given as $E_s(t) = I(t) + i \cdot H\{I(t)\}$, and $H\{I(t)\}$ represents the

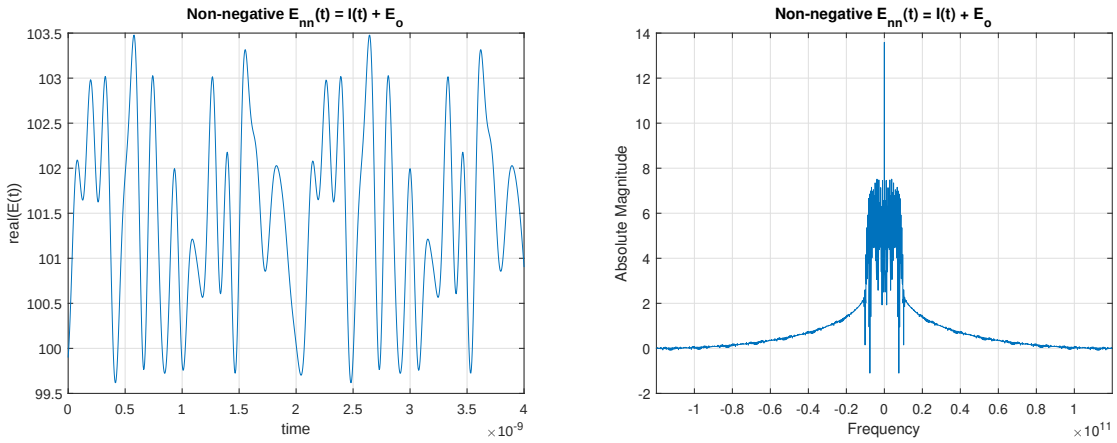


(a) Real-valued 4-PAM signal

(b) Magnitude spectrum

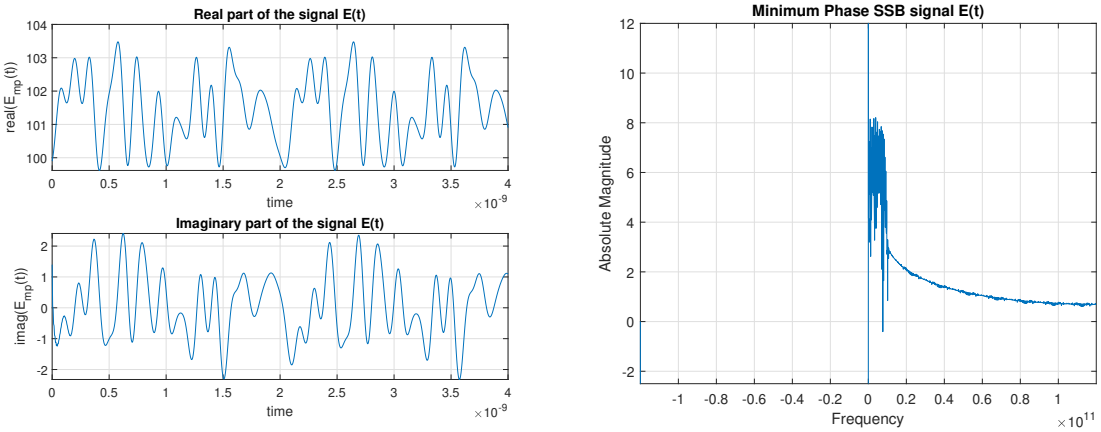
Figure 6.29: 4-PAM signal $E_s(t) = I(t)$

Hilbert transformed of the signal $I(t)$. This SSB minimum phase signal $E(t)$ is launched into the optical fiber for the transmission. At the receiver end, a photo detector will detect the intensity of the signal $|real\{E(t)\}|^2$ (see Figure 6.32). By employing the Kramers-Kronig algorithm on the $|real\{E(t)\}|^2$, we can recover the full complex signal $E(t)$ as shown in Figure 6.33. From the recovered complex signal, we can select only real part and removing the DC bias content, we can recover the signal $I(t)$ as shown in Figure 6.34.



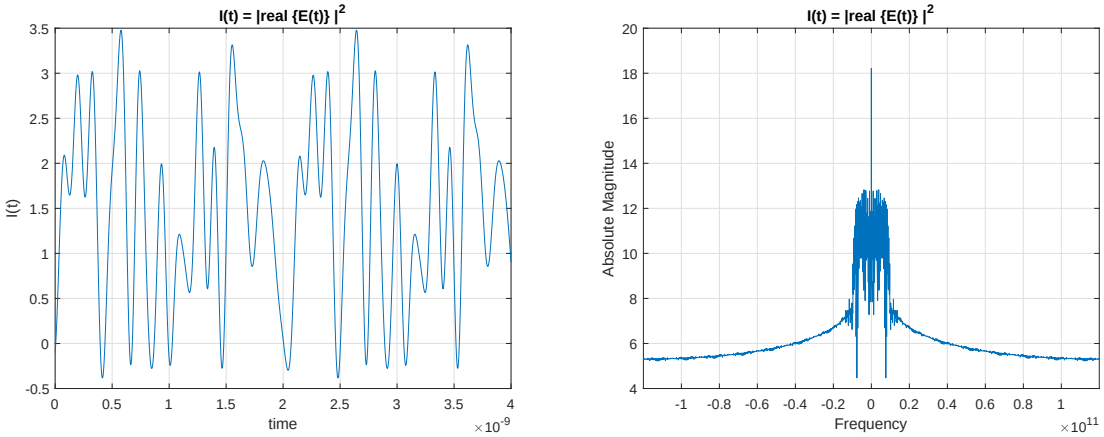
(a) Non-negative 4-PAM signal (b) Magnitude spectrum

Figure 6.30: Non-negative real-valued information signal $E(t)$



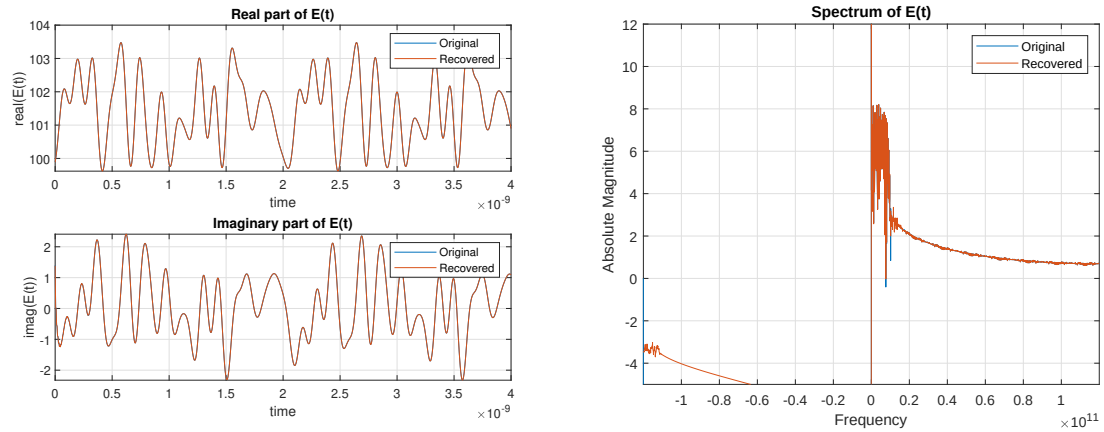
(a) Minimum phase SSB signal (b) Magnitude spectrum

Figure 6.31: SSB minimum phase signal $E(t)$



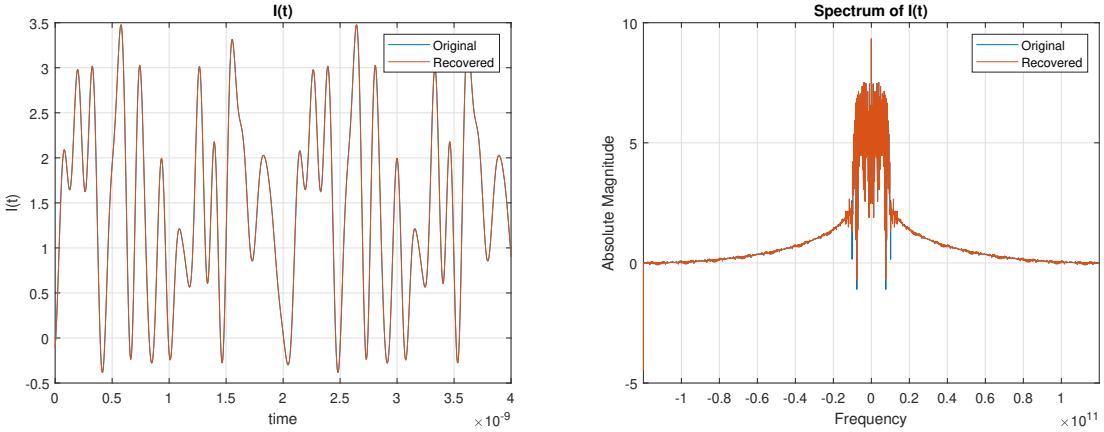
(a) Minimum phase SSB signal (b) Magnitude spectrum

Figure 6.32: Detected SSB minimum phase signal $I(t) = |real\{E(t)\}|^2$



(a) Minimum phase SSB signal (b) Magnitude spectrum

Figure 6.33: Recovered non-negative signal $E(t)$



(a) Minimum phase SSB signal (b) Magnitude spectrum

Figure 6.34: Comparison between original and recovered signal $I(t)$

Transceiver configurations of KK detection

This section contains the various possible configuration for the transmitter set-up to generate the minimum phase SSB signal, however, the receiver architecture will remain same irrespective to the transmitter set-up. The proposed transmitter configurations rely on the type of signal to be used either a real-valued or a complex-valued and it's also based on the complexity of generating the minimum phase SSB signal.

1. Transmitter configurations for real-valued signal

In the quest for the cost-effectiveness and highly performing transmission solutions, a simplified transmitter setup has been proposed which helps to transmit the real-valued signal for the KK detection [5]. The transmitter setup in configuration 1 is shown in the Figure 6.35 which consists of a laser followed by a single-drive MZM and a sharp optical filter (OF) suppressing one of the optical sidebands. The modulator is driven by an arbitrary waveform generator or we can feed any real-valued signal, i.e. an RF signal to it. The MZM bias level was set in the vicinity of the middle between the quadrature and null-points. The driving signal amplitude was adjusted so that it never crosses the MZM null-point, thus preventing the optical field from assuming negative values. This setup offers the reduced complexity and low cost transmitter configuration that relies on the amplitude modulation using a single drive MZM and a single-sideband optical filtering using SSBOF. However, it demands the stringent configurations of the SSBOF with a very steep roll-off of $\sim$ 40dB/0.1 nm.

On the other hand, transmitter configuration 2 allows to generating SSB minimum phase

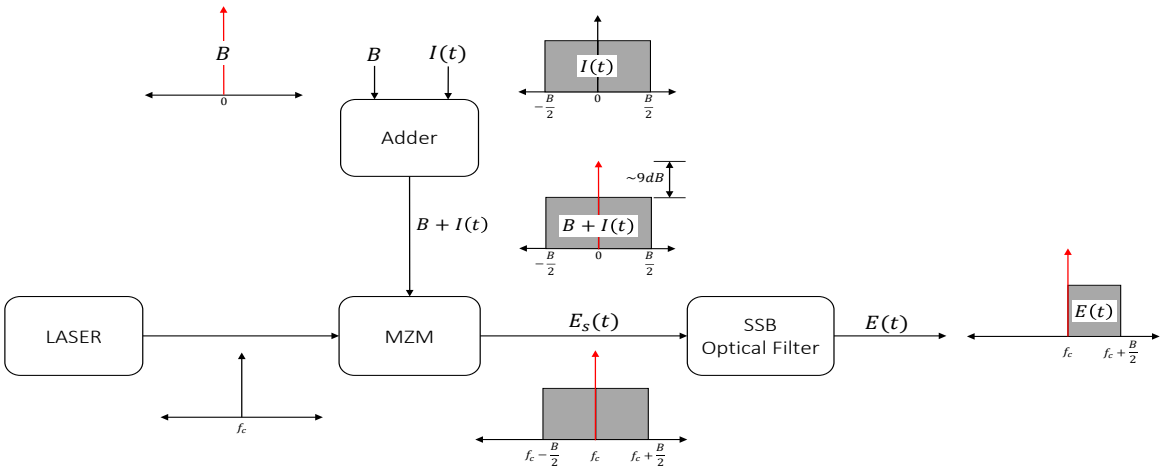


Figure 6.35: Transmitter setup for the real-valued signal : Configuration 1

signal with the help of the Hilbert transformation method. This scheme is based on separately modulating two field quadratures. One field quadrature contains a non-negative PAM signal, where as the other contains its Hilbert transform, such that the overall fields that is combined from the two quadrature is single-sideband minimum phase signal. It is worth

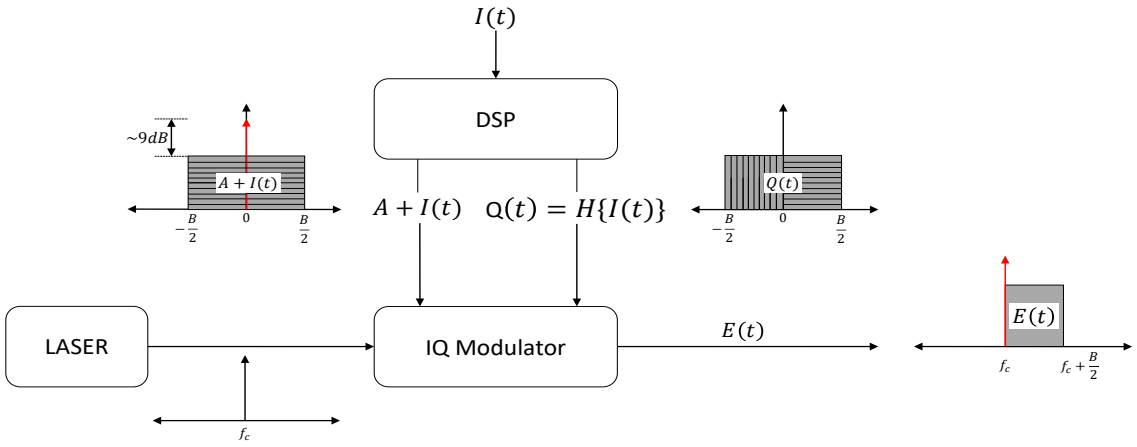


Figure 6.36: Transmitter setup for the real-valued signal : Configuration 2 (H{·} represents the Hilbert transformation)

mentioning that the resultant SSB signal is a minimum phase because the real part $A + I(t)$ is non-negative, where A is the DC bias which should be high enough to ensure the non-negativity of the signal. The quadrature part of the signal contains the Hilbert transformed version of the $H\{I(t)\}$ as mentioned before in the in the discussion. It is important to mention that the Hilbert transform of $H\{I(t)\}$ or $H\{A + I(t)\}$ will led to the same result since the Hilbert transform of any signal does not reflect the DC part in the resultant signal (see Equation 6.6).

2. Transmitter configurations for complex-valued signal

The various transmitter configurations for the KK detection scheme are shown in the Figure 6.37 and 6.38. In the configuration 1, the transmitter consists of a laser and an IQ modulator fed by two DACs for data generation and modulation. The CW tone can be generated optically. In order to generate a CW tone, we split a portion of unmodulated laser carrier and pass it through the frequency shifter (Δf) and added it with the IQ modulated signal as shown in the Figure 6.37. The CW tone does not need to be synchronized with the signal in order work with the KK algorithm.
Since the method displayed in Figure 6.37 involves the complexity of optical domain, an alternative method (See Figure 6.38) was proposed where the CW tone added in the digital domain to get rid of the frequency shifter. In the proposed configuration 2, the CW tone with an amplitude E_o and its frequency approximately coincides with the lower-frequency edge of the information carrying bandwidth added in the digital domain to both $I(t)$ and $Q(t)$ channels. Next, these digital signals are converted into the analog form using DAC and supplied to the IQ modulator as shown in Figure 6.38. Generating and adding the tone digitally before DACs reduces the DAC resolution for the information bearing signal and affects both attainable interface rate and spectral efficiency [6]. The following table displays the main difference between all the transmitter configurations to generating the minimum

phase SSB signal:

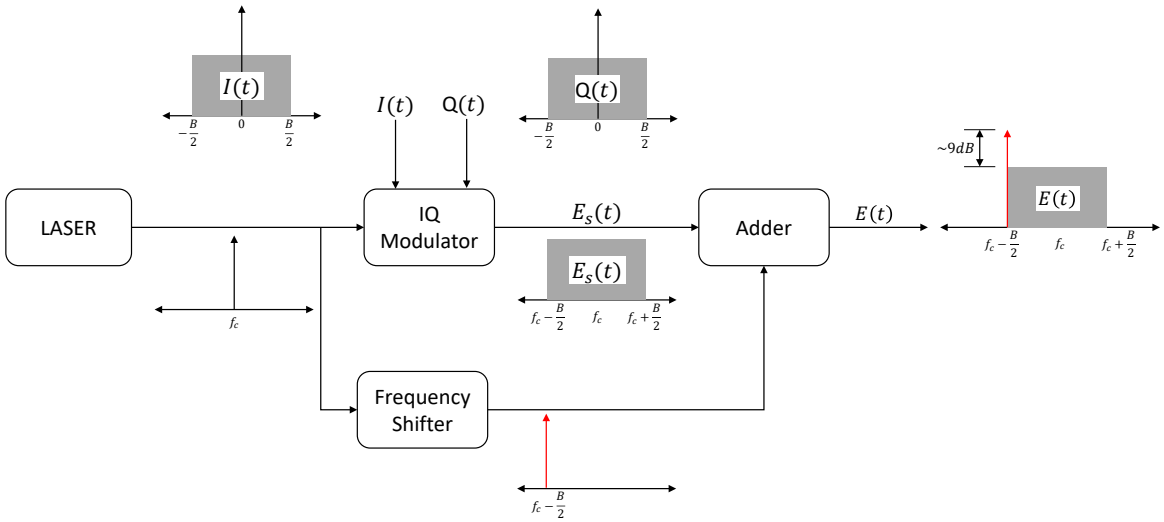


Figure 6.37: Transmitter setup : Configuration 3

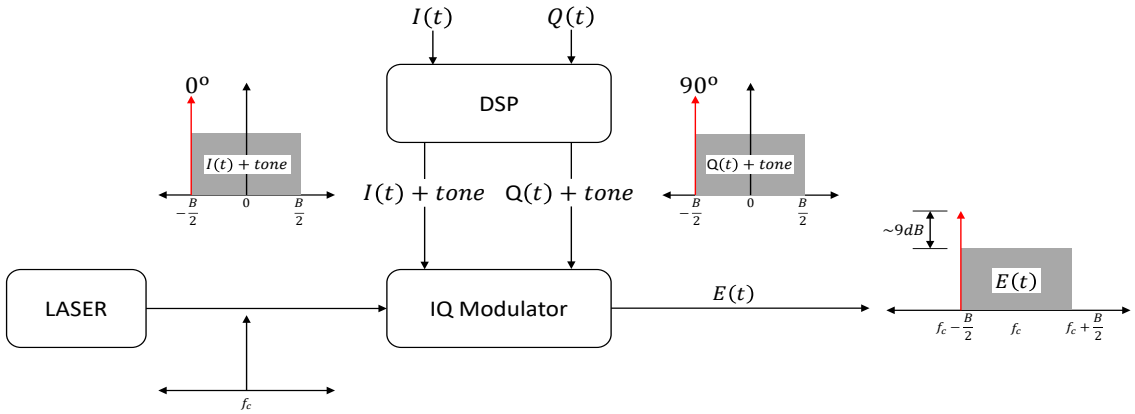


Figure 6.38: Transmitter setup : Configuration 4

Method	**Signal Type**	**Hardware**	**Disadvantage**
Config. 1 (Optical)	Real	Adder, MZM, Optical filter	Requires sharp optical filter
Config. 2 (Digital)	Real	DSP + Two DACs, IQ modulator	Requires Hilbert transform at Tx
Config. 3 (Optical)	Complex	Two DACs, IQ Modulator, frequency shifter, and adder	Optical complexity
Config. 4 (Digital)	Complex	DSP + Two DACs, IQ Modulator	Requires adding a tone in the Tx DSP

Receiver architecture for the KK detection

At the receiver end, the signal is detected by direct detection method and full complex envelope recovered by employing the Kramers-Kronig algorithm. The dotted box in the Figure 6.39 displays the DSP of the kramres-Kronig algorithm. First, the square root data of the photo-detected signal is calculated and then converted into the frequency domain after computing its logarithmic value. In the frequency domain, we can most conveniently apply the Hilbert transformation algorithm. Here, sgn(ω) is the sign function, which is equal to 1 for $\omega > 0$, and and 0 for the $\omega = 0$ and -1 for $\omega < 0$. Finally, the Hilbert transformed signal is converted into the time domain and then its exponential value calculated to get the phase information of the detected signal. Finally, we can recover the full complex signal and applied to the post-processing section to recover the data.

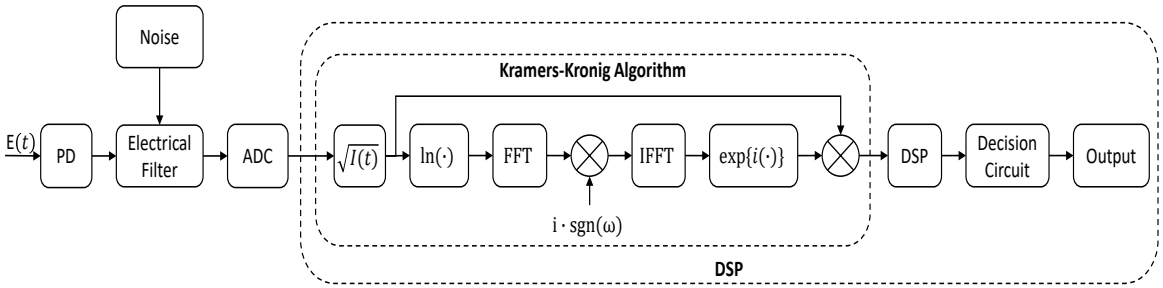


Figure 6.39: Receiver setup for the real-valued signal

Upsampling free KK algorithm

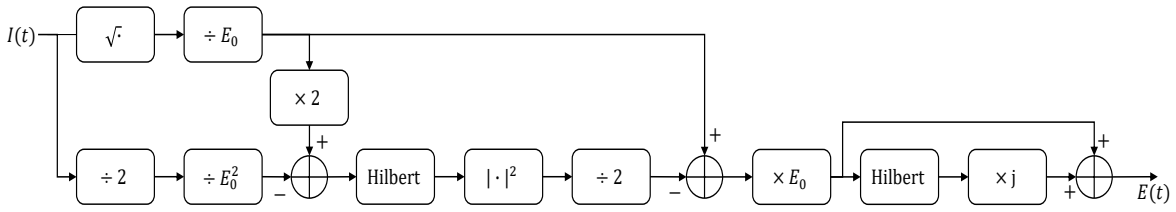


Figure 6.40: Up sampling free KK algorithm

AM-AM and AM-PM analysis

The nonlinear behavior of MZM introduces the amplitude and the phase distortion onto the transmitted signal. These effects, if not properly controlled, cause unacceptable spectral regrow for the actual communication system. The AM-AM and AM-PM measurement provide the way to characterize these nonlinear distortion of the MZM. The AM-AM and AM-PM distortion in a nonlinear system can be observed by increasing the power of the

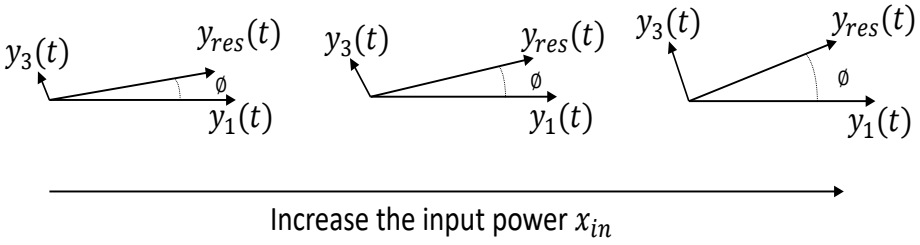


Figure 6.41: Illustration of AM-AM and AM-PM distortion

input signal (see Figure 6.41). where, $y_1(t)$ is the linear component; $y_3(t)$ is the third-order signal correlated distortion component; $y_r(t)$ is the resultant output component; and ϕ is the resultant output phase. The figure shows that addition of the signal correlated third-order components ($y_3(t)$) to the linear components ($y_1(t)$) constitutes a vector addition, which means that variation in input amplitude will produce changes in output amplitude, but also in output phase. Alternatively, we can also illustrate the AM-AM and AM-PM behavior of the nonlinear device as shown Figure 6.42.

There are several ways to characterizing the AM-AM and AM-PM distortion. The method

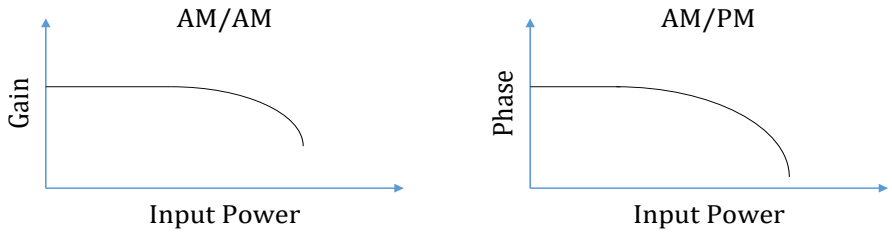


Figure 6.42: Illustration of AM-AM and AM-PM distortion

discussed here minimizes the use of the expensive and complex instrumentations, allowing a fast and low cost assessment of the AM-AM and AM-PM behavior of DUT in a few steps.

