NetXPTO - LinkPlanner

July 18, 2019

Contents

1 Preface **7**

2 Introduction **8**

3 Simulator Structure **9**
3.1 System 9
3.2 Blocks 9
3.3 Signals 9
3.3.1 Circular Buffer 10
3.4 Log File 10
3.4.1 Introduction 10
3.4.2 Parameters 11
3.4.3 Output File 11
3.4.4 Testing Log File 12
Bibliography 13
3.5 Input Parameters System 13
3.5.1 Introduction 13
3.5.2 How To Include The IPS In Your System 14
3.6 Documentation 16

4 Development Cycle **17**

5 Visualizer **18**

6 Case Studies **19**
6.1 BB84 with Simplified Transmission 20
6.1.1 Protocol Overview 20
6.1.2 External usage 30
6.1.3 Input parameters 34
6.1.4 Results 35
6.1.5 Open Issues 36

Bibliography . . . 39
6.2 Discrete Variables Quantum Tx/Rx system - Physical Layer . . . 39
6.2.1 Simulation Analysis - Algorithm for polarization compensation . . . 39
6.2.2 Simulation Analysis - TestBench . . . 40
6.2.3 Open Issues . . . 41
Bibliography . . . 42

7 Library **42**
7.1 ADC . . . 43
7.2 Add . . . 46
7.3 Arithmetic Encoder . . . 47
7.4 Arithmetic Decoder . . . 49
7.5 Alice QKD . . . 51
7.6 Alice QKD . . . 53
7.7 Ascii Source . . . 56
7.8 Ascii To Binary . . . 58
7.9 Balanced Beam Splitter . . . 60
7.10 Basis Reconciliation . . . 61
7.11 Bit Error Rate . . . 63
Bibliography . . . 69
7.12 Binary Source . . . 69
7.13 Binary To Ascii . . . 73
7.14 Bob QKD . . . 75
7.15 Bob QKD . . . 76
7.16 BobQuantumRx . . . 78
7.17 Bit Decider . . . 81
7.18 Clock . . . 82
7.19 Clock_20171219 . . . 84
7.20 Complex To Real . . . 87
7.21 Coupler 2 by 2 . . . 89
7.22 Carrier Phase Compensation . . . 90
7.23 Decision Circuit . . . 94
7.24 Decoder . . . 96
7.25 Discrete To Continuous Time . . . 98
7.26 DownSampling . . . 100
7.27 DSP . . . 102
7.28 EDFA . . . 104
7.29 Electrical Signal Generator . . . 107
7.29.1 ContinuousWave . . . 107
7.30 Entropy Estimator . . . 109
7.31 Entropy Estimator . . . 110
7.32 Error correction . . . 112
7.33 Fork . . . 114

7.34 Gaussian Source . . . 115
7.35 Hamming Decoder . . . 117
7.36 Hamming Encoder . . . 118
7.37 MQAM Receiver . . . 119
7.38 Huffman Decoder . . . 123
7.39 Huffman Encoder . . . 124
7.40 Ideal Amplifier . . . 125
7.41 IQ Modulator . . . 127
Bibliography . . . 132
7.42 IIR Filter . . . 132
Bibliography . . . 133
7.43 Local Oscillator . . . 133
7.44 Local Oscillator . . . 135
7.45 Message Processors . . . 138
7.46 Mutual Information Estimator . . . 142
Bibliography . . . 145
7.47 MQAM Mapper . . . 145
7.48 M-QAM Receiver . . . 148
7.49 MQAM Transmitter . . . 155
7.50 Netxpto . . . 165
7.50.1 Version 20180118 . . . 165
7.50.2 Version 20180418 . . . 165
7.51 Alice QKD . . . 166
7.52 Polarizer . . . 168
7.53 Probability Estimator . . . 169
7.54 Bob QKD . . . 172
7.55 Eve QKD . . . 173
7.56 Rotator Linear Polarizer . . . 174
7.57 MS Windows IP Tunnel . . . 176
7.58 Mutual Information Estimator . . . 178
Bibliography . . . 181
7.59 Optical Switch . . . 181
7.60 Optical Hybrid . . . 182
7.61 Parameter Estimation . . . 184
7.62 Photodiode pair . . . 187
7.63 Photoelectron Generator . . . 190
Bibliography . . . 195
7.64 Power Spectral Density Estimator . . . 195
Bibliography . . . 201
7.65 Privacy amplification . . . 201
7.66 Pulse Shaper . . . 204
7.67 Quantizer . . . 206

7.68 Quantum Channel Emulator . . . 209
7.69 Resample . . . 211
7.70 SNR Estimator . . . 213
Bibliography . . . 218
7.71 Sampler . . . 218
7.72 SNR of the Photoelectron Generator . . . 220
Bibliography . . . 225
7.73 Sink . . . 225
7.74 SNR Estimator . . . 227
Bibliography . . . 232
7.75 Single Photon Receiver . . . 232
7.76 SOP Modulator . . . 236
Bibliography . . . 240
7.77 Source Code Efficiency . . . 240
7.78 White Noise . . . 241
7.79 Ideal Amplifier . . . 244
7.80 Phase Mismatch Compensation . . . 246
Bibliography . . . 248
7.81 Frequency Mismatch Compensation . . . 248
Bibliography . . . 251
7.82 Cloner . . . 251
Bibliography . . . 252
7.83 Error Vector Magnitude . . . 252
Bibliography . . . 254
7.84 Load Ascii . . . 254
7.85 Load Signal . . . 256
7.86 Matched Filter . . . 257
7.87 Timing Deskew . . . 258
7.88 DC component removal . . . 260
7.89 Orthonormalization . . . 262
Bibliography . . . 264
7.90 Matched Filter . . . 264
7.91 Upsampler . . . 265
7.92 Matched Filter . . . 267

8 Mathlab Tools 269
8.1 Generation of AWG Compatible Signals . . . 270
8.1.1 sgnToWfm.m . . . 270
8.1.2 sgnToWfm_20171121.m . . . 271
8.1.3 Loading a signal to the Tekatronix AWG70002A . . . 273
8.2 Polarization Analysis Signals . . . 277
8.2.1 jonesToStokes.m . . . 277
8.2.2 ACF.m . . . 278

8.2.3 plotPhotonStream_20180102.m . . . 279
8.2.4 plot_sphere.m . . . 280

9 Algorithms **281**
9.1 Fast Fourier Transform . . . 282
Bibliography . . . 298
9.2 Overlap-Save Method . . . 298
Bibliography . . . 327
9.3 Filter . . . 327
Bibliography . . . 336
9.4 Hilbert Transform . . . 336
Bibliography . . . 340

10 Code Development Guidelines **340**
10.0.1 Integrated Development Environment . . . 340
10.0.2 Compiler Switches . . . 340

11 Building C++ Projects Without Visual Studio **341**
11.1 Installing Microsoft Visual C++ Build Tools . . . 341
11.2 Adding Path To System Variables . . . 341
11.3 How To Use MSBuild To Build Your Projects . . . 342
11.4 Known Issues . . . 342
11.4.1 Missing ucrtbased.dll . . . 342

12 Git Helper **343**
12.1 Starting with Git . . . 343
12.2 Data Model . . . 343
12.2.1 Objects Folder . . . 345
12.3 Refs . . . 345
12.3.1 Refs Folder . . . 346
12.3.2 Branch . . . 346
12.3.3 Heads . . . 346
12.4 Git Logical Areas . . . 346
12.5 Merge . . . 346
12.5.1 Fast-Forward Merge . . . 346
12.5.2 Three-Way Merge . . . 347
12.6 Remotes . . . 347
12.6.1 GitHub . . . 347
12.7 Commands . . . 348
12.7.1 Porcelain Commands . . . 348
12.7.2 Pluming Commands . . . 352
12.8 Navigation Helpers . . . 353
12.9 Configuration Files . . . 353

12.10 Pack Files 353
12.11 Applications 354
12.11.1 Meld 354
12.11.2 GitKraken 354
12.12 Error Messages 354
12.12.1 Large files detected 354
12.13 Git with Overleaf 354
Bibliography 355

13 Simulating VHDL Programs with GHDL 355
13.1 Adding Path To System Variables 355
13.2 Using GHDL To Simulate VHDL Programs 356
13.2.1 Simulation Input 356
13.2.2 Executing Testbench 356
13.2.3 Simulation Output 356

Chapter 1

Preface

Th

Chapter 2

Introduction

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

Chapter 3

Simulator Structure

LinkPlanner is a signals open-source simulator.

The major entity is the system.

A system comprises a set of blocks.

The blocks interact with each other through signals.

3.1 System

3.2 Blocks

3.3 Signals

List of available signals:

- Signal

PhotonStreamXY

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

PhotonStreamXY_MP

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

3.3.1 Circular Buffer

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

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

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

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

3.4 Log File

3.4.1 Introduction

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

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

Figure 3.1: Disabling Log File

3.4.2 Parameters

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

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

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

3.4.3 Output File

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

Figure 3.2: Output File Example

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

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

Figure 3.3: I-Homodyne Receiver Block Declaration

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

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

Figure 3.4: Four input parameters where loaded from a file

3.4.4 Testing Log File

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

3.5 Input Parameters System

3.5.1 Introduction

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

Format of the Input File

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

1. Input parameter values can be changed by adding a line in the following format: **paramName=newValue**, where **paramName** is the name of the input parameter and **newValue** is the value to be assigned.

2. IPS supports scientific notation. This notation works for the lower case character **e** and the upper case character **E**.

3. If an input parameters is assigned the wrong type of value, method **readSystemInputParameters()** will throw an exception.

4. Not all input parameters need to be changed.

5. The IPS supports comments in the form of the characters //. The comments will only be recognized if placed at the beginning of a line.

```
input_parameters_0.txt
//----------BPSK PARAMETERS--------------
//Changes the value of parameter pLength
pLength=5
//Is able to change the value of different types of parameters
//Of type int
numberOfBitsGenerated=1e3
//Of type double
bitPeriod=20e-12
//Of type bool
shotNoise=false
```

Figure 3.5: Content of file input_parameters_0.txt

Loading Input Parameters From A File

Execute the following command in the Command Line:

some_system.exe <input_file_path> <output_directory>

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

3.5.2 How To Include The IPS In Your System

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

1. Your system must include **netxpto_20180418.h** or later.

2. A class that will contain the system input parameters must be created. This class must be a derived class of **SystemInputParameters**. In this case the created class is called **BPSKInputParameters**.

3. The created class must have 2 constructors. The implementation of these constructors is the same as **BPSKInputParameters**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   BPSKInputParameters param(argc, argv);

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

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

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

   //System Run
  MainSystem.run();