Proposed Method

The proposed approach is based on the differential measurement between two branches, one composed by MZM and the other one made up by using linear element. This method uses the Wilkinson power divider at the input end, and a 90 degree hybrid coupler at the output end. The scattering matrix for the 90 degree hybrid coupler (see Figure 6.43) can be written as,

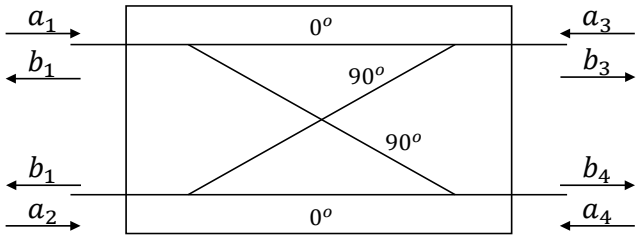


Figure 6.43: 90 degree hybrid coupler

$$S = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & -j & 1 \\ 0 & 0 & 1 & -j \\ -j & 1 & 0 & 0 \\ 1 & -j & 0 & 0 \end{bmatrix} \tag{6.49}$$

Thus, the following equations for the output wave can be derived:

$$b_3 = -j\frac{|a_1|}{\sqrt{2}} + \frac{|a_2|}{\sqrt{2}}cos(\phi) + j\frac{|a_2|}{\sqrt{2}}sin(\phi) \tag{6.50}$$

$$b_4 = \frac{|a_1|}{\sqrt{2}} - j\frac{|a_2|}{\sqrt{2}}cos(\phi) + \frac{|a_2|}{\sqrt{2}}sin(\phi) \tag{6.51}$$

where ϕ is the phase difference between a_1 and a_2, if a_1 is assumed to define a reference phase. From Equation 6.2.2, it is possible to obtain the power at the output ports of the coupler:

$$\begin{aligned} P_3 = |b_3|^2 &= \frac{|a_1|^2}{2} + \frac{|a_2|^2}{2} - |a_1||a_2|\ sin(\phi) \\ &= \frac{P_1}{2} + \frac{P_2}{2} - \sqrt{P_1P_2}\ sin(\phi) \end{aligned} \tag{6.52}$$

$$\begin{aligned} P_4 = |b_4|^2 &= \frac{|a_1|^2}{2} + \frac{|a_2|^2}{2} + |a_1||a_2|\ sin(\phi) \\ &= \frac{P_1}{2} + \frac{P_2}{2} + \sqrt{P_1P_2}\ sin(\phi) \end{aligned} \tag{6.53}$$

where, $P_1 = |a_1|^2$ and $P_2 = |a_2|^2$ are the incident power at the input port 1 and 2 respectively. The value of ϕ can be extracted by subtracting Equation 6.2.2 and 6.2.2,

$$P_4 - P_3 = 2\sqrt{P_1P_2}\ sin(\phi) \tag{6.54}$$

and inverting 6.2.2,

$$\phi = sin^{-1}\left(\frac{P_4 - P_3}{2\sqrt{P_1P_2}}\right) \tag{6.55}$$

If we consider the 90^o degree coupler is lossless then we can write,

$$P_2 = P_3 + P_4 - P_2 \tag{6.56}$$

which leads to

$$\phi = sin^{-1}\left(\frac{P_4 - P_3}{2\sqrt{P1(P_3 + P_4) - P_1^2}}\right) \tag{6.57}$$

Measurement Setup

bla bla bla bla

6.2.3 Simulation Analysis

The transmitter setup includes various blocks to generating the minimum phase SSB signal. The presented block B1 randomly generates the sequence of binary numbers 0 and 1 as a S_1 signal. The generated binary sequence S_1 was 16-QAM mapped which outputs two discrete data signals S_2 and S_3, which are converted into the continuous time form using the B3 and B4 blocks, respectively. The continuous time signals S_4 and S_5 are passed through the pulse shaping filter blocks B5 and B6 respectively to get rid of inter-symbol-interference in the communication system. The output of the pulse shaping filter blocks are represented by signal S_6 and S_7.
The Real to complex block accepts two real signal to generate the complex output. Here, we have supplied S_6 and S_7 signal to generate a complex signal S_8. The oscillator block B8 will generate complex CW tone whose frequency coincides at the left edge of the information signal (S_9) spectrum. The signals S_8 and S_9 are supplied to the B9 adder blocks which outputs the signal S_10. The signal S_10 is then converted into minimum phase signal by mixer block B11. The mixer block will accept the CW tone added information signal S_{10} and the complex LO signal S_11 and generates the minimum phase signal S_{12}. The output of the mixer blocks B11 supplied to the ideal IQ modulator block B12 which outputs the signal S_{13} same as an input signal S_{12}.

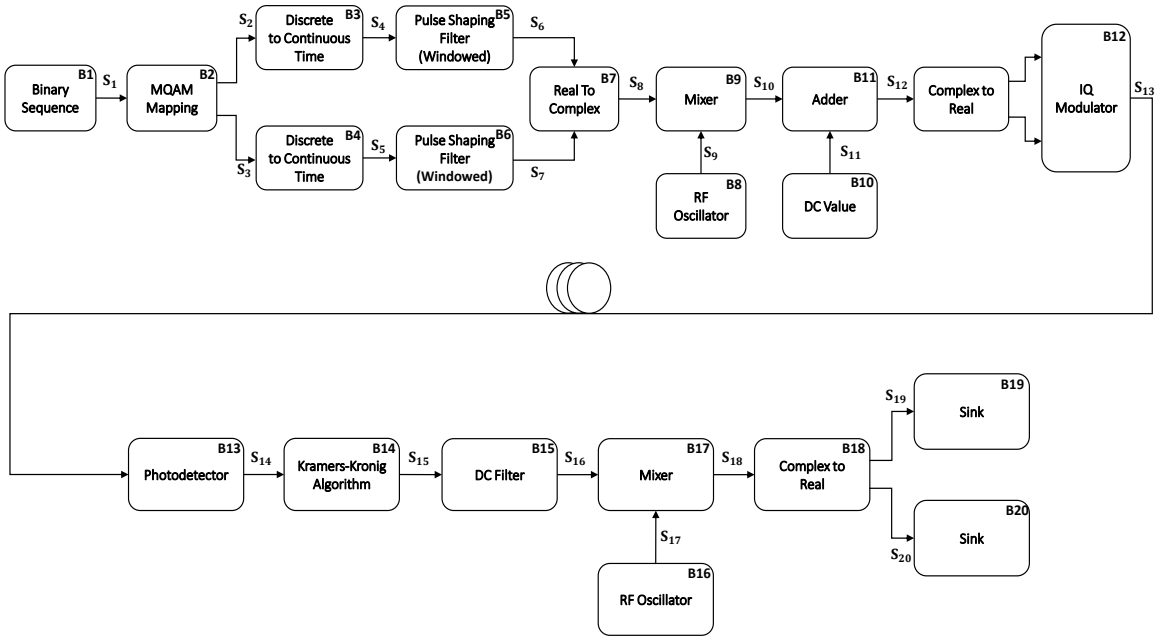
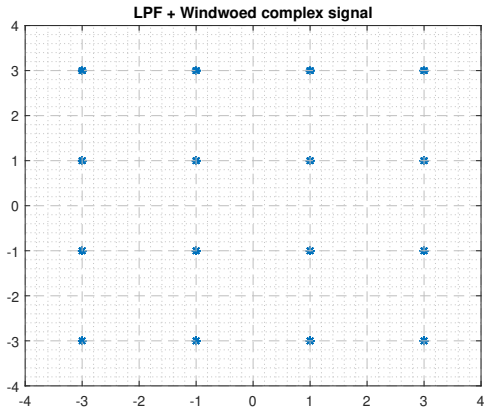


Figure 6.44: Transceiver simulation setup

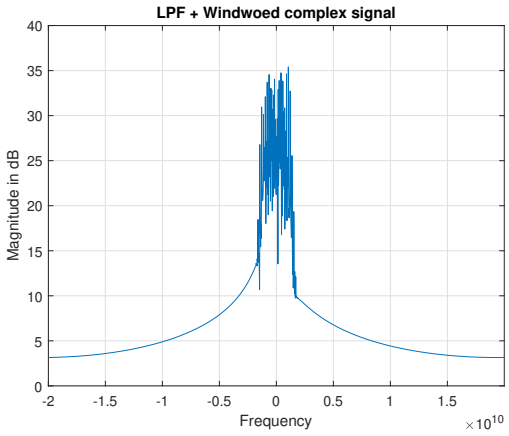
At the receiver end, the received minimum phase signal S_{13} is detected by a single photo detector which outputs the electrical signal S_{14}. Proceeding next, the signal S_{14} is passed

through the block B14 which reconstructs the full complex envelope of the message signal using Kramers-Kronig algorithm and outputs the recovered complex signal S_{15}. Next, the DC filter block B16 filter out the DC content presents in the recovered signal and and passed through mixer block B17 to recover the original baseband signalS_{18}. The reconstructed complex signal is passed through the complex to real block B18 which outputs the signals S_{19} and S_{20}.

Signal analysis

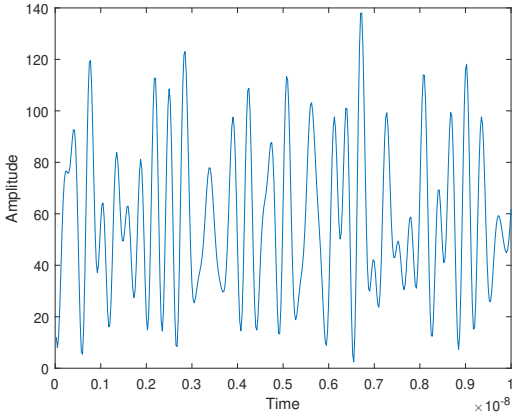


(a) Constellation of 16-QAM signal $S8$

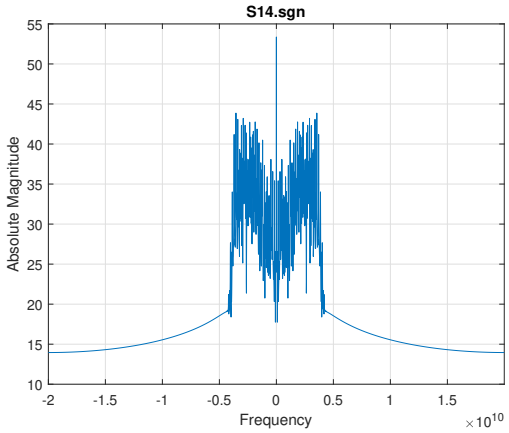


(b) Magnitude spectrum of $S8$

Figure 6.45: 16-QAM signal $S8$

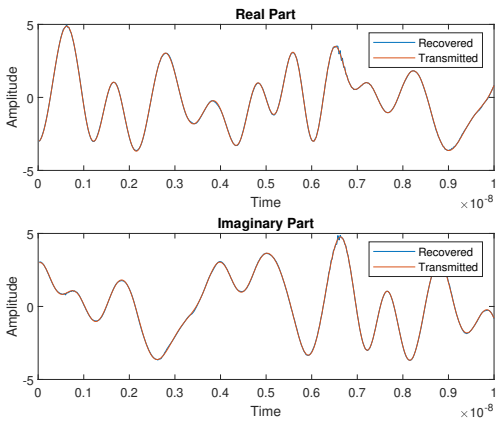


(a) Photo-detected signal $S14$

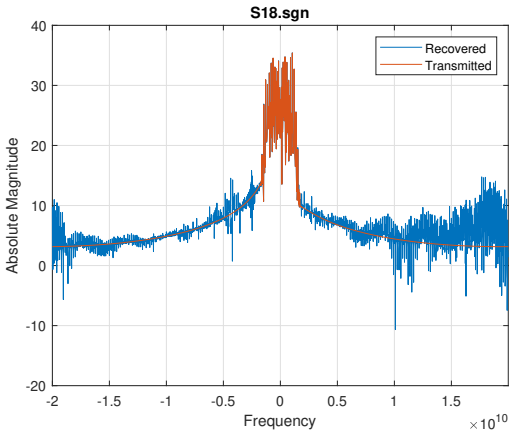


(b) Magnitude spectrum of $S14$

Figure 6.46: Photo-detected signal $S14$



(a) Recovered complex signal $S18$



(b) Magnitude spectrum of $S18$

Figure 6.47: Recovered signal $S18$

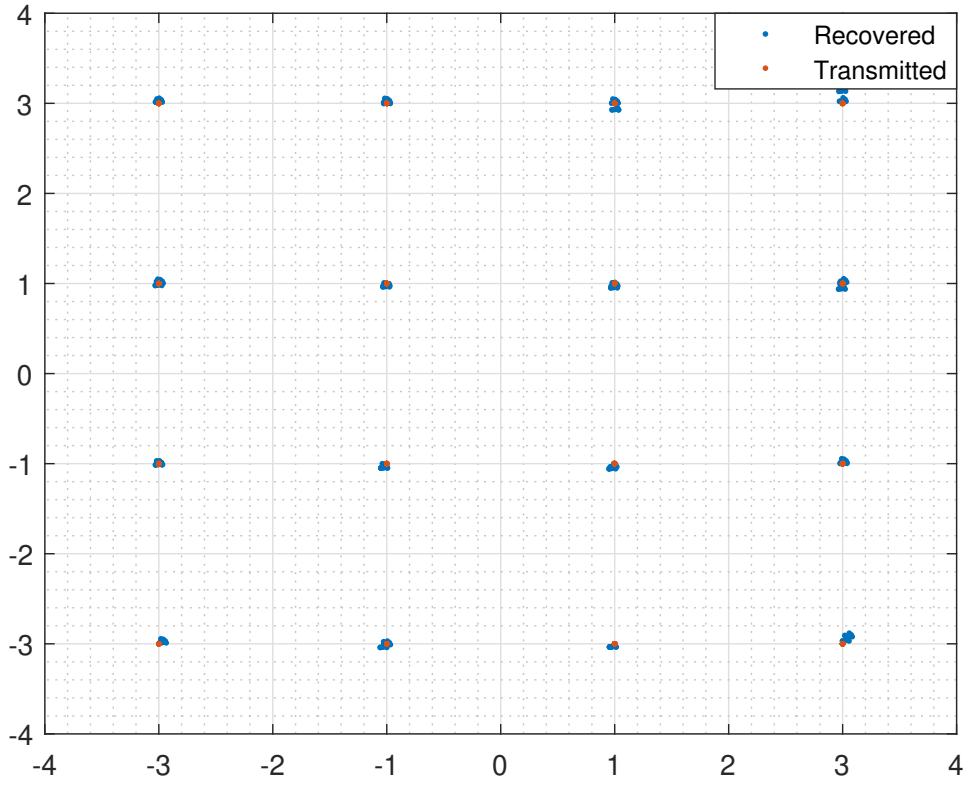


Figure 6.48: Constellation of the recovered($S18$) and transmitted ($S8$) complex signal

6.2.4 Iterative method

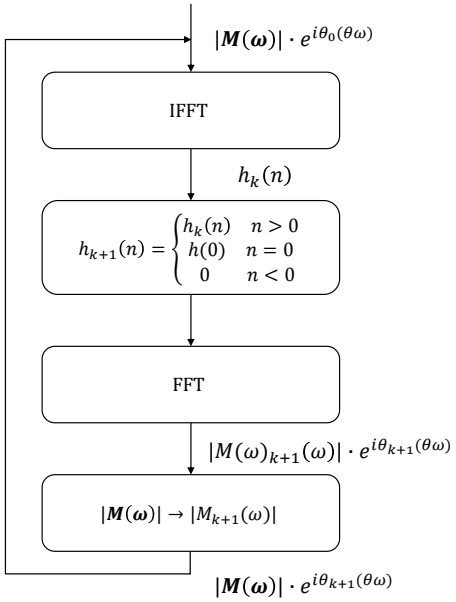


Figure 6.49: Iterative method for frequency domain magnitude data

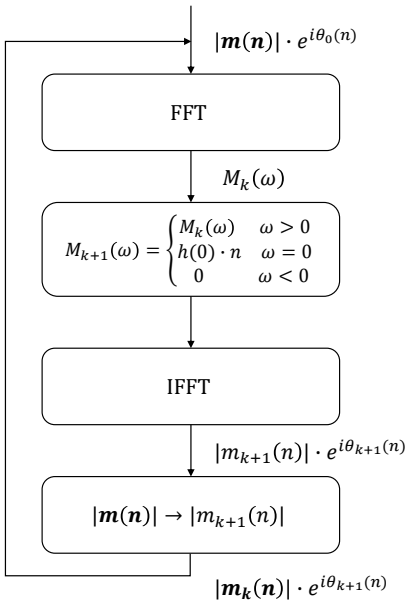
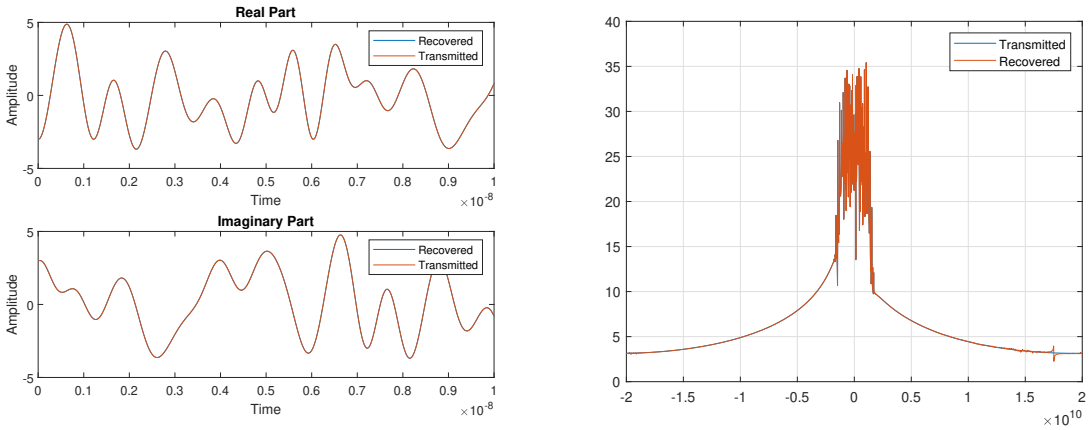


Figure 6.50: Iterative method for time domain magnitude data



(a) Recovered complex signal (b) Magnitude spectrum of of recovered signal

Figure 6.51: Recovered signal using iterative method

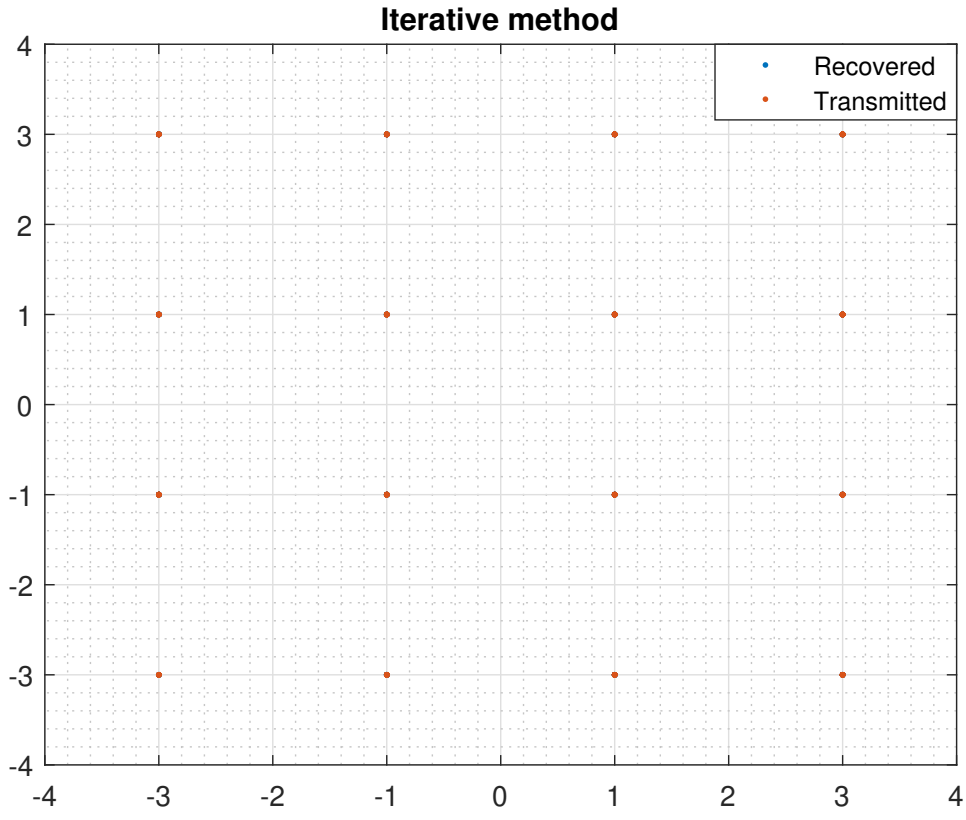


Figure 6.52: Constellation of the recovered and transmitted signal using iterative method

References

[1] Antonio Mecozzi, Cristian Antonelli, and Mark Shtaif. "Kramersâ€"Kronig coherent receiver". In: *Optica* 3.11 (Nov. 2016), p. 1220. ISSN: 2334-2536. DOI: 10.1364/OPTICA.3.001220. URL: https://www.osapublishing.org/abstract.cfm?URI=optica-3-11-1220.

[2] Antonio Mecozzi. "Retrieving the full optical response from amplitude data by Hilbert transform". In: *Optics Communications* 282.20 (Oct. 2009), pp. 4183–4187. ISSN: 0030-4018. DOI: 10.1016/J.OPTCOM.2009.07.025. URL: https://www.sciencedirect.com/science/article/pii/S0030401809006907.

[3] Matilde Legua and Luis Sánchez-Ruiz. "Cauchy Principal Value Contour Integral with Applications". In: *Entropy* 19.5 (May 2017), p. 215. ISSN: 1099-4300. DOI: 10.3390/e19050215. URL: http://www.mdpi.com/1099-4300/19/5/215.

[4] Antonio Mecozzi. "A necessary and sufficient condition for minimum phase and implications for phase retrieval". In: *IEEE TRANSACTIONS ON INFORMATION THEORY* 13.9 (2014). URL: https://pdfs.semanticscholar.org/1ae2/690a2a435f94b74320d14f135f3e4928f08a.pdf.

[5] M. Presi et al. "Transmission in 125-km SMF with 3.9 bit/s/Hz spectral efficiency using a single-drive MZM and a direct-detection Kramers-Kronig receiver without optical CD compensation". In: *Optical Fiber Communication Conference*. Washington, D.C.: OSA, Mar. 2018, Tu2D.3. ISBN: 978-1-943580-38-5. DOI: 10.1364/OFC.2018.Tu2D.3. URL: https://www.osapublishing.org/abstract.cfm?URI=OFC-2018-Tu2D.3.

[6] S. T. Le et al. "8Ã—256Gbps Virtual-Carrier Assisted WDM Direct-Detection Transmission over a Single Span of 200km". In: *2017 European Conference on Optical Communication (ECOC)*. IEEE, Sept. 2017, pp. 1–3. ISBN: 978-1-5386-5624-2. DOI: 10.1109/ECOC.2017.8346088. URL: https://ieeexplore.ieee.org/document/8346088/.

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 Balanced Beam Splitter

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

Input Parameters

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

Functional Description

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

$$M_{BS} = \frac{\sqrt{2}}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{7.1}$$

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \tag{7.2}$$

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

Input Signals

Number: 1 or 2
Type: Complex

Output Signals

Number: 2
Type: Complex

7.3 Bit Error Rate

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

Input Parameters

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

Input Signals

Number: 2
Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional Description

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

The block allows for mid-reports to be generated, the number of bits between reports is customizable, if it is set to 0 then the block will only output the final report.

Theoretical Description

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

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \tag{7.3}$$

The upper and lower bounds, BER_{UB} and BER_{LB} respectively, are calculated using the Clopper-Pearson confidence interval, which returns the following simplified expression for

$N_T > 40$ **[almeida2016continuous]**:

$$\text{BER}_{\text{UB}} = \widehat{\text{BER}} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1 - \widehat{\text{BER}})} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - \widehat{\text{BER}} \right) z_{\alpha/2}^2 + (2 - \widehat{\text{BER}}) \right] \quad (7.4)$$

$$\text{BER}_{\text{LB}} = \widehat{\text{BER}} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\widehat{\text{BER}}(1 - \widehat{\text{BER}})} + \frac{1}{3N_T} \left[2 \left(\frac{1}{2} - \widehat{\text{BER}} \right) z_{\alpha/2}^2 - (1 + \widehat{\text{BER}}) \right], \quad (7.5)$$

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

7.4 Binary Source

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

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

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

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

Input Parameters

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

Table 7.1: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

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

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

Functional description

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

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

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

DeterministicCyclic Mode Generates the sequence of 0's and 1's specified by *bitStream* and then repeats it.

DeterministicAppendZeros Mode Generates the sequence of 0's and 1's specified by *bitStream* and then it fills the rest of the buffer space with zeros.

Input Signals

Number: 0

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1 or more

Type: Binary (DiscreteTimeDiscreteAmplitude)

Examples

Random Mode

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

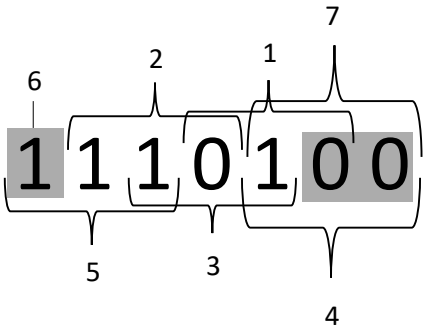


Figure 7.1: Example of a pseudorandom sequence with a pattern length equal to 3.

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

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

Sugestions for future improvement

Implement an input signal that can work as trigger.

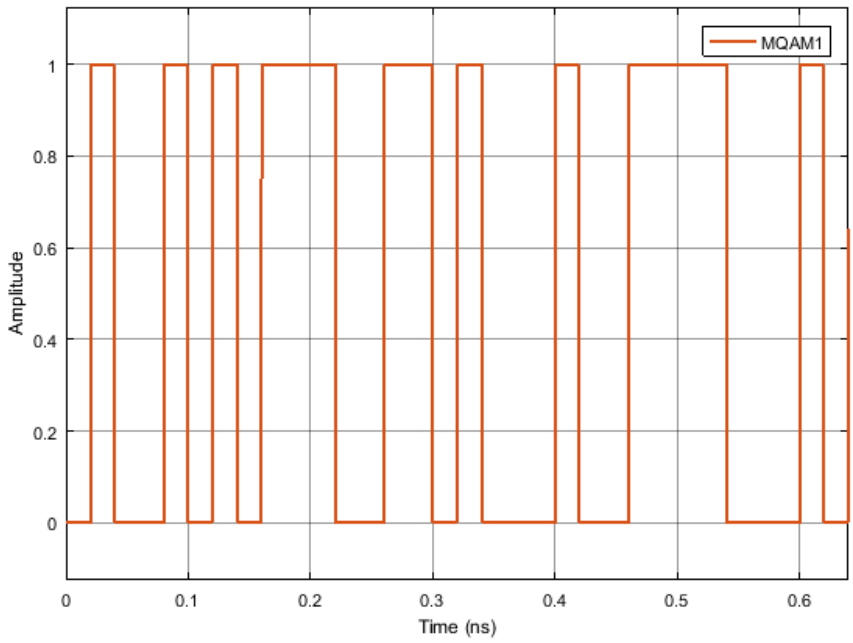


Figure 7.2: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

7.5 cwTone

Header File	:	cwTone.h
Source File	:	cwTone.cpp
Version	:	20180705 (**Student Name**: Romil Patel)

Input Parameters

frequency: The frequency of CW tone depends on the bandwidth of the complex signal $I(t) + i\,Q(t)$. The frequency of the signal should coincide to the right or left edge of the information carrying signal spectrum.
amplitude: The amplitude of the signal should provide the Carrier to Signal Power Ratio (CSPR) of about ∼9dB.

Input Signals

Number: 1, 2
Type: RealValue

Output Signals

Number: 3, 4
Type: RealValue

Functional Description

This block accepts two real valued signal, one is $I(t)$ and other $Q(t)$, and add a CW tone to both these signals. The tone added to both the signals are 90^o phase shifted to other. The signal output at port 3 and 4 can be written as $I(t) + A \cdot cos(2\pi Ft)$ and $I(t) + A \cdot sin(2\pi Ft)$, respectively. Where A and F represent the amplitude and frequency of the CW tone.

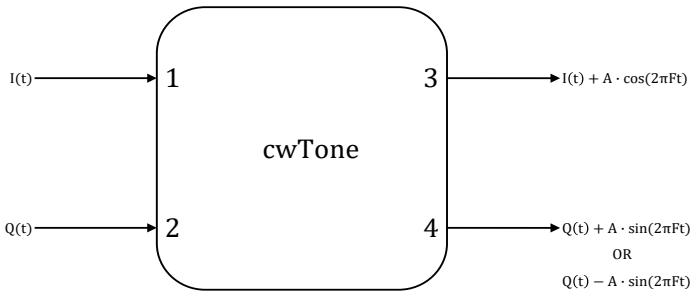


Figure 7.3: cwTone

7.6 Bit Decider

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

Input Parameters

Name	Type	Default Value
decisionLevel	double	0.0

Functional Description

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

Input Signals

Number: 1
Type: Real signal (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

7.7 Clock

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

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

Input Parameters

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

Table 7.2: Binary source input parameters

Methods

Clock()

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

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per)

void setSamplingPeriod(double sPeriod)

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

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

Examples

Sugestions for future improvement

7.8 Clock_20171219

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

Input Parameters

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

Methods

Clock()

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

void initialize(void)

bool runBlock(void)

void setClockPeriod(double per) double getClockPeriod()

void setClockPhase(double pha) double getClockPhase()

void setSamplingPeriod(double sPeriod) double getSamplingPeriod()

Functional description

Input Signals

Number: 0

Output Signals

Number: 1

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

Examples

Sugestions for future improvement

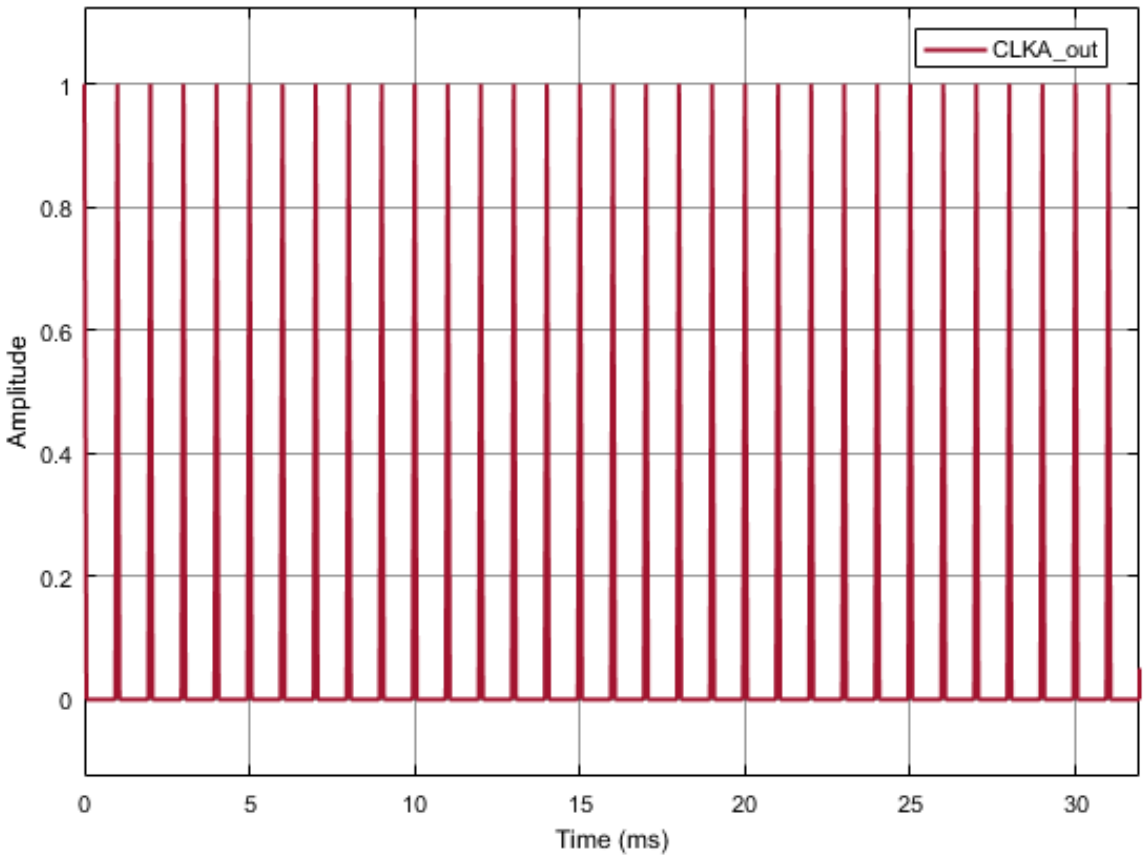


Figure 7.4: Example of the output signal of the clock without phase shift.

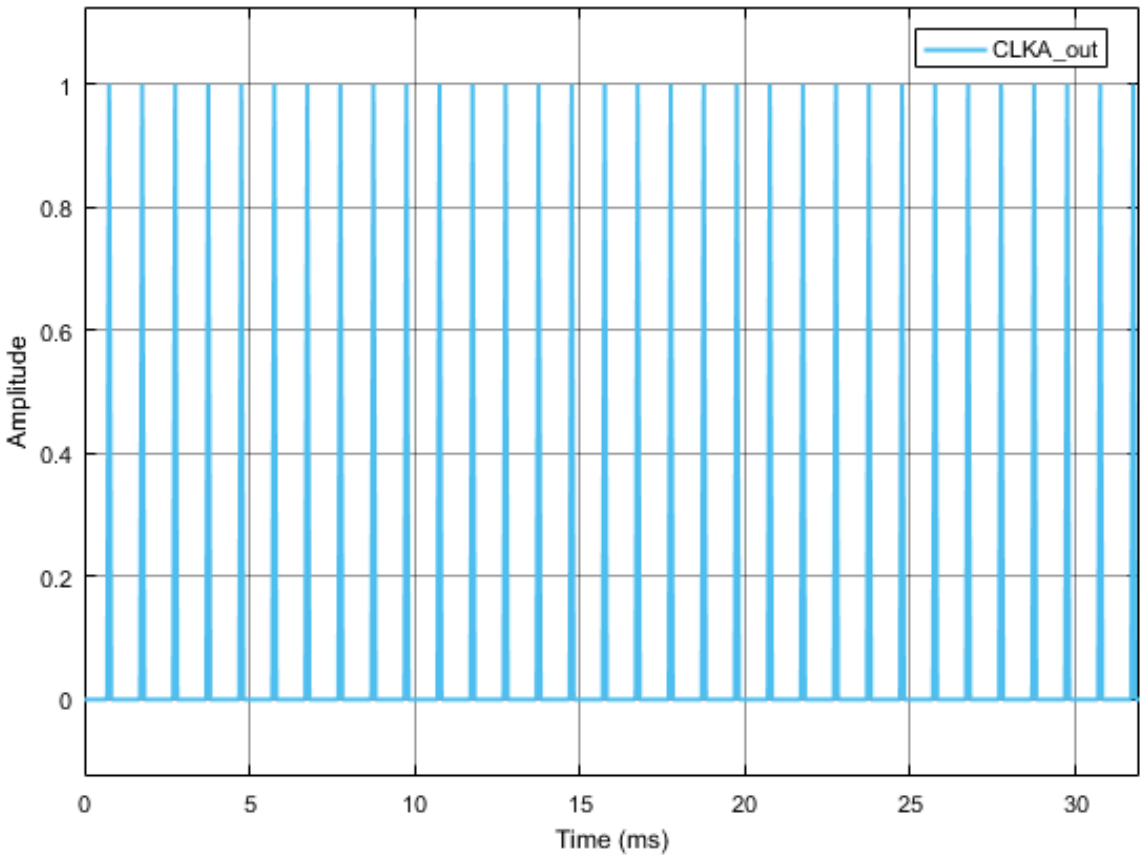


Figure 7.5: Example of the output signal of the clock with phase shift.

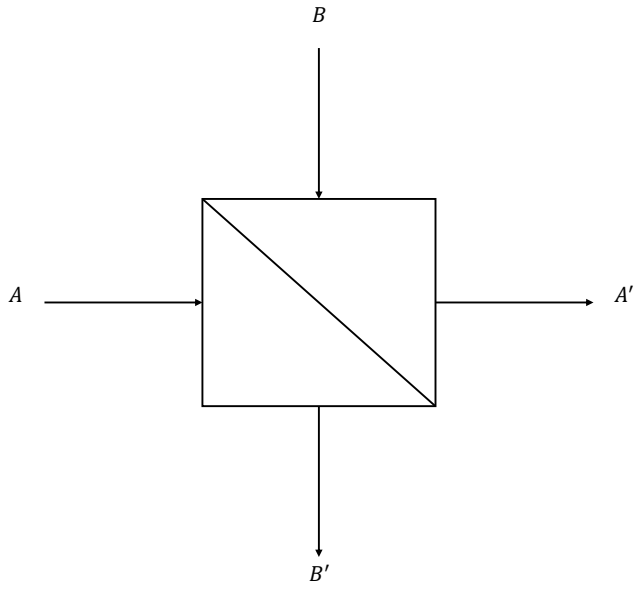


Figure 7.6: 2x2 coupler

7.9 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} . \begin{bmatrix} A \\ B \end{bmatrix} \tag{7.6}$$

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

$$T = \sqrt{1 - \eta_R} \tag{7.7}$$

$$R = \sqrt{\eta_R} \tag{7.8}$$

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

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

7.10 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.

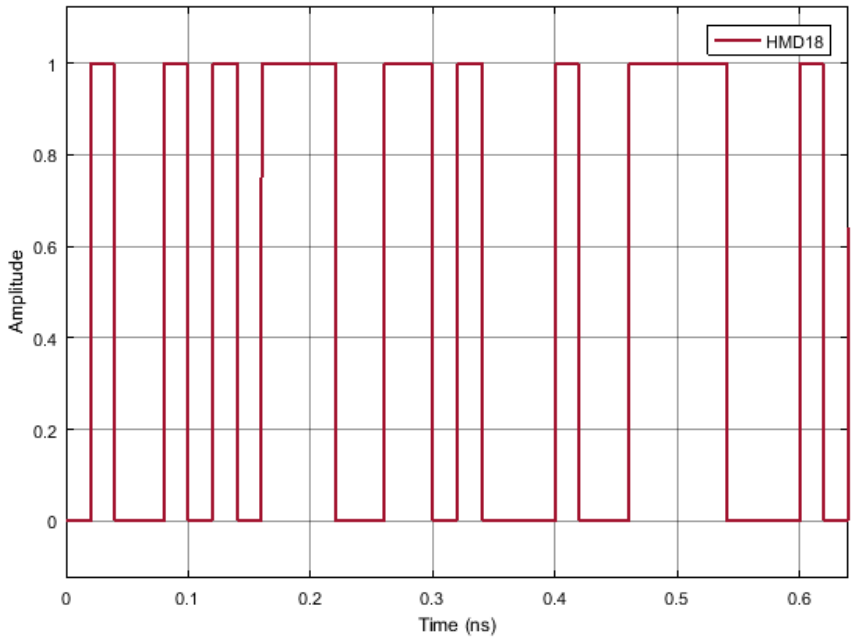


Figure 7.7: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

Sugestions for future improvement

7.11 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

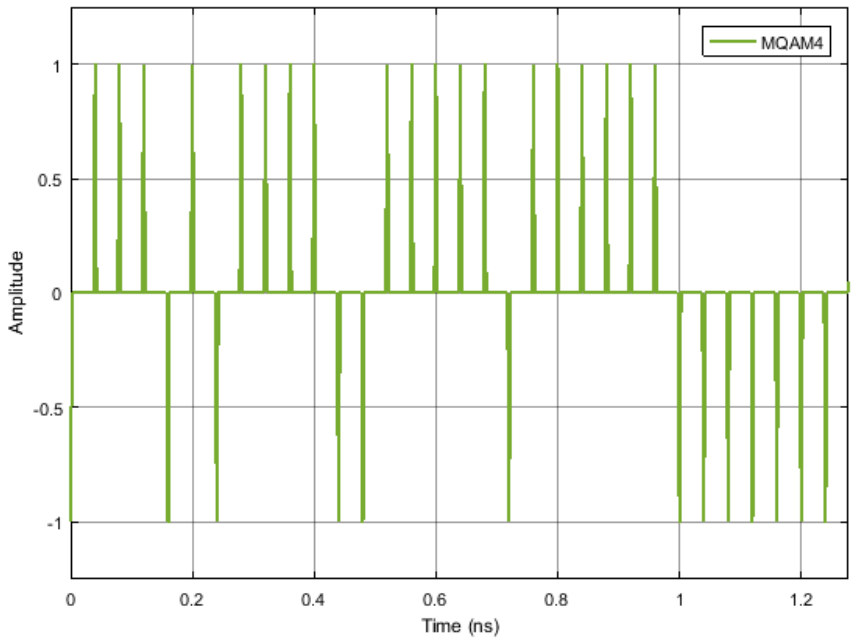


Figure 7.8: Example of the type of signal generated by this block for a binary sequence 0100...

7.12 Electrical Signal Generator

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

7.12.1 ContinuousWave

Continuous Wave the function of the desired signal. This must be introduce by using the function *setFunction(ContinuousWave)*. This function generates a continuous signal with value 1. However, this value can be multiplied by a specific gain, which can be set by using the function *setGain()*. This way, this block outputs a continuous signal with value $1 \times$ gain.

Input Parameters

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

Methods

ElectricalSignalGenerator() {};

void initialize(void);

bool runBlock(void);

void setFunction(ElectricalSignalFunction fun) ElectricalSignalFunction getFunction()

void setSamplingPeriod(double speriod) double getSamplingPeriod()

void setSymbolPeriod(double speriod) double getSymbolPeriod()

void setGain(double gvalue) double getGain()

Functional description

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

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

Input Signals

Number: 0

Type: No type

Output Signals

Number: 1

Type: TimeContinuousAmplitudeContinuous

Examples

Sugestions for future improvement

Implement other functions according to the needs.

7.13 Fork

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

Input Parameters

— NA —

Input Signals

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

Output Signals

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

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

Functional Description

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

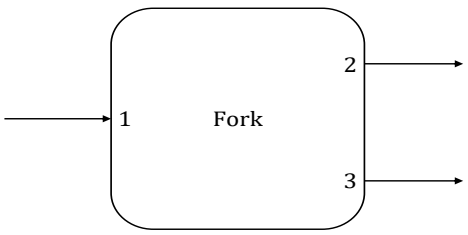


Figure 7.9: Fork

7.14 Gaussian Source

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

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

Input Parameters

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

Table 7.5: Gaussian source input parameters

Methods

GaussianSource()

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

void initialize(void);

bool runBlock(void);

void setAverage(double Average) ;

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Continuous signal (TimeDiscreteAmplitudeContinuousReal)

Examples

Sugestions for future improvement

7.15 MQAM Receiver

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

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

This block of code simulates the reception and demodulation of an optical signal (which is the input signal of the system) outputing a binary signal. A simplified schematic representation of this block is shown in figure 7.10.

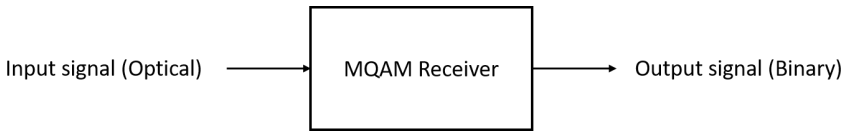


Figure 7.10: Basic configuration of the MQAM receiver

Functional description

This block accepts one optical input signal and outputs one binary signal that corresponds to the M-QAM demodulation of the input signal. It is a complex block (as it can be seen from figure 7.11) of code made up of several simpler blocks whose description can be found in the *lib* repository.

In can also be seen from figure 7.11 that there's an extra internal (generated inside the homodyne receiver block) input signal generated by the *Clock*. This block is used to provide the sampling frequency to the *Sampler*.

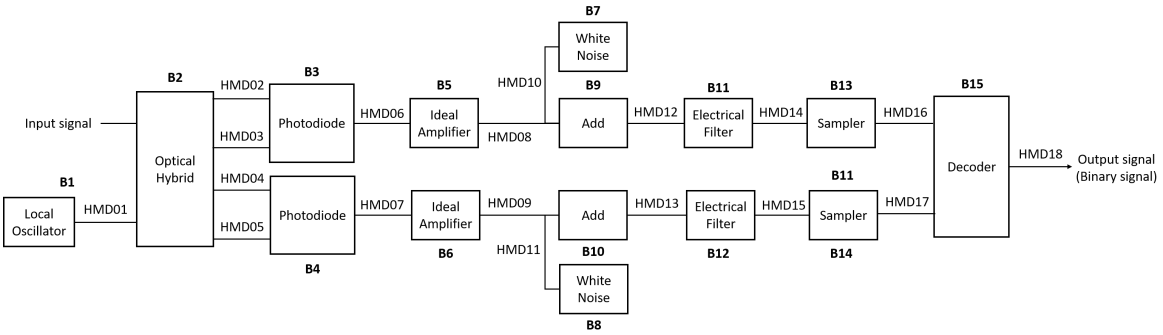


Figure 7.11: Schematic representation of the block homodyne receiver.

Input parameters

This block has some input parameters that can be manipulated by the user in order oto change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 7.11) the input parameters and corresponding functions are summarized.

Input parameters	**Function**	**Type**	**Accepted values**
IQ amplitudes	setIqAmplitudes	Vector of coordinate points in the I-Q plane	**Example** for a 4-QAM mapping: { { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }
Local oscillator power (in dBm)	setLocalOscillatorOpticalPower_dBm	double(t_real)	Any double greater than zero
Local oscillator phase	setLocalOscillatorPhase	double(t_real)	Any double greater than zero
Responsivity of the photodiodes	setResponsivity	double(t_real)	$\in$ [0,1]
Amplification (of the TI amplifier)	setAmplification	double(t_real)	Positive real number
Noise amplitude (introduced by the TI amplifier)	setNoiseAmplitude	double(t_real)	Real number greater than zero
Samples to skipe	setSamplesToSkip	int(t_integer)	
Save internal signals	setSaveInternalSignals	bool	True or False
Sampling period	setSamplingPeriod	double	Given by *symbolPeriod/samplesPerSym*

Table 7.6: List of input parameters of the block MQAM receiver

Methods

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

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setLocalOscillatorSamplingPeriod(double sPeriod)

void setLocalOscillatorOpticalPower(double opticalPower)

void setLocalOscillatorOpticalPower_dBm(double opticalPower_dBm)

void setLocalOscillatorPhase(double lOscillatorPhase)

void setLocalOscillatorOpticalWavelength(double lOscillatorWavelength)

void setSamplingPeriod(double sPeriod)

void setResponsivity(t_real Responsivity)

void setAmplification(t_real Amplification)

void setNoiseAmplitude(t_real NoiseAmplitude)

void setImpulseResponseTimeLength(int impResponseTimeLength)

void setFilterType(PulseShaperFilter fType)

void setRollOffFactor(double rOffFactor)

void setClockPeriod(double per)

void setSamplesToSkip(int sToSkip)

Input Signals

Number: 1

Type: Optical signal

Output Signals

Number: 1

Type: Binary signal

Example

Sugestions for future improvement

7.16 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

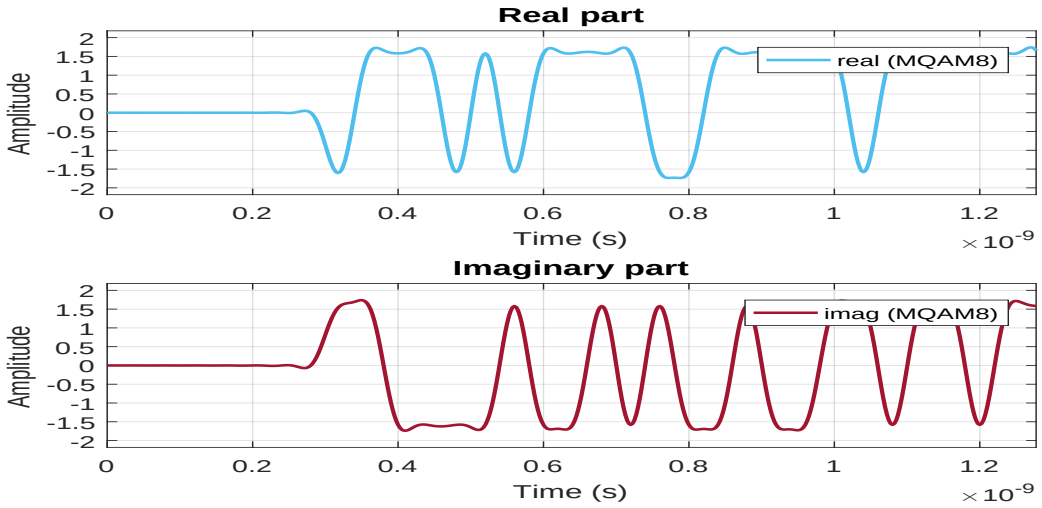


Figure 7.12: Example of a signal generated by this block for the initial binary signal 0100...

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.

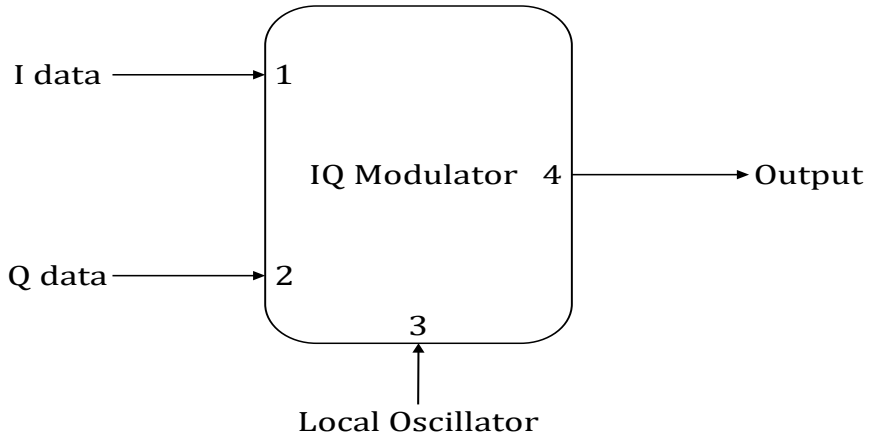


Figure 7.13: IQ Modulator block

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,

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

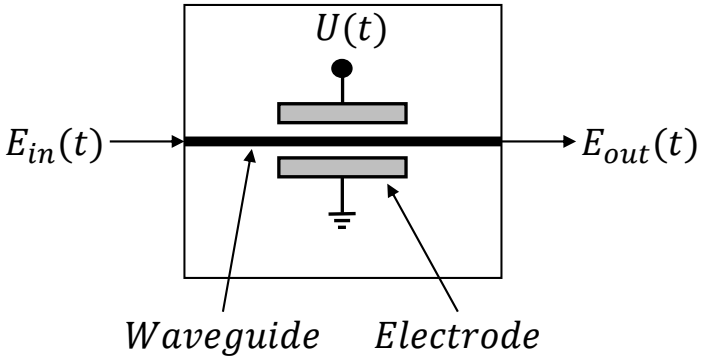


Figure 7.14: Phase Modulator

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,

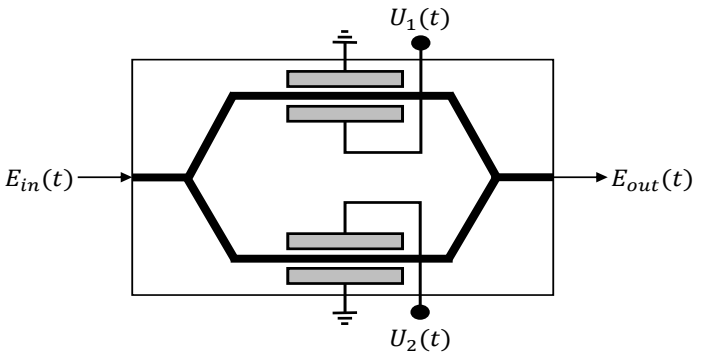


Figure 7.15: Mach-Zehnder Modulator

$$\frac{E_{out}(t)}{E_{in}(t)} = \frac{1}{2} \cdot (e^{j\phi_1(t)} + e^{j\phi_2(t)}) \tag{7.9}$$

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 $E_{out}(t)$ [1]. The transfer function of

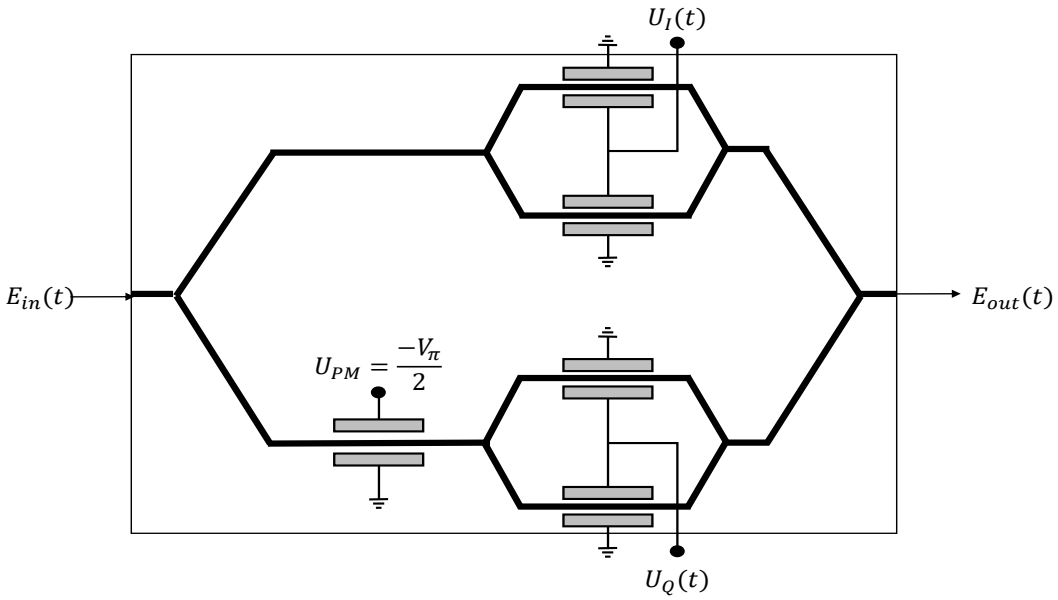


Figure 7.16: IQ Mach-Zehnder Modulator

the IQ MZM can be written as,

$$E_{out}(t) = \frac{1}{2}E_{in}(t)\left[cos\left(\frac{\pi U_I(t)}{2V_\pi}\right) + j \cdot cos\left(\frac{\pi U_Q(t)}{2V_\pi}\right)\right] \tag{7.10}$$

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

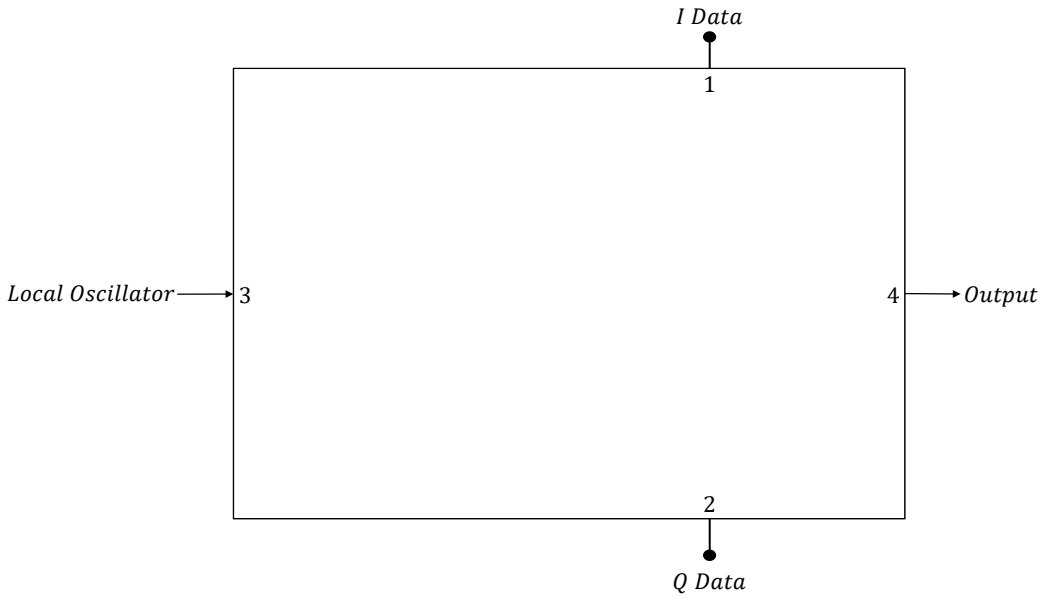


Figure 7.17: Simulation model of the IQ Mach-Zehnder Modulator

References

[1] *National Programme on Technology Enhanced Learning (NPTEL) :: Electronics & Communication Engineering : Optical communications*. URL: https://nptel.ac.in/courses/117104127/5.