  return 0;
}
```

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

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

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

3.6 Documentation

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

Chapter 4

Development Cycle

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

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

Chapter 5

Visualizer

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

Chapter 6

Case Studies

6.1 BB84 with Simplified Transmission

Students Name	:	Andoni Santos (10/04/2019 - ...)
Starting Date	:	April 10, 2019
Goal	:	BB84 implementation with discrete variables and simplified transmission.

BB84 is a quantum key distribution protocol which involves three parties: Alice, Bob and Eve [**Bennet84**]. The objective of the protocol is for Alice and Bob to generate private key that only they know, while assuming a third party, Eve, is trying to eavesdrop on their communications. For this, they use both a quantum channel and a classical channelThe protocol allows them to share information between to create the shared keys, while minimizing the amount of information leaked to Eve.

The BB84 with simplified transmission simulation implements the top layers of the BB84 protocol, while emulating the physical aspects of the communication system in a simplified fashion. Its purpose is to allow implementing, testing, analyzing and improving the software layers of the protocol. The emulation of the quantum physical transmission channel is achieved by using a probabilistic model, where each possible transmission outcome has a predefined probability of happening.

6.1.1 Protocol Overview

The structure of the implemented system is shown in Figure 6.1. Alice and Bob are connected through two different channels, and use them to communicate with each other: a public, authenticated channel represented by the *Messages* signals, and a private, unidirectional quantum channel, represented through the *quantum_channel_emulator* block. The general flow of the system in the simulation is as follows:

The information flowing between Alice and Bob is transmitted through the *Messages* signals and the *Quantum Channel Emulator* block. Data transmitted through the messages signals is publicly available, with its integrity ensured. On the other hand, the information transmitted through the quantum channel is not public, although an eavesdropper could still try to access it. In addition, the integrity of the information in this channel is not guaranteed, as we shall see further ahead.

Two random bitstreams, *AliceData_In* and *AliceBasis_In* are generated and passed to Alice, to serve as the sources for the transmitted key and the basis used to encode the key. Alice sends a copy of those signals to the *quantumChannelEmulator* block, where they will be used to transmit the key to Bob. In addition, Alice saves the information she sent (key and basis) to prepare to do basis reconciliation with Bob. Alice also sends *AliceTransmissionMode_Out* to the quantum channel. This signal indicates if the system is currently transmitting data or doing the control process. Another random independent bitstream is generated and passed to Bob, to provide the measurement basis. Bob sends these to the *quantumChannelEmulator* block and also saves the information to prepare for

Figure 6.1: Implemented QKD Model. Alice and Bob are connected by a public classical channel, represented by the message signals, and a private quantum channel.

basis reconciliation. The *quantumChannelEmulator* block now reads all the input and it processes the information before sending it to Bob. We assume a sort of black box model, where we can define the channel's behavior based on certain key parameters. This block has several probabilities defined during startup which affect its behavior. Depending on a set of generated random numbers, it sends the following data to Bob:

- If the first number is lower than *probabilityOfBothClick*, the channel sends a predefined *doubleClickNumber*, which signals that both the detectors in the physical system clicked;
- Else, if the second number is lower than *probabilityOfNoClick*, the channel sends a predefined *noClickNumber*, which indicates that none of the detectors in the physical system clicked;
- Else, if the third number is lower than *probabilityOfError*, the channel send the wrong bit;
- If none of these happen, the channel compares Alice and Bob's basis. If they are different, the channel checks a fourth random number, and it has a 50% chance of sending the right bit, and 50% chance of sending the wrong one.
- If Alice and Bob's basis are equal, the channel sends the correct bit.

The *quantumChannelEmulator* also sends a *BobTransmissionMode_In* to Bob, which he will then to Alice via Messages.

Bob then receives the data from the quantum channel and starts the basis reconciliation process. Alice and Bob then do basis reconciliation, error correction and privacy amplification, in this order, communicating through the *Message* signals. This process will explored in more detail further ahead. After doing basis reconciliation, both Alice and Bob send the sifted key to a *ParameterEstimation* block. The sifted key is the key obtained after eliminating every bit where Bob used the wrong basis, and where the measurement either had no click or a double click.

The *ParameterEstimation* block establishes an estimate on the error rate of the transmitted key by sharing (and consequently discarding) some bits between Bob and Alice. We use this block to get an upper bound on the error rate, which is then used to decide if a secret key can be obtained. If the error rate is low enough, the key and the error rate are sent to the *BitErrorRate* block, where the error rate and the total number of bits is measured.If everything is working properly, the BER should be similar to the *probabilityOfError* defined in the quantum channel.

After basis reconciliation, Bob initiates the error correction process. This process implements the Cascade algorithm [**brassard93**, **martinez14**], where Alice and Bob exchange the parities of subsets from the current bit string in order to identify and correct errors. After the correction, one bit is discarded for each exchanged parity, in order to avoid giving information to Eve.

In addition, after privacy amplification, Alice and Bob send the final keys to another *BitErrorRate* block. Here, the BER should be either zero or at least much smaller than the value measured with the sifted key, and the size will also be significantly reduced.

In order to understand in detail what is happening, we need to take a look into the *AliceQKD* and *BobQKD* superblocks.

We can see that they have dedicated blocks to receiving and transmitting messages, and also dedicated blocks for basis reconciliation, error correction and privacy amplification. Each signal that goes from one of this blocks to one of the message processors carries a certain type of information. The message processors identify the type of information from the signal that they arrive in, and send the data in a message with the appropriate data type. Only one message of each type can be sent in a single run of the block. This way, all three processes (basis reconciliation, error correction and privacy amplification) can be happening simultaneously for different sets of data.

Now that we have explained the system for Alice and Bob to communicate using the classical channel, we can delve into the details of basis reconciliation, error correction and privacy amplification.

Basis Reconciliation

During basis reconciliation, Bob sends Alice the basis he used to measure the received bits. Alice then compares with the basis she used to transmit the information. If the basis used is the same, Alice sends a "1" to Bob and sends the current bit to the stage of Error Correction; otherwise, Alice sends a "0" to Bob and discards the current bit. In addition, for the cases where the measurement returned a double click or no clicks, Bob sends Alice the signal

Figure 6.2: AliceQKD superblock.

corresponding to that event instead of the basis, so that Alice can readily discard these bits. In these cases, she also sends a zero as a confirmation that she discarded them. Bob can then analyze the response from Alice and discard the bits for which he didn't receive a "1", while sending the others to the stage of error correction.

It's worth noting that this stage, unlike the ones that follow it, does not benefit from processing a higher number of bits at once. This reduces the number of messages Alice and Bob need to communicate, but has no other effect on the sifted key.

Parameter estimation

Even after removing the bits that were measured with the wrong basis, there will still be errors in Bob's key. These may be due either to transmission errors or to interference by an

Figure 6.3: BobQKD superblock.

eavesdropper. However, at this point the origin does not matter, and it is safer to assume that all errors are due to Eve.

An estimate of the QBER is required for two purposes: to choose appropriate parameters for error correction, maximizing error correction efficiency for the estimated error rate; and estimating the information that Eve has obtained, which is related to the error rate.

The process for obtaining the error rate is straightforward: Alice and Bob publicly share a given number of bits, and count how many of them are different. After they do this, they discard all the shared bits, so that Eve does not gain any knowledge about the key.

There is an obvious trade-off that needs to be taken into account: increasing the number of shared bits directly decreases the amount of bits; however, by using a higher amount of bits, a better estimate of the error rate can be achieved, which is translated into better privacy (as the amount of errors is indicative of Eve's activities) and possibly into a better error correction efficiency, particularly if we use the upper bound of this estimate to chose the partition sizes. Therefore, increasing the number of bits wasted at this stage may diminish the number of wasted bits on the long run. We should choose the number of bits to be shared taking into account the total number of bits they are extracted from, how often we will do this estimate and even the total size of the key that is processed during error correction.

Error Correction

Alice and Bob need to make sure they have the same key, and they obviously cannot compare them directly. In these circumstances error correcting codes might not be practical to use, so other methods need to be used.

Cascade is one of the most used algorithms for correcting the errors in these cases. The current implementation will now be described in detail:

After parameter estimation, Alice and Bob's error correction blocks store the sifted keys into a vector, while they wait for the number of available bits to be enough to ensure proper performance. There are three reasons to do this, instead of working normally with the values contained in the input buffer:

1. **It keeps the data flowing.** One of the problems with using Cascade for error correction is the amount of classical messages transmitted between Bob and Alice. If the error correction block only emptied its input buffer when the process ended, it is likely that some of the other blocks would have to pause due to filled buffers, as they wait for Cascade to stop;

2. **Reducing communication between Alice and Bob.** As previously mentioned, this method requires Alice a Bob to communicate in order to find the errors. For a single run of the algorithm, the same average amount of channel uses is required whether the key to correct contains with 500 or 2000 bits. However, if the key being corrected contains 500 bits, it would take four runs of the algorithm to correct a key with 2000 bits, which implies four times more channel uses.

3. **Effectiveness of error correction.** This error correction process greatly reduces the probability of an error between Alice and Bob's bit strings. It is not guaranteed to reduce this probability to zero, but usually it becomes a value low enough so that there will not be an error in a bit string of a given size. The original proposed algorithm used strings of 10000 bits and claimed that these would be kept error free within the given parameters [**brassard93**]. It has been shown that increasing the length of the bit strings has a positive impact on the probability of failing to remove all errors [**martinez14**]. The achieved error rate becomes lower when more bits are processed at once. For instance, in our simulations, using the same set of parameters, the measured error rate after the cascade dropped from 1.0×10^{-5} when using sets of 512 bits to 6.2×10^{-7} when using sets of 2048 bits. A possibly more relevant metric than the bit error rate is the failure error rate [**martinez14**], which describes the probability of at least one error not being corrected. As we intend to have completely equal keys in both sides, this is indeed a better statistic to study the probability of not achieving the key. It also presents a different aspect, as the increased key length improves the error correction but makes the probability of at least one error more likely. Nevertheless, this value is a directly related to the bit error rate, so the BER is still an accurate measurement of the error correction effectiveness.

For this reasons, the values are stored so that the input buffer has space available for incoming sifted keys, and once enough values have been saved, the cascade protocol starts. The minimum number of values is defined in a variable of the *ErrorCorrection* block, *minimumSyndromeSize*. In addition, a maximum possible value is established, defined by *maximumSyndromeSize*. This is necessary due to limits in the buffer sizes. Currently, the message processors transform the entire contents of a buffer into a single message. When that message arrives, it is also completely read into the output buffer of the message processor. Alice and Bob consider that when they read these buffers, all the necessary content is there, without any extra information. Therefore, in the current implementation, a set of data (parities, permutations, etc.) cannot be split in several messages, or read in segments. This means that the maximum data transmitted by a message may never be longer than a buffer. Therefore, *maximumSyndromeSize* defines the maximum length of bit string processed at once. This way, the transmitted data is never longer than that value.

Remember that normally the error rate would have to be calculated, and the ideal partitioning sizes derived from it. However, the blocks accept instead a partition size as an input variable, and this input variable is calculated from the value defined for the BER. Also, the vector that Alice and Bob use to store the incoming data is used as a FIFO, where incoming data is pushed to its end, and retrieved from the start.

Once the process starts, it works as follows. For the extent of this explanation, assume that any communication between Alice and Bob is done by Messages:

1. Bob retrieves a given amount bits from the stored data, where the number is largest multiple of the *partitionSize* which does not exceed the amount of available data, or *maximumSyndromeSize*. Having that string, Bob will partition it according to *partitionSize* and calculate the parity of each partition. He then sends these parities to Alice.

2. Alice retrieves the parities sent by Bob. As she also knows the *partitionSize*, she can also retrieve the same amount bits from the data she stored. She then partitions it, calculates the parities of her partitions and compares her results to what Bob sent. For each of the parities, if they are equal, she returns a "2", to indicate that the parities are equal and no more comparison is required. Otherwise, she returns a "0". She also stores her previous parities for the next step.

3. When Bob receives the response, he processes it in the following fashion: if the returned value for a given partition is "2", he leaves it alone and also sends back a "2". Otherwise, Bob "updates" the partition he uses to calculate the parities, calculates the new parity and sends the parity to Alice. By "updating", we mean that Bob changes the number of bits he is currently considering. We shall explain this in detail further ahead.

4. When Alice receives the new response from Bob she does two things: first, she looks over her previous list of parities to also "update" her partitions, so that they match Bob's; secondly, she calculates her parities and sends her response to Bob: the "2" values remain intact, she sends "1" if the parities are equal and "0" if they are different;

5. Bob receives the response, "updates" his partitions accordingly and sends his new parities. This process is repeated until an updated partition has only one bit. When this happens, that bit is flipped and Bob sends a "5" instead of a parity (for now the "5" works as sending a "2", but the difference will be explained further ahead). When all values Bob sends are either "5" or "2", he has finished this iteration of the cascade, and can proceed to the next.

6. Alice knows the process is over when all parities she receives are either "5" or "2". She then proceeds to the next iteration.

Before proceeding we need to clear up two issues.

First, to keep Eve from gaining additional information, we need to discard a bit for every parity check. However, the Cascade process relies on not discarding any bits until the process is complete. Therefore, to achieve both these objectives, the following is done: each time a parity check is performed, one of the bits is marked. If the bit is "1", it is changed to a "7"; if it is "0", it is changed to an "8". As we are treating "0" and "1" as simple integers anyway, it does not have an impact on performance. This way, when the process finally concludes and the key is written to the output buffer, all values which are not "0" or "1" are ignored, and in the meantime, they are still even or odd.

Secondly, we shall now explain the partition "updating" process. In order to find the errors, we start with partitions of a given size, and if the parities don't match, we have to compare shorter partitions, which are continuous subsets of the original partition. These shorter partitions are calculated based on the responses from Alice. The process of calculating these shorter subsets based on Alice's responses is what we referred to as "updating" the partitions. The process is as follows: if the partition has its original size, it is split in half, and the first half becomes the new partition; if the partition has not the original size, and Alice response was "0", the new partition is the first half of the current one; if the partition has not the original size, and Alice response was "1", the new partition is the first half of the second half of the previous one. We can see a representation of these three cases in Figure 6.4.

Now we can continue explaining the other iterations. All iterations after the first are similar, only the first one is different.

1. Bob takes the key obtained at the end of the previous iteration and creates a scrambled copy of it using random permutations. He then sends the permutations to Alice, so she can also create a scrambled copy of her key. This is required, because at the end of the previous iteration, all partitions either contained no errors, or and even number of errors. Scrambling the key allows to spread the errors, so that the likelihood of a partition containing an even number of errors is smaller. The scrambling process and the corresponding rationale is demonstrated in Figure 6.5. This scrambled key is then treated as the initial data: partitioned, parities calculated, etc. It works mostly like the first iteration. One relevant aspect is that the size of the partition does not necessarily need to remain the same. As the number of errors diminishes, the partition size can be increased. The original article recommends doubling the partition size with each

Figure 6.4: Process of updating partitions, demonstrated with a 10 bit partition. In order to find the error, the partition is sequentially divided in smaller subsets, with the particular choice of subset depending on whether Alice and Bob's parities agree. Bob's key is on the left, Alice's is one the right and the error in Bob's key is shown in red. The green squares represent the bits chosen to calculate the parity to compare, and the orange line delineates all the possible error positions. In the last line, no parity is required, as there is only a single bit left, which must be the error. That bit is then flipped. In addition, each time a parity is compared, one of the bits in the green squares is marked to be discarded. In this case, 5 in 10 bits would be discarded from this bit string.

subsequent iteration. That is an option on the simulation, however it currently does not perform as we would hope. Therefore, for now, the partition size remains constant, which limits the efficiency of key generation, but ensures a very low error rate.

2. Alice also creates a scrambled copy her key, so it has the same one as Bob. Again the process remains mostly the same.

3. When an error is finally found, Bob sends a "5" in the parities. He corrects the error in the original version of the key. In addition, he also stores the position of the corrected bit in the original key (the one obtained at the end of the first pass). He also activates a Boolean flag to indicate that the lookback process is to be done when this iteration ends;

4. When Alice receives a "5", she also stores the original position of that bit, corrects the original version of the key and activates a similar flag.

5. After all errors have been corrected, Alice and Bob start the lookback process. The rationale for this is that for each error corrected in this pass, in the original key that error was paired with at least one other error. As we now corrected some errors, their old pairs should now be easily found, if they haven't already been corrected. Therefore, we look into the smallest previous partition which contained the bit we corrected, and

look for the error with the process described above (they are all the same size, but the first one would be the smallest otherwise).

6. After the lookback process is completed, we can proceed to the next iteration. The error correction finishes after a predetermined number of iterations.

Figure 6.5: Process of creating a scrambled copy of the key. Demonstrated here using only two partitions of 10 bits each. Initially, no error is detected. Alice and Bob have the same parity for both partitions. However, after generating a random list of indexes and permutating the bits according to it, the errors become spread out between both partitions. Using only two partitions, the likelihood of the errors staying in the same partition is still high. Processing longer keys improves this aspect.

Privacy Amplification

The purpose of privacy amplification is to reduce as much as possible the information that Eve has on the transmitted key, while keeping it as long as possible.

There are two sources where Eve can get information about the transmitted key: she can directly intercept the communications in the quantum channel; and she also learns information during the error correction stage, during which Alice and Bob share the parities of bit strings in order to find and correct errors.

The error correction process already discards a bit for each parity check that Alice and Bob conduct, so that Eve does not learn any new information. However, Eve likely still retains some information about the key, at least from intercepting qubits in the quantum channel.

For instance, let us assume that Bob knows that the QBER is equal to a certain percentage z, and that every error in the received key is due to a qubit which Alice wrongly measured and subsequently re-sent to Bob.

From Eve perspective, she will measure with the correct basis half of the time. When she uses the wrong basis, she has a 50% probability of introducing an error. Therefore, the

total number of bits she might know is given by the number of times she measured correctly, which is twice the number of times she introduced an error, $2z$.

In order to reduce her information, we can transform the key so that the information Eve retains from it is almost certainly none (or, at most, a single bit). Again, this necessarily implies a reduction of the key size. If we assume that Eve probably knows K bits, and our key is N bits long, the final key would be $R = N - K - S$. Here, S is a security parameter that can be chosen as a function of the desired security.

We can use a hash function to transform the key that Alice and Bob have generated into a key of size R. To do this, the hash function should be randomly chosen from a class of strongly universal hash functions, by randomly selecting a key and having Alice a Bob use it. The amount of information that Eve is expected to have on the final key in this case is less than $2^{-S}/\ln(2)$ bits. If the hash function is instead chosen from an almost strongly universal class this still works, and the expected knowledge is no higher than $1 + \log(1 + 2^{-(S+1)})$ **[bennett88]**.

Another option is to generate a new key by checking the parities of random subsets from the initial key. This reduces the knowledge that Eve has on the key, as she cannot know the parity of a subset containing bits that she does not know. The key still needs to be reduced by the same amount as in the previous case.

6.1.2 External usage

The work described in this case study was done with the intention of developing and testing the *AliceQKD* and *BobQKD* blocks. Their function and signal interface was designed so that it would be compatible with other simulations and with the experimental setup currently in development. We shall now detail these blocks, their signal interfaces and what their expected inputs, outputs and functions are expected to be.

General overview

If we look at Figure 6.1, we can immediately take notice that some components were included with the intention of being easily replaceable.

First we have the binary sources. These are easily replaceable with any other source of binary data, such as a quantum random number generator.

Secondly, we have the quantum channel emulator, which simply tries to emulate the expected behavior of a quantum channel. Currently, it simply directly passes the data, introducing errors according to the defined error probabilities. We can replace it by another channel that follows its guidelines. Thus, it would allow connecting the *AliceQKD* and *BobQKD* blocks, either for using a more complex simulation, testing different configuration or slightly different transmission protocol, or even for implementing the Alice and Bob software parts to use on an experimental setup.

Messages

A classical channel between Alice and Bob is required for most of the software processing, namely basis reconciliation, error correction, and even for sharing seeds to use with hash functions during privacy amplification. This is done using the two message signals which directly connect *AliceQKD* to *BobQKD*. In a simulation setting, they should always remain connected to each other. In a future implementation where Alice and Bob run in different computers, these are should be the only signals which need to communicate directly in a classical channel between both computers.

The message signals implementation is a bit more liberal than the other signals, but all messages share some common properties. In all of them the data is transmitted in strings. They are identified by their type. In the blocks responsible for sending and receiving messages, *MessageProcessorReceiver* and *MessageProcessorTransmitter*, each message type is associated with a correspondent input or output signal, respectively.

Each message usually contains a complete set of data, corresponding to a single iteration step of the corresponding type. It is assumed that the blocks won't send a message if the previous one hasn't been received. This is ensured by the way the data flows between the different processing layers.

- During basis reconciliation, Alice never sends a message by herself, she only responds to Bob's messages. In addition, Bob sends a message to Alice, and does not send a second one until he has received an answer for the first.
- The process for parameter estimation is essentially the same, only one element starts it and it does not proceed until an answer is received.
- During error correction, the process is a little more complex, but the principle remains. Bob starts the process, and from there he requires Alice's answer to proceed until the current processing stage is done. Then he starts it again, following the same *modus operandi*.
- During privacy amplification or changing the cascade blocks size, the requirement to only have one message is not as important, as the amount of data sent is much smaller, and accumulation of a small number of messages has little impact. However, the ratio of the number of bits processed at different stages should ensure that there are never enough accumulated messages to fill the buffers (which typically won't happen easily).

The way it currently works is that when data is sent to a *MessageProcessorTransmitter* or from a *MessageProcessorReceiver*, the first value contains the total number of samples to retrieve (excluding this value from the count). This way, the transmission of data is not constrained by the buffer sizes, and a full set of data can be sent over as many iterations as necessary. Of course, reducing the number of iterations required to transmit the data to and from the message processors greatly improves performance.

Therefore, as it stands, they need to be delivered in one piece. Otherwise, the message processors need to be adjusted to split the data in multiple messages. Nevertheless, if this is

done, all the messages containing the same set of data need to be read in sequence, without exiting the *MessageProcessorReceiver* block. For instance, if Alice's *ErrorCorrection* block starts before having received all the parities from Bob, it will still try to process them, even though some are missing. This will inevitably lead to errors.

Currently there is no enforced size limit on a single message. While the message buffers have the default space for 512 values, there is no specified maximum size for a message. This means that each message can, in principle, have an arbitrary size. In practice, they are currently limited by the size of the buffers connecting to the message processors. The largest messages currently sent are the permutation messages, which contain the list of indexes used to scramble the key. For instance, for key with 512 bits, the transmitted message would be $10 \times 1 + 90 \times 2 + 412 \times 3 + 512 = 2338$ characters long. A possible alternative would for this particular case would be to calculate permutations based on a pseudorandom number generator, using the same generator in both Alice and Bob, and sending the seed instead of the index array. This should be trivial to implement. Its just a matter of altering the function that generates the permutations, sending the seed instead of the permutations, and make Alice calculate the permutations instead of sending them.

The second highest message length is needed by the parity checks. All other exchanges don't really depend on message size. For a particular message containing the parity checks, the required message length s would be $s = K/p$, where K is the total length of the key being corrected, and p is the size of each partition. In this case, assuming an error rate of 8% (so that the partition size is very small, containing only 9 bits) and a key of 4068 bits, the required message string size would be only 512 characters.

One could think that reducing the key size in order to process less bits at a time, therefore reducing message size, would be practical. However, this would both reduce the error correction effectiveness and increase the total number of channel uses, delaying the error correction process. Both of these effects are highly undesirable. In fact, in an ideal scenario, the key would be processed all at once, after the transmission ended. This would maximize the effectiveness of the error correction, and the number of classical channel uses would be minimal (the partition size would be the same, and the number of classical channel uses done by the Cascade algorithm is independent of key length, assuming message size is arbitrary).

Quantum channel emulator

The quantum channel emulator is currently a black box which communicates the bits from Alice to Bob, randomly introducing errors, double clicks or no clicks. These are the signals expected from a BB84 implementation, and as long as the the interface is correct, it should be possible to easily replace the *QuantumChannelEmulator* block with a different, more complex channel model, or even a real quantum channel.

Lets start on Alice's side. As mentioned before, *AliceQKD* block expects input from two binary sources. These can be replaced by other sources, but they should contain only the bits to be transmitted or the bits indicating which base to to use. Any synchronization bits, or any other kind of deterministic data, is not expected, and will be treated as if they were either data or basis. That would be a problem, as the randomness of the data is crucial to

security. The other input are the messages from Bob, but those were already described.

In addition to the messages to Bob, Alice outputs three different signals. The first and second are binary signals, copies of the two binary input signals, intended to be used as data and basis by the quantum channel. Again, they contain only bits to be used as data and basis, they don't contain any headers or other type of additional information.