7.17 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

7.18 Local Oscillator

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

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

Input Parameters

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

Table 7.9: Local oscillator input parameters

Methods

LocalOscillator()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setOpticalPower(double oPower);

void setOpticalPower_dBm(double oPower_dBm);

void setWavelength(double wlength);

void setPhase(double lOscillatorPhase);

void setLaserLinewidth(double laserLinewidth);

double getLaserLinewidth();

void setLaserRIN(double LOlaserRIN);

double getLaserRIN();

Functional description

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

Input Signals

Number: 0

Output Signals

Number: 1

Type: Optical signal

Examples

Sugestions for future improvement

7.19 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 7.18.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

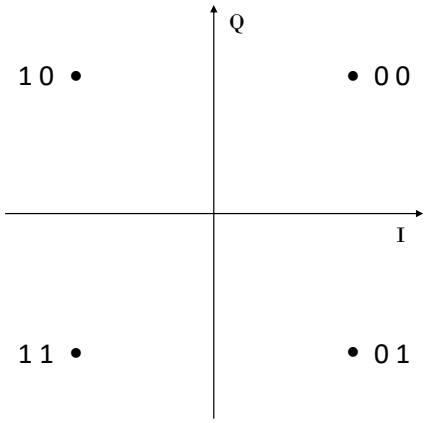


Figure 7.18: Constellation used to map the signal for m=4

Output Signals

Number : 2

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

Example

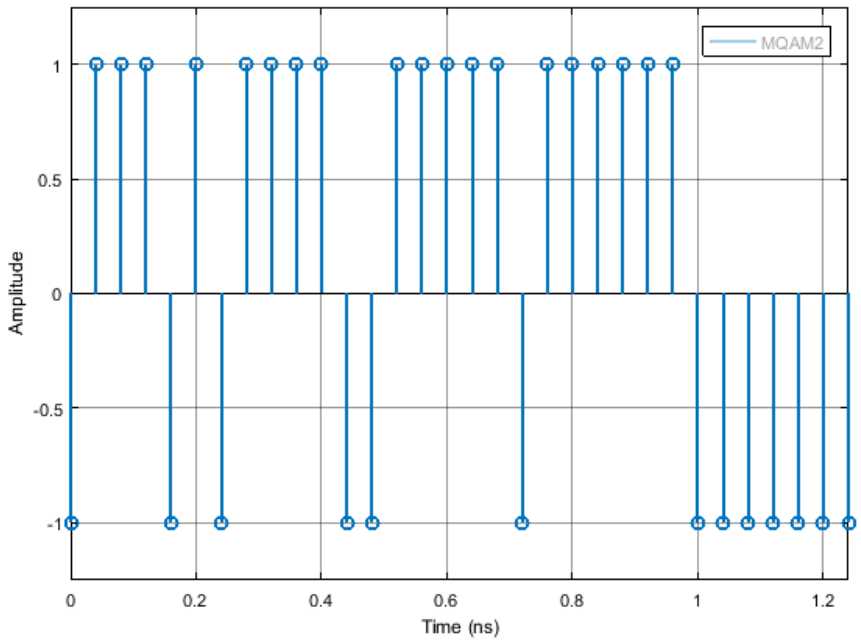


Figure 7.19: Example of the type of signal generated by this block for the initial binary signal 0100...

7.20 MQAM Transmitter

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

This block generates a MQAM optical signal. It can also output the binary sequence. A schematic representation of this block is shown in figure 7.20.

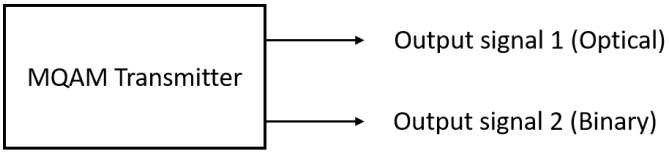


Figure 7.20: Basic configuration of the MQAM transmitter

Functional description

This block generates an optical signal (output signal 1 in figure 7.21). The binary signal generated in the internal block Binary Source (block B1 in figure 7.21) can be used to perform a Bit Error Rate (BER) measurement and in that sense it works as an extra output signal (output signal 2 in figure 7.21).

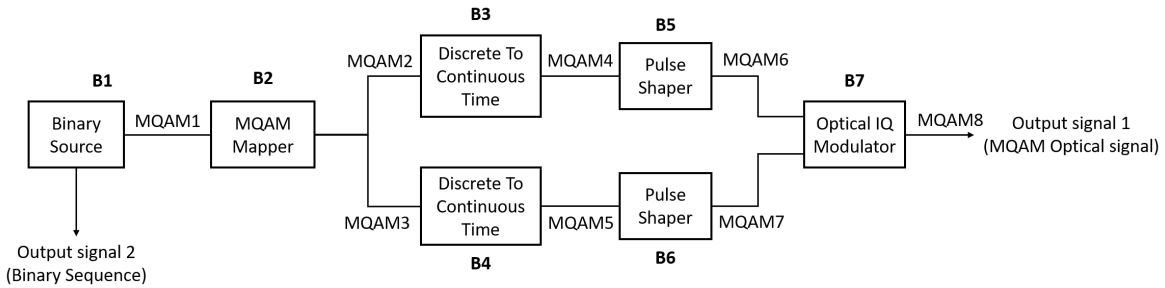


Figure 7.21: Schematic representation of the block MQAM transmitter.

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.

Input parameters	**Function**	Type	**Accepted values**
Mode	setMode()	string	PseudoRandom Random DeterministicAppendZeros DeterministicCyclic
Number of bits generated	setNumberOfBits()	int	Any integer
Pattern length	setPatternLength()	int	Real number greater than zero
Number of bits	setNumberOfBits()	long	Integer number greater than zero
Number of samples per symbol	setNumberOfSamplesPerSymbol()	int	Integer number of the type 2^n with n also integer
Roll of factor	setRollOfFactor()	double	$\in$ [0,1]
IQ amplitudes	setIqAmplitudes()	Vector of coordinate points in the I-Q plane	**Example** for a 4-qam mapping: { { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }
Output optical power	setOutputOpticalPower()	int	Real number greater than zero
Save internal signals	setSaveInternalSignals()	bool	True or False

Table 7.11: List of input parameters of the block MQAM transmitter

Methods

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

void set(int opt);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

void setM(int mValue) int const getM(void)

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setNumberOfSamplesPerSymbol(int n)

int const getNumberOfSamplesPerSymbol(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor(void)

void setSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)

double const getSeeBeginningOfImpulseResponse(void)

void setOutputOpticalPower(t_real outOpticalPower)

t_real const getOutputOpticalPower(void)

void setOutputOpticalPower_dBm(t_real outOpticalPower_dBm)

t_real const getOutputOpticalPower_dBm(void)

Output Signals

Number: 1 optical and 1 binary (optional)

Type: Optical signal

Example

Sugestions for future improvement

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

7.21 Netxpto

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

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

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

Input Parameters

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

Table 7.12: Sampler input parameters

Methods

Sampler()

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

void initialize(void)

bool runBlock(void)

void setSamplesToSkip(`t_integer` sToSkip)

Functional description

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

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

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

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

Input Signals

Number: 1

Type: Electrical real (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical real (TimeDiscreteAmplitudeContinuousReal)

Examples

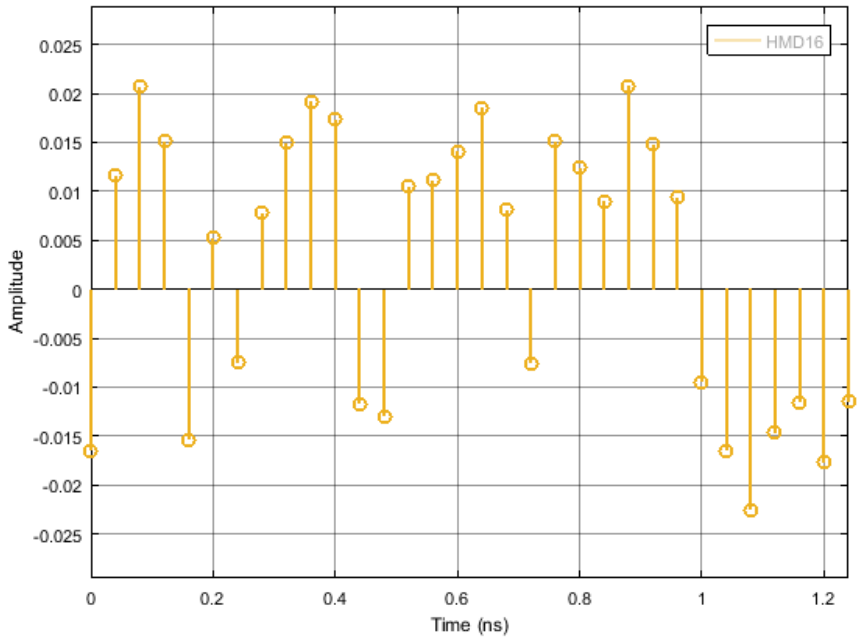


Figure 7.22: Example of the output signal of the sampler

7.21.1 Version 20180118

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

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

Sugestions for future improvement

7.22 Alice QKD

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

Input Parameters

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

Methods

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

Functional description

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

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

Input Signals

Number : 3

Type : Binary, Real Continuous Time and Messages signals.

Output Signals

Number : 3

Type : Binary, Real Discrete Time and Messages signals.

Examples

Sugestions for future improvement

7.23 Polarizer

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

Input Parameters

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

Methods

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

Functional description

Considering m=4, this block atributes for each pair of bits a point in space. In this case, it is be considered four possible polarization states: $0°$, $45°$, $90°$ and $^{1}35°$.

Input Signals

Number : 2

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

Output Signals

Number :1

Type : Photon Stream

Examples

Sugestions for future improvement

7.24 Probability Estimator

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

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

$$\begin{aligned} X(\omega) &= 1 \quad \text{,if } \omega \in A \\ X(\omega) &= 0 \quad \text{,if } \omega \in \bar{A} \end{aligned} \tag{7.11}$$

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

$$\begin{aligned} P(X = 1) = &\quad P(A) &&= p \\ P(X = 0) = &\quad P(\bar{A}) &&= 1 - p \end{aligned} \tag{7.12}$$

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

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

$$\frac{X - np}{\sqrt{np(1-p)}} \backsim N(0, 1). \tag{7.13}$$

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

$$\frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} \backsim N(0, 1) \tag{7.14}$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} < z_{\alpha/2}\} &\approx 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})} < np - X < z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})}\} &\approx 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}\} &\approx 1 - \alpha \end{aligned} \tag{7.15}$$

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

Input Parameters

- zscore
  (double)
- fileName
  (string)

Methods

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

void initialize(void);

bool runBlock(void);

void setProbabilityExpectedX(double probx) double getProbabilityExpectedX()

void setProbabilityExpectedY(double proby) double getProbabilityExpectedY()

void setZScore(double z) double getZScore()

Functional description

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

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \tag{7.16}$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \tag{7.17}$$

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

$$\text{ME} = \text{z}_{\text{score}} \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \tag{7.18}$$

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

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

Input Signals

Number: 1

Type: Binary

Output Signals

Number: 2

Type: Binary

Type: txt file

Examples

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

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

$$\begin{aligned} \text{ME} &= \pm z_{\alpha/2}\frac{\sigma}{\sqrt{N}} \\ \text{ME} &= \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{N}}, \end{aligned} \tag{7.19}$$

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

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

Sugestions for future improvement

7.25 Bob QKD

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

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

7.26 Eve QKD

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

1.

2.

Input Parameters

-
-

Methods

Functional description

Input Signals

Examples

Sugestions for future improvement

7.27 Rotator Linear Polarizer

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

Input Parameters

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

Methods

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

Functional description

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

Input Signals

Number : 2

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

Output Signals

Number : 1

Type : Photon Stream

Examples

Sugestions for future improvement

7.28 Optical Switch

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

Input Parameters

No input parameters.

Methods

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

Functional description

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

Input Signals

Number : 1

Type : Photon Stream

Output Signals

Number : 2

Type : Photon Stream

Examples

Sugestions for future improvement

7.29 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 7.23 shows a schematic representation of this block.

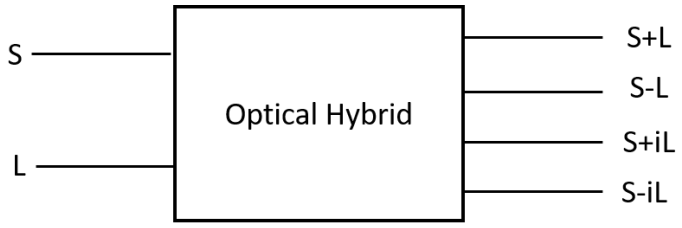


Figure 7.23: Schematic representation of an optical hybrid.

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 7.13: Optical hybrid input parameters

Methods

OpticalHybrid()

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

void initialize(void)

bool runBlock(void)

void setOutputOpticalPower(double outOpticalPower)

void setOutputOpticalPower_dBm(double outOpticalPower_dBm)

void setOutputOpticalWavelength(double outOpticalWavelength)

void setOutputOpticalFrequency(double outOpticalFrequency)

void setPowerFactor(double pFactor)

Functional description

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

Input Signals

Number: 2

Type: Optical (OpticalSignal)

Output Signals

Number: 4

Type: Optical (OpticalSignal)

Examples

Sugestions for future improvement

7.30 Photodiode pair

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

This block simulates a block of two photodiodes assembled like in figure 7.24. 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.

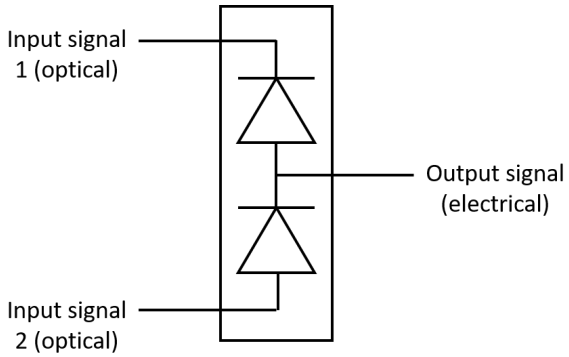


Figure 7.24: Schematic representation of the physical equivalent of the photodiode code block.

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

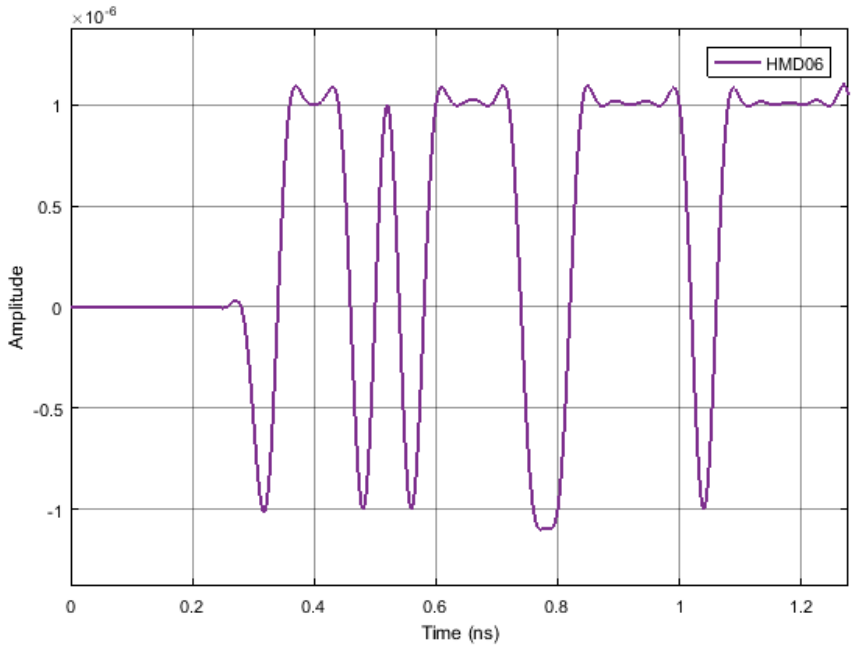


Figure 7.25: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

7.31 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 7.14: Pulse shaper input parameters

Methods

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

void initialize(void);

void setImpulseResponseTimeLength(int impResponseTimeLength)

int const getImpulseResponseTimeLength(void)

void setFilterType(PulseShaperFilter fType)

PulseShaperFilter const getFilterType(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor()

Functional Description

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

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

Input Signals

Number : 1

Type : Sequence of Dirac Delta functions (ContinuousTimeDiscreteAmplitude)

Output Signals

Number : 1

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

Example

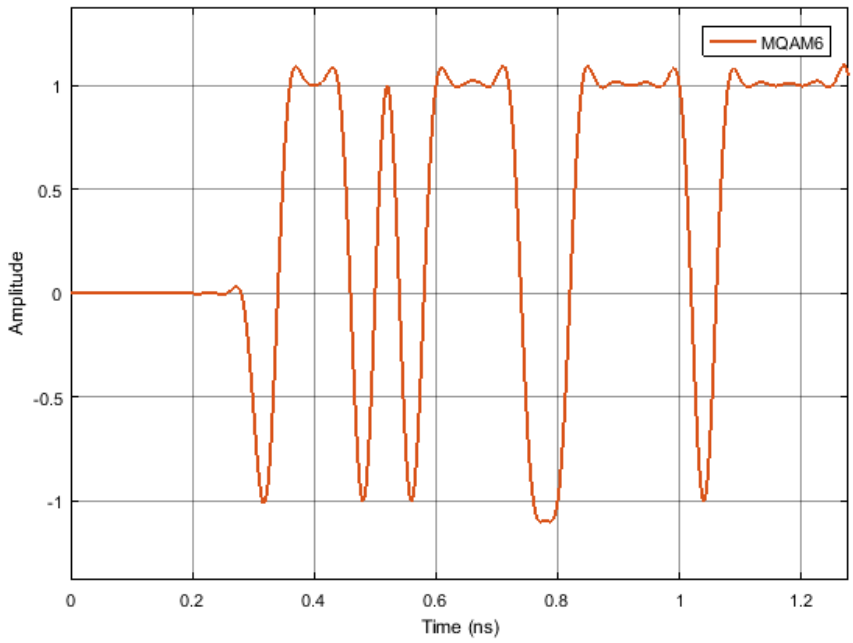


Figure 7.26: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

7.32 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 7.15: Sampler input parameters

Methods

Sampler()

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

void initialize(void)

bool runBlock(void)

void setSamplesToSkip(`t_integer` sToSkip)

Functional description

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

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

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

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

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

Input Signals

Number: 1

Type: Electrical real (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical real (TimeDiscreteAmplitudeContinuousReal)

Examples

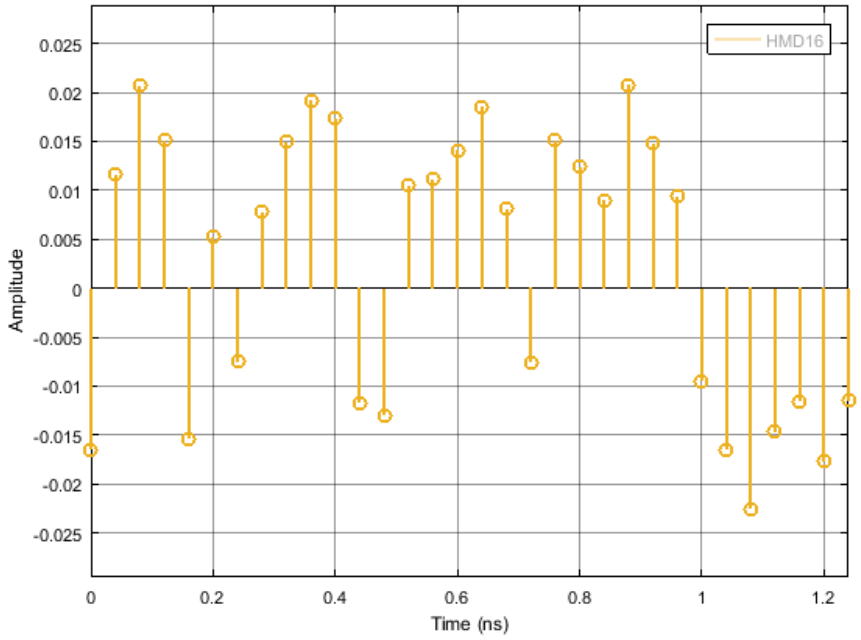


Figure 7.27: Example of the output signal of the sampler

Sugestions for future improvement

7.33 Sink

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

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

Input Parameters

Parameter	Type	Values	Default
numberOfSamples	long int	any	-1

Table 7.16: Sampler input parameters

Methods

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

bool runBlock(void)

void setNumberOfSamples(long int nOfSamples)

void setDisplayNumberOfSamples(bool opt)

Functional Description

7.34 White Noise

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

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

1. DefaultDeterministic
2. RandomDevice
3. Selected

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

Input Parameters

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

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

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: RealValue, ComplexValue or ComplexValueXY

Examples

Random Mode

Suggestions for future improvement

7.35 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 7.18: Ideal Amplifier input parameters

Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;

Functional description

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

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

Chapter 8

Mathlab Tools

8.1 Working with the Tektronix AWG70002A

Goal	:	Convert simulation signals into waveform files compatible with the Tektronix AWG70002A Arbitrary Waveform Generator available in the laboratory.
MATLAB file	:	sgnToWfm_*.m
Version	:	20170930 (Francisco Santos)
	:	20171121 (Romil Patel)

This matlab function generates a *.wfm file from a give input signal file (*.sgn). The waveform file is compatible with the laboratory's Arbitrary Waveform Generator (Tekatronix AWG70002A). The input signal must be real, with no more than $8x10^9$ samples and have a sampling rate equal to 16 GS/s or less (version 20171121 and above supports greater sampling rates).

8.1.1 Generating a signal for the Tektronix AWG70002A

Function prototype

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm_20170930(fname_sgn, nReadr, fname_wfm);

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

Using one argument:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm_20170930('S6.sgn');

This creates a waveform file with the same name as the *.sgn, S6.sgn in the example above, file and uses all of the samples it contains.

Using two arguments:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm_20170930('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., 256 x 16 in the above example, note that the samplesPerSymbol constant is defined in the *.sgn file.

Using three arguments:

[data, symbolPeriod, samplingPeriod, type, numberOfSymbols, samplingRate] = sgnToWfm_20170930('S6.sgn',256,'myWaveform.wfm');

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

Inputs

Name of input signals	Description
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 summarized un the following table:

Name of output signals	Description
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.

Version : 20171121

The function version 20171121 supports decimation of the signal if its sampling rate is greater than 16 GS/s. Since the maximum sampling rate supported by the Tektronix AWG70002A is 16 GS/s, the signal with the sampling rate greater than 16 GS/s can not be generated using AWG70002A. In order to overcome this limitation, it is essential to decimate the signal if its sampling rate is greater than 16 GS/s and reduce the sampling rate to be less than 16 GS/s.

Function prototype

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

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

Same as discussed above in the version 20170930.

Inputs

Same as discussed above in the version 20170930.

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:

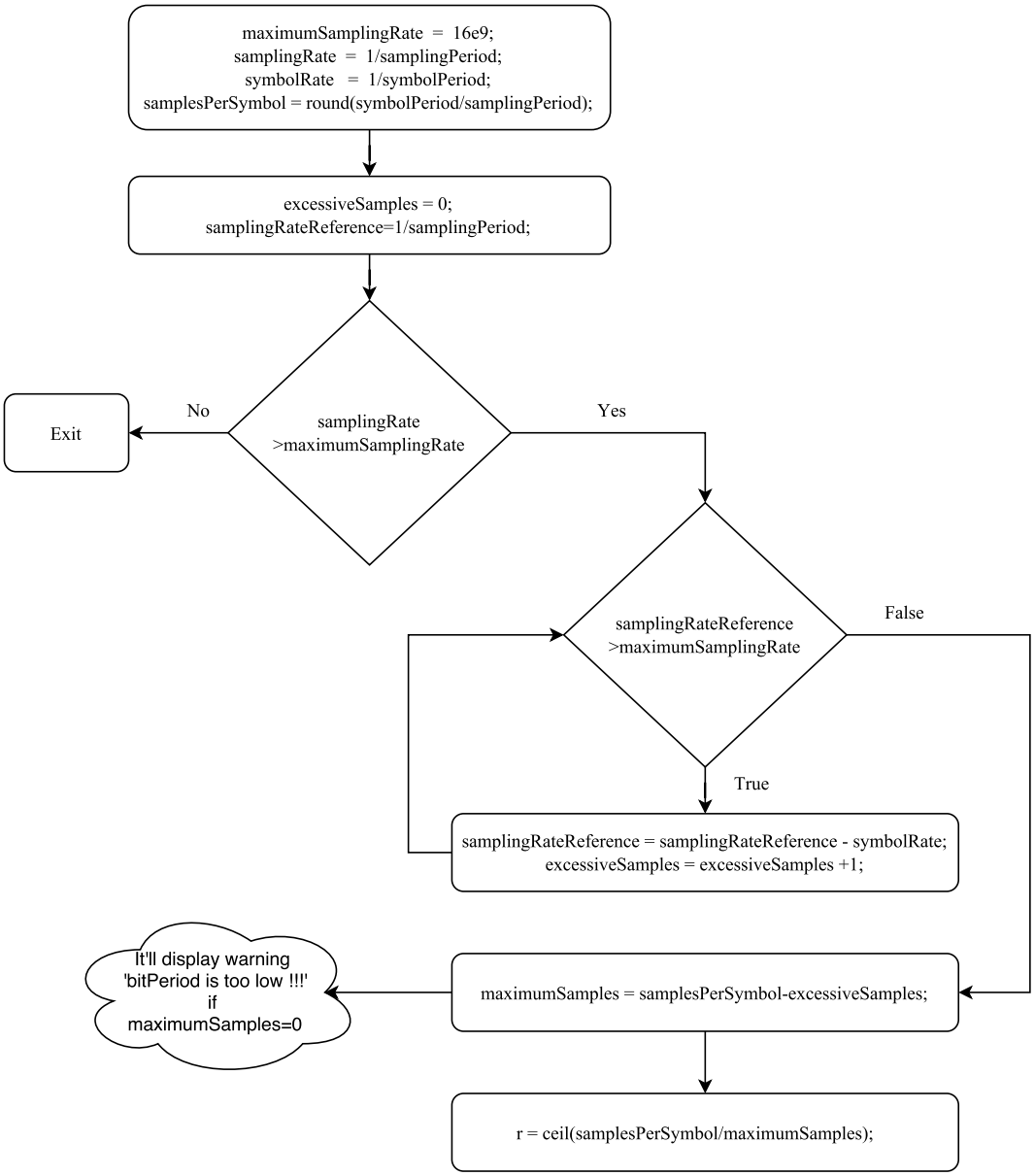


Figure 8.1: Flowchart to calculate decimation factor

8.1.2 Loading a signal to the Tekatronix AWG70002A

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

Sampling rate up to 16 GS/s: This is the most important characteristic because it

determines the maximum sampling rate that your signal can have. It must not be over 16 GS/s or else the AWG will not be able to recreate it appropriately.

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

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

1. Using the function sgnToWfm_20171121: 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).

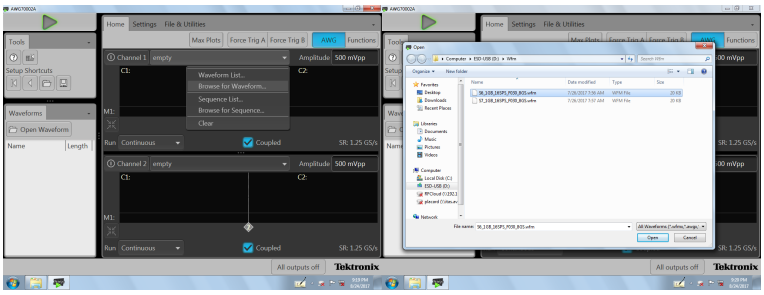


Figure 8.2: Selecting your waveform in the AWG

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).

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

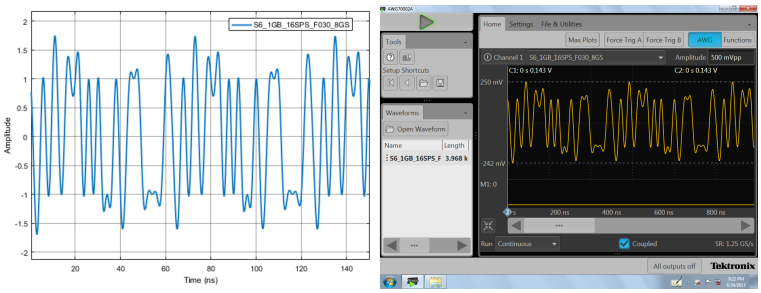


Figure 8.3: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

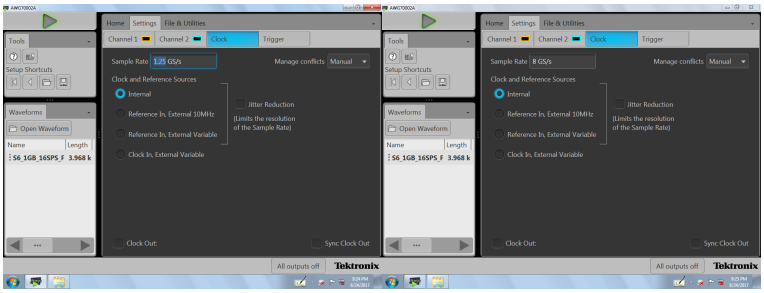


Figure 8.4: Configuring the right sampling rate

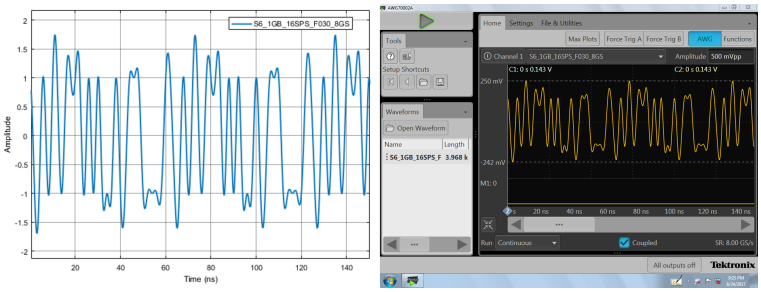


Figure 8.5: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

Chapter 9

Algorithms

9.1 Fast Fourier Transform

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

Algorithm

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

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

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.

Function description

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

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

or

$$y = fft(x)$$

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

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

or

$$x = ifft(y)$$

Flowchart

The figure 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 [1]. The computational complexity of Radix-2 and Bluestein algorithm is $O(Nlog_2N)$, however, the computation of Bluestein algorithm involves the circular convolution which increases the number of computations. Therefore, to reduce the computational time it is advisable to work with the vectors of length 2^N [2].

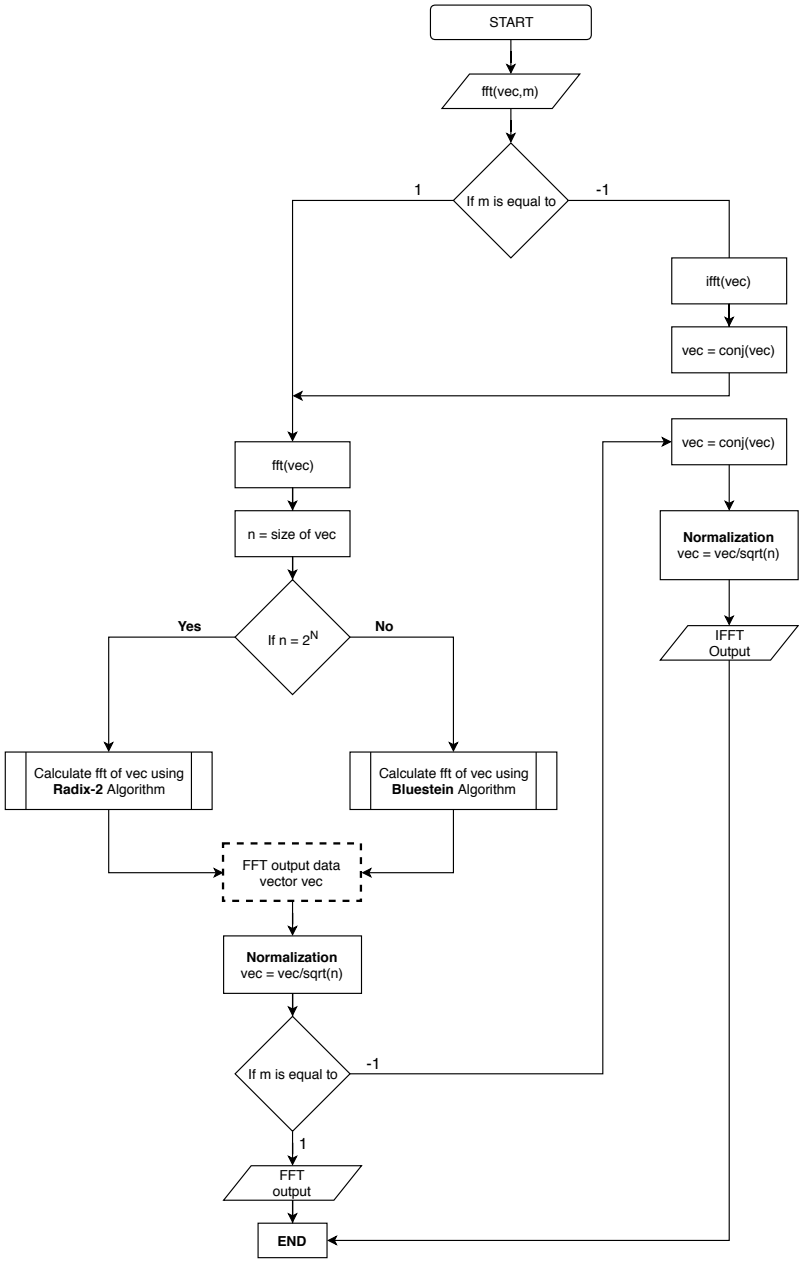


Figure 9.1: Top level architecture of FFT algorithm

Radix-2 algorithm

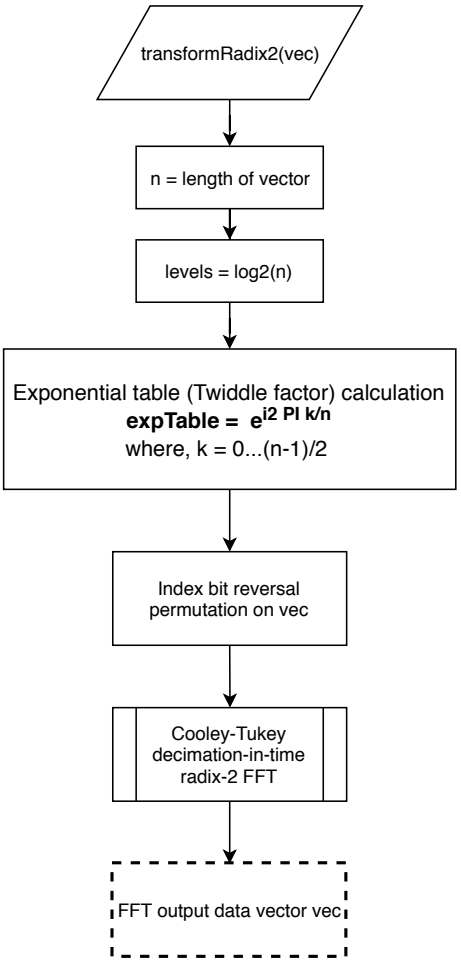


Figure 9.2: Radix-2 algorithm

Cooley-Tukey algorithm

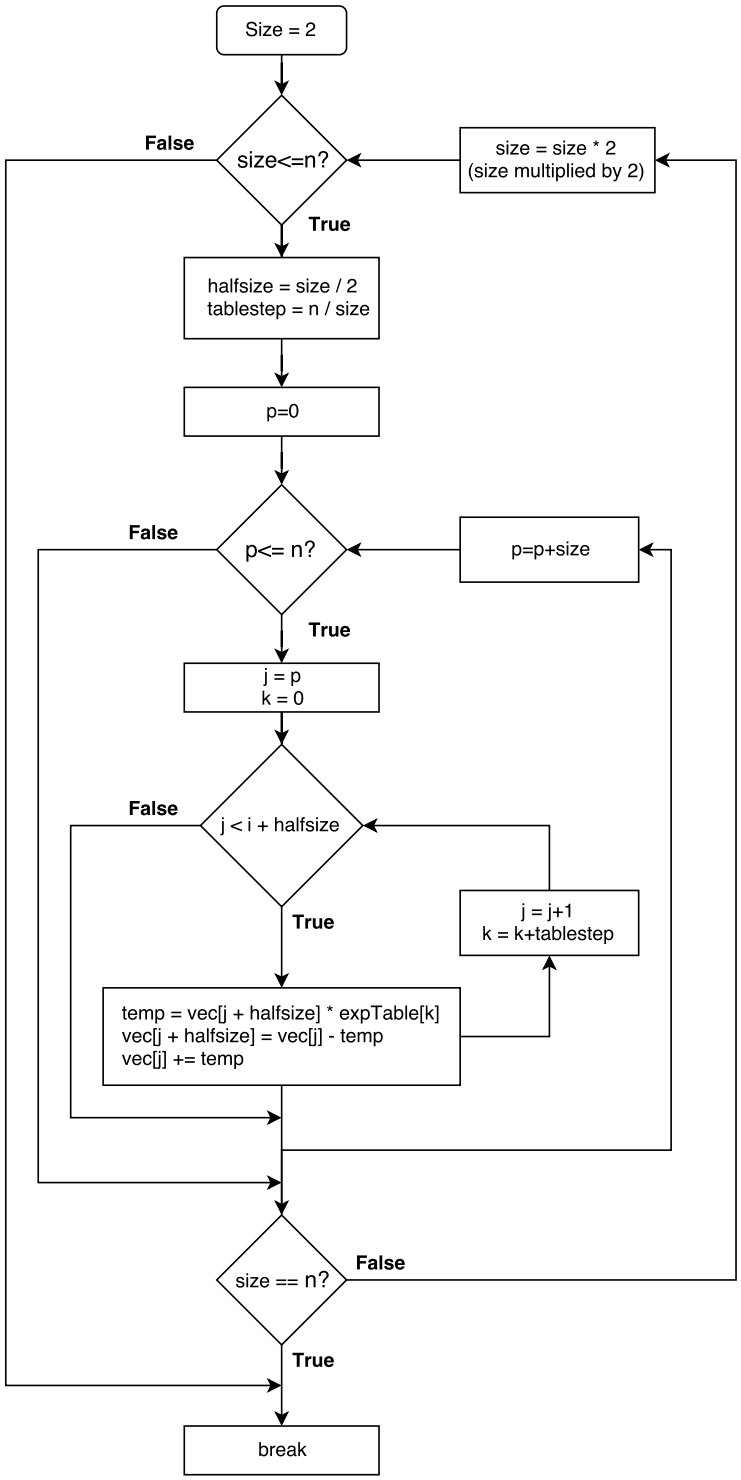


Figure 9.3: Cooley-Tukey algorithm

Bluestein algorithm

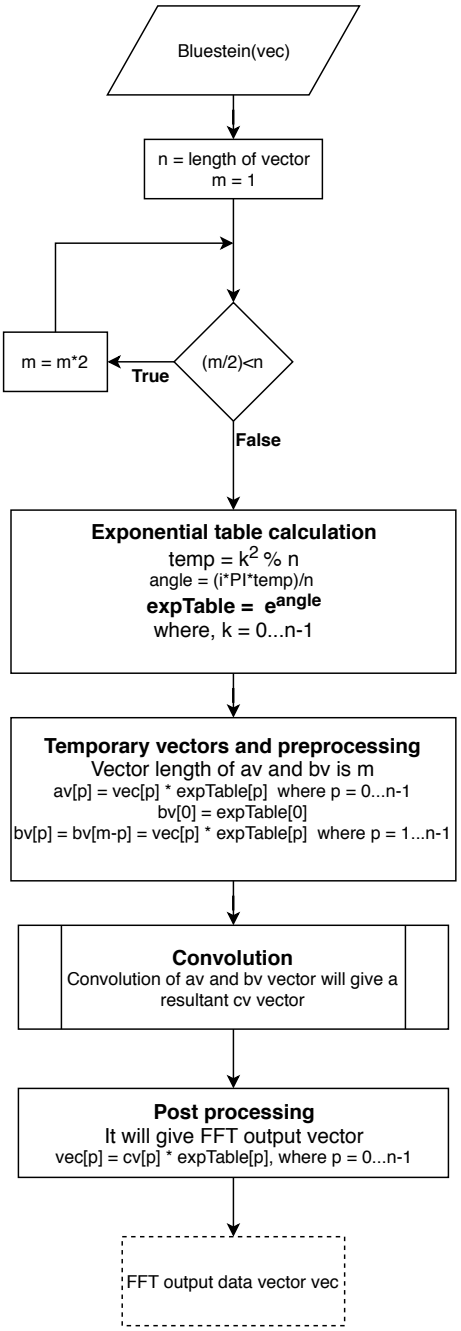


Figure 9.4: Bluestein algorithm

Convolution algorithm

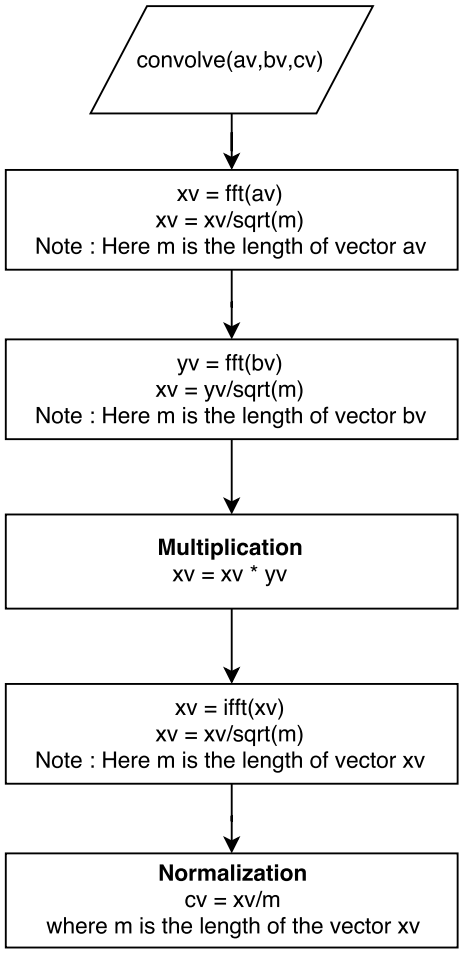


Figure 9.5: Circular convolution algorithm

Test example

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

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

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

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

Fs = 1e5;                 % Sampling frequency
T = 1/Fs;                 % Sampling period
L = 2^10;                 % Length of signal
t = (0:L-1)*(5*T);        % Time vector
f = linspace(-Fs/2,Fs/2,L);

%Choose for sig a value between [1, 7]
sig = 7;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
        X = S + 2*randn(size(t));
    case 2
        signal_title = 'Sinusoids with Random Noise';
        S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
        X = S + 2*randn(size(t));
    case 3
        signal_title = 'Single sinusoids';
        X = sin(2*pi*t);
    case 4
        signal_title = 'Summation of two sinusoids';
        X = sin(2*pi*t) + cos(2*pi*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';
        X = sin(2*pi*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 = ifft(X)*sqrt(length(X));% According to the definition of "optical fft"
%  with the (1/sqrt(N)) concept.
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(ifft(X)*sqrt(length(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') % fft from C++
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(ifft(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.

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

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

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

  // Next, Real and Imaginary vector to complex vector conversion
```

```
53    inTimeDomainComplex = reImVect2ComplexVector(re, im);

55    // calculate FFT
      clock_t begin = clock();
57    fourierTransformed = fft(inTimeDomainComplex);
      clock_t end = clock();
59    double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;

61    ////////////////////////////////// Section 3 ////////////////////////////////////
      //////// Save FFT calculated data (export "frequency_function.txt" ) /////////
63    ////////////////////////////////////////////////////////////////////////////////
      ofstream outFile;
65    complex<double> outFileData;
      outFile.open("frequency_function.txt");
67    outFile.precision(15);
      for (unsigned int i = 0; i <fourierTransformed.size(); i++){
69      outFile <<  fourierTransformed[i].real() << endl;
        outFile << fourierTransformed[i].imag() << endl;
71    }
      outFile.close();
73
      ////////////////////////////////// Section 4 ////////////////////////////////////
75    ////////////////////////////////// Display Section //////////////////////////////
      ////////////////////////////////////////////////////////////////////////////////
77    for (unsigned int i = 0; i < fourierTransformed.size(); i++){
        cout << fourierTransformed[i] << endl;  // Display all FFT calculated data
79    }
      cout << "\n\nTime elapsed to calculate FFT : " << elapsed_secs << " seconds"<<
        endl;
81    cout <<"\nTotal length of of data :"<< count << endl;
      getchar();
83    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.

Resultant analysis of various test signals

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

1. Signal with two sinusoids and random noise

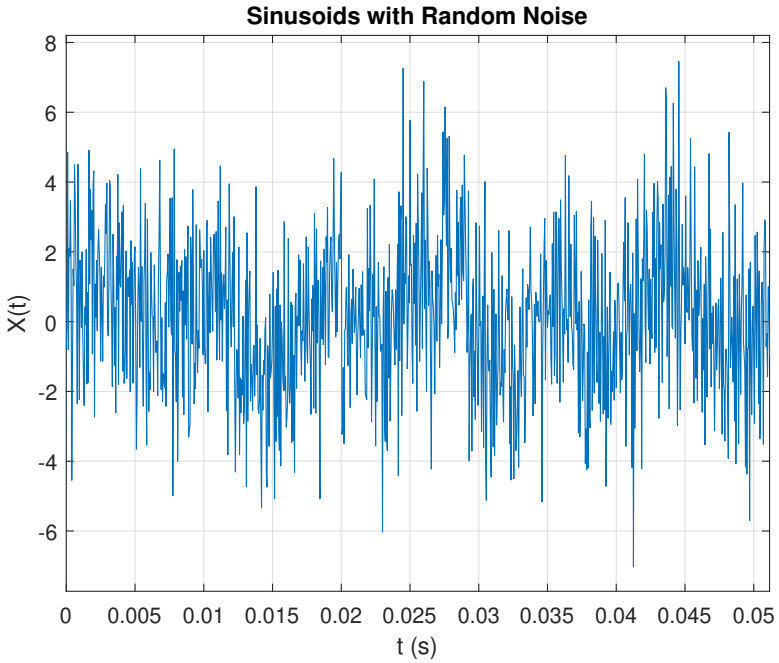


Figure 9.6: Random noise and two sinusoids

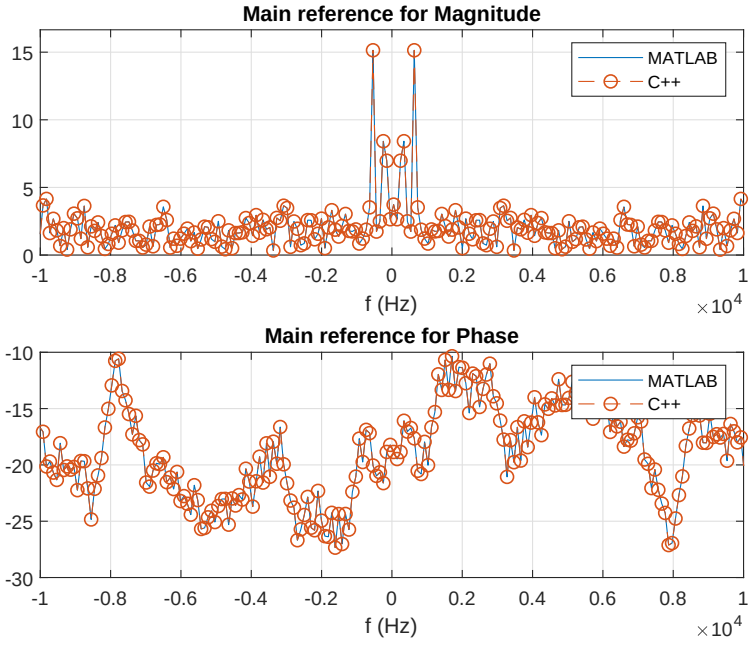


Figure 9.7: MATLAB and C++ comparison

2. Sinusoid with an exponent

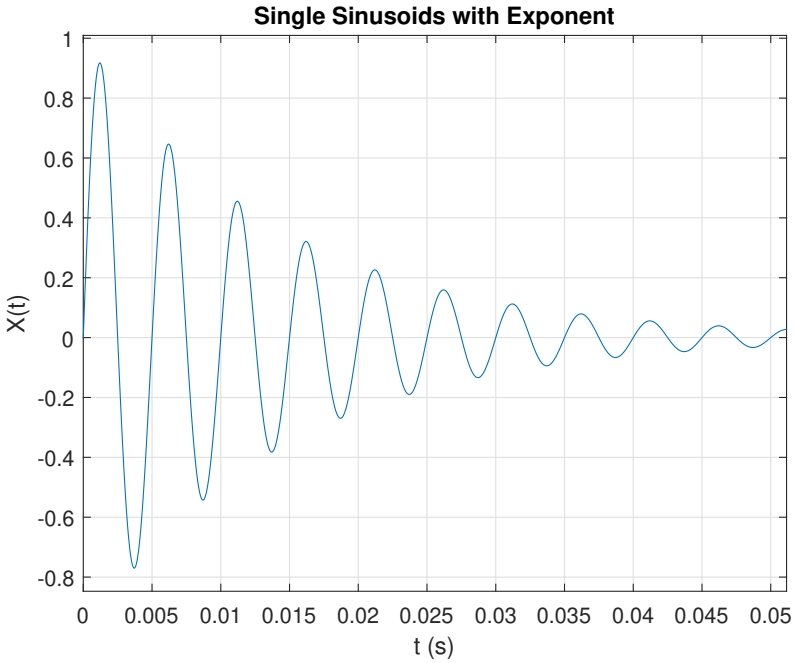


Figure 9.8: Sinusoids with exponent

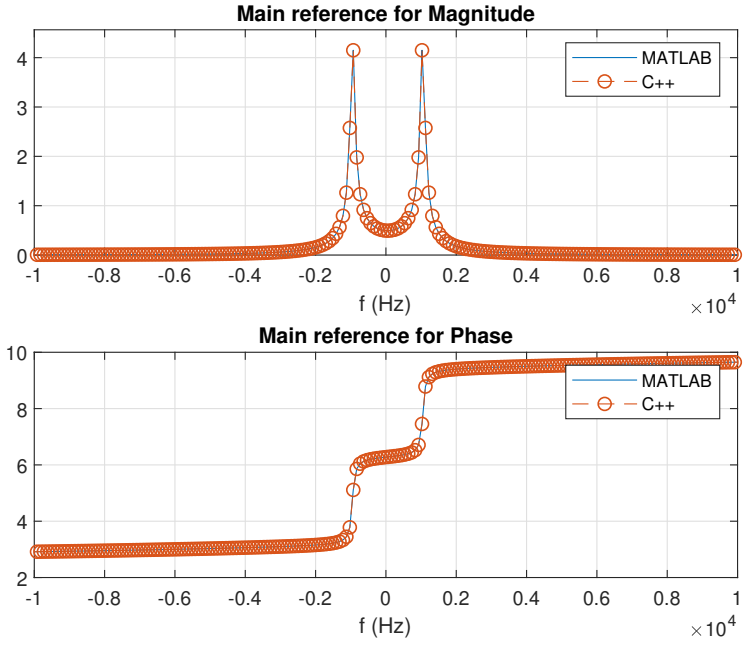


Figure 9.9: MATLAB and C++ comparison

3. Mixed signal

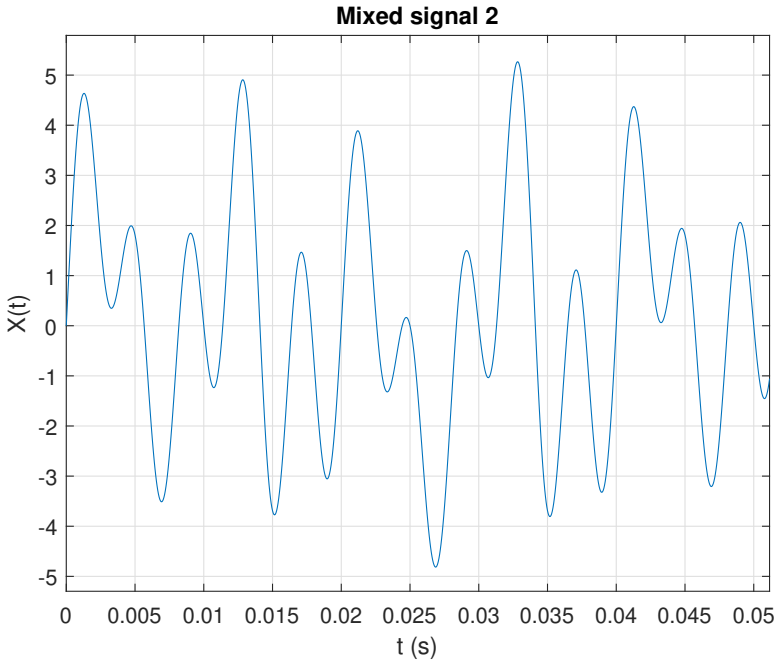


Figure 9.10: mixed signal

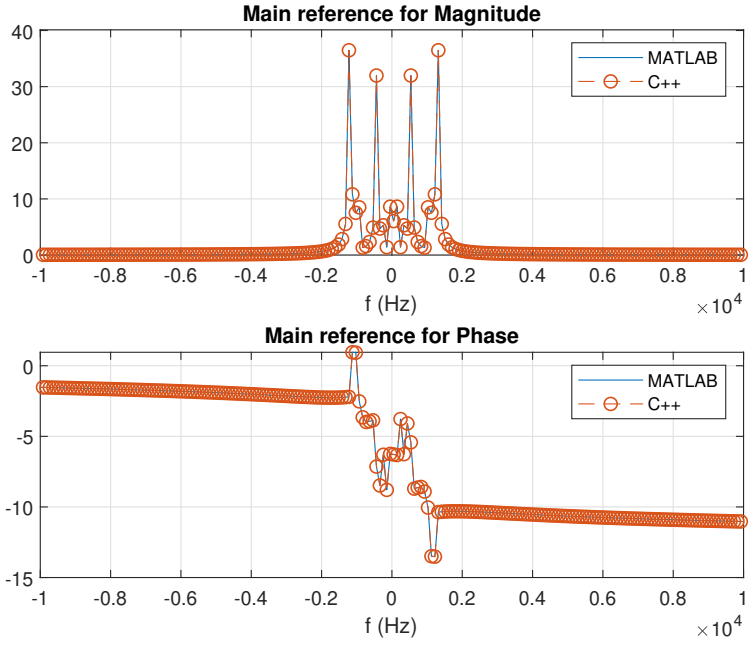


Figure 9.11: MATLAB and C++ comparison

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

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.

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.

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

References

[1] K. Ramamohan (Kamisetty Ramamohan) Rao, D. N. Kim, and J. J. Hwang. *Fast Fourier transform : algorithms and applications*. Springer, 2010, p. 423. ISBN: 9781402066290.

[2] Eleanor Chin-hwa Chu and Alan. George. *Inside the FFT black box : serial and parallel fast Fourier transform algorithms*. CRC Press, 2000, p. 312. ISBN: 9781420049961. URL: https://www.crcpress.com/Inside-the-FFT-Black-Box-Serial-and-Parallel-Fast-Fourier-Transform-Algorithms/Chu-George/p/book/9780849302701.

9.2 Overlap-Save Method

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

Overlap-save using impulse response

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 [1, 2] :
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.