Lastly, Alice outputs a third binary signal, although it's functionality is not currently implemented. *AliceTransmissionMode_Out* is a signal Alice generates, based on messages received from Bob, to indicate whether it is currently transmitting data or working on control processes. Further details about this signal will be added when it is implemented.

These are all the signals relevant to the quantum channel on Alice's side. Let us now observe Bob's side.

Bob also has the messages signals to communicate with Alice in the classical channel, as previously discussed. He also makes use of a binary source to generate the bases used to measure the incoming signal in the quantum channel. Again, it could easily replaced by any other binary source. A direct copy of this signal is then output into the *QuantumChannelEmulator*.

Other than that, Bob has two additional inputs . One is the *BobData_In* signal. Unlike all the other signals discussed here, this is not binary. Although the data transmitted in the quantum channel should be binary, the channel also outputs certain numbers when it was unable to make a proper measurement, either due no detector clicking, or both the detectors clicking at the same time. Therefore, this signal is *TimeContinuousAmplitudeContinuousReal*. The numbers used to identify the *doubleClick* or *noClick* events can be defined in the the blocks, both in *BobQKD* and in the *QuantumChannelEmulator*. By default, these numbers are set to 2 and 3, respectively.

The last signal is *BobTransmissionMode_In*. As with Alice, the functionality of this signal is currently not really implemented.

6.1.3 Input parameters

Table 6.1: Main system files

Variable	Type	Default Value	Description
numberOfBits	int	1000000	Total number of bits created on the bit sources
~~*probabilityOfTransmissionError*~~	t_real	0.03	Transmission error rate. Also used in error correction and privacy amplification
probabilityOfDoubleClick	t_real	0.1	Rate of double clicks
probabilityOfNoClick	t_real	0.1	Rate of events without clicks
doubleClickNumber	t_integer	2	Number to signal the double click events
noClickNumber	t_integer	3	Number to signal the events without clicks
numPasses	t_integer	4	Number of iterations in the error correction process
qValue	t_real	0.73	Parameter for choosing the initial size of error correction segments. Size is equal to *qValue* divided by *probabilityOfTransmissionError*.
minimumNumberOfPartitions	t_integer	10	Minimum number of segments to use for error correction
maximumSyndromeSize	t_integer	8192	Maximum number of samples to be processed at once during error correction
minimumSyndromeSize	t_integer	8192	Minimum number of samples required to start error correction.
bypassHash	bool	false	Control whether to hash the key or not, to better analyze error correction results.
bitRate	t_real	1000	Bits per second.
systematicSecurityParameter	t_integer	1	Number of extra bits to discard after hashing. Discarded after each round.

6.1.4 Results

Figure 6.6: Residual BER after error correction as a function of the processed key size, for several QBER values.

Figure 6.7: Cascade efficiency: ratio between number of bits at the beginning of cascade and number of bits after discarding one bit for each new bit revealed during parity check. These results assume that the initial block sizes were chosen based on the upper bound for the BER estimated during parameter estimation. 1000 bits were used for each BER estimate.

6.1.5 Open Issues

1. Currently working with a fixed value of BER defined at startup on the main sdf file. Should be dynamically estimated and some parameters should be changed based on the obtained value (*partitionSize*). In addition, the error estimation would likely require discarding additional bits.

   - ***2019-05-28 Update:*** Implementing parameter estimation block. It should provide a top bound for the BER to the error correction and privacy amplification blocks.
   - ***2019-05-28 Update:*** Sort of implemented, but it raises critical issues: it either puts error correction in an infinite loop or, for some reason, error correction continuously outputs values, more than the output of basis reconciliation.

2. Cascade is not working as well as it should. It is able to correct errors with a reasonably high efficiency and achieving low BERs. However, in the literature, the partition sizes should double for each pass after the first. Doing this should reduce the number of shared parity bits while still reducing the number of errors by at least a factor of 2. However, doing this currently does not work, and the BER does not go down from a relatively high value (1×10^{-3}).

   - ***2019-05-21 Update:*** Using default parameters, with a key size of 8192 for error correction, the BER stays around 10^{-4}. However, by making a simple change and using parameters different from the default ones, it is possible to go around this. The implemented change is to establish a minimum number of partitions. By making sure that, even if the partition size grows, the number of partitions is always at least 10 (10 bits discarded per pass, if there no error is found), we can achieve much better results. With this, changing the number of passes to 7 is enough to go below 10^{-6}, while not affecting efficiency significantly.
   - ***2019-05-22 Update:*** Comparing with [**martinez14**], we cannot achieve the same key performance as them. Frame error rate usually does not fall below $10^{-}1$, and even bit error rate does not behave as they claim. The BER for original cascade with a frame of 1000 bits should output a BER of $10^{-}4$. However, it seems that this can achieved for frame size of $10^{-}4$. There are other inconsistencies. Something is wrong. Going to look for the error now.
   - ***2019-05-22 Update:*** The lookback process was not always running when it should. However, the effect of this was not particularly significant, as it only caused an issue in the last iteration for each run. Performance is mostly the same.
   - ***2019-05-23 Update:*** It appears that not only is Cascade failing to perform as well as it should, it sometimes also introduces new errors (Typically, new errors appear much less frequently than the rate of the uncorrected old ones.). Currently trying to find the source of this issue.
   - ***2019-05-23 Update:*** It appears that no new errors appear with the lookback procedure disabled. Also, the error rate for the error correction with the lookback

procedure disabled is close to the expected theoretical results (assuming the frame is large enough) [**brassard93**]. For this, we consider the expression for the expected number of errors to generalize for all iterations, which should be valid as the lookback procedure is disabled, and therefore all iterations are essentially equal.

- ***2019-05-23 Update:*** Seems that during lookback, the used original partition sizes were wrong, although they were equal between Alice and Bob. That meant that they didn't always contain the original error pair. Fixing this
- ***2019-05-24 Update:*** So, the Cascade algorithm was incomplete. The lookback procedure was not as extensive as it should be. It only performed lookback once, with the original key. However, it is supposed to be done for every scrambled key, and for every new detected error.
- ***2019-05-27 Update:*** This should now be implemented, and is currently being tested. Error correction seems better, although values are not as good as described in [**martinez14**]. Still, testing this properly takes a long time, as the expected BER is on the order of 10^{-7} (or lower, depending on the QBER). We could try to compare higher BERS, but the variation is much smaller in those, making it harder to detect issues. In order to get good statistics, we should look for at least around 100 errors. At this point, that is unpractical. Testing for a QBER of 0.01 and a frame of 10000 bits, the final raw BER is approximately 2×10^{-6}, and the frame error rate is 10^{-3}, using four iterations.
- ***2019-05-28 Update:*** Some additional bugs were fixed relating to error correction. Nevertheless, the error rate still does not read 10^{-7}.

3. Recalculating random permutations each time in the Cascade might not be optimal. Maybe the permutations can be kept between successive incoming sifted key strings, if their size is the same.

   - ***2019-05-21 Update:*** Permutations can now saved between sequential runs. However, this carries a risk: if one of obtained the permutation vectors is of low quality, it will perpetuate itself until the end of the simulation. Should verify if performance is worth this risk. A good tradeoff might be to renew matrixes after a few uses. However, it does not seem to improve speed. No significantly, at least. Therefore, the previous behavior was maintained, which minimizes the risk of a bad matrix propagating through the whole simulation.

4. Hash function currently used in privacy amplification is probably not adequate. The function currently used is *hash* from the standard library, and is not seeded. A 2 universal hash function should be used, and should use random seeds shared between Bob and Alice (so that a random strongly universal hash function is used, instead of a deterministic one).

5. The number of bits to ignore in privacy amplification should be verified. Different sources claim different values (number of parities exchanged, estimated number of

measurements from Alice, etc.). Currently, privacy amplification discards a number of bits k per each processed block, where $k = 2n\text{BER} + s$. Here, n is the number of bits in the processed block and and s is a constant number similar to the security parameter. The security parameter is usually only needed for the whole key, and implementing it this way may discard much more bits than necessary, particularly for small keys.

- ***2019-05-28 Update:*** Updating the system so it works in discrete segments of data. Process starts once N data points are available for doing parameter estimation using K random bits. Error correction proceeds once this is done, using N-K bits. M bits are discarded by this process. Privacy amplification should start once it has N-K-M bits available, providing a secure key with N-K-N-S bits.

6. While the implementation is mostly independent, increasing the number of bits processed in a run of Cascade improves its performance. To increase that number, the size of the buffer connecting the error correction with the message processors needs to be at least as high as the number of processed bits. This can be changed in Bob and Alice by modifying the value *maximumECSyndromeSize*. However, this needs to be changed in the code, as currently it is the only way to change the size of a signal buffer.

   - **Fixed:** The specific buffer size for the relevant signals can now be defined during superblock creation.

7. In addition, the messages for sharing the permutation indexes are huge. The buffer size mentioned previously needs to be big because of these indexes. A better alternative could be to instead use a a pseudorandom number generator to create the permutations, sharing a string between Alice and Bob. The buffer size then would instead limit the number of partitions to be used at once, instead of the total number of bits. This could easily increase the number of bits to be processed at once by 8 to 40 times, for the same buffer size (depending on the BER).

   - **Fixed:** Bob sends Alice the seed for generating permutations. The permutations are then generated by both independently, and as they use the same seed, they should be equal. This reduces the size of required communications for using the matrices and increases the maximum frame size.

8. In an ideal scenario, the key would be processed all at once, after the transmission ended. This would maximize the effectiveness of the error correction, and the number of classical channel uses would be minimal (the partition size would be the same, and the number of classical channel uses done by the Cascade algorithm is independent of key length, assuming message size is arbitrary).

   - ***2019-05-28 Update:*** Latest work in parameter estimation and error correction tries to move toward processing data in discrete blocks, doing parameter estimation, error correction and privacy amplification for each block. I believe this is the best way to do this.

6.2 Discrete Variables Quantum Tx/Rx system - Physical Layer

Student Name	:	Mariana Ramos
Starting Date	:	May 05, 2019
Goal	:	Algorithm for polarization drift compensation with discrete variables.
Directory	:	sdf/dv_polarization_encoding_system.

6.2.1 Simulation Analysis - Algorithm for polarization compensation

Figure 6.8: Block diagram of a DV quantum Tx/Rx system to emulate the physical layer.

This system emulates the physical layer of a quantum communication system. In this way, it comprises three super blocks. The first is the AliceQTx, which emulates the quantum transmitter of single-photons using polarization encoding. This block has three inputs: the ModeSelection, when it has value '0' selects the monitoring mode in which Alice transmits 1 control qubit in each **N** transmitted qubits. The remaining 99 qubits are data qubits which are encoded taking into account the other two inputs that are binary signals with basis, AliceQTxBasis, and other with the bit values, AliceQTxBits. Combining these two binary signals Alice is capable of encode 4 different states of polarization in two non-orthogonal basis in the photonStreamXY signal AliceQTxOut. When ModeSelection has the value '1' AliceQTx only transmits control qubits continuously according with a pre-defined sequence. The super block quantum channel accepts and outputs a photonStreamXY signal, and intends to emulate the polarization random drift throughout the optical fiber and the attenuation induced by it. The super block BobQRx receives the photonStreamXY signal and will measure and decode the single photon. This block outputs a binary signal ModeSelection that later inputs Alice super block. It also outputs BobQRxDataOut, which is a signal with Bob's measurements. This is a real signal has five possible value: 5 if it is a control qubit, and the upper layer should simply discard it, 3 if no-click happened, 2 if both

detectors clicked, 0 if bit zero was measured and 1 if bit one was measured. It also outputs a real signal with the quantum channel QBER, QBER_Qchannel, and the ModeSelection. BobQRx accepts a binary signal, BobQRxBasis, which comprises the basis for data qubits measurements.

6.2.2 Simulation Analysis - TestBench

Figure 6.9: TestBench to test the DV quantum Tx/Rx system the physical layer emulator.

This testbench was developed for testing the quantum physical layer presented in figure 6.8. In this way, the **modeSelection** signal was directly connected from **BobQRx** to **AliceQTx**.

This system must be set with the following input parameters:

- *double* bitRate{ 1e3 },
- *double* channelAttenuationDb{ 0.2 },
- *sop_rotation_type* channelModel{ sop_rotation_type::Stocastic },
- *double* PolarizationLinewidth{ 900e-9 },
- *double* fiberLength_km{ 50 },
- *double* nPhotonsData{ 1 },
- *double* nPhotonsControl{ 80 },
- *int* nIncrementControl{ 100 }.

Furthermore, six more blocks were added with the following input parameters:

- BSAliceQTxBasis_:

  - setMode(BinarySourceMode::DeterministicCyclic)
  - setBitPeriod(1 / bitRate)
  - setBitStream("1")
  - setNumberOfBits(nBits);

- BSAliceQTxBits_:
  - setMode(BinarySourceMode::DeterministicCyclic)
  - setBitPeriod(1 / bitRate)
  - setBitStream("0");

- BSBobQRxBasis_:
  - setMode(BinarySourceMode::DeterministicCyclic)
  - setBitStream("1")
  - setBitPeriod(1 / bitRate);

- QberRaw_;
- BobQRx_Sink_;
- QBERQChannel_Sink_.

The three main blocks were defined with the following parameters:

- AliceQTx_:
  - setControlBasisSequence("0")
  - setControlBitsSequence("0")
  - setNumberOfPhotonsData(nPhotonsData)
  - setNumberOfPhotonsControl(nPhotonsControl)
  - setIncrementControl(nIncrementControl);

- QuantumChannel_:
  - setAtennuationDbmPerKm(channelAttenuationDb)
  - setFiberLength(fiberLength_km)
  - setFiberModel(channelModel)
  - setPolarizationLinewidth(PolarizationLinewidth);

- BobQRx_:
  - setControlSequenceBits("0")
  - setControlBasisMeasurement("0")
  - setIncrementControl(nIncrementControl).

6.2.3 Open Issues

Chapter 7

Library

7.1 ADC

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

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

Input Parameters

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

Table 7.1: ADC input parameters

Methods

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

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

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

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

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

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

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

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

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.2 Add

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

Input Parameters

This block takes no parameters.

Functional Description

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

Input Signals

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

Output Signals

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

7.3 Arithmetic Encoder

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

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

Input Parameters

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

Table 7.2: Arithmetic encoder block input parameters.

Methods

bool runBlock(void)

void initialize(void);

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

Functional description

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

Input Signals

Number: 1

Output Signals

Number: 1

Type: binary

Examples

Sugestions for future improvement

7.4 Arithmetic Decoder

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

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

Input Parameters

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

Table 7.3: Arithmetic decoding block input parameters.

Methods

bool runBlock(void)

void initialize(void);

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

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.5 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

7.6 Alice QKD

This block is the main processor for Alice.

AliceQKD is a superblock intended work as Alice's main processor. It performs a series of functions, including classical channel communication with Bob and data processing for basis reconciliation, error correction and privacy amplification.

Input Parameters

Methods

Functional description

AliceQKD is a superblock intended to do classical channel communication and data processing as required by the BB84 protocol. This superblock requires four different inputs:

- Messages from Bob, which are used to coordinate the required data processing for key extraction;
- A binary signal to be used for the transmitted key;
- A binary signal corresponding to the basis which will be used for encoding the data.
- A binary signal controlling the transmission mode, enabling or disabling transmission.

In addition, this block produces a total of five output signals:

- Messages intended to be read by Bob, used to coordinate the required data processing for key extraction;

- A binary signals of the data, to be transmitted through the quantum channel;
- A binary signal corresponding to the basis which will be used to encoded the transmitted data;
- A binary signal with the generated key;
- A binary signal corresponding to the transmission control.

The process inside the superblock is as follows:

The incoming data and basis binary signals are provided as inputs to the *AliceIncomingDataProcessor*, together with the *transmissionMode* signal. The transmission mode controls the behaviour of the *AliceIncomingDataProcessor*: it either enables it to continue working or tells it to stop reading the input data. The data and basis read at the block are sent twice each to the output signals, keeping their sync. Each pair of basis and data has a different destination: one of the pair is directed to the superblock outputs, in order to transmit the data and control the basis for the transmission; and the other pair is sent to the *BasisReconciliation* block.

The *BasisReconciliation* block in AliceQKD is configured to responding to Bobs messages, instead of sending them first. Therefore, it now waits for a signal from the *MessageProcessorReceiver*. When Bob has enough samples, he should send a message containing the basis he used to measure the received data. This message is received, identified by its type `BasisReconciliation1`, and interpreted at the *MessageProcessorReceiver*. When a message of the correct type arrives, this block sends a signal to the *BasisReconcilliation* block, containing the list of basis Bob used to measure the data on his side. The *BasisReconciliation* block then compares the basis received from Bob with the ones that Alice used, and based on this is outputs two signals:

- A binary evaluation of which basis used by Bob are correct. This is sent to the *MessageProcessorTransmitter*, so that a message can be sent to Bob for him to know which basis were correct.
- A binary signal containing the data sent when those basis were used. This is the data that Bob should have measured correctly. This signal is sent to the *ErrorCorrection* block.

The *ErrorCorrection* block will be responsible for correcting any errors that occurred during transmission. Similarly to the *BasisReconciliation* block, it is configured to act in response to Bobs messages. As such, it waits for data from the *MessageProcessorReceiver*, and outputs a response to the *MessageProcessorTransmitter*. It also outputs a copy of the signal from basis reconciliation, after it has been used for error correction. This is will be the transmitted key, free from errors.

When the *MessageProcessorReceiver* receives a message with the type `ErrorCorrection1`, it starts the Cascade error correction process. That message contains the necessary information used

in the Cascade process. The explanation of the whole error correction process is somewhat long and not required to understand how the AliceQKD block works. Therefore, further details can be found in the *ErrorCorrection* block's documentation.

After each set of bits is corrected, the data is output to the *PrivacyAmplification* block, which does the necessary transformation to ensure the information that any attacker might have gained (Eve) is nullified.

The end result, and final output of the superblock, is a secure key shared known only to Alice and Bob.

As mentioned before, this whole process is controlled by the *transmissionMode* signal.

Input Signals

Number : 3

Type : Binary, Real Continuous Time and Messages signals.

Output Signals

Number : 4

Type : Binary, Real Discrete Time and Messages signals.

Examples

Sugestions for future improvement

7.7 Ascii Source

Header File	:	ascii_source_*.h
Source File	:	ascii_source_*.cpp
Version	:	20180828 (André Mourato)

This block generates an ascii signal and can work in different modes:

1. Terminate
2. Cyclic
3. AppendZeros

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

Input Parameters

Parameter	Type	Values	Default
mode	AsciiSourceMode	Terminate, AppendZeros, Cyclic	Terminate
asciiString	string	any	“"
asciiFilePath	string	any	“text_file.txt"
numberOfCharacters	int	$\in [0, \infty[$	1000

Table 7.4: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setMode(AsciiSourceMode m);

AsciiSourceMode const getMode(void);

void setAsciiString(string s);

string getAsciiFilePath();

void setNumberOfCharacters(int n);

int getNumberOfCharacters();

Functional description

The *mode* parameter allows the user to select one of the operation modes of the ascii source.

Terminate The resulting ascii signal will be the $asciiString$ sequence of characters.

AppendZeros The resulting ascii signal will be a sequence of characters starting with $asciiString$ with zeros (null character) concatenated to the right until $numberOfCharacters$ characters have been generated.

Cyclic The resulting ascii signal will be a sequence of characters in which $asciiString$ is concatenated to the right of the string until $numberOfCharacters$ characters have been generated.

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: Ascii

7.8 Ascii To Binary

Header File	:	ascii_to_binary_*.h
Source File	:	ascii_to_binary_*.cpp
Version	:	20180905 (André Mourato)

Methods

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

void initialize(void);

bool runBlock(void);

Functional description

Figure 7.6 shows an example of an input signal that can be passed as argument. This signal contains three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. The values to these three characters are respectively: 01100001, 1100010 and 1100011. The block AsciiToBinary will convert the characters a, b and c to their respective binary codes. The resulting output signal will be of type Binary. The output signal to this example, shown in figure 7.5, is the concatenation of the previous binary codes. The full binary sequence is 011000011100010110001 1.

Figure 7.1: Ascii signal passed as input to the AsciiToBinary block

Figure 7.2: Resulting Binary signal from the output of the AsciiToBinary block

Input Signals

Number: 1

Type: Ascii

Output Signals

Number: 1

Type: Ascii

7.9 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

7.10 Basis Reconciliation

Header File	:	basis_reconciliation_*.h
Source File	:	basis_reconciliation_*.cpp
Version	:	20190410 (Andoni Santos)

This block is used to reconcile information between two parties in Quantum Key Distribution, Alice and Bob, by exchanging information about the basis used to measure the data or the failure to properly measure the qubit.

Signals

Number of Input Signals	3
Type of Input Signals	TimeContinuousAmplitudeDiscreteReal, Binary and TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	3
Type of Output Signals	Binary, Binary and TimeDiscreteAmplitudeContinuousReal

Table 7.5: Binary source signals

Input Parameters

Parameter	Type	Values	Default
basisReconciliationRole	basis_reconciliation_role	Alice, Bob, Undefined	Undefined
doubleClickNumber	int	any	2
noClickNumber	int	any	3

Table 7.6: Basis reconciliation input parameters

Methods

BasisReconciliation(initializer_list<Signal *> InputSig, initializer_list<Signal *> OutputSig);

void initialize(void);

bool runBlock(void);

void setBasisReconciliationRole(basis_reconciliation_role role);

void setDoubleClickNumber(t_integer dcn);

void setNoClickNumber(t_integer ncn);

long int getReceivedQubits(void);

long int getReconciledBits(void);

Functional description

This block has two different behaviors: one for Alice (the transmitter), and another for Bob (the receiver). Initially, after Alice encodes the data with the chosen basis and sends it to Bob, she waits until Bob decides to start the process.

Bob, after successfully receiving data from the quantum channel, sends a message to Alice with the information required to reconcile the data. This message contains the either the basis that Bob used to measure the incoming qubits, or, in case it happens, a code indicating that either the detectors both clicked, or none of them clicked. After sending the message, Bob stores the data until he gets a response from Alice.

When Alice receives the message, she compares the basis with the ones she used, and prepares a response. If the basis were equal, she writes a 1 in the message and outputs that bit; otherwise, she writes 0 in the message and discards that bit.

Bob then receives the message and, according to the results for each bit, he either outputs it or discards it. He then waits until he has more data so that this process can begin again.

Suggestions for future improvement

Figure out if a minimum number of values should be implemented, in order to avoid wasting time communicating to reconcile a small number of bits. Typically, communication using TCP on the classical channel should be minimized, as each transmission requires an acknowledgement.,The time needed for a round trip is not negligible, particularly if done frequently.

7.11 Bit Error Rate

Header File	:	bit_error_rate_*.h
Source File	:	bit_error_rate_*.cpp
Version	:	20180815
Contributors	:	Daniel Pereira
	:	Mariana Ramos
	:	Manuel Neves
	:	Armando Pinto

This block estimates the bit-error rate. It compares two input binary signals and outputs a binary signal with a zero if both input bits are equal and a one if they differ. Additionally, this block does statistical analysis on the BER and outputs a file with the results of this analysis. The statistical analysis consists on the upper and lower bounds of the BER value, as well as its average value.
The block also allows having breakpoints on the BER analysis, outputting mid-reports (instead of only one final report).

Signals

Input Signals

Number: 2
Type: Binary

These two input signals are (interchangeably) the received binary sequence, from the MQAM Receiver output, and the transmitted signal, directly from the Binary Source.

Output Signals

Number: 1
Type: Binary

The output signal is a binary vector of '1's and '0's, with the same length as the input signals, in which a '0' indicates the occurrence of an error and a '1' a successfully transmitted bit.

Input Parameters

Name	Type	Default Value	Description
alpha	double	0.05	Alpha is one minus the confidence level in the BER interval.
m	integer	0	Defines how many bits are analysed in between mid-reports. When null there aren't any mid-reports.
lMinorant	double	1×10^{-10}	Defines the minimum value of the lower bound for the BER.

Block Constructor

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

This block's constructor expects, as any *Block* object, an input of two vectors of Signal pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

Methods

- void **initialize**(void);
  - Sets symbol and sampling period and the *firstValueToBeSaved* of the output signal, based on the input signals.
- bool **runBlock**(void);
  - Computes output statistics and writes to BER.txt and mid-report#.txt the obtained results.
- double const **getConfidence**(double P);
  - Returns the value of the confidence level based on the private parameter *alpha*.
- void **setConfidence**(double P);
  - Defines the value of the *alpha* parameter, based on the wished confidence.
- int const **getMidReportSize**(int M);
  - Returns the value of the private parameter *m*.
- void **setMidReportSize**(int M);
  - Alters the value of the *m* parameter, from its default.

- double **getLowestMinorant**(double lMinorant);
  - Returns the value of the private parameter *lMinorant*.
- void **setLowestMinorant**(double lMinorant);
  - Alters the value of the *lMinorant* parameter, from its default.

Examples

- One interesting test we can do with this block, is analyse how a certain parameter of the system affects de BER. In the following graph we demonstrate an example of how the Noise Spectral Density of the TI Amplifier on the receiver affects the BER:

  To obtain this result a Transmitter+Receiver QPSK system was ran several times, applying different values to the spectral density of the noise at the input of the TI amplifier.

- Another interesting aspect facilitated by the BER block is to analyse how the BER tends to its real value as the number of the number of received bits increases. We can do this simply by analysing the multiple mid-reports, by giving a sufficiently low value to the *m* parameter.
  The following graph was obtained in the same configuration as the previous graph, for a fixed noise spectral density value of 5e-17W/Hz:

It is interesting to observe how the confidence bounds shrink, and that the BER value is within the previous values of these bounds (in at least $(1-\alpha)\%$ of the cases).

Output Signals

Number: 1
Type: Binary (DiscreteTimeDiscreteAmplitude)

Functional and Theoretical Description

This block accepts two binary strings and outputs a binary string. For each pair of input bits, it outputs a '1' if the two coincide and '0' if not.
The $\widehat{\text{BER}}$ is thus obtained by counting both the total number received bits, N_{bits}, and the number of coincidences, K, and calculating their relative ratio:

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

The confidence interval of the $\widehat{\text{BER}}$ value is also calculated. To easily obtain the upper and lower bounds, BER_{UB} and BER_{LB} respectively, an approximation of the Clopper-Pearson

confidence interval [**Almeida16**] is used, which holds valid for $N_{bits} > 40$:

$$\mathrm{BER_{UB}} = \widehat{\mathrm{BER}} + \frac{z_{\alpha/2}}{\sqrt{N_{bits}}}\sqrt{\widehat{\mathrm{BER}}(1-\widehat{\mathrm{BER}})} + \frac{1}{3N_{bits}}\left[2\left(\frac{1}{2}-\widehat{\mathrm{BER}}\right)z_{\alpha/2}^2 + 2 - \widehat{\mathrm{BER}}\right] \quad (7.4)$$

$$\mathrm{BER_{LB}} = \widehat{\mathrm{BER}} - \frac{z_{\alpha/2}}{\sqrt{N_{bits}}}\sqrt{\widehat{\mathrm{BER}}(1-\widehat{\mathrm{BER}})} + \frac{1}{3N_{bits}}\left[2\left(\frac{1}{2}-\widehat{\mathrm{BER}}\right)z_{\alpha/2}^2 - 1 - \widehat{\mathrm{BER}}\right] \quad (7.5)$$

where $z_{\alpha/2}$ is the $100\left(1-\frac{\alpha}{2}\right)$th percentile of a standard normal distribution and this is where the value of the *alpha* parameter influences the result.
Upon finishing running, this block outputs to the *signals* directory a file named *BER.txt* with a report of the estimated $\widehat{\mathrm{BER}}$, the estimated confidence bounds for a given confidence level $(1-\alpha)$ and the total number of received bits.
The file format is the following:

```
BER= 4.34783e-05
Upper and lower confidence bounds for 95% confidence level
Upper Bound= 9.33303e-05
Lower Bound= 1.47979e-05
Number of received bits =92000
```

Another functionality of the BER block is that it allows for mid-reports to be generated. A mid-report has exactly the same structure of the final report showed above, but it's generated for every *m* pairs of input bits. These files are saved in the same directory of the running solution and they're saved on files named *midreport#.txt*, where the '#' stands for the number of the respective mid-report and is in the interval $[1; \frac{N_{bits}}{m}]$.
The number of bits between reports is the parameter *m*, customizable, and if it is set to '0' then the block will only output the final report.

Open Issues

- The output value for the Lower bound is negative when the BER is null, and in these cases it should be set to '0'. This error is hidden in the *BER.txt* file, because when the Lower Bound value is lower than the *lMinorant* parameter, an *if* statement sets it to the value of *lMinorant*. This correction can however lead to errors in the cases that the *lMinorant* is smaller than the actual BER we're attempting to measure.

- Mid-reports are not formatted exactly like the BER output file. And they are stored in the current directory, instead of being saved in the same directory as the *BER.txt* file, which would probably be more desirable.

Future Improvements

- fixing the Lower Bound issue mentioned above;

- the printing of all the mid-reports could be done in the same file, for ease of BER convergence analysis;

- the method *getLowestMinorant()* should have a *const* return;

- there could be only one private method for printing an output file, thus avoiding having two times the same code repeated for the mid-report printing and for the BER final report printing;
- exactness required for the convergence of the z-score should also be a parameter, for it may be desirable to have a different value of exactness.

7.12 Binary Source

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

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

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

Signals

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

Table 7.7: Binary source signals

Input Parameters

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

Table 7.8: Binary source input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

Functional description

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

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

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

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

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

Input Signals

Number: 0

Type: Binary (DiscreteTimeDiscreteAmplitude)

Output Signals

Number: 1 or more

Type: Binary (DiscreteTimeDiscreteAmplitude)

Illustrative Examples

Random Mode

PseudoRandom Mode Consider a pseudorandom sequence with *patternLength*=3 which contains a total of 7 ($2^3 - 1$) bits. In this sequence it is possible to find every combination of 0's and 1's that compose a 3 bit long subsequence with the exception of 000. For this example the possible subsequences are 010, 110, 101, 100, 111, 001 and 100 (they appear in figure 7.3 numbered in this order). Some of these require wrap.

Figure 7.3: Example of a pseudorandom sequence with a pattern length equal to 3.

Figure 7.4: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

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

DeterministicAppendZeros Mode Take the *bit stream* '0100011101010101'. The generated binary signal is displayed in 7.4.

Suggestions for future improvement

Implement an input signal that can work as trigger.

7.13 Binary To Ascii

Header File	:	binary_to_ascii_*.h
Source File	:	binary_to_ascii_*.cpp
Version	:	20180905 (André Mourato)

Methods

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

void initialize(void);

bool runBlock(void);

Functional description

Figure 7.5 shows an example of an input signal that can be passed as argument. This signal contains the binary representation of three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. This signal contains the following stream of bits: 011000011100010110001 1. There are 24 bits in this stream, therefore we can divide it into three segments of 8 bits each. The resulting segments are: 01100001, 1100010 and 1100011. Each of these segments is the binary code of a character. 01100001 represents the character a, 1100010 represents the character b and 1100011 represents the character c. The block BinaryToAscii will convert the binary codes into the respective characters. The resulting output signal will be of type Ascii. The output signal to this example, shown in figure 7.6, contains the characters that can be represented with the previous binary codes.

Figure 7.5: Binary signal passed as input to the BinaryToAscii block

Figure 7.6: Resulting Ascii signal from the output of the BinaryToAscii block

Input Signals

Number: 1

Type: Ascii

Output Signals

Number: 1

Type: Ascii

7.14 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

7.15 Bob QKD

This block is the main processor for Alice.

AliceQKD is a superblock intended work as Bob's main processor. It performs a series of functions, including classical channel communication with Alice and data processing for basis reconciliation, error correction and privacy amplification.

Input Parameters

-
-

Methods

Functional description

BobQKD is a superblock intended to do classical channel communication and data processing as required by the BB84 protocol. This superblock requires four different inputs:

- Messages from Aliec, which are used to coordinate the required data processing for key extraction;
- A binary signal corresponding to the measured trasmitted data;
- A binary signal corresponding to the basis used by Bob for decoding the data sent by Alice.
- A continuous amplitude signal of the estimated Bit Error Rate, which is used to control the transmission process.

In addition, this block produces a total of three output signals:

- Messages intended to be read by Alice, used to coordinate the required data processing for key extraction;
- A binary signal with the generated key;
- A binary signal corresponding to the transmission mode sent out.

The process inside the superblock is as follows:

The incoming BER signal is analyzed at the *TransmissionControl* block. If the BER is too high, the process stops

The incoming data and basis signals are received in the *BobIncomingDataProcessor* block. If the transmission is enabled, the incoming data and basis are sent into the *BasisReconciliation* block

The *BasisReconciliation* block in BobQKD is configured to start the basis reconciliation process. Thus, when enough samples are available, Bob sends a signal with the list of used basis to the *MessageProcessorTransmitter* block. The block then waits. When Alice responds, a message arrives at the *MessageProcessorReceiver* block. It contains a binary string indicating whether Bob used the correct basis or not. This information is then sent to the *BasisReconciliation* block. There, the bits where Bob got the right basis are sent to the *ErrorCorrection* block, while the remaining ones are promptly discarded.

The *ErrorCorrection* block will be responsible for correcting any errors that occurred during transmission. Similarly to the *BasisReconciliation* block, in BobQKD it is configured to start acting when enough samples are available. When this condition is verified, the *ErrorCorrection* block starts the Cascade process by dividing the available bits in segments, calculating their parities and sending them to the *MessageProcessorTransmitter*. Once this is done, it waits until a response from Alice arrives at *MessageProcessorReceiver*. The message should contain a reponse to Bob on whether the parities are similar or not. The Cascade process continues until it is concluded. The Cascade process is explained in the section of the *ErrorCorrection* block, but is not fundamental to understanding the BobQKD superblock.

When the cascade process ends, the corrected data is output to the *PrivacyAmplification* block, which does the necessary transformation to ensure the information that any attacker might have gained (Eve) is nullified.

The end result, and final output of the superblock, is a secure key shared known only to Alice and Bob.

Input Signals

Examples

Sugestions for future improvement

7.16 BobQuantumRx

Header File	:	BobQRx.h
Source File	:	BobQRx.cpp
Version	:	20190410

This block is quantum channel Bob's receiver. This block accepts a binary signal, which comprises the values of basis to measure the encode single photons. These values must be randomly chosen between two different basis corresponding to the two non-orthogonal basis needed. Furthermore, this block also accepts a PhotonStreamXY signal corresponding to the single photon that arrives from quantum channel. It produces a binary signal which contains the modeSelection.

- inputSignal[0] → BobQRxBasis (Binary)
- inputSignal[1] → BobQRx_In (PhotonStreamXY)

This block produces the following output signals,

- outputSignal[0] → BobQRx_DataOut (TimeDiscreteAmplitudeContinuousReal)
- outputSignal[1] → ModeSelection (Binary)
- outputSignal[2] → QBER_QChannel (TimeDiscreteAmplitudeContinuousReal)

Input Parameters

- string controlBasisSequence - This string is the sequence of basis used to control qubits. Its default value is "0".
- string controlSequence - This string is the sequence of bits used to control qubits. Its default value is "0".

Methods

- void setControlBasisMeasurement(string cBasis) - This method allows to set the control basis measurement sequence accepting a string value.
- void setControlSequenceBits(string cSeq) - This method allows to set the control bits sequence to use for quantum bit error rate estimation, taking into account this sequence. It accepts a string value.

Figure 7.7: Schematic of Bob Quantum Receiver

Functional description

This block receives 1 input binary signal that will have the information about basis ("BobQRxBasis") to measure the single-photons that arrives from the quantum channel. This signal must comprise two possible values corresponding to two non-orthogonal basis used in the communication protocol. It also accepts a PhotonStreamXY signal corresponding to the encoded single photon that arrives from quantum channel sent by AliceQTx.

The mode selection signal that outputs this superblock will inform Alice about which mode she should selects, and so that if she should send data or control qubits.

Apart from the detection scheme, this block also comprises an active random polarization drift compensation scheme. This scheme includes a randomSopCompensation block that is responsible for all post processing tasks, including QBER estimation, and sends three feedback signals with information about the rotations to be performed by an electronic polarization controller (EPC). In this way, the random polarization drift is compensated before the single photons reach the detection scheme.

Input Signals

Number : 2

Type : Binary, PhotonStreamXY.

Output Signals

Number : 3

Type : TimeDiscreteAmplitudeContinuousReal, Binary.

Examples

Sugestions for future improvement

7.17 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)

7.18 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 7.9: 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

7.19 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 7.8: Example of the output signal of the clock without phase shift.

Figure 7.9: Example of the output signal of the clock with phase shift.

7.20 Complex To Real

Header File	:	complex_to_real_∗.h
Source File	:	complex_to_real_∗.cpp
Version	:	20180717 (Celestino Martins)

This super block converts a complex input signal into two real signals.

Input Parameters

Methods

- ComplexToReal() {};
- ComplexToReal(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);

Functional description

This super block converts a complex input signal into two real signals.

Input Signals

Number: 1

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

Figure 7.10: 2x2 coupler

7.21 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.

7.22 Carrier Phase Compensation

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

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

Input Parameters For VV Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
methodType	string	VV	VV
mQAM	int	any	4

Table 7.10: CPE input parameters

Input Parameters For BPS Algorithms

Parameter	Type	Values	Default
nTaps	int	any	25
NtestPhase	int	any	32
methodType	string	BPS	VV
mQAM	int	any	4

Table 7.11: CPE input parameters

Methods

CarrierPhaseCompensation() ;

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

void initialize(void);

bool runBlock(void);

void setnTaps(int ntaps) nTaps = ntaps;

double getnTaps() return nTaps;

void setmQAM(int mQAMs) mQAM = mQAMs;

double getmQAM() return mQAM;

void setTestPhase(int nTphase) nTestPhase = nTphase;

double getTestPhase() return nTestPhase;

void setmethodType(string mType) methodType = mType;

string getmethodType() return methodType;

void setBPStype(string tBPS) BPStype = tBPS;

string getBPStype() return BPStype;

Functional description

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

Viterbi-Viterbi Algorithm

Figure 7.11: Block diagram of Viterbi-Viterbi algorithm for carrier phase recovery.

VV algorithm is a n-th power feed-forward approach employed for uniform angular distribution characteristic of m-PSK constellations, where the information of the modulated

phase is removed by employing the n-th power operation on the received symbols. The algorithm implementation diagram is shown in Figure 7.11, starting with M-th power operation on the received symbols. In order to minimize the impact of additive noise in the estimation process, a sum of $2N + 1$ symbols is considered, which is then divided by M. The resulting estimated phase noise is then submitted to a phase unwrap function in order to avoid the occurrence of cycle slip. The final phase noise estimator is then used to compensate for the phase noise of the original symbol in the middle of the symbols block.

Blind Phase Search Algorithm

Figure 7.12: Block diagram of blind phase search algorithm for carrier phase recovery.

An alternative to the VV phase noise estimator is the so-called BPS algorithm, in which the operation principle is shown in the Figure 7.12. Firstly, a block of $2N + 1$ consecutive received symbols is rotated by a number of B uniformly distributed test phases defined as,

$$\phi_b = \frac{b}{B}\frac{\pi}{2}, b \in \{0, 1, ..., B - 1\}. \tag{7.9}$$

Then, the rotated blocks symbols are fed into decision circuit, where the square distance to the closest constellation points in the original constellation is calculated for each block. Each resulting square distances block is summed up to minimize the noise distortion. After average filtering, the test phase providing the minimum sum of distances is considered to be the phase noise estimator for the symbol in the middle of the block. The estimated phase noise is then unwrapped to reduce cycle slip occurrence, which is then used employed for the compensation for the phase noise of the original symbols.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.23 Decision Circuit

Header File	:	decision_circuit_∗.h
Source File	:	decision_circuit_∗.cpp
Version	:	20181012 (Celestino Martins)

This block performs the symbols decision, by calculating the minimum distance between the received symbol relatively to the constellation map. It receives one input complex signal and outputs one complex signal.

Input Parameters

Parameter	Type	Values	Default
mQAM	int	any	4

Table 7.12: Decision circuit input parameters

Methods

DecisionCircuitMQAM() ;

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

void initialize(void);

bool runBlock(void);

void setmQAM(int mQAMs) mQAM = mQAMs;

double getmQAM() return mQAM;

Functional description

This block performs the symbols decision, by minimizing the distance between the received symbol relatively to the constellation map. It perform the symbol decision for the modulations formats, 4QAM and 16QAM. The order of modulation format is defined by the parameter $mQAM$.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

Extend the decision to higher order modulation format.

7.24 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 7.13: 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 7.13: 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

7.25 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 7.14: 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 7.14: Example of the type of signal generated by this block for a binary sequence 0100...

7.26 DownSampling

Header File	:	down_sampling_∗.h
Source File	:	down_sampling_∗.cpp
Version	:	20180917 (Celestino Martins)

This block simulates the down-sampling function, where the signal sample rate is decreased by integer factor.

Input Parameters

Parameter	Type	Values	Default
downSamplingFactor	int	any	2

Table 7.15: DownSampling input parameters

Methods

DownSampling() ;

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

void initialize(void);

bool runBlock(void);

void setSamplingFactor(unsigned int dSamplingfactor) downSamplingFactor = dSamplingfactor;

unsigned int getSamplingFactor() return downSamplingFactor;

Functional description

This block perform decreases the sample rate of input signal by a factor of $downSamplingFactor$. Given a down-sampling factor, $downSamplingFactor$, the output signal correspond to the first sample of input signal and every $downSamplingFactor^{th}$ sample after the first.

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical real signal

Examples

Sugestions for future improvement

7.27 DSP

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

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

Input Parameters

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

Table 7.16: DSP input parameters

Methods

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

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

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

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

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

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

Functional description

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

Input Signals

Number: 2

Output Signals

Number: 2

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.28 EDFA

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

This block mimics an EDFA in the simplest way, by accepting one optical input signal and outputting an amplified version of that signal, affected by white noise.

Figure 7.15: Basic configuration of the MQAM receiver

Functional description

This block of code simulates the basic functionality of an EDFA: it amplifies the optical signal by a given gain, and adds noise according to a certain noise figure. Currently the only parameters are the gain and noise figure, and it's assumed that the output power is always far below the EDFA's saturation power. Therefore, the gain and noise spectral density are independent of the signal. The noise spectral density is calculated from the noise figure, gain and wavelength.

This block is made of smaller blocks, and its internal constitution is shown in Figure 7.16.

Figure 7.16: Schematic representation of the block homodyne receiver.

Input parameters

Input parameters	Function	Type
powerGain_dB	setGain_dB	t_real
noiseFigure	setNoiseFigure	t_real
samplingPeriod	setNoiseFigure	t_real
wavelength	setWavelength	t_real
dirName	setDirName	string

Table 7.17: List of input parameters of the EDFA block

Methods

Edfa(vector<Signal *> &inputSignal, vector<Signal *> &outputSignal); (**constructor**)0

void setGain_dB(t_real newGain)

t_real getGain_dB(void)

void setNoiseFigure(t_real newNoiseFigure)

t_real getNoiseFigure(void)

void setNoiseSamplingPeriod(t_real newSamplingPeriod)

t_real getNoiseSamplingPeriod(void)

void setWavelength(t_real newWavelength)

t_real getNoiseFigure(void)

void setDirName(string newDirName);

string getDirName(void)

Input Signals

Number: 1

Type: Optical signal

Output Signals

Number: 1

Type: Optical signal

Example

Sugestions for future improvement

7.29 Electrical Signal Generator

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

7.29.1 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.

7.30 Entropy Estimator

Header File	:	entropy_estimator_*.h
Source File	:	entropy_estimator_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal,a binary, and it produces an output signal with the entropy value. No input variables in this block.

Functional Description

This block calculates the entropy of a binary source code composed 0 and 1.

Input Signals

Number: 1
Type: Binary

Output Signals

Number: 1
Type: Real (TimeContinuousAmplitudeContinuousReal)

7.31 Entropy Estimator

Functional Description

entropyEst(vector<Signal *> &InputSig, int window)

The estimator sweeps the full range of the binary input computing an entropy estimation for each window. Then, the entropy mean, the variance and the individual entropy estimations are outputted to a file.

Input Parameters

The block accepts as input parameter the window size, which defines the length over which each entropy estimation is computed.

Note: If the length of the binary stream is not an integer multiple of the window size, the estimator considers the window size equal to the full length of the input signal.

Input Signals

Number: 1
Type: Binary Stream

Output Data

Entropy mean, entropy variance and entropy estimations.

The entropy estimator generates a file with the name `"entropy_est.txt"` where the output data is written.

Results

Binary entropy estimator
Estimator
Theoretical
Binary Entropy (bits)
p
Input length: 10^6 bits
Window length: $5x10^3$ bits
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

7.32 Error correction

Header Files	:	error_correction_*.h
Source Files	:	error_correction_*.cpp
Version	:	20190410 (Andoni Santos)

This block is used to correct the errors in the reconciled keys shared by two parties in a quantum key distribution protocol. It currently implements the Cascade error correction method, which resorts to iteratively exchanging the parities of subsets from the sifted key in order to find errors.

Signals

If errorCorrectionRole is Alice:

Number of Input Signals	5
Type of Input Signals	Binary, TimeDiscreteAmplitudeContinuousReal, TimeDiscreteAmplitudeContinuousReal, TimeDiscreteAmplitudeContinuousReal, TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	2
Type of Output Signals	Binary, TimeDiscreteAmplitudeContinuousReal

Table 7.18: Error correction signals - Alice

If errorCorrectionRole is Bob:

Number of Input Signals	3
Type of Input Signals	Binary, TimeDiscreteAmplitudeContinuousReal TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	4
Type of Output Signals	Binary, TimeDiscreteAmplitudeContinuousReal TimeDiscreteAmplitudeContinuousReal TimeDiscreteAmplitudeContinuousReal

Table 7.19: Error correction signals - Bob

Input Parameters

Methods

ErrorCorrection(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig) : Block(InputSig, OutputSig) ;

initialize(void);

runBlock(void);

setErrorCorrectionRole(error_correction_role role)

setPartitionSize(t_integer wsz)

setBer(t_real berValue)

setNumberOfPasses(t_integer np)

setDoublePartitionSize(bool db)

setMaximumSyndromeSize(t_integer maxss)

setMinimumSyndromeSize(t_integer minss)

setMinimumNumberOfPartitions(t_integer mnp)

getInputBits(void)

getOutputBits(void)

getNumberOfExchangedParities(void)

Functional description

Suggestions for future improvement

7.33 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 7.17: Fork

7.34 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 7.20: 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

7.35 Hamming Decoder

Header File	:	hamming_decoder_*.h
Source File	:	hamming_decoder_*.cpp
Version	:	20180806 (Luís Almeida)

Input Parameters

This block accepts two input parameters ($nBits$ and $kBits$) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair (n, k) are present in columns n and k.

Parity Bits	n **(bits)**	k **(bits)**	**Hamming Code**	**Rate**
2	3	1	(3, 1)	1/3
3	7	4	(7, 4)	4/7
4	15	11	(15, 11)	11/15
5	31	26	(31, 26)	26/31
6	63	57	(63, 57)	57/63
7	127	120	(127, 120)	120/127
8	255	247	(255, 247)	247/255

Functional Description

This block performs the decoding of the input signal using the selected Hamming Algorithm and outputs the decoded signal.

Input Signals

Number: 1
Type: Binary Signal

Output Signals

Number: 1
Type: Binary Signal

7.36 Hamming Encoder

Header File	:	hamming_encoder_*.h
Source File	:	hamming_encoder_*.cpp
Version	:	20180806 (Luís Almeida)

Input Parameters

This block accepts two input parameters ($nBits$ and $kBits$) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair (n, k) are present in columns n and k.

Parity Bits	n **(bits)**	k **(bits)**	**Hamming Code**	**Rate**
2	3	1	(3, 1)	1/3
3	7	4	(7, 4)	4/7
4	15	11	(15, 11)	11/15
5	31	26	(31, 26)	26/31
6	63	57	(63, 57)	57/63
7	127	120	(127, 120)	120/127
8	255	247	(255, 247)	247/255

Functional Description

This block performs the encoding of the input signal using the selected Hamming Algorithm and outputs the encoded signal.

Input Signals

Number: 1
Type: Binary Signal

Output Signals

Number: 1
Type: Binary Signal

7.37 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 7.18.

Figure 7.18: 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 7.19) of code made up of several simpler blocks whose description can be found in the *lib* repository.

In can also be seen from figure 7.19 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 7.19: 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 7.46) 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 7.21: 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

7.38 Huffman Decoder

Header File	:	huffman_decoder_*.h
Source File	:	huffman_decoder_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal, a binary signal with the message to decode, and it produces an output signal (message decoded). Two inputs are required, the probabilityOfZero and the sourceOrder.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 7.22: Huffman Decoder input parameters

Functional Description

This block decodes a message using Huffman method for a source order of 2, 3 and 4.

Input Signals

Number: 1
Type: Binary

Output Signals

Number: 1
Type: Binary

7.39 Huffman Encoder

Header File	:	huffman_encoder_*.h
Source File	:	huffman_encoder_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

The block accepts one input signal,a binary signal with the message to encode, and it produces an output signal (message encoded). Two inputs are required, the probabilityOfZero and the sourceOrder.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 7.23: Huffman Encoder input parameters