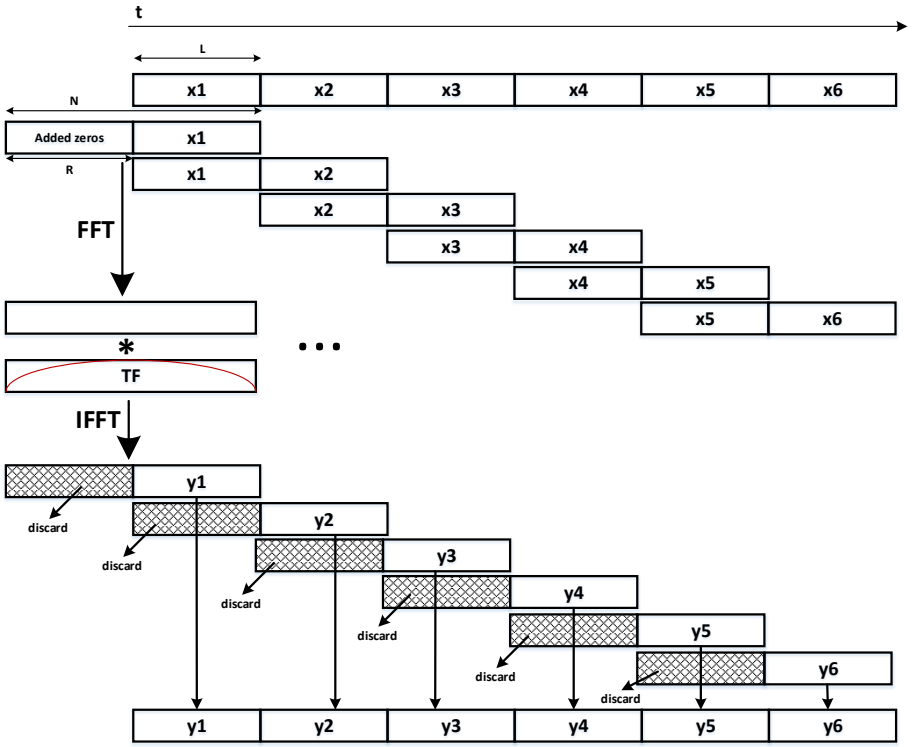


Figure 9.12: Illustration of Overlap-save method.

Function description

Traditionally, overlap-save method performs the linear convolution between discrete time-domain signal $x(n)$ and the filter impulse response $h(n)$ with the help of circular convolution. 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,

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

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

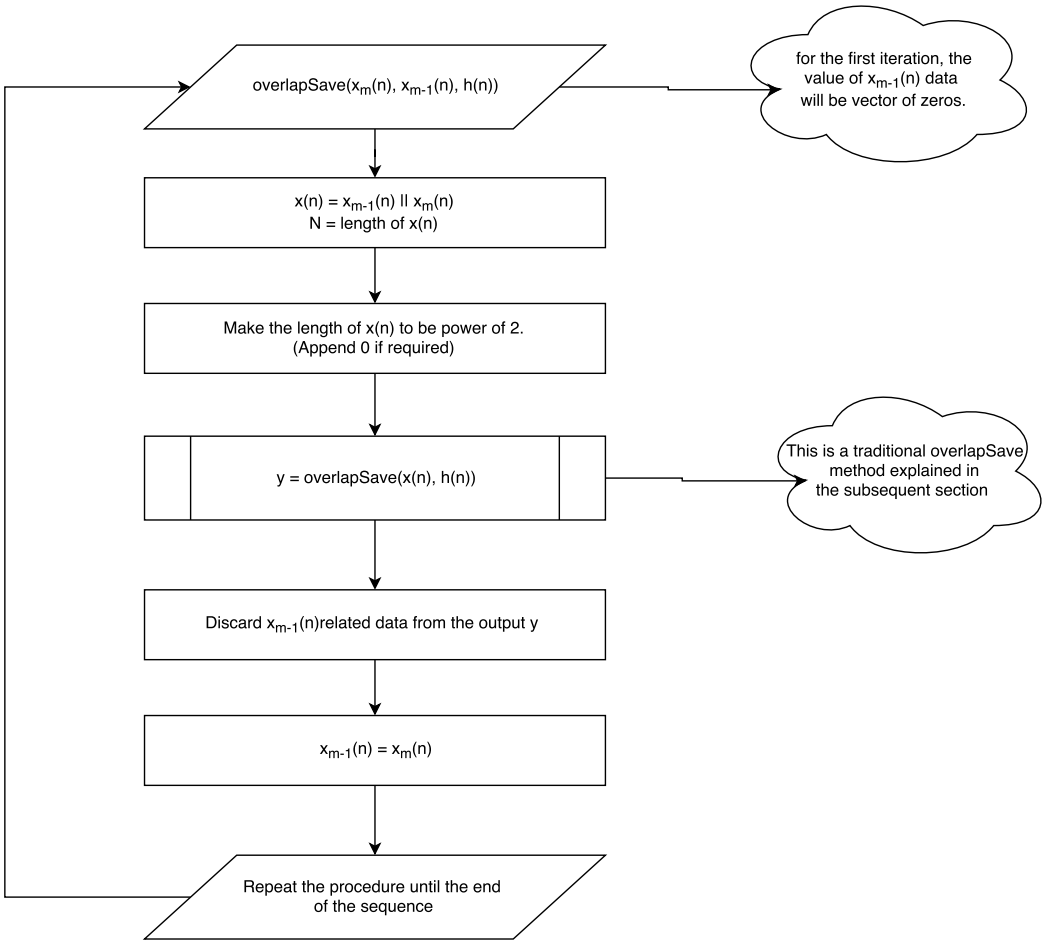


Figure 9.13: Flowchart for real-time overlap-save method

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

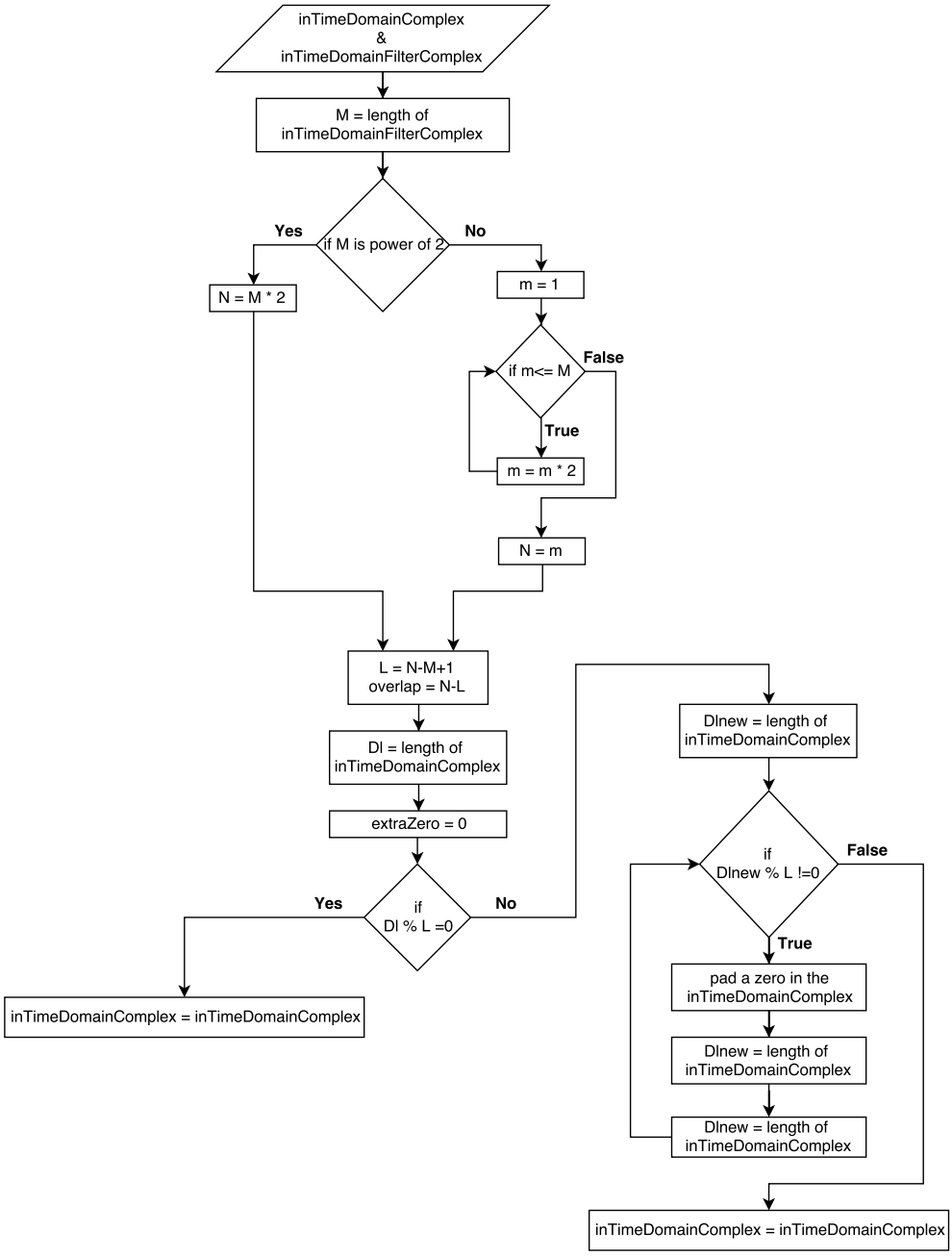


Figure 9.14: Flowchart for calculating length of FFT, data block and filter

2. Create matrix with overlap

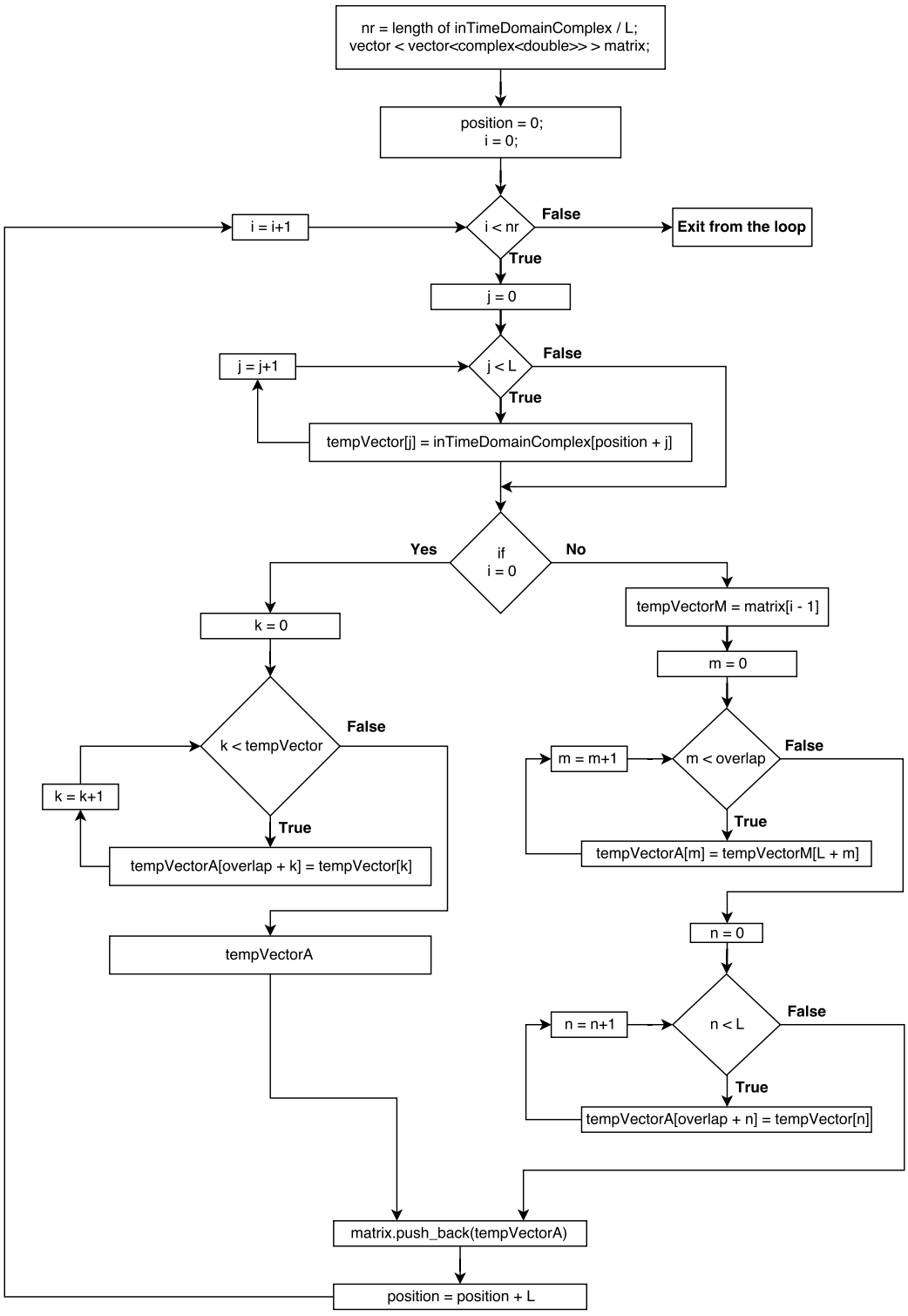


Figure 9.15: Flowchart of creating matrix with overlap

3. Convolution between filter and data blocks

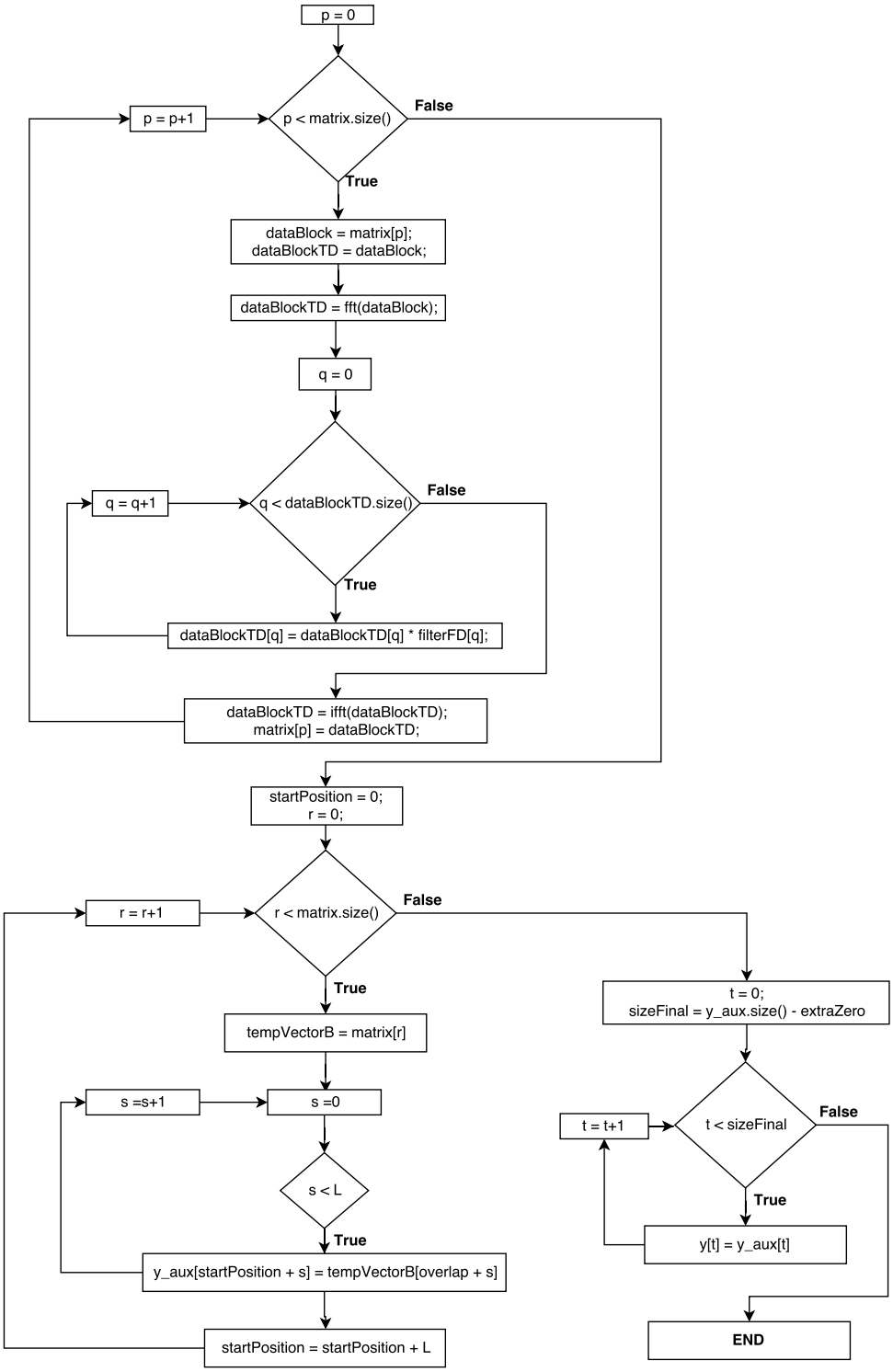


Figure 9.16: Flowchart of the convolution

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 = 7;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
        X = S + 2*randn(size(t));
    case 2
        signal_title = 'Sinusoids with Random Noise';
        S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
        X = S + 2*randn(size(t));
```

```
    case 3
        signal_title = 'Single sinusoids';
        X = sin(2*pi*t);
    case 4
        signal_title = 'Summation of two sinusoids';
        X = sin(2*pi*1205*t) + cos(2*pi*1750*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';
        X = sin(2*pi*250*t).*exp(-70*abs(t));
    case 6
        signal_title = 'Mixed signal 1';
        X = sin(2*pi*10*t).*exp(-t)+sin(2*pi*t)+7*sin(2*pi*+5*t)+7*cos(2*pi*+20*t
    )+5*sin(2*pi*+50*t);
    case 7
        signal_title = 'Mixed signal 2';
        X = 2*sin(2*pi*100*t).*exp(-t)+2.5*sin(2*pi*+250*t)+sin(2*pi*+50*t).*cos
    (2*pi*+20*t)+1.5*sin(2*pi*+50*t).*sin(2*pi*+150*t);
end

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


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

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


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

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

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

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

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

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

```

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

tic
for k=1:nr
    Ma(k,:)=x1(((k-1)*L+1):k*L);
    if k==1
        Ma1(k,:)=[zeros(1,overlap) Ma(k,:)];
        % Ma1(k,:)=[Ma(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;
```

```
14
int main()
16 {
  ////////////////////////////////// Section 1 ////////////////////////////////////
18  ////// Read the time_domain_data.txt and time_domain_filter.txt files  ///////
  ////////////////////////////////////////////////////////////////////////////////
20  ifstream inFile;
  double ch;
22  vector <double> inTimeDomain;
  inFile.open("time_domain_data.txt");
24
  // First data (at 0th position) appled to the ch it is similar to the "cin".
26  inFile >> ch;

28  // It'll count the length of the vector to verify with the MATLAB
  int count = 0;
30
  while (!inFile.eof())
32  {
    // push data one by one into the vector
34    inTimeDomain.push_back(ch);

36    // it'll increase the position of the data vector by 1 and read full vector.s
    inFile >> ch;
38    count++;
  }
40
  inFile.close(); // It is mandatory to close the file at the end.
42
  ifstream inFileFilter;
44  double chFilter;
  vector <double> inTimeDomainFilter;
46  inFileFilter.open("time_domain_filter.txt");
  inFileFilter >> chFilter;
48  int countFilter = 0;

50  while (!inFileFilter.eof())
  {
52    inTimeDomainFilter.push_back(chFilter);
    inFileFilter >> chFilter;
54    countFilter++;
  }
56  inFileFilter.close();

58  ////////////////////////////////// Section 2 ////////////////////////////////////
  ////////////////////////// Real to complex conversion //////////////////////////
60  ////////////////////////////////////////////////////////////////////////////////
  ////////////// For signal data  //////////////
62  vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
  vector <complex<double>> fourierTransformed;
64  vector <double> re(inTimeDomain.size());
  vector <double> im(inTimeDomain.size());
```

```
66
   for (unsigned int i = 0; i < inTimeDomain.size(); i++)
68 {
     re[i] = inTimeDomain[i]; // Real data of the signal
70   im[i] = 0;              // Imaginary data of the signal
   }
72 // Next, Real and Imaginary vector to complex vector conversion
   inTimeDomainComplex = reImVect2ComplexVector(re, im);
74
   ////////////// For filter data  //////////////
76 vector <complex<double>> inTimeDomainFilterComplex(inTimeDomainFilter.size());
   vector <double> reFilter(inTimeDomainFilter.size());
78 vector <double> imFilter(inTimeDomainFilter.size());

80 for (unsigned int i = 0; i < inTimeDomainFilter.size(); i++)
   {
82   reFilter[i] = inTimeDomainFilter[i];
     imFilter[i] = 0;
84 }
   inTimeDomainFilterComplex = reImVect2ComplexVector(reFilter, imFilter);
86
   ////////////////////////////////// Section 3 ////////////////////////////////
88 ////////////////////////////// Overlap & save ///////////////////////////////
   ////////////////////////////////////////////////////////////////////////////
90 vector <complex<double>> y;
   y = overlapSave(inTimeDomainComplex, inTimeDomainFilterComplex);
92
   ////////////////////////////////// Section 4 ////////////////////////////////
94 ////////////////////////////////// Save data ////////////////////////////////
   ////////////////////////////////////////////////////////////////////////////
96 ofstream outFile;
   complex<double> outFileData;
98 outFile.precision(20);
   outFile.open("overlap_save_data.txt");
100
   for (unsigned int i = 0; i <y.size(); i++)
102 {
     outFile << y[i].real() << endl;
104  outFile << y[i].imag() << endl;
   }
106 outFile.close();

108 cout << "Execution finished! Please hit enter to exit." << endl;
   getchar();
110 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.

Resultant analysis of various test signals

1. Signal with two sinusoids and random noise

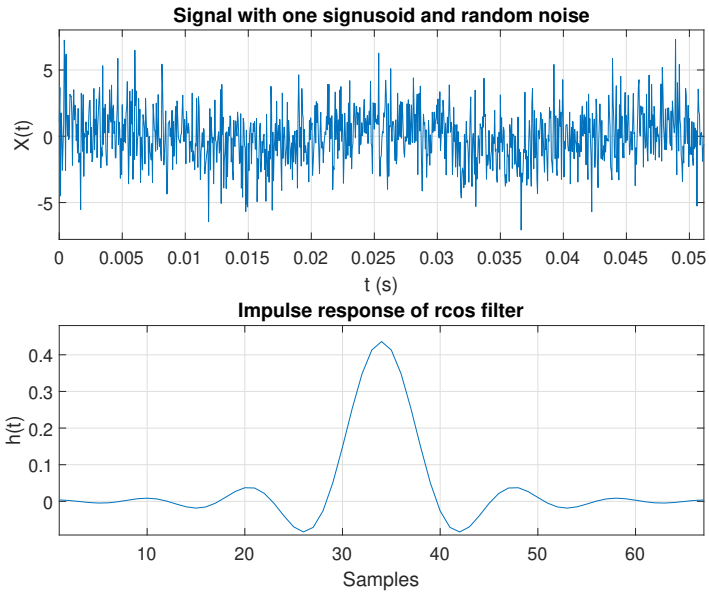


Figure 9.17: Random noise and two sinusoids signal & Impulse response of rcos filter

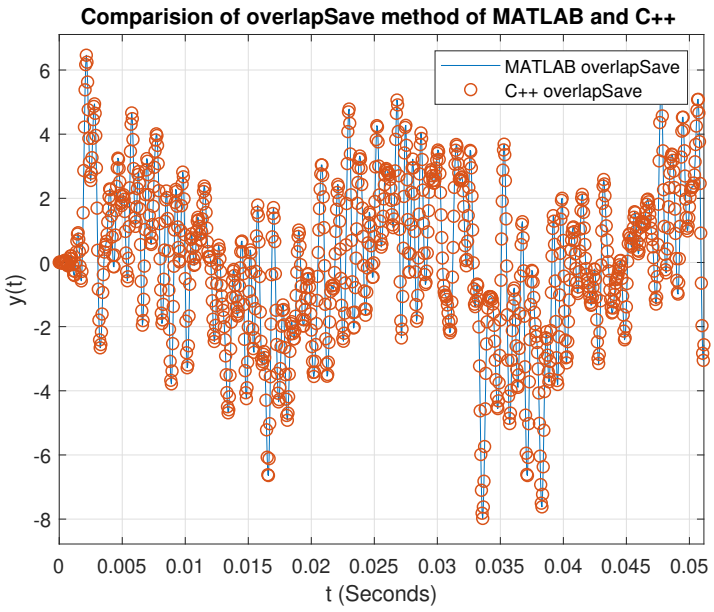


Figure 9.18: MATLAB and C++ comparison

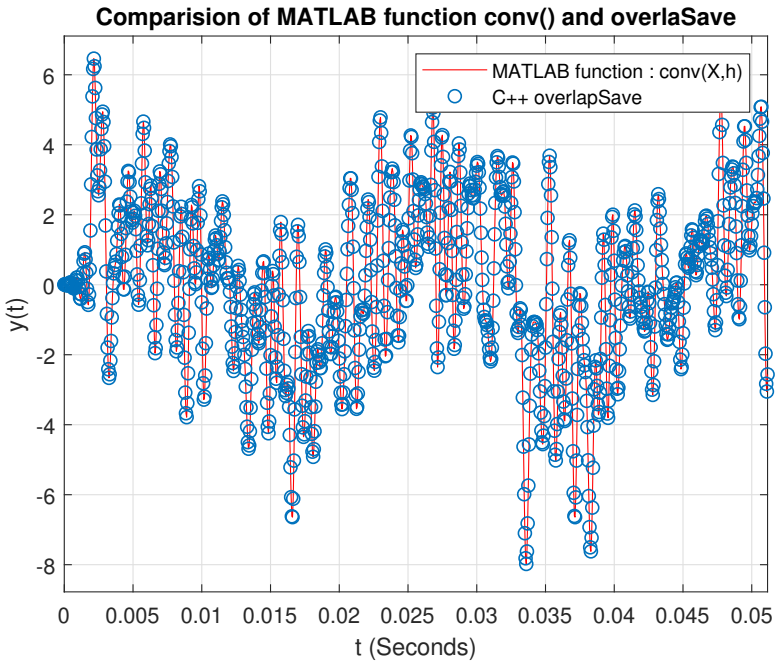


Figure 9.19: MATLAB function conv() and C++ overlapSave comparison

2. Mixed signal2

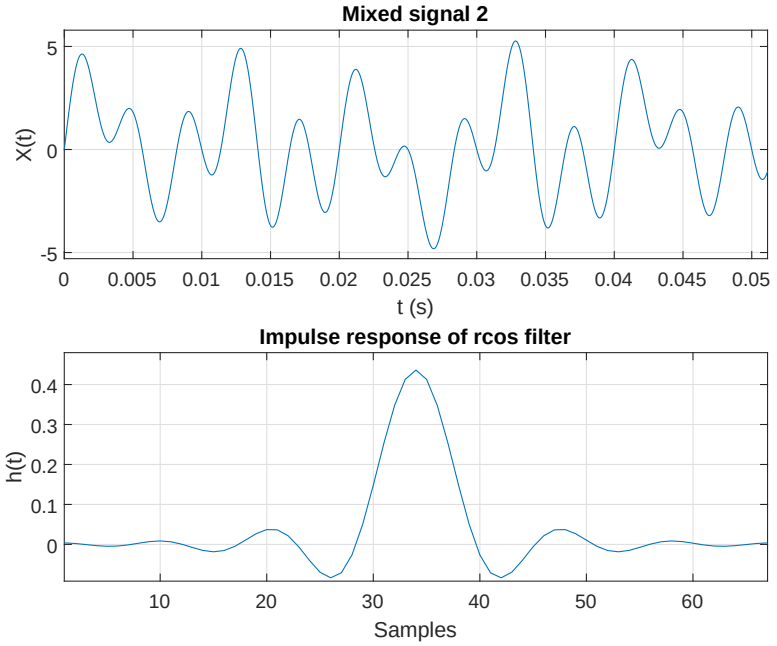


Figure 9.20: Mixed signal2 & Impulse response of rcos filter

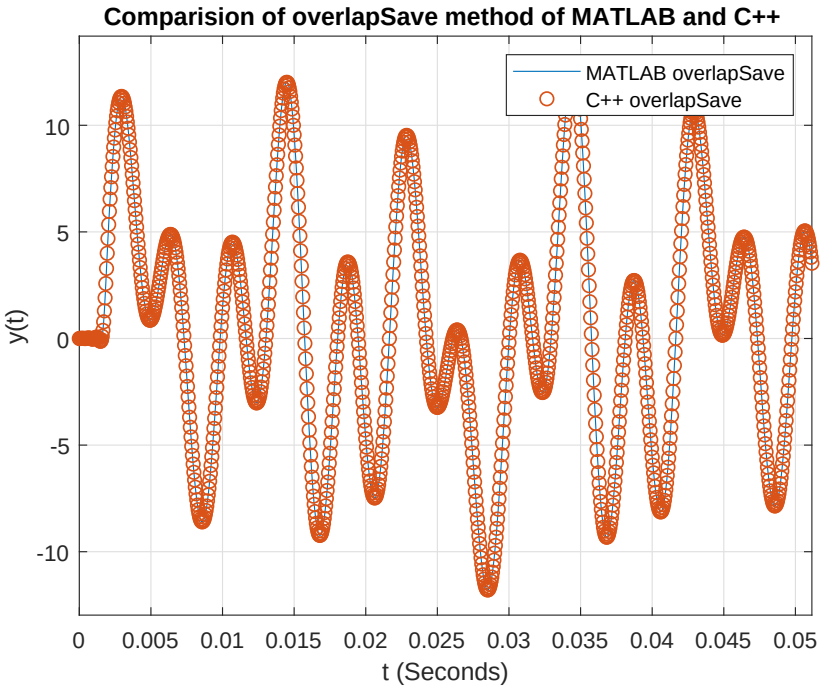


Figure 9.21: MATLAB and C++ comparison

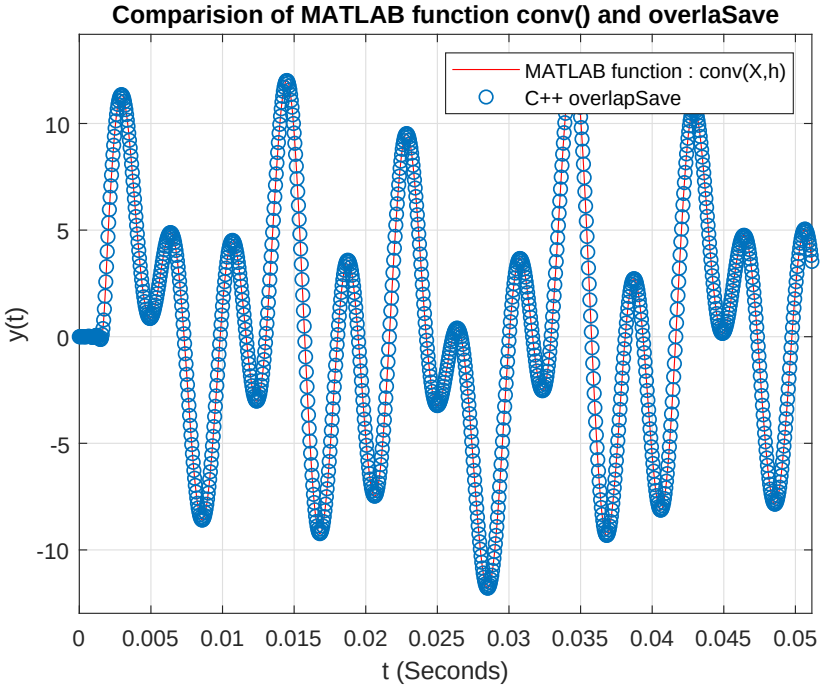


Figure 9.22: MATLAB function conv() and C++ overlapSave comparison

3. Sinusoid with exponent

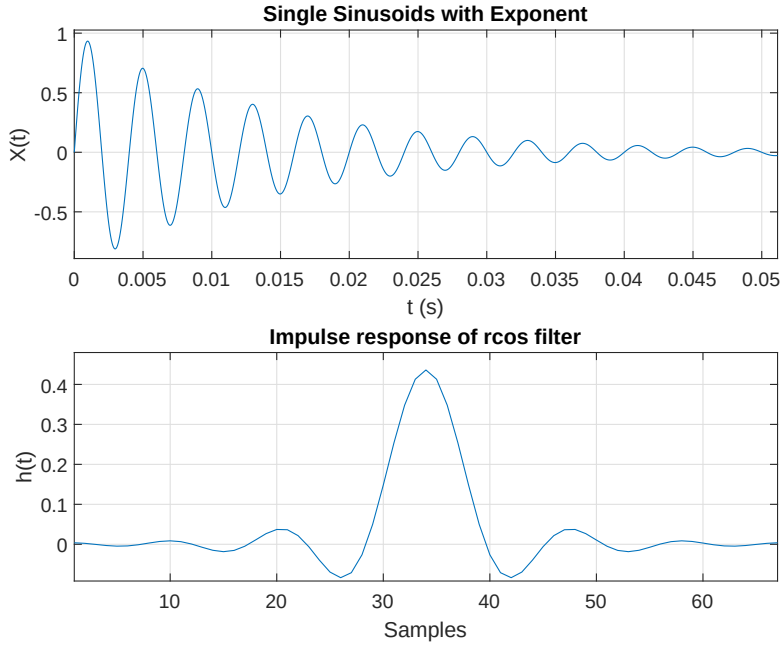


Figure 9.23: Sinusoid with exponent & Impulse response of Gaussian filter

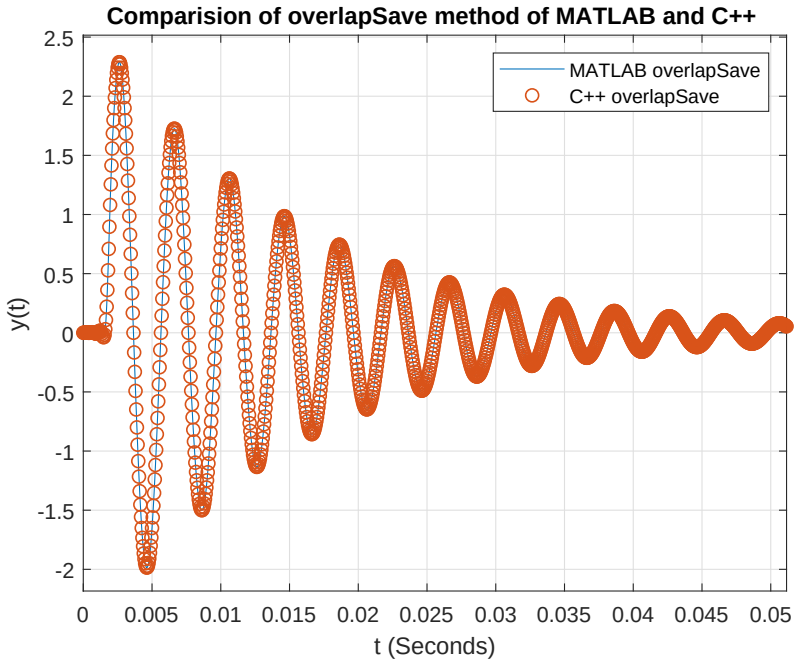


Figure 9.24: MATLAB and C++ comparison

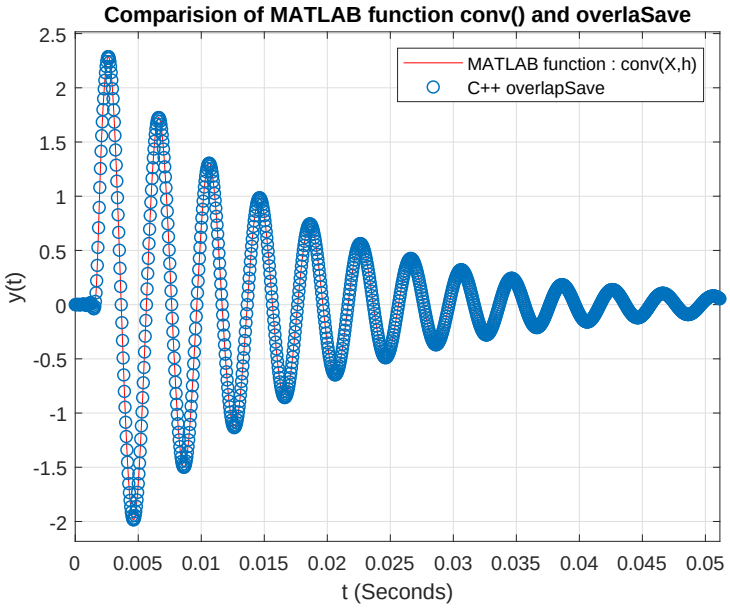


Figure 9.25: MATLAB function conv() and C++ overlapSave comparison

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

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

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

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