Functional Description

This block encodes a message using Huffman method for a source order of 2, 3 and 4.

Input Signals

Number: 1
Type: Binary

Output Signals

Number: 1
Type: Binary

7.40 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 7.24: Ideal Amplifier input parameters

Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;

Functional description

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

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

7.41 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 7.25: 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 7.20: 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 7.21: 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 7.22: 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 7.23: Mach-Zehnder Modulator

$$\frac{E_{out}(t)}{E_{in}(t)} = \frac{1}{2} \cdot (e^{j\phi_1(t)} + e^{j\phi_2(t)}) \tag{7.10}$$

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

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

Figure 7.24: IQ Mach-Zehnder Modulator

function of the IQ MZM can be written as,

$$E_{out}(t) = \frac{1}{2}E_{in}(t)\left[cos(\frac{\pi U_I(t)}{2V_\pi}) + j \cdot cos\left(\frac{\pi U_Q(t)}{2V_\pi}\right)\right] \tag{7.11}$$

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

Figure 7.25: Simulation model of the IQ Mach-Zehnder Modulator

7.42 IIR Filter

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

Input Parameters

Name	Type	Default Value

Methods

IIR_Filter()

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

void initialize(void)

bool runBlock(void)

void setBCoeff(vector<double> newBCoeff)

void setACoeff(vector<double> newACoeff)

int getFilterOrder(void)

Input Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

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

Theoretical Description

Known Issues

7.43 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 7.26: 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

7.44 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 7.27: 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

7.45 Message Processors

Header Files	:	message_processors_common_*.h
	:	message_processor_transmitter_*.h
	:	message_processor_receiver_*.h
Source Files	:	message_processors_common_*.cpp
	:	message_processor_transmitter_*.cpp
	:	message_processor_receiver_*.cpp
Version	:	20190410 (Andoni Santos)

These blocks are used to send and receive well defined sets of data in organized groups. They are meant to be used as a means of classically transmitting data between two parties. Currently these allow doing all the necessary classical communication needed to implement the BB84 protocol, transmitting discrete sets of data of arbitrary size and type at once.

The *MessageProcessorTransmitter* block is used to convert the data it receives into a Message structure, and send it on an appropriate Message signal. The *MessageProcessorReceiver*, on the other hand, is used to receive Messages from appropriate signals, convert from the structure into numeric data, and output the data into the appropriate signal, according to its type.

The *message_processors_common* files don't contain a particular block. Instead, they have auxiliary functions, variables and types which are useful working with messages and the message processors. Both the transmitter and receiver make use of these files, but other blocks may use resort to them in order to get access to useful functions.

Signals

Message Processor Transmitter

Number of Input Signals	6
Type of Input Signals	TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	1
Type of Output Signals	Message

Message Processor Receiver

Number of Input Signals	1
Type of Input Signals	Message
Number of Output Signals	6
Type of Output Signals	TimeDiscreteAmplitudeContinuousReal

Input Parameters

No input parameters.

Methods

Message Processor Transmitter

MessageProcessorReceiver(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig)

void initialize(void);

bool runBlock(void);

Message Processor Receiver

MessageProcessorTransmitter(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig)

void initialize(void);

bool runBlock(void);

Message Processor Common

t_message_data MessageProcessors::getMessageData(const t_message& msg, t_message_data_length dataLength);

string MessageProcessors::generateMessageData(const t_message_data &msgData, const t_message_type &msgType);

t_message_data_length MessageProcessors::getMessageDataLength(const t_message& msg);

t_message_type MessageProcessors::getMessageType(const t_message& msg);

bool MessageProcessors::sendToMsgProc(vector <t_integer>& data, Signal& signalToMsg, bool& started);

bool MessageProcessors::recvFromMsgProc(vector <t_integer>& data, Signal& signalToMsg, bool& started, int& numberofValuesToRecv);

Functional description

In the case that any of the signals in question is not required, a signal leading to nowhere may be used both in the transmitter and the receiver to bypass using that signal. That may allow using only later types of messages.

Message Processor Transmitter

When run, the *MessageProcessorTransmitter* verifies if there is anything in any of its input signals. If there is, it starts getting values $n + 1$ values from the buffer, where n is the first value it receives. If n is bigger than the buffer, it continues to retrieve values in subsequent runs until it has all the values. When it finally has all the values to send in the message, it builds the message, with its structure depending on which signal were the values retrieved from.

Each
message is a structure containing three *string* fields: *messageType*, *messageDataLength*, and *messageData*. The first defines the type of information that the message carries. The possible types are defined in the MessageProcessorsCommon header. The second indicates the data string length, and the third carries the data itself, in string form.

Afterwards, it sends the message through the output signals. Each time it is run, it does this for all the input signals, and can send as many messages at once as it has signals. In order for blocks to send data to the message processor, the method *MessageProcessors::sendToMsgProc* from the *MessageProcessorsCommon* can be used. This method provides the functionality to send $n + 1$ values to the buffer, where the first value is the total number of values to send. In addition, if the buffer it too small for all the values, it continues to do so after it is has more space, until it has sent all the relevant values.

At this moment, the signals index and type correspondence is fixed as follows:

- 0 - BasisReconciliationMsg
- 1 - ErrorCorrectionMsg
- 2 - ErrorCorrectionPermutationsMsg
- 3 - PrivacyAmplificationSeedsMsg
- 4 - ParameterEstimationSeedAndBitsMsg
- 5 - ToggleBERChangeMsg

Message Processor Transmitter

This block does the inverse work: it checks the input *Message* signal to see if any message has arrived. If it has, it retrieves it, identifies its type, extracts its information and outputs the data to the relevant signal, preceded by a total count of the number of values. Again, like in the *MessageProcessorTransmitter* block, the data can be read using the

MessageProcessors::recvFromMsgProc method from the *MessageProcessorsCommon* files. This allows receiving more data than the buffer can carry at any one time.

At this moment, the signals index and type correspondence is fixed as follows:

- 0 - BasisReconciliationMsg
- 1 - ErrorCorrectionMsg
- 2 - ErrorCorrectionPermutationsMsg
- 3 - PrivacyAmplificationSeedsMsg
- 4 - ParameterEstimationSeedAndBitsMsg
- 5 - ToggleBERChangeMsg

Current Message Types:

Currently, the following message types are implemented, each required by a block needed by the BB84 protocol:

- BasisReconciliationMsg - this messages carry the information for Alice and Bob to do Basis reconciliation. All the values are integers between 0 and 5.
- ErrorCorrectionMsg - this messages carry information for the cascade process. All the transmitted data is composed of integers between 0 and 5.
- ErrorCorrectionPermutationsMsg - this messages carry the seed for generating the random permutations used to scramble the data in sequential cascade iterations. It should contain a single large integer.
- PrivacyAmplificationSeedsMsg - this message carries seeds used for hashing the key during error correction. It should contain one or more large integers.
- ParameterEstimationSeedAndBitsMsg - this message carries the seeds for selecting random bits from the data in order to estimate the bit error rate. In addition, it also carries the values of said bits. It is composed of a single large integer followed by binary values.
- ToggleBERChangeMsg - this message signals to Alice when Bob updates the Cascade parameters based on a new BER estimation. It contains a single "1".

Suggestions for future improvement

Currently, the number and type of signals is fixed and cannot be easily changed adapted. In the future, this could be modified to allow adapting the message processors to situations other than the BB84 protocol. In order to do that, it should be possible to choose a given number of signals and define the desired type for each of them.

7.46 Mutual Information Estimator

Header File	:	mutual_information_estimator_20180723.h
Source File	:	mutual_information_estimator_20180723.cpp

This block estimates the mutual information between the input and output channel symbols X and Y, respectively. Each input signal x_j (j = 1,2,...,J) has a specific probability being possible calculate the entropy of the alphabet X, $H(X)$. The uncertainty of the observation regarding with the occurrence of the input symbol is measured by the entropy $H(X)$, which is maximum when all inputs have the same probability. Another concept important to estimate the mutual information is the conditional entropy $H(X|Y = y_k)$, which represents the uncertainty related with the channel input symbols X that stays after the observation of the output symbols Y. This way, the difference $H(X) - H(X|Y)$ is on average the amount of information gained by the observer with respect with the channel input based on the channel output symbol. This difference is called the mutual information:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^{K}\sum_{j=1}^{J} P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \tag{7.12}$$

where K corresponds to the number of possible output symbols and J corresponds to the number of possible input symbols.

This block uses the property 1.9.3 in **[Wesolowski09]** of Mutual information which tells that it can be determined using the formula:

$$I(X;Y) = H(Y) - H(Y|X). \tag{7.13}$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0 $P(X = 0)$ and assumes that the complementary of this probability corresponds to the probability of the input bit is 1 $P(X = 1)$. $P(X = 0)$ corresponds to the α probability calculated in this block. Furthermore, another probability of interest is p which corresponds to the error probability of the channel. Both α and p are estimated in this block.

In order to calculate the mutual information, from equation 7.31 we should calculate the conditional entropy $H(Y|X)$ and the entropy of the channel outputs $H(Y)$. First, lets calculate $H(Y|X)$:

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^{K}\sum_{j=1}^{J} P(y_k|x_j)P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) + \bar{\alpha}(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) \\ &= H(p), \end{aligned} \tag{7.14}$$

which means that the conditional entropy depends only on the channel properties and it does not depends on the channel input statistics. Now, to calculate the entropy of the channel output symbols we need to calculate the channel output probabilities, i.e $P(Y = 0)$ and $P(Y = 1)$.

$$\begin{aligned} P(Y=0) &= P(Y=0|X=0)P(X=0) + P(Y=0|X=1)P(X=1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \tag{7.15}$$

Since the output channel symbol only has two possible values (0 or 1),

$$\begin{aligned} P(Y=1) &= P(Y=1|X=1)P(X=1) + P(Y=1|X=0)P(X=0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y=0). \end{aligned} \tag{7.16}$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \tag{7.17}$$

and the mutual information should be calculated using the following formula:

$$I(X;Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \tag{7.18}$$

The upper and lower bounds, $\mathrm{I(X;Y)_{UB}}$ and $\mathrm{I(X;Y)_{LB}}$ respectively, are calculated using the method of Coppler-Pearson as described in section 7.11 for bit error rate calculation. This way, it returns the simplified expression:

$$\mathrm{I(X;Y)_{UB}} = \mathrm{I(X;Y)} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\mathrm{I(X;Y)}(1-\mathrm{I(X;Y)})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \mathrm{I(X;Y)}\right) z_{\alpha/2}^2 + (2 - \mathrm{I(X;Y)})\right] \tag{7.19}$$

$$\mathrm{I(X;Y)_{LB}} = \mathrm{I(X;Y)} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\mathrm{I(X;Y)}(1-\mathrm{I(X;Y)})} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \mathrm{I(X;Y)}\right) z_{\alpha/2}^2 - (1 + \mathrm{I(X;Y)})\right], \tag{7.20}$$

where $z_{\alpha/2}$ is the $100\left(1 - \frac{\alpha}{2}\right)$th percentile of a standard normal distribution and N_T the total number of bits used to calculate the mutual information.

Input Parameters

Name	Type	Default Value
m	integer	0
alpha_bounds	double	0.05

Methods

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

- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setConfidence(double P) { alpha = 1-P; }

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 0 if the two input samples are equal to each other and 1 if not. This block also outputs *.txt* files with a report of the estimated Mutual Information, as well as the error probability of the channel estimated p and the estimated probability of $X = 0$, α. Furthermore, the mutual information estimator block can output middle report files with size m set by the user using the method *setMidReportSize(int M)*, i.e the mutual information calculated uses m input symbols in its calculation. However, a final report is always outputted using all symbols transmitted.

The block receives two input binary strings, one with the sequence of input channel symbols and from this it calculates the probability of the input symbol is equals to 0, α, and other sequence with the output channel symbols and it compares the bit from this sequence with the correspondent bit from the first sequence (i.e the sequence with the input channel symbols). If the bit in the output channel symbol is different from the correspondent bit in the input channel symbol sequence, it counts as an error and the error probability of the channel is calculated based on the final number of errors, p. Both probabilities α and p allow the block to estimate the conditional entropy and the entropy of the output channel symbols, allowing the calculation of the mutual information.

7.47 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 7.28: 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 7.26.

Input Signals

Number : 1

Type : Binary (DiscreteTimeDiscreteAmplitude)

Figure 7.26: Constellation used to map the signal for m=4

Output Signals

Number : 2

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

Example

Figure 7.27: Example of the type of signal generated by this block for the initial binary signal 0100...

7.48 M-QAM Receiver

Header File	:	m_qam_receiver.h
Source File	:	m_qam_receiver.cpp
Version	:	20180815 (Manuel Neves)

This block simulates the reception and demodulation of an optical signal (which is the input signal of the system) and outputs a binary signal corresponding to the reconstructed transmitted bitstream. A simplified schematic representation of this block is shown in figure 7.28.

Figure 7.28: Simplified model of the MQAM receiver

It also optionally logs in a file the information about the system flow and stores all internal signals for posterior analysis.

Signals

Input Signals

Number: 2
Type: Optical

These two input signals are the optical signal modulated with the information, from the optical fiber channel, and the optical signal from the local oscillator of the receiver.

Output Signals

Number: 1
Type: Binary

The output signal is the binary sequence demodulated from the optical signal.

Input Parameters

Name	Type	Default Value	Description
logValue	bool	true	If true, the log file will be printed.
logFileName	string	"SuperBlock_MQamReceiver.txt"	Name of the log file.
signalsFolderName	string	"signals/SuperBlock_MQamReceiver"	Name of the directory where the internal signals are saved.

Note that, due to the big amount of parameters involved in this super block, here there's only mention of the input parameters that are strictly related with the M-QAM Receiver super block. To address input parameters related to a block included in the receiver, please refer to the documentation of such block.
You can however, by having a look on the available methods, infer which parameters can be controlled from the scope of the M-QAM Receiver.

Methods

Block Constructor

- **MQamReceiver**(initializer_list<Signal *> inputSig, initializer_list<Signal *> outputSig);

This block's constructor expects, as any *Block* object, an input of two vectors of Signal pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

Methods

Methods to start and run the block:

- void **initialize**(void);
  - Sets up the input and output signals and updates the parameters related to the super block.
- bool **runBlock**(void);

  - This method is responsible for iterating over the several blocks contained on the receiver and running them until the signal buffers are full or all data has been processed.

Methods to configure the Photodiodes:

- void **setPhotodiodesResponsivity**(t_real Responsivity);

Methods to configure the TI Amplifiers:

- void **setGain**(t_real gain);
- t_real **getGain**(void);
- void **setAmplifierInputNoisePowerSpectralDensity**(t_real NoiseSpectralDensity);
- t_real **getAmplifierInputNoisePowerSpectralDensity**(void);
- void **setTiAmplifierFilterType**(Filter fType);
- void **setTiAmplifierCutoffFrequency**(double ctfFreq);
- void **setTiAmplifierImpulseResponseTimeLength_symbolPeriods**(int irl);
- void **setElectricalFilterImpulseResponse**(vector<t_real> ir);
- void **setElectricalImpulseResponseFilename**(string fName);
- void **setElectricalSeeBeginningOfImpulseResponse** (bool sBeginningOfImpulse Response);
- double const **getElectricalSeeBeginningOfImpulseResponse**(void);

Methods to configure the General Noise:

- void **setNoiseSamplingPeriod**(t_real SamplingPeriod);
- void **setNoiseSymbolPeriod**(t_real nSymbolPeriod);

Methods to configure the Thermal Noise:

- void **setThermalNoiseSpectralDensity**(t_real NoiseSpectralDensity);
- void **setThermalNoisePower**(t_real NoiseSpectralDensity);
- void **setThermalConstantPower**(bool cp);
- void **setSeeds**(array<int, 2> noiseSeeds);
- void **setSeedType**(SeedType seedType);

Methods to configure the Pulse Shapers:

- void **setImpulseResponseTimeLength**(int impResponseTimeLength);
- void **setFilterType**(pulse_shapper_filter_type fType);
- void **setRollOffFactor**(double rOffFactor);
- void **usePassiveFilterMode**(bool pFilterMode);
- void **setRrcNormalizeEnergy**(bool ne);
- void **setMFImpulseResponseFilename**(string fName);
- void **setMFSeeBeginningOfImpulseResponse** (bool sBeginningOfImpulse Response);
- double const **getMFSeeBeginningOfImpulseResponse**(void);

Methods to configure the Samplers:

- void **setSamplesToSkip**(int sToSkip);

Methods to configure the Decoder:

- void **setIqAmplitudes**(vector<t_iqValues> iqAmplitudesValues);
- vector<t_iqValues> const **getIqAmplitudes**(void);

Examples

– Impact of the noise on the TI Amplifier on the BER:

We can observe that the BER decreases at an enlarging rate, as we augment the spectral density of the noise on the TI Amplifier. Giving us an estimate of the sensitivity of the receiver to thermal noise.

– Impact of the bandwidth of the TI Amplifier:

On the following graph we can see the impact of the bandwith of the electrical amplifier on the BER.
For the horizontal axes the used metric will be the ratio between the signal bandwidth and the TI Amplifier bandwidth, to better understand how these to relate, and the minimum requirements for a well suited electrical amplifier.
For the test, thermal noise will also be inserted, to also evaluate the impact of a too broad bandwidth.

We can see that, on the simulator, the bandwidth of the TI Amplifier only has a significant effect when it is smaller than 80% of the transmitted signal bandwidth. For

higher bandwidths of the TI Amplifier (higher than the signal bandwidth) the simulator demonstrates no increase in the BER.

Functional description

This block has an input of two optical signals, the received signal from the transmission channel and the signal from the local oscillator. These signals pass through a set of blocks, as it can be seen in figure7.29, and then it outputs one binary signal that corresponds to the decoded information transmitted in the input signal from the transmission channel.
With this block we needn't to manually make all the connections between the different blocks needed to implement an optical receiver. All we need to do is create an instance of the block and only modify the parameters whose default values are not in accordance with our needs. All the methods available to do so have already been presented above.
It is a super block of higher complexity than the M-QAM Transmitter, making it harder to give a general overview of the signal flow, thus it is important to emphasize that, for any more specific doubts the documentation of the singular blocks should be addressed.

Figure 7.29: Schematic representation of the block homodyne receiver.

The block Optical hybrid alongside with the photodetectors mix the oscillator signal with the input signal in order to convert a complex optical signal into two real electrical signals, these represent the in-phase and the quadrature components of the received signal.
We are now in the electrical domain and the signals are a function of the current in the photodetectors, but still working with very low amplitudes, thus we need to have an electrical transimpedance amplifier, that will increase the signal power. Then, noise is added to simulate thermal noise present in all electronics.
After having strong electrical signals, we need to sample them, for them to be processed by a digital signal processor.
Finally, the in-phase and quadrature signals, discrete in time but continuous in amplitude, are processed by a decoder, which translates pairs of amplitudes to a sequence of binary digits, using the maximum likelihood approach.

Open issues

- The decoder only works for constellations in which all points have the same distance from the origin (example: QPSK). This is due to the fact that the decoder only takes into consideration which point of the constellation is closer to the received coordinates, but not the amplitude of the received signal. This means the decoder doesn't work properly for 16-QAM signals, making in impossible to analyse BER for cases that don't comply with the condition of all points being at the same distance from the origin;
- The formula that is responsible for inserting the effect of the quantum noise on the photodetectors seems po be wrong, since the shot noise power is always much smaller than the signal power, even for values of 100dBm on the receiver's local oscillator;
- There are a lot of parameters from the blocks included in the Receiver super block that can't be acessed/modified through methods of the Receiver, making it hard to configure the various parameters available for each block that constitutes the Receiver block.

Future improvements

- Besides the binary signal, there could be an additional output signal that would easily allow us to observe the received constellation with the noise added by the receiver block;
- Correction of the several issues.

7.49 MQAM Transmitter

Header File	:	m_qam_transmitter.h
Source File	:	m_qam_transmitter.cpp
Contributors	:	Andoni Santos
	:	Manuel Neves
	:	Armando Pinto

This block receives a binary sequence and an optical signal from the local oscillator, with these, it generates a MQAM optical signal. A schematic representation of this block is shown in figure 7.30.

Figure 7.30: Basic configuration of the MQAM transmitter

Signals

Input Signals

Number: 2
Type: Binary and Optical

These two input signals are the binary sequence to be transmitted, from the binary source, and the optical signal, directly from the local oscillator of the transmitter.

Output Signals

Number: 1
Type: Optical

The output signal is an optical signal resultant from the modulation of the binary signal at the input with the optical signal from the local oscillator.

This block generates an optical signal (output signal 1 in figure 7.35). The binary signal generated in the internal block Binary Source (block B1 in figure 7.35) 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 7.35).

Figure 7.31: Schematic representation of the block MQAM transmitter.

Input Parameters

Name	Type	Default Value	Description
logValue	bool	true	If true, the log file will be printed.
logFileName	string	"SuperBlock_MQamTransmitter.txt"	Name of the log file.
signalsFolderName	string	"signals/SuperBlock_MQamTransmitter"	Name of the directory where the internal signals are saved.

Table 7.29: Parameters related to the super block

Name	Type	Default Value	Description
m	int	4	Cardinality.
iqAmplitudes	vector<vector<t_real>>	{{1,1}, {-1,1}, {1,-1}, {-1,-1}}	Amplitudes of the points of the constellation.
firstTime	bool	true	First time value on the block M-QAM Mapper.
numberOfSamplesPerSymbol	int	8	Number of samples per symbol.
impulseResponseTimeLength	int	16	Temporal length of the impulse response in multiples of the symbol period.
filterType	pulse_shapper_filter_type	RaisedCosine	Type of the shaping filter on the pulse shaper block.
rollOffFactor	double	0.9	Roll-off factor for the raised-cosine filter.
pulseWidth	double	5e-10	Width of the pulse.
passiveFilterMode	bool	false	When true the filter is passive, meaning the amplitudes are normalized.

Table 7.30: Configurable parameters related to the blocks inside the MQAM Transmitter

Block Constructor

- **MQamTransmitter**(initializer_list<Signal *> inputSig, initializer_list<Signal *> outputSig);

This block's constructor expects, as any *Block* object, an input of two vectors of Signal pointers, *InputSig* and *OutputSig*, containing the input/output signals described above.

Methods

Methods to start and run the block:

- void **initialize**(void);
  - Sets up the input and output signals and updates the parameters related to the super block.
- bool **runBlock**(void);
  - This method is responsible for iterating over the several blocks contained on the transmitter and running them until the signal buffers are full or all data has been processed.

Methods to access/modify system parameters:

- void **setM**(int mValue);
- void **setIqAmplitudes**(vector<vector<double>> iqAmplitudesValues);