```
1 ////////////////////////////////////////////////////////
  ////////////////////// FD_Filter ///////////////////////
```

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

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

  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 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;

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

  ////////////////////////// Impulse response ////////////////////
  ////////////////////////////////////////////////////////////////
  vector<t_complex> impulseResponseComplex(impulseResponse.size());
  vector<t_real> irImag(impulseResponse.size());

  impulseResponseComplex = reImVect2ComplexVector(impulseResponse, irImag);

  ////////////////////// OverlapSave function ////////////////////
```

```
55  ////////////////////////////////////////////////////////////
    vector<t_complex> pcinitialize(process);
57  if (K == 0)
    {
59    if (symmetryTypeIr == "Symmetric")
        previousCopy = pcinitialize;
61
      else
63      previousCopy = currentCopy;
    }
65
    vector<t_complex> OUT = overlapSaveImpulseResponse(currentCopy, previousCopy,
      impulseResponseComplex);
67
    previousCopy = currentCopy;
69  K = K + 1;

71  // Bufferput
    for (int i = 0; i < process; i++){
73    t_real val;
      val = OUT[i].real();
75    outputSignals[0]->bufferPut((t_real)(val));
    }
77
    return true;
79  }


81

83  ///// Overlap save::Transfer Function : 2018 - 11 - 10 //////
    ////////////////////////////////////////////////////////////
85  ////////////////////// FD_Filter_20181110 ///////////////////
    ////////////////////////////////////////////////////////////
87  void FD_Filter_20181110::initializeFD_Filter_20181110(void)
    {
89    outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
      outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
91    outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

93    if (!getSeeBeginningOfTransferFunction()) {
        int aux = (int)(((double)transferFunctionLength) / 4) + 1;
95      outputSignals[0]->setFirstValueToBeSaved(aux);
      }
97
      if (saveTransferFunction)
99    {
        ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
101     fileHandler << "// ### HEADER TERMINATOR ###\n";

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

```
107     t_real f;
        for (int k = 0; k < transferFunction.size(); k++)
109     {
          f = -fWindow / 2 + k * df;
111       fileHandler << f << " " << transferFunction[k] << "\n";
        }
113     fileHandler.close();
    }
115 }

117 bool FD_Filter_20181110::runBlock(void)
    {
119   bool alive{ false };

121   int ready = inputSignals[0]->ready();
      int space = outputSignals[0]->space();
123   int process = min(ready, space);
      if (process == 0) return false;
125
      ////////////////////  currentCopy ////////////////////////
127   ////////////////////////////////////////////////////////////
      vector<t_complex> currentCopy(process); // Get the Input signal
129   t_real input;
      for (int i = 0; i < process; i++) {
131     inputSignals[0]->bufferGet(&input);
        currentCopy.at(i) = { input, 0};
133   }

135   vector<t_complex> pcinitialize(process);
      if (K == 0)
137   {
        if (symmetryTypeTf == "Symmetric")
139       previousCopy = pcinitialize;

141     else
          previousCopy = currentCopy;
143   }

145   if (previousCopy.size() != currentCopy.size())
      {
147     vector<t_complex> currentCopyAux;
        while (currentCopyAux.size() <= previousCopy.size())
149     {
          for (int i = 0; i < currentCopy.size(); i++)
151       {
            currentCopyAux.push_back(currentCopy[i]);
153       }
        }
155
        vector<t_complex> cc(previousCopy.size());
157     for (int i = 0; i < previousCopy.size(); i++)
```

```
    {
159   cc[i] = currentCopyAux[i];
    }
161
    currentCopy = cc;
163
  }
165
  vector<t_complex> out;
167   out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(
    transferFunction));

169 // Bufferput
  for (int i = 0; i < process; i++) {
171   t_real val;
    val = out[i].real();
173   outputSignals[0]->bufferPut((t_real)(val));
  }
175
  K = K + 1;
177 previousCopy = currentCopy;

179 return true;
}
```

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 9.29.

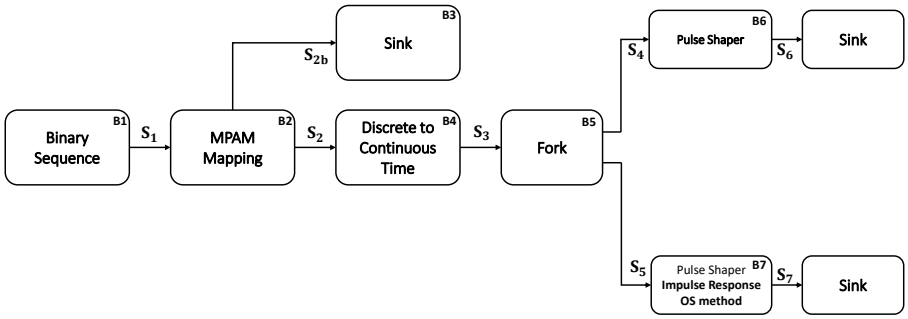


Figure 9.26: Real-time overlap-save example setup

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

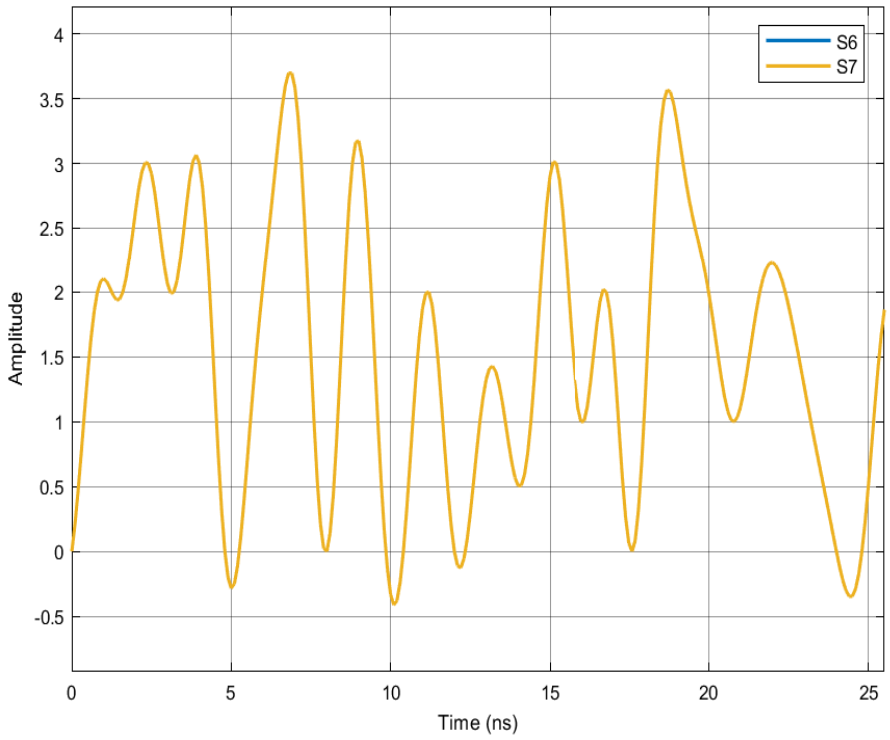


Figure 9.27: Comparison of signal S6 and S7

Overlap-save using transfer function

The process flow of the overlap save method using transfer function is shown in th Figure 9.28. The input signal is divided into the overlapping blocks x_i of M samples. The amount overlapping is $M - L$ and for each block the following computations are performed.

Step 1: The block x_i is transformed into the M-point DFT.

Step 2: The transformed block vector coefficients are multiplied term by term with transfer function of length M.

Step 3: Computer M-point IDFT of the result achieved from multiplication of transfer function and block x_i.

Step 4: Only L central points of the resultant block are kept (Discard d point at both ends of the result). Here the value of the $M - L$ must be an even number to preserve the symmetry.

Step 5: Concatenation of the output blocks to form a filtered signal.

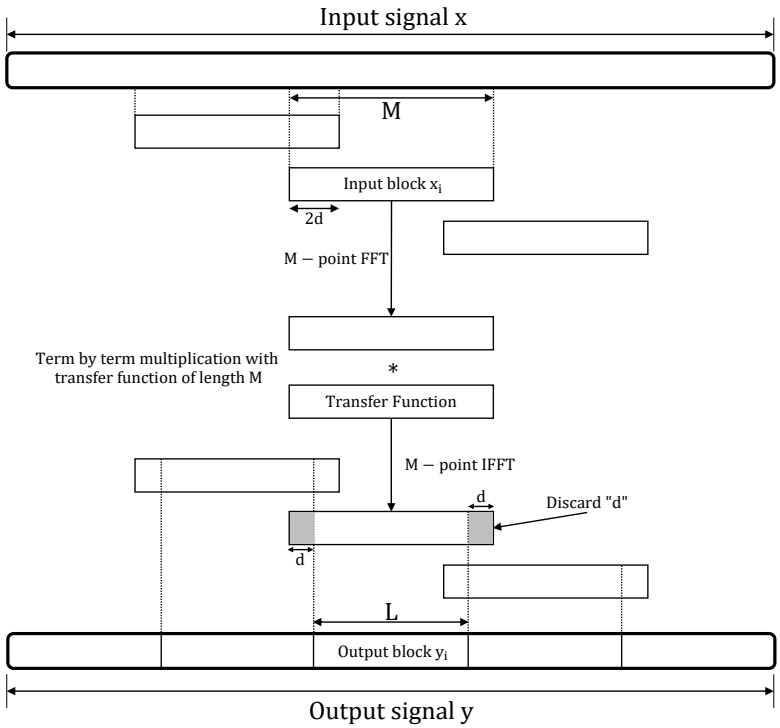


Figure 9.28: Process of the overlap save using transfer function

Function description

The structure of the new function is as follows,

$$y(n) = overlapSaveTransferFunction(x_m(n), x_{m-1}(n), tf(k))$$

Here, $x_m(n)$, $x_{m-1}(n)$ and $tf(k)$ 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 $tf(k)$.

Test example of overlap-save impulse response and transfer function in Netxpto simulator

This section explains the steps of comparing the result of impulse response and transfer function based overlap-save methods.
Step 1 : Open the folder namely **overlapSaveRealTimeIrTf_test_test** by following the path "/algorithms/overlapSave/overlapSaveRealTimeIrTf_test_test".
Step 2 : Find the **overlapSaveRealTimeIrTf_test_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 definitions of two different filter class namely **FD_Filter** and **FD_Filter_20181110** for applying overlap-save method using impulse response and transfer function, respectively.

```
////// Overlap save :: Impulse Response : 2018-03-06 ////////
////////////////////////////////////////////////////////////
////////////////////////// FD_Filter ///////////////////////
////////////////////////////////////////////////////////////
void FD_Filter::initializeFD_Filter(void)
{
  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

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

  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 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;

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

  ////////////////////////// Impulse response ////////////////////
  ////////////////////////////////////////////////////////////////
  vector<t_complex> impulseResponseComplex(impulseResponse.size());
  vector<t_real> irImag(impulseResponse.size());
```

```
53    impulseResponseComplex = reImVect2ComplexVector(impulseResponse, irImag);

55    ////////////////////// OverlapSave function ////////////////////
      ////////////////////////////////////////////////////////////////
57    vector<t_complex> pcinitialize(process);
      if (K == 0)
59    {
        if (symmetryTypeIr == "Symmetric")
61        previousCopy = pcinitialize;

63      else
          previousCopy = currentCopy;
65    }

67    vector<t_complex> OUT = overlapSaveImpulseResponse(currentCopy, previousCopy,
        impulseResponseComplex);

69    previousCopy = currentCopy;
      K = K + 1;
71
      // Bufferput
73    for (int i = 0; i < process; i++){
        t_real val;
75      val = OUT[i].real();
        outputSignals[0]->bufferPut((t_real)(val));
77    }

79    return true;
}
81


83
///// Overlap save::Transfer Function : 2018 - 11 - 10 //////
85////////////////////////////////////////////////////////////////
//////////////////////// FD_Filter_20181110 ////////////////////
87////////////////////////////////////////////////////////////////
void FD_Filter_20181110::initializeFD_Filter_20181110(void)
89{
    outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
91  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
    outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;
93
    if (!getSeeBeginningOfTransferFunction()) {
95    int aux = (int)(((double)transferFunctionLength) / 4) + 1;
      outputSignals[0]->setFirstValueToBeSaved(aux);
97  }

99  if (saveTransferFunction)
    {
101   ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
      fileHandler << "// ### HEADER TERMINATOR ###\n";
103
```

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

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

bool FD_Filter_20181110::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;

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

  vector<t_complex> pcinitialize(process);
  if (K == 0)
  {
    if (symmetryTypeTf == "Symmetric")
      previousCopy = pcinitialize;

    else
      previousCopy = currentCopy;
  }

  if (previousCopy.size() != currentCopy.size())
  {
    vector<t_complex> currentCopyAux;
    while (currentCopyAux.size() <= previousCopy.size())
    {
      for (int i = 0; i < currentCopy.size(); i++)
      {
        currentCopyAux.push_back(currentCopy[i]);
      }
    }
```

```
157     vector<t_complex> cc(previousCopy.size());
        for (int i = 0; i < previousCopy.size(); i++)
159     {
          cc[i] = currentCopyAux[i];
161     }

163     currentCopy = cc;

165   }

167   vector<t_complex> out;
        out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(
        transferFunction));
169
      // Bufferput
171   for (int i = 0; i < process; i++) {
        t_real val;
173     val = out[i].real();
        outputSignals[0]->bufferPut((t_real)(val));
175   }

177   K = K + 1;
      previousCopy = currentCopy;
179
      return true;
181 }
```

Listing 9.6: filter_20180306.cpp code

Step 3 : Next, open **overlapSaveRealTimeIrTf_test.cpp** file in the same project and run it. Graphically, this files represents the following Figure 9.29.

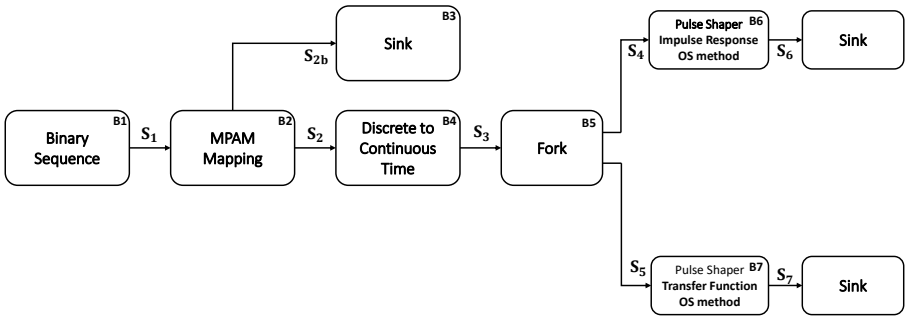


Figure 9.29: Real-time overlap-save example setup

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

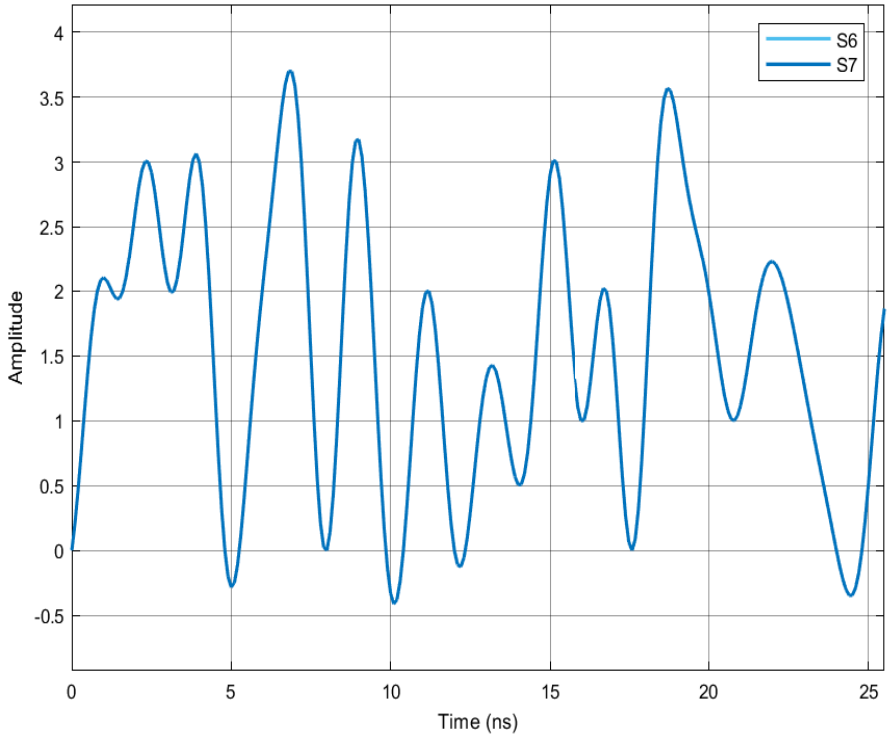


Figure 9.30: Comparison of signal S6 and S7

References

[1] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Pub, 1999. ISBN: 0966017676.

[2] Richard E. Blahut and Richard E. *Fast algorithms for digital signal processing*. Addison-Wesley Pub. Co, 1985, p. 441. ISBN: 0201101556. URL: https://dl.acm.org/citation.cfm?id=537283.

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 three distinct class namely **FIR_Filter**, **FD_Filter**, and **FD_Filter_20181110** which facilitate filtering operation in time-domain (using impulse response), overlap-save (using impulse response), and overlap-save (using transfer function), respectively (see Figure 9.31). The $filter\_*.cpp$ file contains the definitions of all the functions declared in the **FIR_Filter**, **FD_Filter** and **FD_Filter_20181110**.

In the Figure 9.31, the declared function **bool** $runblock(void)$ in all the classes

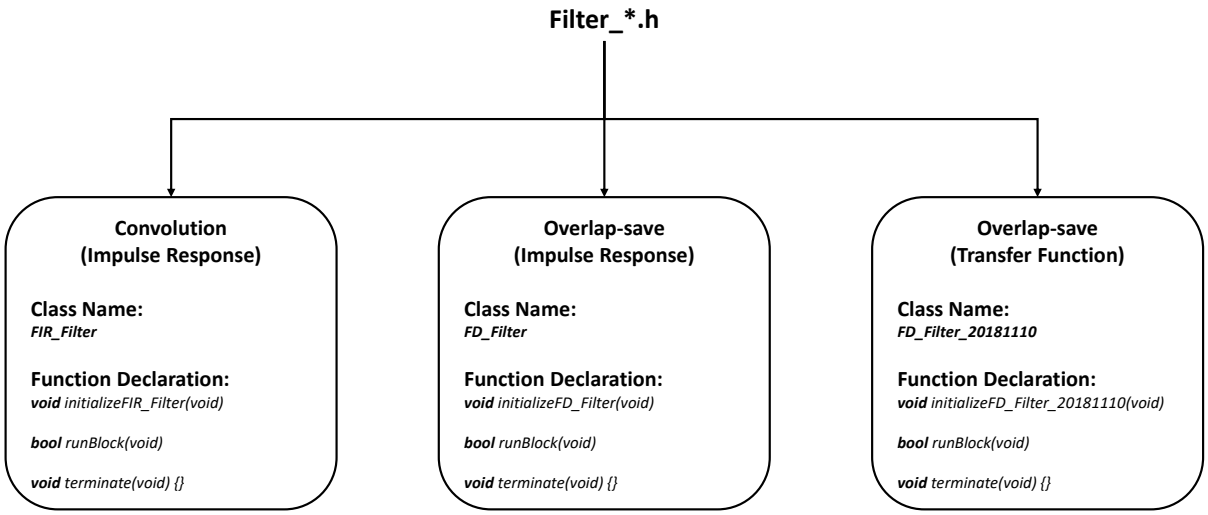


Figure 9.31: Filter class

FIR_Filter, **FD_Filter** and **FD_Filter_20181110** represent the definition of time domain convolution filtering (using impulse response), frequency domain overlap-save filtering (using impulse response), and frequency-domain overlap-save filtering (using transfer function), respectively. [1, 2, 3] .

All those function declared in the $filter\_*.h$ file are defined in the $filter\_*.cpp$ file. The definition of **bool** $runblock(void)$ function for all three classes are the following,

```
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;
};
```

Listing 9.7: 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;

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

  ////////////////////////// Impulse response ////////////////////
  ////////////////////////////////////////////////////////////
  vector<t_complex> impulseResponseComplex(impulseResponse.size());
  vector<t_real> irImag(impulseResponse.size());

  impulseResponseComplex = reImVect2ComplexVector(impulseResponse, irImag);

  ////////////////////// OverlapSave function ////////////////////
  ////////////////////////////////////////////////////////////
  vector<t_complex> pcinitialize(process);
  if (K == 0)
  {
    if (symmetryTypeIr == "Symmetric")
      previousCopy = pcinitialize;
```

```
    else
      previousCopy = currentCopy;
  }

  vector<t_complex> OUT = overlapSaveImpulseResponse(currentCopy, previousCopy,
    impulseResponseComplex);

  previousCopy = currentCopy;
  K = K + 1;

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

  return true;
}
```

Listing 9.8: Definition of **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;

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

  vector<t_complex> pcinitialize(process);
  if (K == 0)
  {
    if (symmetryTypeTf == "Symmetric")
      previousCopy = pcinitialize;

    else
      previousCopy = currentCopy;
  }

  if (previousCopy.size() != currentCopy.size())
  {
    vector<t_complex> currentCopyAux;
    while (currentCopyAux.size() <= previousCopy.size())
```

```
32      {
          for (int i = 0; i < currentCopy.size(); i++)
34        {
            currentCopyAux.push_back(currentCopy[i]);
36        }
        }
38
        vector<t_complex> cc(previousCopy.size());
40      for (int i = 0; i < previousCopy.size(); i++)
        {
42        cc[i] = currentCopyAux[i];
        }
44
        currentCopy = cc;
46
      }
48
      vector<t_complex> out;
50      out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(
        transferFunction));

52    // Bufferput
      for (int i = 0; i < process; i++) {
54      t_real val;
        val = out[i].real();
56      outputSignals[0]->bufferPut((t_real)(val));
      }
58
     K = K + 1;
60   previousCopy = currentCopy;

62   return true;
}
```

Listing 9.9: Definition of **bool FD_Filter_20181110::runBlock(void)**

All three classes 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. As described earlier, 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**, **FD_Filter** and **FD_Filter_20181110** classes can be used for the filtering operation with the help of impulse response and transfer function of the pulse shaping filter. The impulse response for the both **FIR_Filter** and **FD_Filter** classes will be generated by a *pulse_shaper.cpp* and *pulse_shaper_20180306.cpp* files respectively and applied to the **bool** $runblock(void)$ block as shown in Figure 9.32. The transfer function for

the class FD_Filter_20181110 will be generated by *pulse_shaper_20181110.cpp* and applied to the **bool** $runblock(void)$ as shown in Figure 9.32.

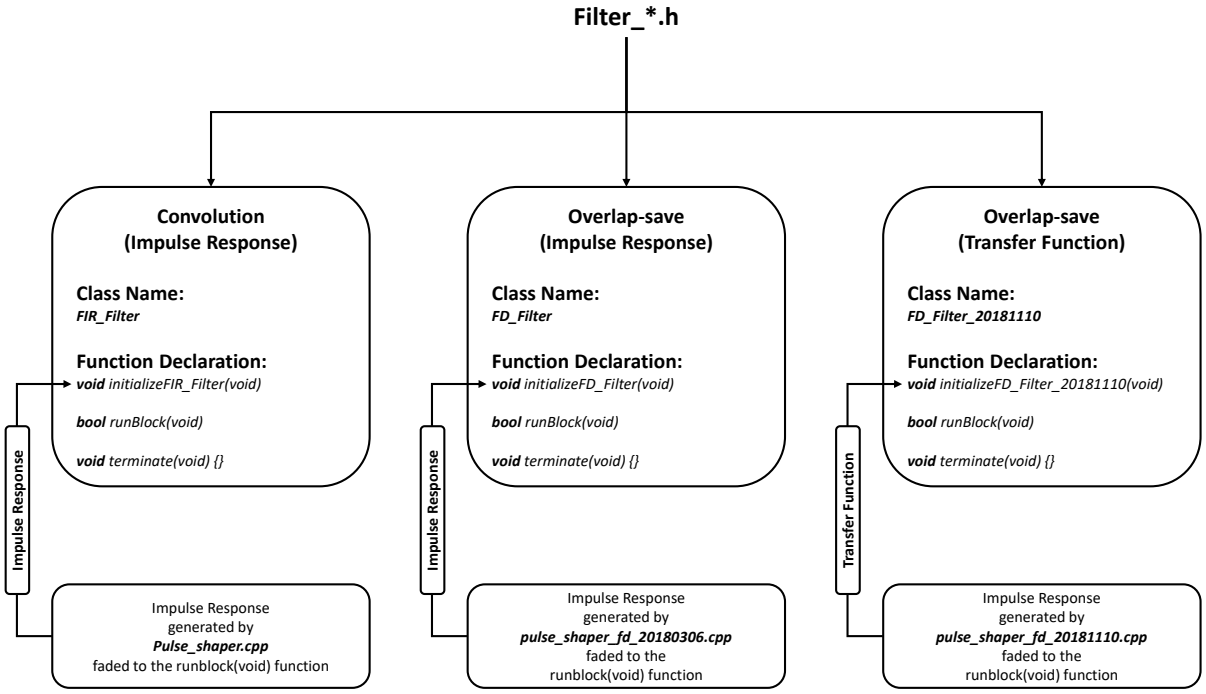


Figure 9.32: Pulse shaping using **filter_*.h**

Example of pulse shaping filtering : Procedural steps

This section explains the steps of pulse shaping filtering with the help of convolution and overlap-save method using its impulse response. 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 9.33.

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

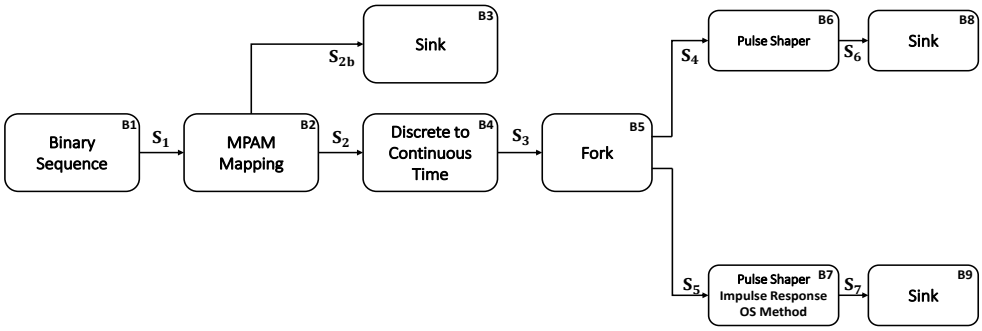


Figure 9.33: Filter test setup

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 9.34, 9.35 and 9.36 display the comparison of the output signals **S6.sgn** and **S7.sgn** for the raised cosine, root raised cosine and Gaussian pulse shaping filter, respectively. Similarly, we can verify the result of the transfer function based overlap-save method.

Case 1 : Raised cosine

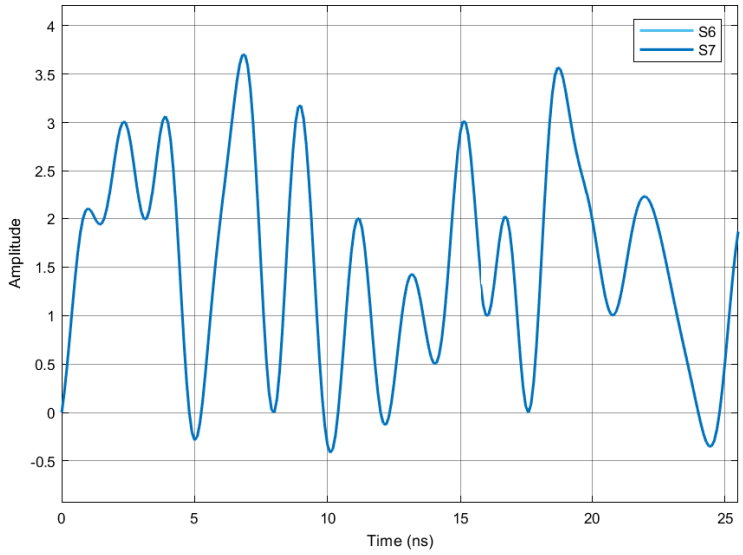


Figure 9.34: Raised cosine pulse shaping results comparison

Case 2 : Root raised cosine

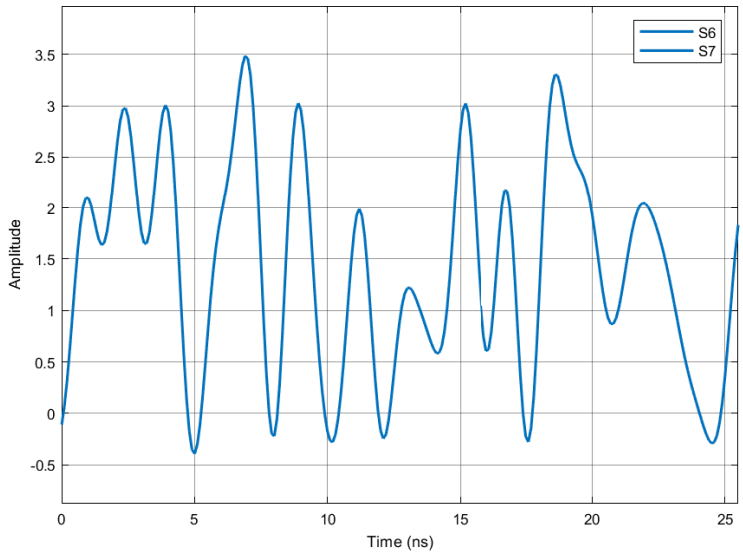


Figure 9.35: Root raised cosine pulse shaping result

Case 3 : Gaussian

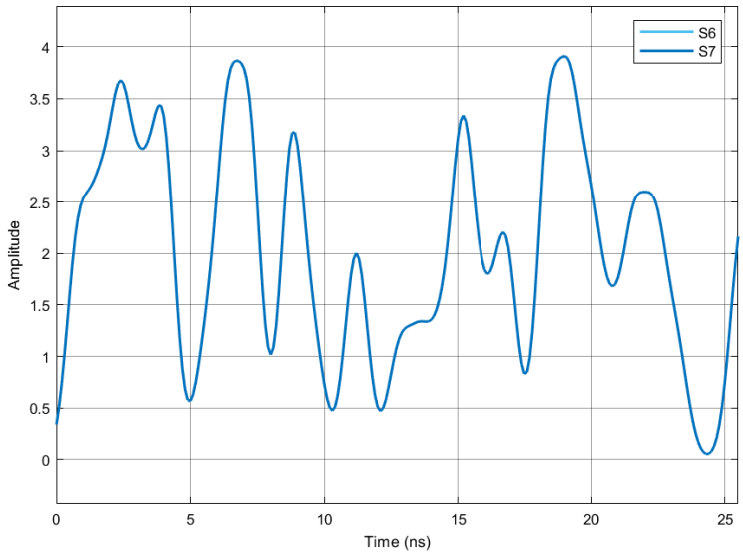


Figure 9.36: Gaussian pulse shaping results comparison

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.

References

[1] Sen M. (Sen-Maw) Kuo, Bob H. Lee, and Wenshun. Tian. *Real-time digital signal processing : fundamentals, implementations and applications*. ISBN: 9781118414323. URL: https://www.wiley.com/en-us/Real+Time+Digital+Signal+Processing%7B%5C%%7D3A+Fundamentals%7B%5C%%7D2C+Implementations+and+Applications%7B%5C%%7D2C+3rd+Edition-p-9781118414323.

[2] Theodore S. Rappaport. *Wireless communications : principles and practice*. Prentice Hall PTR, 2002, p. 707. ISBN: 0130422320.

[3] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, 1999, p. 870. ISBN: 0137549202. URL: https://dl.acm.org/citation.cfm?id=294797.

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 [1] (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 [2]

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

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 9.37. The filtering process will start only after acquiring first

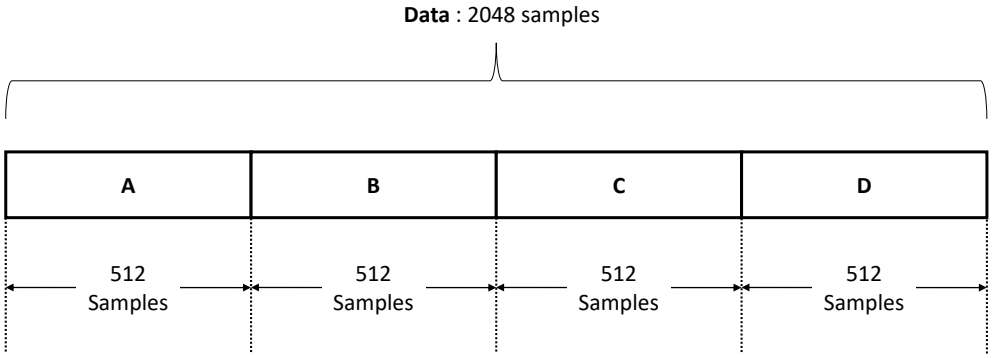


Figure 9.37: Logical flow

two blocks A and B (see iteration 1 in Figure 9.38), 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"

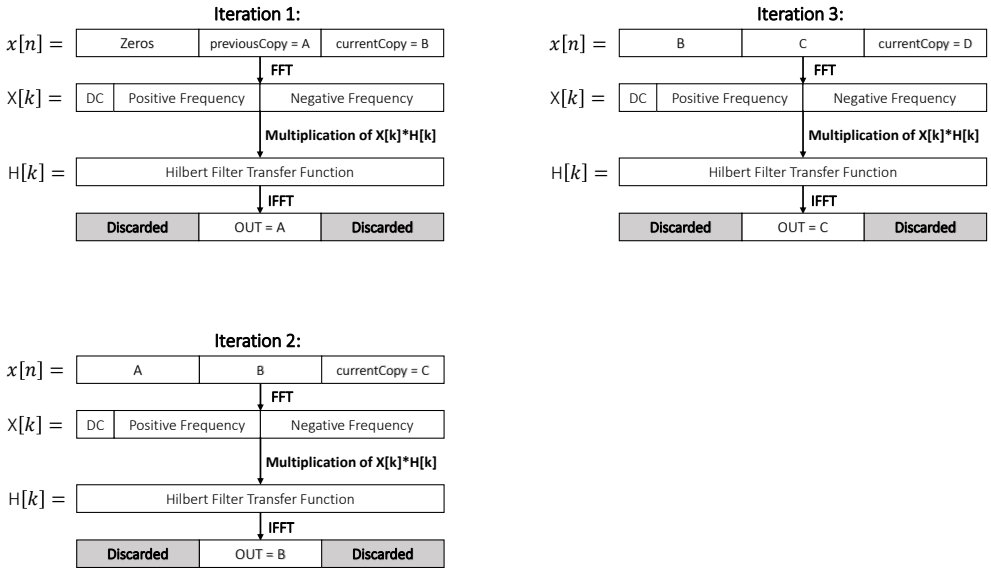


Figure 9.38: Logical flow of real-time Hilbert transform

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

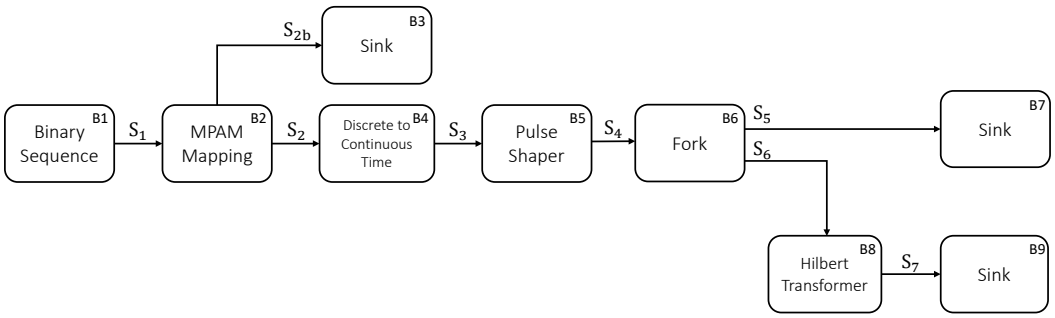


Figure 9.39: Test setup for the real time Hilbert transform

Real-time Hilbert transform : Results

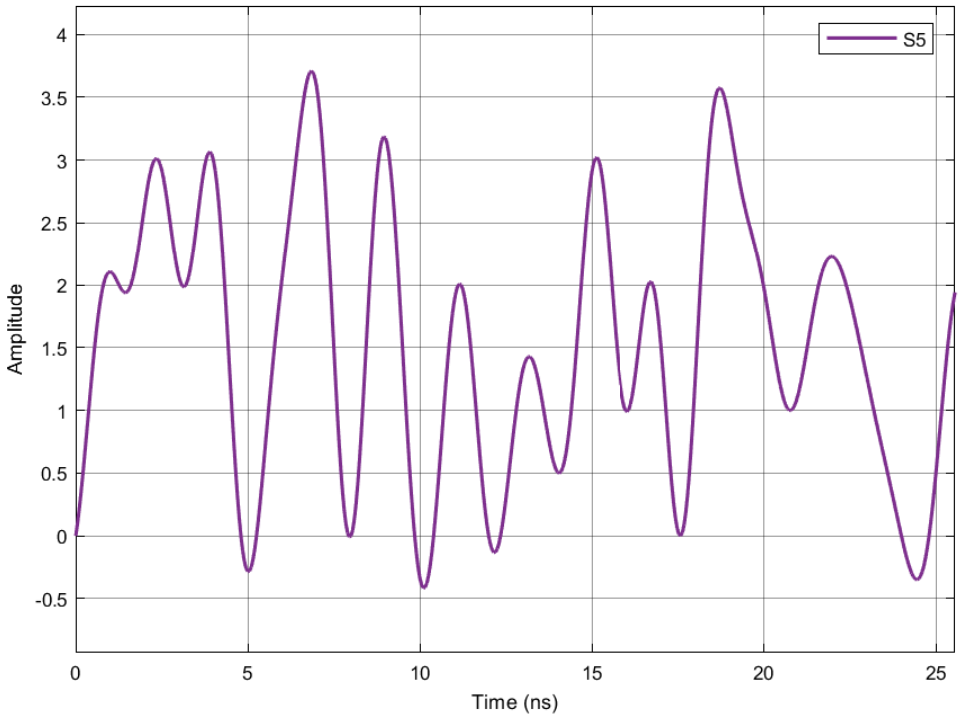


Figure 9.40: $S5$ signal

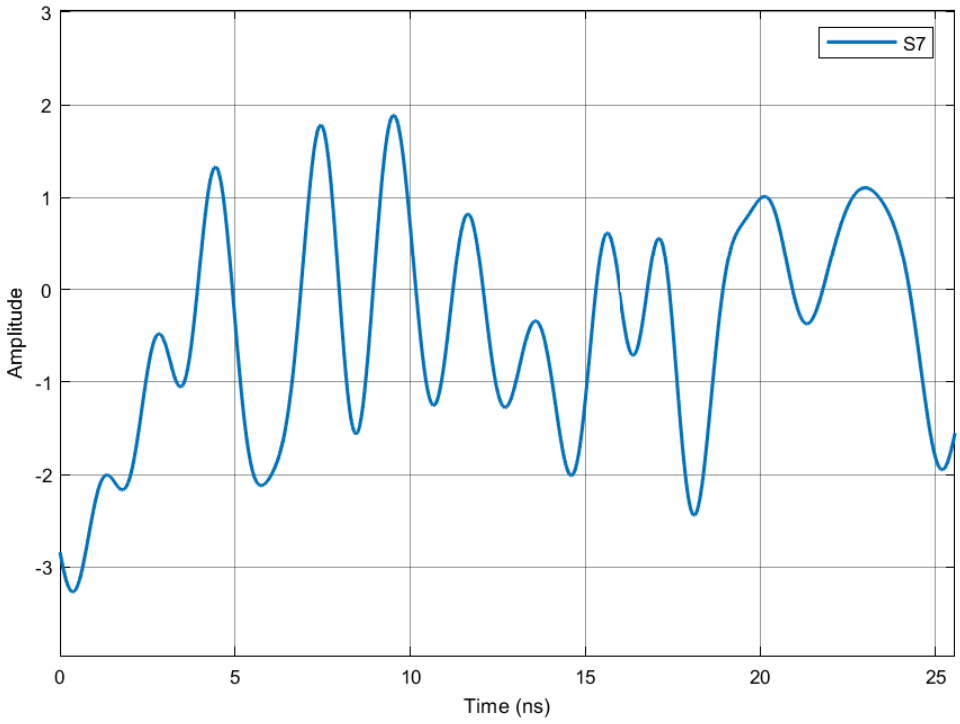


Figure 9.41: $S7$ signal

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.

References

[1] S.L. Marple. "Computing the discrete-time 'analytic' signal via FFT". In: *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers (Cat. No.97CB36136)*. Vol. 2. IEEE Comput. Soc, pp. 1322–1325. ISBN: 0-8186-8316-3. DOI: 10.1109/ACSSC.1997.679118. URL: http://ieeexplore.ieee.org/document/679118/.

[2] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, 1999, p. 870. ISBN: 0137549202. URL: https://dl.acm.org/citation.cfm?id=294797.

Chapter 10

Code Development Guidelines

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

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

git init

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

12.1 Data Model

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

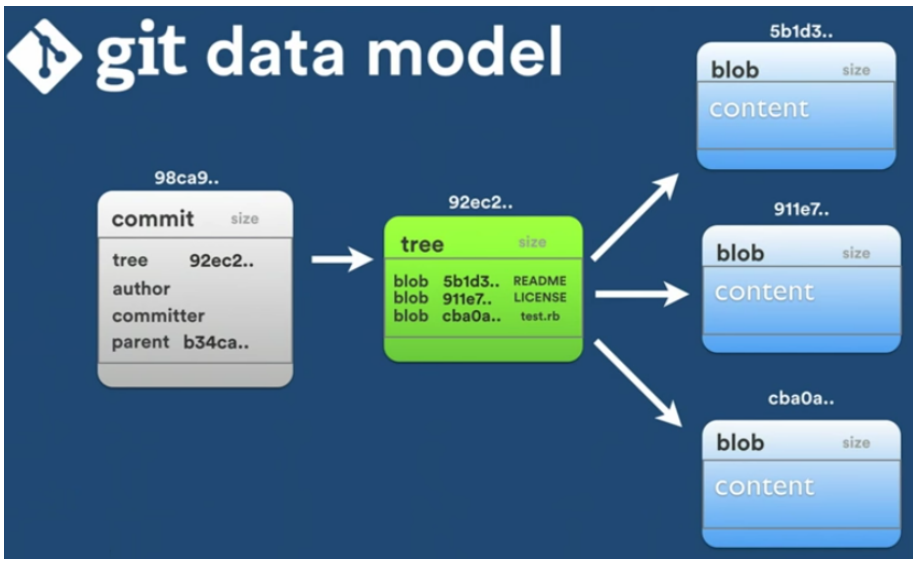


Figure 12.1: Git data model.

Git manipulates the following objects:

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

- blobs - the files that exist in your repository.

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

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

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

Here goes the commit message.

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

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

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

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

12.2 Refs

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

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

12.3 Tags

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

12.4 Branch

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

12.5 Heads

Heads is a pointer for the commit where you are.

12.6 Database Folders and Files

12.6.1 Objects Folder

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

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

12.6.2 Refs Folder

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

12.7 Git Spaces

Git uses several spaces.

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

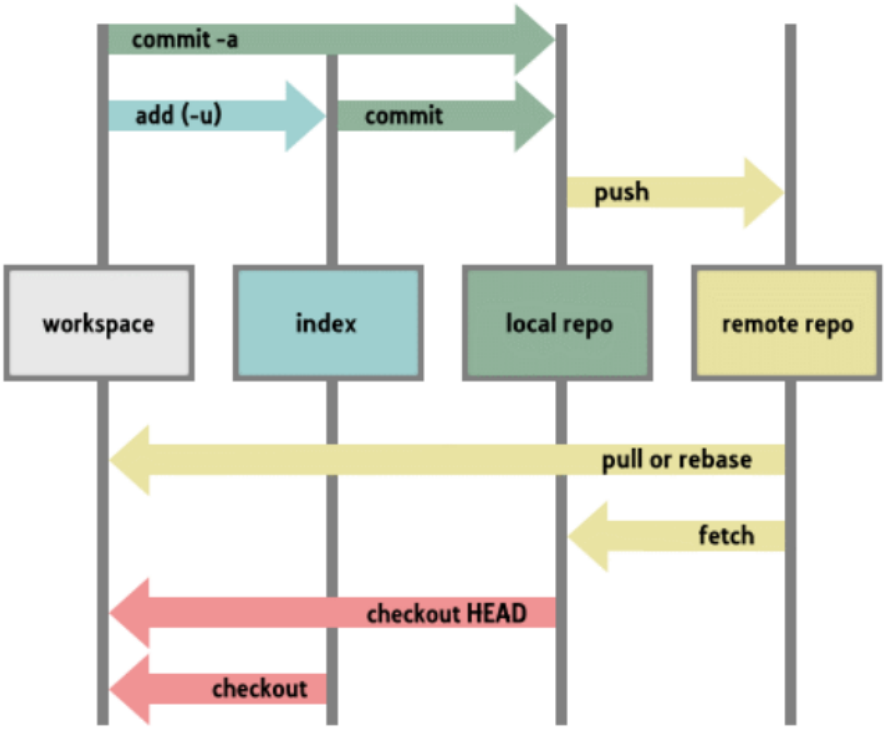


Figure 12.2: Git spaces.

12.8 Workspace

12.9 Index

12.10 Merge

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

12.10.1 Fast-Forward Merge

12.10.2 Resolve

12.10.3 Recursive

12.10.4 Octopus

12.10.5 Ours

12.10.6 Subtree

12.10.7 Custom

12.11 Commands

12.11.1 Porcelain Commands

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

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

git init

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

git log

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

options

–graph

shows a graphical representation of the repository commits history.

git rebase

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

git reset

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

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

git reflog

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

git show

Shows what is new in the last commit.

git stash

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

git stash, save the present state of your repository.

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

git status

12.11.2 Pluming Commands

git count-object

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

git gc

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

git gc –prune=all

git hash-object

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

git cat-files

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

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

git update-index

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

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

12.12 The Configuration Files

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

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

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

git config –global -e

12.13 Pack Files

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

12.14 Applications

12.14.1 Meld

12.14.2 GitKraken

12.15 Error Messages

12.15.1 Large files detected

Clean the repository with the BFG Repo-Cleaner.

Run the Java program:

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

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

do:

git push −−force.

Chapter 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\vhdl_simulation\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\vhdl_simulation\ghdl\bin**.
   Jump to step 10.
8. **If it exists**, click on the variable **Path** and press **Edit**. This should open another window;
9. Click on **New** to add another value to this variable. Enter your absolute path to the folder **\LinkPlanner\vhdl_simulation\ghdl\bin**.
   Example: **C:\repos\LinkPlanner\vhdl_simulation\ghdl\bin**.
10. Press **Ok** and you're done.

13.2 Using GHDL To Simulate VHDL Programs

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

13.2.1 Requirements

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

13.2.2 Option 1

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

13.2.3 Option 2

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

ghdl -a –std=08 signal_processing.vhd

ghdl -a –std=08 vhdl_simulation.vhd

ghdl -e –std=08 vhdl_simulation

ghdl -r –std=08 vhdl_simulation

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

13.2.4 Simulation Output

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