- void **setFirstTime**(bool fTime);
- bool **getFirstTime**(void);
- void **setNumberOfSamplesPerSymbol**(int nSamplesPerSymbol);
- int const **getNumberOfSamplesPerSymbol**(void);
- void **setImpulseResponseTimeLength_symbolPeriods**(int impResponseTime Length);
- int const **getImpulseResponseTimeLength_symbolPeriods**(void);
- void **setFilterType**(pulse_shapper_filter_type fType);
- pulse_shapper_filter_type const **getFilterType**(void);
- void **setRollOffFactor**(double rOffFactor);
- double const **getRollOffFactor**(void);
- void **setPulseWidth**(double pWidth);

- double const **getPulseWidth**(void);
- void **setPassiveFilterMode**(bool pFilterMode);
- bool const **getPassiveFilterMode**(void);

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)

Examples

– Impact of the constellation coding on the BER:

For this example we will analyse how Grey encoding on the constellation optimizes the BER at the receiver in the presence of noise. To achieve that we will use the two following constellations:

Figure 7.32: Two different constellations for QPSK.

Even though they both seem randomly chosen, after testing these two constellations with all the other parameters constant (the same configuration of the system, apart from the constellation) we get a BER of 10.5% for the constellation on the left and a BER of 8% for the constellation on the right.
This happens because, in the presence of noise, the symbols can be decoded to adjacent symbols. And that obviously has an effect on the BER. What we can see here is how the Symbol Error Rate (SER) affects the BER. It's obvious that, once the only configuration that was changed was the constellation, the SER is constant for both cases.
So, what do these two constellations have different to cause different measured BER?
The difference is that, the constellation on the right has Grey encoding, meaning that adjacent symbols only differ in one bit. In this way, a misevaluated symbol will only result in one out of two bits wrong, thus minimizing the BER of a system.

– Inter-symbolic interference (ISI) for different pulse shapes:

For this example we will analyse the eye diagrams at the receiver side, after the matched receiver filter of the M-QAM Receiver.

(a) Eye diagram - Raised Cosine

(b) Eye diagram - Root Raised Cosine

Figure 7.33: Eye diagrams without noise

In the figure 7.33a we have the eye diagram of the received signal, whose pulse was shaped with a Raised Cosine and the matched filter at the receiver had the same filter shape. The result is not what we are expecting, because the Raised Cosine pulse shape is known for having null ISI, but we can clearly see that it is not null. This happens because, thanks to the matched filter, whose shape is the same as the pulse shaper, we actually get a Squared Raised Cosine shape, which does not present the same null ISI as the Raised Cosine impulse. To solve this problem we must instead apply both on the pulse shaper and on the matched filter a Root Raised Cosine filter, in such way that at the receiver we obtain the expected null ISI. The result obtained from this can be observed in figure 7.33b.
Even though that it is clear to see the differences between figures 7.33a and 7.33b, we have to keep in mind that these signals are ideal and have no noise. It is then of interest to observe if the difference is still noticeable when we have a noise figure with the same order of the ISI seen before.
To analyse that we can add noise to the both cases mentioned before, and analyse the differences between both pulse shapes on the eye diagram. The result of doing so can be seen in figures **??** and 7.34b.

(a) Eye diagram - Raised Cosine with noise

(b) Eye diagram - Root Raised Cosine with noise

Figure 7.34: Eye diagrams with noise

From this we can see that, in spite of one pulse shape having null ISI at the receiver and the other not having, when the noise is of a significant order we can't differentiate both types

of pulse shapes as one might expect.

Number: 1 optical and 1 binary (optional)

Functional description

The M-QAM Transmitter is a super block that contains all the necessary blocks to generate a modulated optical signal using only as an input the binary code to modulate and the optical signal from the local oscillator, as we can observe in figure 7.35.

Figure 7.35: Schematic representation of the block MQAM transmitter.

With this block we needn't to manually make all the connections between the different blocks needed to implement an optical transmitter (B1-6). All we need to do is create an instance of the block and only modify the parameters whose default values are not in accordance with our needs. All the methods available to do so have already been presented above.

On the next paragraphs, a general overview of the flow of the signal from the input to the output will be made (please note that this is only a simple high-level description, for a detailed analysis of each block you should resort to the respective's block documentation).

Firstly, the input signal passes through the M-QAM Mapper, which converts the groups of $\log_2 M$ consecutive bits into two streams of *TimeDiscreteAmplitudeContinuousReal* signals (MQAM 1 and 2), these signals have for amplitude the values inserted on the *iqAmplitudes* parameter, in such way that together, they map the binary sequence that originated them.

Then, these signals are converted to *TimeContinuousAmplitudeContinuousReal* signals, by up-sampling the previous signals of a factor given by *numberOfSamplesPerSymbol*, obtaining (MQAM 3 and 4).

Following, the pulse shaper block uses the previously configured pulse shape (through the input parameters) to turn the pulses into impulses, obtaining signals MQAM 5 and 6, that are the In-phase and Quadrature components used to modulate the optical signal from the input signal 2, henceforth obtaining the output signal 1.

Sugestions for future improvement

Open issues

- the default QPSK constellation isn't the same as the one set if you call the function *setM(4)*, which may originate problems concerning the compatibility with the transmitter;
- the block has an unused parameter called *firstTime*, which may cause confusion with the parameter from the M-QAM Mapper, which is accessed and modified through the methods *set/getFirstTime()*;
- setting a square pulse shape throws a runtime error;
- not all methods available at the pulse shaper block can be accessed from the transmitter block.

Future improvements

- add missing methods from the pulse shaper to the transmitter block;
- add methods to modify the settings of the IQ Modulator;
- it would be interesting to have a noise input at the pulse shaper, to be able to see the constellations with noise, because currently there's no easy way to observe this.

7.50 Netxpto

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

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

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

7.50.1 Version 20180118

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

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

7.50.2 Version 20180418

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

Sugestions for future improvement

7.51 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

7.52 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

7.53 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

$$\begin{aligned} X(\omega) &= 1 \quad \text{,if } \omega \in A \\ X(\omega) &= 0 \quad \text{,if } \omega \in \bar{A} \end{aligned} \tag{7.21}$$

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 $P(X = 0)$,

$$\begin{aligned} P(X = 1) = &\quad P(A) &= p \\ P(X = 0) = &\quad P(\bar{A}) &= 1 - p \end{aligned} \tag{7.22}$$

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

$$\frac{X - np}{\sqrt{np(1-p)}} \backsim N(0, 1). \tag{7.23}$$

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:

$$\frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} \backsim N(0, 1) \tag{7.24}$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{n\hat{p}(1-\hat{p})}} < z_{\alpha/2}\} \approx &\qquad 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})} < np - X < z_{\alpha/2}\sqrt{n\hat{p}(1-\hat{p})}\} \approx &\qquad 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1-\hat{p})/n}\} \approx &\qquad 1 - \alpha \end{aligned} \tag{7.25}$$

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:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \tag{7.26}$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \tag{7.27}$$

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

$$\text{ME} = \text{z}_{\text{score}} \times \sqrt{\frac{\hat{p}(1-\hat{p})}{N}} \tag{7.28}$$

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,

$$\begin{aligned} \text{ME} &= \pm z_{\alpha/2}\frac{\sigma}{\sqrt{N}} \\ \text{ME} &= \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{N}}, \end{aligned} \tag{7.29}$$

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

7.54 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

7.55 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

7.56 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

7.57 MS Windows IP Tunnel

Header File	:	ms_windows_ip_tunnel_∗.h
Source File	:	ms_windows_ip_tunnel_∗.cpp
Version	:	20180815 (João Coelho)

This block works in two ways: one way with one input and zero output signals (as client/sender) and the other one with zero input and one output signal (as server/receiver). MS Windows IP Tunnel is duplicated onto two machines; the first takes samples out of the input buffer until it's empty and transmits them to the IP Tunnel on the second machine through a TCP/IP connection. The IP Tunnel on the second machine receives the signal and outputs it to the output buffer. It's only necessary to specify the ip address of the remote machine and its TCP port. Only works on windows due to the Ws2tcpip.h header and ws2_32 library and it is compatible and tested with SDK v8.1 and v10.0.18362.0.

Input Parameters

Parameter	Type	Values	Default	Brief Description
displayNumberOfSamples	bool	true/false	true	If true, number of samples sent and received are displayed.
numberOfTrials	int	any	10	Number of trials after a connection has been refused.
remoteMachineIpAddress	string	any	"127.0.0.1"	IP Address of Remote Machine.
tcpPort	int	any	54000	TCP port used to connect to server.
timeIntervalSeconds	int	any	3	Time interval before trying to connect again after a connection has been refused (seconds).

Table 7.31: MS Windows IP Tunnel input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void terminate(void);

void ipTunnelSendInt(int space);

int ipTunnelRecvInt();

int ipTunnelPut(T object);

void setDisplayNumberOfSamples(bool opt);

bool getDisplayNumberOfSamples();

bool server();

bool client();

Functional Description

The IP Tunnel block transfers signals from one machine to another so it continues the simulation in two different computers. This block is duplicated onto two machines, one with only input (client) and other with only output signals (server). After being executed, the input signal's buffer (1) will be empty and this block will transmit this signal to the other block. The second block will output this signal to the output buffer (4). An architecture "Server - Client" is used to establish a TCP/IP channel between the two blocks (2 and 3). The block without input signal is the server (receiver) and the block with input signal is the client (transmitter). After the connection is established, server sends the "space" of its buffer (maximum signal size that can be received) and client responds with the "process" (signal size that is going to be transmitted) and with an integer representing the type of signal. Subsequently, the transmission of the signal starts and the simulation continues normally on both machines.

Figure 7.36: MS Windows IP Tunnel block

7.58 Mutual Information Estimator

Header File	:	mutual_information_estimator_20180723.h
Source File	:	mutual_information_estimator_20180723.cpp

This block estimates the mutual information between the input and output channel symbols X and Y, respectively. Each input signal x_j (j = 1,2,...,J) has a specific probability being possible calculate the entropy of the alphabet X, $H(X)$. The uncertainty of the observation regarding with the occurrence of the input symbol is measured by the entropy $H(X)$, which is maximum when all inputs have the same probability. Another concept important to estimate the mutual information is the conditional entropy $H(X|Y = y_k)$, which represents the uncertainty related with the channel input symbols X that stays after the observation of the output symbols Y. This way, the difference $H(X) - H(X|Y)$ is on average the amount of information gained by the observer with respect with the channel input based on the channel output symbol. This difference is called the mutual information:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^{K}\sum_{j=1}^{J} P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \tag{7.30}$$

where K corresponds to the number of possible output symbols and J corresponds to the number of possible input symbols.

This block uses the property 1.9.3 in **[Wesolowski09]** of Mutual information which tells that it can be determined using the formula:

$$I(X;Y) = H(Y) - H(Y|X). \tag{7.31}$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0 $P(X = 0)$ and assumes that the complementary of this probability corresponds to the probability of the input bit is 1 $P(X = 1)$. $P(X = 0)$ corresponds to the α probability calculated in this block. Furthermore, another probability of interest is p which corresponds to the error probability of the channel. Both α and p are estimated in this block.

In order to calculate the mutual information, from equation 7.31 we should calculate the conditional entropy $H(Y|X)$ and the entropy of the channel outputs $H(Y)$. First, lets calculate $H(Y|X)$:

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^{K}\sum_{j=1}^{J} P(y_k|x_j)P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) + \bar{\alpha}(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p}\log\frac{1}{\bar{p}} + p\log\frac{1}{p}) \\ &= H(p), \end{aligned} \tag{7.32}$$

which means that the conditional entropy depends only on the channel properties and it does not depends on the channel input statistics. Now, to calculate the entropy of the channel output symbols we need to calculate the channel output probabilities, i.e $P(Y = 0)$ and $P(Y = 1)$.

$$\begin{aligned} P(Y=0) &= P(Y=0|X=0)P(X=0) + P(Y=0|X=1)P(X=1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \tag{7.33}$$

Since the output channel symbol only has two possible values (0 or 1),

$$\begin{aligned} P(Y=1) &= P(Y=1|X=1)P(X=1) + P(Y=1|X=0)P(X=0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y=0). \end{aligned} \tag{7.34}$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \tag{7.35}$$

and the mutual information should be calculated using the following formula:

$$I(X;Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \tag{7.36}$$

The upper and lower bounds, $\mathrm{I(X;Y)_{UB}}$ and $\mathrm{I(X;Y)_{LB}}$ respectively, are calculated using the method of Coppler-Pearson as described in section 7.11 for bit error rate calculation. This way, it returns the simplified expression:

$$\mathrm{I(X;Y)_{UB}} = \mathrm{I(X;Y)} + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\mathrm{I(X;Y)(1-I(X;Y))}} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \mathrm{I(X;Y)}\right) z_{\alpha/2}^2 + (2 - \mathrm{I(X;Y)})\right] \tag{7.37}$$

$$\mathrm{I(X;Y)_{LB}} = \mathrm{I(X;Y)} - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{\mathrm{I(X;Y)(1-I(X;Y))}} + \frac{1}{3N_T}\left[2\left(\frac{1}{2} - \mathrm{I(X;Y)}\right) z_{\alpha/2}^2 - (1 + \mathrm{I(X;Y)})\right], \tag{7.38}$$

where $z_{\alpha/2}$ is the $100\left(1 - \frac{\alpha}{2}\right)$th percentile of a standard normal distribution and N_T the total number of bits used to calculate the mutual information.

Input Parameters

Name	Type	Default Value
m	integer	0
alpha_bounds	double	0.05

Methods

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

- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setConfidence(double P) { alpha = 1-P; }

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 0 if the two input samples are equal to each other and 1 if not. This block also outputs *.txt* files with a report of the estimated Mutual Information, as well as the error probability of the channel estimated p and the estimated probability of $X = 0$, α. Furthermore, the mutual information estimator block can output middle report files with size m set by the user using the method *setMidReportSize(int M)*, i.e the mutual information calculated uses m input symbols in its calculation. However, a final report is always outputted using all symbols transmitted.

The block receives two input binary strings, one with the sequence of input channel symbols and from this it calculates the probability of the input symbol is equals to 0, α, and other sequence with the output channel symbols and it compares the bit from this sequence with the correspondent bit from the first sequence (i.e the sequence with the input channel symbols). If the bit in the output channel symbol is different from the correspondent bit in the input channel symbol sequence, it counts as an error and the error probability of the channel is calculated based on the final number of errors, p. Both probabilities α and p allow the block to estimate the conditional entropy and the entropy of the output channel symbols, allowing the calculation of the mutual information.

7.59 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

7.60 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 7.37 shows a schematic representation of this block.

Figure 7.37: 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 7.32: 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

7.61 Parameter Estimation

Header Files	:	parameter_estimation_*.h
Source Files	:	parameter_estimation_*.cpp
Version	:	20190410 (Andoni Santos)

This block is used to estimate the error rate between two sets of data shared by two parties in Quantum Key Distribution, Alice and Bob. It does this by randomly sampling a predefined number of bits from the sifted key, exchanging and comparing them. The error rate of the exchanged bits provides an estimative of the overall error rate of the sifted key.

Signals

Number of Input Signals	2
Type of Input Signals	Binary, TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	3
Type of Output Signals	Binary, TimeDiscreteAmplitudeContinuousReal, TimeDiscreteAmplitudeContinuousReal

Table 7.33: Parameter estimation signals

Input Parameters

Parameter	Type	Values	Default
alpha	t_real	[0,1]	0.05
numberOfBits	t_integer	> 1	500
numberOfBitsRequiredToStart	t_integer	> numberOfBits	10500
threshold	t_real	[0,1]	0.12
bypassParameterEstimation	bool	[true, false]	false

Table 7.34: Parameter estimation input parameters

Methods

ParameterEstimation(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig)

ParameterEstimation(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig, string bFolderName)

ParameterEstimation(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig, string bFolderName, string bFileName)

ParameterEstimation(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig, string bFolderName, string bFileName, string bLFileName)

void initialize(void);

bool runBlock(void);

void setConfidence(double P)

double const getConfidence(void)

void setNumberOfBitsToUse(const unsigned int nb)

void setNumberOfBitsRequiredToStart(const unsigned int nbs)

void setBypassParameterEstimation(const bool bypass)

void setParameterEstimationRole(const parameter_estimation_role role)

void setThreshold(const t_real thr)

long int getInputBits(void)

long int getOutputBits(void)

t_real getAverageEstimatedBER(void)

Functional description

This block estimates the error rate between sifted keys held by two different parties, while sharing only a relatively small random subset of the keys. This keeps most of the data away from the communication channels. The small amount of shared data is discarded afterwards to maintain the privacy of the remaining data.

Let's assume that, for instance, in a Quantum Key Distribution protocol, there are two parties who want to generate a shared private key. These two parties are Alice, who transmits the data, and Bob, who receives the data. At a certain point, they both have sifted keys, where both have the data corresponding to the qubits which Bob correctly received and measured. However, the sifted keys are still not equal, as they are affected by transmission errors. These errors can be due to multiple reasons, the most important of which is attempted interference by a third party. Thus, it becomes necessary to estimate the error rate for two reasons:

1. Estimate the potential amount of data which has been intercepted, to find if it is possible to distill a secure key, and how much of the key may have been compromised;

2. Choose the parameters to efficiently correct the errors, minimizing the amount of data that will need to be discarded in the process.

To do this, some data needs to be shared between Alice and Bob, and that data will be compromised. Therefore, the amount of shared data should be minimized, as it will have to be discarded afterwards. This is currently done as follows.

The *ParameterEstimation* block receives the binary sifted keys in a queue. After Bob has accumulated enough bits, as defined by *numberOfBitsRequiredToStart*, it starts the parameter estimation process. First, it generates a random seed, which will be used to seed a random generator. This generator will randomly choose *numberOfBits* among the first *numberOfBitsRequiredToStart* bits in the queue. Then, it will send the seed and the bits themselves to Alice. Alice can then sample the same bits and compare them to Bob's. The ratio between the number of coincidences and *numberOfBits* provides an estimate to the overall error rate in the accumulated *numberOfBitsRequiredToStart*. Alice then sends the bits information to Bob, outputs the accumulated *numberOfBitsRequiredToStart* (except for the shared bits), and the remaining other signals it sends the upper bound of the BER confidence interval and the total number of remaining bits. Bob then receives the data sent by Alice, get its own estimate and outputs the same data.

Suggestions for future improvement

7.62 Photodiode pair

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

This block simulates a block of two photodiodes assembled like in figure 7.38. 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 7.38: 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 7.39: Example of the output singal of the photodiode block for a bunary sequence 01

Sugestions for future improvement

7.63 Photoelectron Generator

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

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

Figure 7.40: Schematic representation of the photoelectron generator code block

Theoretical description

The operation of a real photodiode is based on the photoelectric effect, which consists on the removal of one electron from the target material by a single photon, with a probability η. Given an input beam with an optical power $P(t)$ in which the photons are around the wavelength λ, the flux of photons $\phi(t)$ is calculated as **[saleh1991]**

$$\phi(t) = \frac{P(t)\lambda}{hc} \tag{7.39}$$

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

$$\bar{n}(t) = \int_t^{t+T} \phi(\tau)\, \mathrm{d}\tau \tag{7.40}$$

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

$$p\,(n) = \frac{\bar{n}^n \exp(-\bar{n})}{n!} \tag{7.41}$$

where $n = n(t)$ and $\bar{n} = \bar{n}(t)$.
For each incident photon, there is a probability η of generating a phototelectron. Therefore, we can model the generation of photoelectrons during this time interval, as a binomial process where the number of events is equal to the number of incident photons, $n(t)$, and the rate of success is η. If we combine the two random processes, binomial photoelectron generation after poissonian photon flux, the number of output photoelectrons in this time interval, $m(t)$, will follow **[saleh1991]**

$$m \sim \text{Poisson}\,(\eta\bar{n}) \tag{7.42}$$

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

Functional description

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

$$m(t) = \eta\overline{n(t)} \tag{7.43}$$

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

$$m(t) \sim \text{Poisson}\left(\eta\overline{n(t)}\right) \tag{7.44}$$

Input Parameters

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

Methods

PhotoelectronGenerator()

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

void initialize(void)

bool runBlock(void)

void setEfficiency(`t_real` efficiency)

void getEfficiency()

void setShotNoise(`bool` shotNoise)

void getShotNoise()

Input Signals

Number: 1

Type: Optical (OpticalSignal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal or TimeContinuousAmplitudeContinuousReal)

Examples

Figure 7.41: Constant power simulation setup

To test the output of this block, we recreated the results of figure $11.2 - 3$ in [**saleh1991**]. We started by simulating the constant optical power case, in which the local oscillator power was fixed to a constant value. Two power levels were tested, $P = 1\mu W$ and $P = 1nW$, using a sample period of 20 picoseconds and photoelectron generator efficiency of 1.0. The simulation code is in folder `lib \photoelectron_generator \photoelectron_generator_test_constant`. The following plots show the number of output electrons per sample when the shot noise is ignored or considered

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

Figure 7.42: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 7.42 shows clearly that turning the quantum noise on or off will produce a signal with or without variance, as predicted. If we compare this result with plot (a) in **[saleh1991]**, in particular $P = 1nW$, we see that they are in conformance, with a slight difference, where a sample has more than one photoelectron. In constrast with the reference result, where only single events are represented, the $P = 1\mu W$ case shows that all samples account many photoelectrons. Given it's input power, multiple photoelectron generation events will occur during the sample time window. Therefore, to recreate the reference result, we just need to reduce the sample period until the probability of generating more than 1 photoelectron per sample goes to 0.

Figure 7.43: Variable power simulation setup

To recreate plot (b) in **[saleh1991]**, a more complex setup was used, where a series of states are generated and shaped by a MQAM, creating a input electric field with time-varying power.
The simulation code is in folder `lib \photoelectron_generator \photoelectron_generator_test_variable`.

Figure 7.44: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 7.44 shows that the output without shot-noise is following the input power perfectly, apart from a constant factor. In the case with shot-noise, we see a that there are only output samples in high power input samples.
These results are not following strictly plot (b) in [**saleh1991**], because has already discussed previously, in the high power input samples, we have a great probability of generating many photoelectrons.

Sugestions for future improvement

7.64 Power Spectral Density Estimator

Header File	:	power_spectral_density_estimator_*.h
Source File	:	power_spectral_density_estimator_*.cpp
Version	:	20180704 (Andoni Santos)

Input Parameters

Name	Type	Default Value
method	enum	WelchPgram
ignoreInitialSamples	int	513
windowType	WindowType	Hann
measuredIntervalSize	int	2048
segmentSize	int	512
overlapPercent	double	0.5
overlapCount	int	256
alpha	double	0.05
tolerance	double	1e-6
filename	string	signals/powerSpectralDensity.txt
active	bool	false

Methods

PowerSpectralDensityEstimator() ;

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

void initialize(void);

bool runBlock(void);

void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

int getMeasuredIntervalSize(void) return measuredIntervalSize;

void setSegmentSize(int sz) segmentSize = sz;

int getSegmentSize(void) return segmentSize;

void setOverlapCount(int olp) overlapCount = olp; overlapPercent = overlapCount/segmentSize;

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olpP) overlapPercent = olpP; overlapCount = segmentSize*overlapPercent;

double getOverlapPercent(void) return overlapPercent;

void setConfidence(double P) alpha = 1-P;

double getConfidence(void) return 1 - alpha;

void setWindowType(WindowType wd) windowType = wd;

WindowType getWindowType(void) return windowType;

void setFilename(string fname) filename = fname;

string getFilename(void) return filename;

void setEstimatorMethod(PowerSpectralDensityEstimatorMethod mtd) method = mtd;

PowerSpectralDensityEstimatorMethod getEstimatorMethod(void) return method;

void setActivityState(bool state) active = state;

bool getActivityState(void) return active;

Input Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This block accepts one OpticalSignal or TimeContinuousAmplitudeContinousReal signal (or two real signals, an In-phase signal and a quadrature signal) and outputs an exact copy of the input signals to the output. As it receives the input signals, it saves until it has enough to perform a power spectral density estimation, in W/Hz. The number of samples required to perform this estimation is defined by the parameter `measuredIntervalSize`. After it has enough samples, it proceeds to estimating the power spectral density of the signal through the procedure chosen through the `method` parameter. Currently only one

Figure 7.45: Estimated power spectral density for a QPSK signal.

method is available, the Welch periodogram (`WelchPgram`). With this method, when enough samples are gathered, a segment of size `segmentSize` is chosen, windowed with a window chosen with `windowType` and a periodogram obtained from it. Another segment is then selected, with a certain number of samples overlapping with the previous one, as defined by `overlapCount` or `overlapPercent`. This process is repeated until the end of the measured interval, and the average of every periodogram obtained so far is the end result. This is then saved to the target file, specified by the `filename` parameter. This file is updated each time that the chosen number of samples is reached. As the simulation continues to run, more samples are acquired and as the estimation is improved.

The file contains data divided in different lines, separated by "\n", to simplify reading it with external programs. The first line identifies the file as a Power Spectral Density (PSD) estimate. The second line lists the number of segments averaged to obtain the estimated. The third line identifies the chosen confidence interval. The remaining lines are grouped in pairs, with the first line describing the contents of the next one. These pairs contain, in order, the center of the frequency bins, the Power Spectral Density estimate, and the corresponding upper and lower confidence bounds.

Theoretical Description

The techniques used for estimating the Power Spectral Density are divided in two, based on different definitions of power spectral density. These are the periodogram (direct method) and the correlogram (indirect method).

Periodogram

It is also possible to define the Power Spectral Density as follows [**jeruchim06**]:

$$S_{XX}(f) = \lim_{T\to\infty} E\left[\hat{S}_{XX}\left(f,T\right)\right] \tag{7.45}$$

with

$$\hat{S}_{XX}(f,T) = \frac{|X_T(f)|^2}{T} \tag{7.46}$$

and

$$X_T(f) = \int_0^T x(t)\exp(-j2\pi ft)dt \tag{7.47}$$

$tS_{XX}(f,T)$ presents a natural estimation for the Power Spectral Density, if the limiting and expectation operations are omitted. This is called a periodogram. Its discrete version is given by:

$$\tilde{P}\left(f\right) = \frac{1}{NT_s}|X_N(f)|^2 \tag{7.48}$$

with

$$X_N\left(f\right) = T_s\sum_{n=0}^{N-1} X(n)\exp\left(-j2\pi fnT_s\right) \tag{7.49}$$

$X_N(f)$ is the DFT of the observed data sequence, which can easily be determined through FFt calculation.

To obtain the periodogram, the data needs to windowed. Windowing implies a function that selects a certain portions of another function, either as is or transforming it in a certain way. Any finite observation is at least implicitly windowed with a rectangular window, as it is a smaller set of the original function. Windows can also modify the data in a given desired way. Any window necessarily alters the true spectrum. The expected value of the periodograms is therefore different from the true spectrum, instead providing a biased estimate. However, the periodogram is asymptotically unbiased. This is because the effects of windowing, which are the origin of the bias, disappear as the number of samples increases. For any window $W(f)$, including the rectangular window, $|W(f)^2| \to \delta(f)$ as the number of observations tends to infinity.

Also, for any periodogram, the standard deviation is at least as large as the expected value of the spectrum, independently of the size of the observation.

Most common methods for obtaining periodograms try to avoid this issue. One operation they resort to is the averaging of periodograms. This simply means to obtain the average periodogram from a given set of periodograms obtained from different signal sequences. The averaging of periodograms turn them into a consistent estimator. In order to average the periodograms, the signal is divided into smaller segments through windowing and periodograms are obtained from different segments, in order to be averaged.

The two most common methods are the Bartlett periodogram and the Welch periodogram. The Bartlett periodogram finds the average of nonoverlapping signal

segments obtained with a rectangular window. The Welch periodogram is calculated by using segments with a certain amount of overlap, which are obtained with other windows, typically Hann or Hamming.

Confidence intervals

The periodogram is Chi-Square distributed [**jeruchim06**]. Therefore, the confidence interval must be calculated accordingly. The confidence interval for a an estimated Power Spectral Density value $\hat{S}(f)$ is defined by [**nsapplication255**]

$$P\left(A < S(f) < B\right) = 1 - \alpha$$

where A and B are the bounds of the confidence interval, and α is the probability of the true Power Spectral Density being outside the confidence interval. For a given number of degrees of freedom, the interval's bounds are given by

$$A = \frac{\nu \hat{S}(f)}{\chi^2_{1-\alpha/2}}$$

$$B = \frac{\nu \hat{S}(f)}{\chi^2_{\alpha/2}}$$

$\chi^2_{\alpha/2}$ can be obtained by finding the values at which the chi-squared cumulative distribution function, for ν degrees of freedom, equals the desired probabilities. This function is calculated as [**weisstein18csd**]

$$\mathrm{CDF}(\chi^2) = \frac{\gamma\left(\nu/2, \chi^2/2\right)}{\Gamma\left(\nu/2\right)} \tag{7.50}$$

where $\gamma\left(\nu/2, \chi^2 i/2\right)$ is an incomplete gamma function and $\Gamma(\nu/2)$ is the gamma function. The values of χ^2 needed for calculating A and B can be found by numerically evaluating 7.50 to find the values which fit the desired probabilities. This is currently done using the bissection method, which provides an approximation better than most available tables ($\frac{|\alpha_{\text{calc}} - \alpha_{\text{desired}}|}{\alpha_{\text{desired}}} \leq 10^{-6}$) in approximately 20 iterations.

The confidence bounds are then calculated using equations 7.64 and 7.64.

Known Issues

The current implementation for obtaining confidence intervals is susceptible to underflow/overflow errors if the degrees of freedom grow too much, due to the calculation of the incomplete gamma functions. This can mostly be avoided by making sure that the segment size for calculating the periodogram is, at least, approximately 1/1000th of the total number of samples.

$$\text{Segment_size} \geq \frac{\text{NumberOfBits} \times \text{SamplesPerSymbol}}{1000 \times \text{BitsPerSymbol}}$$

In order to completely correct this issue, two possibilities exist:

1. Improve the algorithm in order to overcome the overflow limitations, by using increased precision numeric types or some other method;

2. By using a gaussian approximation for the distribution of the periodogram, which is valid for larger degrees of freedom [**jeruchim06**].

7.65 Privacy amplification

Header Files	:	privacy_amplification_*.h
Source Files	:	privacy_amplification_*.cpp
Version	:	20190410 (Andoni Santos)

This block implements privacy amplification by hashing segments of the key, and discarding bits from the corrected key, eliminating the residual knowledge that an attacker may still have at the end of the error correction process. It can share a seed between the parties to allow randomly choosing a common hash family, and may be easily adapted to use a different hash function.

Signals

If paRole is seedTransmitter:

Number of Input Signals	2
Type of Input Signals	Binary, TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	2
Type of Output Signals	Binary, TimeDiscreteAmplitudeContinuousReal

Table 7.35: Privacy amplification signals - Seed Tx

If paRole is seedReceiver:

Number of Input Signals	3
Type of Input Signals	Binary, TimeDiscreteAmplitudeContinuousReal, TimeDiscreteAmplitudeContinuousReal
Number of Output Signals	1
Type of Output Signals	Binary

Table 7.36: Privacy amplification signals - Seed Rx

Input Parameters

Methods

PrivacyAmplification(initializer_list<Signal*> InputSig, initializer_list<Signal*> OutputSig) : Block(InputSig, OutputSig) {};

initialize(void);

runBlock(void);

Parameter	Type	Values	Default
ber	t_real	[0,1]	0.03
paRole	privacy_amplification_role paRole	{SeedTransmitter, SeedReceiver, Undefined}	Undefined
securityParameter	int	Any natural number	1
bypassHashing	bool	true or false	false

Table 7.37: Privacy amplification input parameters

setRole(privacy_amplification_role role)

setBER(t_real berValue)

setSystematicSecurityParameter(t_integer sp)

setBypassHashing(bool mrb)

getInputBits(void)

getOutputBits(void)

Theoretical introduction

As mentioned at the beginning, this block is used to amplify privacy on a quantum key distribution protocol. For instance, after the error correction part of the BB84 protocol, both parties share equal keys. During the previous steps of the key distillation process, information about the key was shared in a public classical channel, but all the shared information has been discarded, and thus, rendered useless for an attacker. However, the attacker may still have residual information about the key, acquired during the transmission stage. In addition, it may be desirable not to discard the bits during error correction, and deal with privacy issues all in one place.

To understand why a third party (lets call her Eve) may still have information, recall that they may attempt to intercept the qubits sent by Alice. Let us assume an intercept-resend arrack. When Eve tries to intercept a certain Qubit, she sends to Bob the value in the basis she used.

In addition, let us assume that Bob has estimated the QBER to be equal to a value z, and that transmission is perfect. This means that there is no source of error other and Eve, and thus every error in the received key is explicitly due to Eve's interference.

There are four possibilities:

1. Eve uses the correct basis, and Bob also uses the correct basis. *The bit is not discarded, and Eve knows it deterministically.*

2. Eve uses the correct basis, and Bob uses the wrong basis. The bit is discarded, and no-one gains anything.

3. Eve uses the wrong basis, and Bob uses the wrong basis. The bit is discarded, and no-one gains anything.

4. Eve uses the wrong basis, and Bob uses the correct basis. *The bit is not discarded, and it has 50% chance to introduce and error. Alice gains no information from this exchange, other than that she used the wrong basis.*

This means that the deterministic knowledge that Eve has on the final key should be at most twice the QBER. Typically, due to the existence of other sources of error, the QBER is actually higher than half of Eve's knowledge.

Therefore, the rate of bits she might know is given by the number of times she measured correctly, which is twice the number of times she introduced an error, $2z$.

It is important to emphasize that this analysis is based on only one type of attack. Nevertheless, it is enough to understand the issue. The important fact is that the information that Eve has is related to the estimated QBER.

One way to remove that information is to use a compression function, which transforms a key of size n into a key of size $n - t - s$, where t is Eve's information, and s is a security parameter. This transformation needs to be random, uniform and unique. One way to do this is with universal hash functions.

Functional description

At this point it is assumed that Alice and Bob share equal keys. Any error will be propagated, as we will explain ahead. Therefore, it is crucial that the error correction is configured with parameters strict enough to avoid failure.

The way this block currently works is that it uses a predefined size between Alice and Bob, which will be used to hash key segments. By always using segments with the same size, Alice and Bob are always synchronized and keep equal keys after privacy amplification.

A seed is generated and shared by the Transmitter. That seed is used to choose from a universal family of hash functions. The function is then used to hash the chosen key segment, ending up with a bit string of the same size. If there is a single bit error in the chosen segment, the hashed string will be completely different, and will likely have a 50% error rate (provided that the hash function is appropriate). We then remove $t + n$ bits from the hashed bit string, and output the value.

When the receiver block receives the seed it replicates the process, ending up with the same hashed string (assuming no error is present).

Suggestions for future improvement

7.66 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 7.38: 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 7.46: Example of a signal generated by this block for the initial binary signal 0100...

Sugestions for future improvement

Include other types of filters.

7.67 Quantizer

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

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

Input Parameters

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

Table 7.39: Quantizer input parameters

Methods

Quantizer() ;

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod) samplingPeriod = sPeriod;

void setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;

void setResolution(double nbits) resolution = nbits;

double getResolution() return resolution;

void setMinValue(double maxvalue) maxValue = maxvalue;

double getMinValue() return maxValue;

void setMaxValue(double maxvalue) maxValue = maxvalue;

double getMaxValue() return maxValue;

Functional description

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

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

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.68 Quantum Channel Emulator

Header Files	:	quantum_channel_emulator_*.h
Source Files	:	quantum_channel_emulator_*.cpp
Version	:	20190410 (Andoni Santos)

The Quantum Channel Emulator block tries to imitate the possible outcomes of the discrete variable quantum channel used in the laboratory. There is another more complex model of this system. However, the additional complexity makes it difficult to test and troubleshoot the software to be used with it. Therefore, this block was created. It is able to simulate the conditions existent on the complex system conditions in a stationary manner.This way, it produces similar results, while being much faster and simpler to interact with.

Signals

Number of Input Signals	4
Type of Input Signals	Binary, Binary, Binary, Binary
Number of Output Signals	3
Type of Output Signals	TimeContinuousAmplitudeDiscreteReal, Binary,TimeContinuousAmplitudeContinuousReal

Table 7.40: Quantum Channel Emulator signals

Input Parameters

Parameter	Type	Values	Default
doubleClickNumber	t_integer	any	2
noClickNumber	t_integer	any	3
~~probabilityOfError~~	t_real	[0,1]	0
probabilityOf~~Both~~Click	t_real	[0,1]	0
probabilityOfNoClick	t_real	[0,1]	0
seed	t_integer	any	~~971931~~

Table 7.41: Binary source input parameters

Methods

QuantumChannelEmulator(initializer_list<Signal *> InputSig, initializer_list<Signal *> OutputSig) : Block(InputSig, OutputSig) {};

void initialize(void);

bool runBlock(void);

initialize(void);

runBlock(void);

setProbabilityOfBothClick(t_real pbc)

setProbabilityOfNoClick(t_real pnc)

setProbabilityOfError(t_real pe)

setNoClickNumber(t_integer ncn)

setDoubleClickNumber(t_integer dcn)

Functional description

This block receives three main input signals: the key provided by Alice, the basis chosen by Alice to encode the key, and the basis chosen by Bob to decode the key.

The process to determine the output of the Bob's measurement is as follows:

1. A random number between zero and one, *randomRollClicks*, is drawn. If that number is lower than *probabilityOfNoClick*, a *noClickNumber* is output;
2. else, if that number is lower than *probabilityOfNoClick* + *probabilityOfDoubleClick*, a *doubleClickNumber* is output;
3. If none of the above happens, another random number between zero and one, *randomRollError*, is drawn. If the number is lower than *probabilityOfError*, the wrong bit is put in the buffer, indicating an error (independently of the measurement basis);
4. If the *randomRollError* is higher than the *probabilityOfError*, then the outcome depends on whether Alice's and Bob's basis are equal. If they are, the correct bit is output;
5. If the basis are different, a random number *randomRollErrorFromWrongBasis* between zero and one is generated. If the value is lower than 0.5, the correct bit is output. Otherwise, the wrong bit is output.

In addition, the third input signal is copied to the second output signal, at the same rate as the key. The third output signal currently is a placeholder and produces no output.

Suggestions for future improvement

7.69 Resample

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

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

Input Parameters

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

Table 7.42: Resample input parameters

Methods

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

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

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

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

Functional description

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

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

Input Signals

Number: 1

Output Signals

Number: 1

Type: Electrical complex signal

Examples

Sugestions for future improvement

7.70 SNR Estimator

Header File	:	snr_estimator_*.h
Source File	:	snr_estimator_*.cpp
Version	:	20180227 (Andoni Santos)

Input Parameters

Name	Type	Value
Confidence	double	0.95
measuredIntervalSize	int	1024
windowType	WindowType	Hamming
segmentSize	int	512
overlapCount	int	256
active	bool	false

Methods

SNREstimator() ;

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

void initialize(void);

bool runBlock(void);

void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

int getMeasuredIntervalSize(void) return measuredIntervalSize;

void setSegmentSize(int sz) segmentSize = sz;

int getSegmentSize(void) return segmentSize;

void setOverlapCount(int olp) overlapCount = olp;

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olp) overlapCount = floor(segmentSize*olp);

double getOverlapPercent(void) return overlapCount/segmentSize;

void setConfidence(double P) alpha = 1-P;

double getConfidence(void) return 1 - alpha;

void setWindowType(WindowType wd) windowType = wd;

WindowType getWindowType(void) return windowType;

void setFilename(string fname) filename = fname;

string getFilename(void) return filename;

vector<complex<double» fftshift(vector<complex<double» &vec);

void setRollOff(double rollOff) rollOffComp = rollOff;

double getRollOff(void) return rollOffComp;

void setNoiseBw(double nBw) noiseBw = nBw;

double getNoiseBw(void) return noiseBw;

void setEstimatorMethod(SNREstimatorMethod mtd) method = mtd;

SNREstimatorMethod getEstimatorMethod(void) return method;

void setIgnoreInitialSamples(int iis) ignoreInitialSamples = iis;

int getIgnoreInitialSamples(void) return ignoreInitialSamples;

void setActivityState(bool state) active = state;

bool getActivityState(void) return active;

Input Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This accepts one OpticalSignal or TimeContinuousAmplitudeContinousReal signal (or two, an In-phase signal and a quadrature signal), estimates the signal-to-noise ratio for a given time interval and outputs an exact copy of the input signal. The estimated SNR value is printed to *cout* after each calculation. The block also writes a *.txt* file reporting the estimated signal-to-noise ratio, a count of the number of measurements and the corresponding bounds for a given confidence level, based on measurements made on consecutive signal segments.

Theoretical Description

This block can use on of several methods to estimate the signal's SNR. Any of them can be usefull for particular use cases.

Currently there are three methods implemented:

- **powerSpectrum** - Spectral analysis [**harris12**, **xiao10**, **kashefi12**];
- **m2m4** - Moments method [**matzner93**];
- **ren** - Modified moments method [**ren05**];

Spectral analysis

Several methods exist for SNR estimation, most of them requiring prior knowledge of the sample time or of the signal's conditions. Spectrum-based SNR estimators have been shown to provide accurate results even these informations are not available, at the cost of being less efficient [**harris12**, **xiao10**, **kashefi12**].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [**john2007digital**]. This averages the values in each bin, providing a smoother estimate of the power spectrum. Using the power spectrum obtained from this sequence, the frequency interval containing the signal is estimated from the sampling rate, symbol rate and modulation type. The power contained in this interval will include both the signal and the white noise in those frequencies.

The power of a signal can be calculated from its power spectrum [**john2007digital**]:

$$P = \frac{1}{N}\sum_{n=0}^{N-1}|x(n)|^2 = \frac{1}{N^2}\sum_{k=0}^{N-1}|X(k)|^2 \tag{7.51}$$

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the the spectrum in the area without signal. This is the used to calculate the total noise power. The signal power is obtained by summing the FFT bins within the signal's frequency interval and subtracting the corresponding superimposed noise, according to what was previously estimated. The power SNR is the ratio between these two values.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \tag{7.52}$$

This
value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [**tranter2004principles**]. The SNR value and confidence interval are then saved to a text file.

Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal wo estimate the SNR [**matzner93**].

Let the second and fourth order moments be M_2 and M_4. In addition, let S_k be the sampled signal, with S_{I_k} and S_{Q_k} as its in-phase and quadrature components.

$$M_2 = E\{|S_k|^2\} = E\{S_{i_k}^2 + S_{Q_k}^2\} = P_S + P_N \tag{7.53}$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{i_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4 P_S P_N + k_n P_N^2 \tag{7.54}$$

Here, k_e and k_n are constants depending on the properties of the probability density function of the signal and noise processes. In practice, they are known from the modulation scheme used and the noise characteristics. For signals with constant energy, for instance, $k_e = 1$, and for two dimensional gaussian noise $k_n = 2$. It follows that the signal and noise powers can be defined as follows:

$$P_S = \sqrt{2M_2^2 - M_4} \tag{7.55}$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \tag{7.56}$$

and therefore

$$\text{SNR} = \frac{\sqrt{2EM_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \tag{7.57}$$

Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [**ren05**].

$$M_4' = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2 P_S P_N + P_N^2 \tag{7.58}$$

Thus, P_S ,P_N and the SNR become:

$$P'_S = \sqrt{M_2^2 - M'_4} \tag{7.59}$$

$$P'_N = M_2\sqrt{M_2^2 - M'_4} \tag{7.60}$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M'_4}}{M_2 - \sqrt{M_2^2 - M'_4}} \tag{7.61}$$

Known Issues

This block currently only works with either one or two *TimeContinuousAmplitudeContinuousReal* signals, depending on the chosen method. It has also only been tested with the in-phase and quadrature components of an MQAM signal with M=4. If the noise's power spectral density is high enough for the signal not to be detected, the block will fail (in the mentioned case, typically values of SNR < -10dB can be unreliable).

7.71 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 7.43: 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 7.47: Example of the output signal of the sampler

Sugestions for future improvement

7.72 SNR of the Photoelectron Generator

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

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

Figure 7.48: Schematic representation of the SNR of the Photoelectron Generator code block

Theoretical description

The input of this block is a stream of samples, y_j, each one of them corresponding to a number of photoelectrons generated in a time interval Δt. These photoelectrons are usually the output of a photodiode (photoelectron generator). To calculate the SNR of this stream, we will use the definition used in **[saleh1991]**

$$\text{SNR} = \frac{\bar{n}^2}{\sigma_n^2} \tag{7.62}$$

in which $\bar{n}$ is the mean value and σ_n^2 is the variance of the photon number in a given time interval T.
To apply this definition to our input stream, we start by separate it's samples in contiguous time windows with duration T. Each time window k is defined as the time interval $[kT, (k+1)T[$. To estimate the SNR for each time window, we will use the following estimators for the mean, μ_k, and variance, s_k^2 **[smith1997]**

$$\mu_k = \langle y \rangle_k \qquad\qquad s_k^2 = \frac{N}{N-1}\left(\langle y^2 \rangle_k - \langle y \rangle_k^2\right) \tag{7.63}$$

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

$$\langle y^n \rangle_k = \frac{1}{N_k} \sum_{j=j_{\text{min}}(k)}^{j_{\text{max}}(k)} y_j^n \tag{7.64}$$

in which

$$j_{min}(k) = \lceil t_k/\Delta t \rceil \tag{7.65}$$
$$j_{max}(k) = \lceil t_{k+1}/\Delta t \rceil - 1 \tag{7.66}$$
$$N_k = j_{max}(k) - j_{min}(k) + 1 \tag{7.67}$$
$$t_k = kT \tag{7.68}$$

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

$$S_1(k) = N_k \langle y \rangle_k \tag{7.69}$$
$$S_2(k) = N_k \langle y^2 \rangle_k \tag{7.70}$$

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

$$\mu_k = \frac{S_1(k)}{N_k} \qquad s_k^2 = \frac{1}{N_k - 1}\left(S_2(k) - \frac{1}{N_k}(S_1(k))^2\right) \tag{7.71}$$

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

$$\text{SNR}_k = \frac{\mu_k^2}{\sigma_k^2} = \frac{N_k - 1}{N_k}\frac{(S_1(k))^2}{N_k S_2(k) - (S_1(k))^2} \tag{7.72}$$

One particularly important case is the phototelectron stream resulting from the conversion of a laser photon stream by a photodiode (photoelectron generator). The resulting SNR will be **[saleh1991]**

$$\text{SNR} = \eta \bar{n} \tag{7.73}$$

in which η is the photodiode quantum efficiency.

Functional description

This block is designed to operate in time windows, dividing the input stream in contiguous sets of samples with a duration `tWindow` $= T$. For each time window, the general process consists in accumulating the input sample values and the square of the input sample values, and calculating the SNR of the time window based on these two variables.
To process this accumulation, the block uses two state variables, `aux_sum1` and `aux_sum2`, which hold the accumulation of the sample values and accumulation of the square of sample values, respectively.
The block starts by calculating the number of samples it has to process for the current time window, using equations 7.66, 7.67 and 7.68. If the duration of `tWindow` is 0, then we assume that this time window has infinite time (infinite samples). The values of `aux_sum1` and `aux_sum2` are set to 0, and the processing of the samples of current window begins.
After processing all the samples of the time window, we obtain $S1(k)$ and $S2(k)$ from the

state variables as $S1(k) =$ `aux_sum1` and $S_2(k) =$ `aux_sum2`, and proceed to the calculation of the SNR$_k$, using equation 7.72.
If the simulation ends before reaching the end of the current time window, we calculate the SNR$_k$, using the current values of `aux_sum1`, `aux_sum2` for $S_1(k)$ and $S_2(k)$, and the number of samples already processed, `currentWindowSample`, for N_k.

Input Parameters

Parameter	Default Value	Description
windowTime	0	SNR time window.

Methods

SnrPhotoelectronGenerator()

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

void initialize(void)

bool runBlock(void)

void setTimeWindow(`t_real` timeWindow)

Input Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeDiscreteAmplitudeContinuousReal)

Examples

Figure 7.49: Simulation setup

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

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

Figure 7.50: Theoretical and simulated results of the average SNR, for two photodiode efficiencies.

The plots in 7.50 show the comparison between the theoretical result 7.73 and the simulation results. We see that for low values of T, the average SNR shows a systematic deviation form the theoretical resul, but for $T > 10^{-6}s$ (10000 samples per time window), the simulation result shows a very good agreement with the theoretical result.

The simulations also show a lack of average SNR results when low power, low efficiency and small time window are combined (see plot 7.50a). This is because in those conditions, the probability of having a time windows with no photoelectrons, creating a invalid SNR, is very high, which will prevent the calculation of the average SNR.
We can estimate the probability of calculating a valid SNR average by calculating the probability of no time window having 0 phototelectrons, $p_{ave} = (1-q)^M$, in which q is the probability of a time window having all it's samples equal to 0 and M is the number of time windows. We know that the input stream follows a Poisson distribution with mean $\bar{m}$, therefore $q = (\exp(-\bar{m}))^N$, in which $\bar{m} = \eta P\lambda/hc$ and $N = T/\Delta t$, is the average number of samples per time window. Using this result, we obtain the probability of calculating a valid SNR average as

$$p_{ave} = (1 - \exp(-N\bar{m}))^M \tag{7.74}$$

Block problems

Future work

The block could also output a confidence interval for the calculated SNR. Given that the output of the Photoelectron Generator follows a Poissonian distribuition when the shot noise is on, the article "Confidence intervals for signal to noise ratio of a Poisson distribution" by Florence George and B.M. Kibria [**george2011**], could be used as a reference to implement such feature.

7.73 Sink

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

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

Input Parameters

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

Table 7.44: Sink input parameters

Methods

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

bool runBlock(void)

void setAsciiFilePath(string newName)

string getAsciiFilePath()

void setNumberOfBitsToSkipBeforeSave(long int newValue)

long int getNumberOfBitsToSkipBeforeSave()

void setNumberOfBytesToSaveInFile(long int newValue)

long int getNumberOfBytesToSaveInFile()

void setNumberOfSamples(long int nOfSamples)

long int getNumberOfSamples const()

void setDisplayNumberOfSamples(bool opt)

bool getDisplayNumberOfSamples const()

Functional Description

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

7.74 SNR Estimator

Header File	:	snr_estimator_*.h
Source File	:	snr_estimator_*.cpp
Version	:	20180227 (Andoni Santos)

Input Parameters

Name	Type	Value
Confidence	double	0.95
measuredIntervalSize	int	1024
windowType	WindowType	Hamming
segmentSize	int	512
overlapCount	int	256
active	bool	false

Methods

SNREstimator() ;

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

void initialize(void);

bool runBlock(void);

void setMeasuredIntervalSize(int misz) measuredIntervalSize= misz;

int getMeasuredIntervalSize(void) return measuredIntervalSize;

void setSegmentSize(int sz) segmentSize = sz;

int getSegmentSize(void) return segmentSize;

void setOverlapCount(int olp) overlapCount = olp;

int getOverlapCount(void) return overlapCount;

void setOverlapPercent(double olp) overlapCount = floor(segmentSize*olp);

double getOverlapPercent(void) return overlapCount/segmentSize;

void setConfidence(double P) alpha = 1-P;

double getConfidence(void) return 1 - alpha;

void setWindowType(WindowType wd) windowType = wd;

WindowType getWindowType(void) return windowType;

void setFilename(string fname) filename = fname;

string getFilename(void) return filename;

vector<complex<double» fftshift(vector<complex<double» &vec);

void setRollOff(double rollOff) rollOffComp = rollOff;

double getRollOff(void) return rollOffComp;

void setNoiseBw(double nBw) noiseBw = nBw;

double getNoiseBw(void) return noiseBw;

void setEstimatorMethod(SNREstimatorMethod mtd) method = mtd;

SNREstimatorMethod getEstimatorMethod(void) return method;

void setIgnoreInitialSamples(int iis) ignoreInitialSamples = iis;

int getIgnoreInitialSamples(void) return ignoreInitialSamples;

void setActivityState(bool state) active = state;

bool getActivityState(void) return active;

Input Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Output Signals

Number: 1 or 2
Type: OpticalSignal or TimeContinuousAmplitudeContinuousReal

Functional Description

This accepts one OpticalSignal or TimeContinuousAmplitudeContinousReal signal (or two, an In-phase signal and a quadrature signal), estimates the signal-to-noise ratio for a given time interval and outputs an exact copy of the input signal. The estimated SNR value is printed to *cout* after each calculation. The block also writes a *.txt* file reporting the estimated signal-to-noise ratio, a count of the number of measurements and the corresponding bounds for a given confidence level, based on measurements made on consecutive signal segments.

Theoretical Description

This block can use on of several methods to estimate the signal's SNR. Any of them can be usefull for particular use cases.

Currently there are three methods implemented:

- **powerSpectrum** - Spectral analysis [**harris12**, **xiao10**, **kashefi12**];
- **m2m4** - Moments method [**matzner93**];
- **ren** - Modified moments method [**ren05**];

Spectral analysis

Several methods exist for SNR estimation, most of them requiring prior knowledge of the sample time or of the signal's conditions. Spectrum-based SNR estimators have been shown to provide accurate results even these informations are not available, at the cost of being less efficient [**harris12**, **xiao10**, **kashefi12**].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [**john2007digital**]. This averages the values in each bin, providing a smoother estimate of the power spectrum. Using the power spectrum obtained from this sequence, the frequency interval containing the signal is estimated from the sampling rate, symbol rate and modulation type. The power contained in this interval will include both the signal and the white noise in those frequencies.

The power of a signal can be calculated from its power spectrum [**john2007digital**]:

$$P = \frac{1}{N}\sum_{n=0}^{N-1}|x(n)|^2 = \frac{1}{N^2}\sum_{k=0}^{N-1}|X(k)|^2 \tag{7.75}$$

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the the spectrum in the area without signal. This is the used to calculate the total noise power. The signal power is obtained by summing the FFT bins within the signal's frequency interval and subtracting the corresponding superimposed noise, according to what was previously estimated. The power SNR is the ratio between these two values.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \tag{7.76}$$

This
value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [**tranter2004principles**]. The SNR value and confidence interval are then saved to a text file.

Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal wo estimate the SNR [**matzner93**].

Let the second and fourth order moments be M_2 and M_4. In addition, let S_k be the sampled signal, with S_{I_k} and S_{Q_k} as its in-phase and quadrature components.

$$M_2 = E\{|S_k|^2\} = E\{S_{i_k}^2 + S_{Q_k}^2\} = P_S + P_N \tag{7.77}$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{i_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4 P_S P_N + k_n P_N^2 \tag{7.78}$$

Here, k_e and k_n are constants depending on the properties of the probability density function of the signal and noise processes. In practice, they are known from the modulation scheme used and the noise characteristics. For signals with constant energy, for instance, $k_e = 1$, and for two dimensional gaussian noise $k_n = 2$. It follows that the signal and noise powers can be defined as follows:

$$P_S = \sqrt{2M_2^2 - M_4} \tag{7.79}$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \tag{7.80}$$

and therefore

$$\text{SNR} = \frac{\sqrt{2EM_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \tag{7.81}$$

Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [**ren05**].

$$M_4' = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2 P_S P_N + P_N^2 \tag{7.82}$$

Thus, P_S ,P_N and the SNR become:

$$P'_S = \sqrt{M_2^2 - M'_4} \tag{7.83}$$

$$P'_N = M_2\sqrt{M_2^2 - M'_4} \tag{7.84}$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M'_4}}{M_2 - \sqrt{M_2^2 - M'_4}} \tag{7.85}$$

Known Issues

This block currently only works with either one or two *TimeContinuousAmplitudeContinuousReal* signals, depending on the chosen method. It has also only been tested with the in-phase and quadrature components of an MQAM signal with M=4. If the noise's power spectral density is high enough for the signal not to be detected, the block will fail (in the mentioned case, typically values of SNR < -10dB can be unreliable).

7.75 Single Photon Receiver

This block of code simulates the reception of two time continuous signals which are the outputs of single photon detectors and decode them in measurements results. A simplified schematic representation of this block is shown in figure 7.51.

Figure 7.51: Basic configuration of the SPR receiver

Functional description

This block accepts two time continuous input signals and outputs one time discrete signal that corresponds to the single photon detection measurements demodulation of the input signal. It is a complex block (as it can be seen from figure 7.51 of code made up of several simpler blocks whose description can be found in the *lib* repository.

In can also be seen from figure 7.51 that there are two extra internal input signals generated by the *Clock* in order to keep all blocks synchronized. This block is used to provide the sampling frequency to the *Sampler* blocks.

Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 7.46) the input parameters and corresponding functions are summarized.

Input parameters	**Function**	Type	**Accepted values**
samplesToSkip	Samples to skip in sampler block	int values	
filterType	Type of the filter applied in pulse shapper	PulseShaperFilter	

Table 7.45: List of input parameters of the block SP receiver

Methods

SinglePhotonReceiver(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals)({constructor})

void setPulseShaperFilter(PulseShaperFilter fType)

void setPulseShaperWidth(double pulseW)

void setClockBitPeriod(double period)

void setClockPhase(double phase)

void setClockSamplingPeriod(double sPeriod)

void setThreshold(double threshold)

Input Signals

Number: 2

Type: Time Continuous Amplitude Continuous Real

Output Signals

Number: 1

Type: Time Discrete Amplitude Discrete Real

Example

Sugestions for future improvement

7.76 SOP Modulator

Header File	:	sop_modulator_*.h
Source File	:	sop_modulator_*.cpp
Version	:	20180514 (Mariana Ramos)

This block of code simulates a modulation of the State Of Polarization (SOP)in a quantum channel, which intends to insert possible errors occurred during the transmission due the polarization rotation of single photons. These SOP changes can be simulated using deterministic or stochastic methods. The type of simulation is one of the input parameters when the block is initialized.

Functional description

This block intends to simulate SOP changes using deterministic and stochastic methods. The required function mode must be set when the block is initialized. Furthermore, other input parameters should be also set at initialization. If a deterministic method was set by the user, he also needs to set the θ and ϕ angles in degrees, which corresponds to the two parameters of Jones Space in Poincare Sphere thereby being the rotation and elevation angle, respectively. On the other hand, if a stochastic method is set by the user a model proposed in [**Czegledi16**] was implemented.

Figure 7.52: Diagram block of SOP block input and outputs.

The block in figure 7.52 was implemented based on a continuous matrix calculation. Lets assume that the input photon stream can be represented by the matrix:

$$S_{in} = \begin{bmatrix} A_x \\ A_y \end{bmatrix}, \tag{7.86}$$

where A_x and A_y are complex numbers. The SOP block will implement the multiplication operation between the input photon S_{in} and the random matrix $\mathbf{J}_k$ obtaining the output photon S_{out} with a different polarization:

$$S_{out} = S_{in}\mathbf{J}_k, \tag{7.87}$$

where,

$$\mathbf{J}_k = J(\dot{\boldsymbol{\alpha}})\mathbf{J}_{k-1}. \tag{7.88}$$

$J(\boldsymbol{\alpha})$ is a random matrix which can be expressed using the matrix exponential parameterized by $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)$:

$$\begin{aligned} J(\boldsymbol{\alpha}) &= e^{-i\boldsymbol{\alpha}\cdot\vec{\boldsymbol{\sigma}}} \\ &= \mathbf{I}_2 \cos(\theta) - i\mathbf{a}\cdot\vec{\boldsymbol{\sigma}}\sin(\theta), \end{aligned} \tag{7.89}$$

where $\vec{\boldsymbol{\sigma}} = (\boldsymbol{\sigma_1}, \boldsymbol{\sigma_2}, \boldsymbol{\sigma_3})$ is the tensor of Pauli Matrices:

$$\boldsymbol{\sigma_1} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}; \boldsymbol{\sigma_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \boldsymbol{\sigma_3} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \tag{7.90}$$

$\mathbf{I}_2$ is the 2×2 identity matrix. In addition, $\boldsymbol{\alpha} = \theta\boldsymbol{a}$, having the length $\theta = \|\boldsymbol{\alpha}\|$ and the direction on the unit sphere $\boldsymbol{a} = (a_1, a_2, a_3)$, where $\|\cdot\|$ denotes the Euclidian norm. Furthermore, the randomness is achieved by $\boldsymbol{\alpha}$ parameters:

$$\dot{\boldsymbol{\alpha}} \sim \mathcal{N}(0, \sigma_p^2\mathbf{I}_3), \tag{7.91}$$

where $\sigma_p^2 = 2\pi\Delta pT$, being T the symbol period and Δp the polarization linewidth, which is a parameter dependent on the fiber installation.

This method allows to analyse the polarization drift over time for a fixed fiber length.

Input parameters

SOP modulator block must have an input signals to set the clock of the operations. This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the SOP modulator. Each parameter has associated a function that allows for its change. In the following table (table 7.46) the input parameters and corresponding functions are summarized.

Input parameters	**Function**	**Accepted values**
sopType	Simulation type that the user intends to simulate.	Deterministic OR Stocastic
theta	Rotation Angle in Jones space for deterministic rotation in degrees	double
phi	Elevation Angle in Jones space for deterministic rotation in degrees	double
deltaP	Polarization Linewidth	double

Table 7.46: List of input parameters of the block sop modulator.

Methods

SOPModulator(vector <Signal*> &inputSignals, vector <Signal*> &outputSignals)({constructor})

void initialize(void)

bool runBlock(void)

void setSOPType(SOPType sType)

void setRotationAngle(double angle)

void setElevationAngle(double angle)

void setDeltaP(long double deltaP)

long double getDeltaP()

Input Signals

Number: 1

Type: PhotonStreamXY

Output Signals

Number: 1

Type: PhotonStreamXY

Example

Sugestions for future improvement

7.77 Source Code Efficiency

Header File	:	source_code_efficiency_*.h
Source File	:	source_code_efficiency_*.cpp
Version	:	20180621 (MarinaJordao)

Input Parameters

This block accepts one input signal and it produces one output signal. To perform this block, two input variables are required, probabilityOfZero and sourceOrde.

Parameter	Type	Values	Default
probabilityOfZero	double	from 1 to 0	0.45
sourceOrder	int	2, 3 or 4	2

Table 7.47: Source Code Efficiency input parameters

Functional Description

This block estimates the efficiency of the message, by calculating the entropy and the length of the message.

Input Signals

Number: 1
Type: Binary

Output Signals

Number: 1
Type: Real (TimeContinuousAmplitudeContinuousReal)

7.78 White Noise

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

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

1. DefaultDeterministic
2. RandomDevice
3. Selected

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

Input Parameters

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

Table 7.48: White noise input parameters

Methods

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

void initialize(void);

bool runBlock(void);

void setNoiseSpectralDensity(double SpectralDensity) spectralDensity = SpectralDensity;

double const getNoiseSpectralDensity(void) return spectralDensity;

void setSeedType(SeedType sType) seedType = sType; ;

SeedType const getSeedType(void) return seedType; ;

void setSeed(int newSeed) seed = newSeed;

int getSeed(void) return seed;

Functional description

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

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

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

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

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

$$N = n_0 B = n_0 \frac{2}{T} \tag{7.92}$$

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

Input Signals

Number: 0

Output Signals

Number: 1 or more

Type: RealValue, ComplexValue or ComplexValueXY

Examples

Random Mode

Suggestions for future improvement

7.79 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 7.49: Ideal Amplifier input parameters

Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;

Functional description

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

Input Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Output Signals

Number: 1

Type: Electrical (TimeContinuousAmplitudeContinuousReal)

Examples

Sugestions for future improvement

7.80 Phase Mismatch Compensation

Header File	:	phase_mismatch_compensation_*.h
Source File	:	phase_mismatch_compensation_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
numberOfSamplesForEstimation	integer	21
pilotRate	integer	2

Methods

- PhaseMismatchCompensation(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig, OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setPilotRate(int pRate) { pilotRate = pRate; };
- void setNumberOfSamplesForEstimation(int nSamplesEstimation) { numberOfSamplesForEstimation = nSamplesEstimation; samplesForEstimation.resize(nSamplesEstimation); };

Input Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts a complex constellation signal and outputs another complex constellation built from its input. The block assumes pilot-aided phase mismatch compensation is being performed, takes the pilot signals before and after each signal, makes an average of the phase in each pilot and uses that estimation to compensate the phase mismatch.

Theoretical Description

The pilot-assisted phase mismatch compensation scheme employed in this block is based on the schemes proposed in [**soh15**, **qi15**]. A representation of the output of the modulation stage of a pilot-assisted LLO CV-QC scheme is presented in Figure 7.53. An unmodulated

Figure 7.53: Representative time dependence of a pilot assisted phase mismatch compensation signal.

pilot signal is sent, time-multiplexed with the information carrying pulses, with the pilot signal being used to estimate the phase mismatch between the two lasers. At the receiver, the quantum and the pilot constellation are given, respectively, by

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)]}, \ n \neq mR_P \\ x_p(t_n)e^{i[\epsilon(t_n)]}, \ n = mR_P \end{cases}, \ n, m \in \mathbb{N}. \tag{7.93}$$

The phase mismatch for the nth pulse is estimated from an average of the phase of the previous and the later pilots, resulting in

$$\hat{\epsilon}(t_n) = \frac{\epsilon(t_{(m+1)R_P}) + \epsilon(t_{mR_P})}{2}, \ mR_P < n < (m+1)R_P, \tag{7.94}$$

with the mismatch compensation being accomplished by multiplying the constellation by $e^{-i(\hat{\epsilon}(t_n))}$, resulting in

$$x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)-\hat{\epsilon}(t_n)]}. \tag{7.95}$$

7.81 Frequency Mismatch Compensation

Header File	:	frequency_mismatch_compensation_*.h
Source File	:	frequency_mismatch_compensation_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
pilotRate	integer	0
numberOfSamplesForEstimation	integer	21

Methods

- FrequencyMismatchCompensation(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setPilotRate(int pRate) { pilotRate = pRate; };
- void setNumberOfSamplesForEstimation(int nSamplesEstimation) { numberOfSamplesForEstimation = nSamplesEstimation; samplesForEstimation.resize(nSamplesEstimation); };
- void setMode(FrequencyCompensationMode m) { mode = m; }

Input Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts a complex constellation and outputs another built from its input. The block attempts to perform frequency mismatch compensation on the input constellation by one of three different methods:

- Pilot aided method
- Blind estimation method
- Spectral method

Theoretical Description

The signal at the receiver for a system with frequency mismatch can be assumed to take the form

$$y(t) = |y(t)|e^{i[\Delta\omega t+\theta(t)+\epsilon(t)]}, \tag{7.96}$$

where $\Delta\omega$ is the frequency mismatch, $\theta(t_n)$ is the phase encoded in the signal and $\epsilon(t_n)$ is the phase noise contribution. Frequency mismatch compensation is accomplished by first estimating the value of $\Delta\omega$ and then removing via

$$y(t)' = y(t) * e^{-i\Delta\hat{\omega}}. \tag{7.97}$$

Three methods for estimating $\Delta\omega$ are presented here.

Pilot aided frequency mismatch compensation

This method uses a pilot signal similar to the ones employed in pilot assisted phase mismatch compensation techniques. In an pilot aided technique a reference signal (the pilot), composed of pre-agreed on symbols, is inserted, time multiplexed, with the data payload at a pre-agreed on rate. A visual representation of the output of the modulation stage of a pilot-assisted scheme is presented in Figure 7.54. At the receiver stage, after coherent detection,

Figure 7.54: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted scheme. Pilot pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

the signal is described by

$$y(t_n) = \begin{cases} x_s(t_n)e^{i[\Delta\omega t_n+\theta_q(t_n)+\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n+\epsilon(t_n)]}, & n = mR_P \end{cases}, \tag{7.98}$$

where $x_s(t)/x_p(t)$ represents the signal/pilot amplitude, R_P is the pilot rate, $\Delta\omega$ is the offset frequency, $\epsilon(t)$ is the instantaneous phase noise contribution and $\theta(t)$ is the phase

modulation at instant t. The first step in this technique is to obtain the auxiliary signal $x_f(m)$, which is defined as

$$\begin{aligned} x_f(m) &= y_r^*(t_{mR_P})y_r(t_{(m+1)R_P}) \\ &= x_p(t_{mR_p})x_p(t_{(m+1)R_p})e^{i(\Delta\omega R_P T_s + \Delta\epsilon(m))}, \end{aligned} \tag{7.99}$$

where

$$\Delta\epsilon(m) = \epsilon(t_{mR_P}) - \epsilon(t_{(m+1)R_P}). \tag{7.100}$$

The frequency estimation is accomplished by taking the expected value of the complex argument of (7.99) and dividing it by $R_P T_s$. This returns

$$\begin{aligned} \Delta\hat{\omega} &= \mathrm{E}\left[\frac{1}{R_P T_s}\arg(x_f(m))\right] \\ &= \mathrm{E}\left[\Delta\omega + 2k\pi(R_P T_s)^{-1} + \frac{\Delta\epsilon(m)}{R_P T_s}\right] \\ &= \Delta\omega + 2k\pi(R_P T_s)^{-1}. \end{aligned} \tag{7.101}$$

Blind estimation frequency mismatch compensation

In this method the frequency is scanned over a predetermined range, symbol decisions are made and the minimum square error used as the frequency-selection criteria. Scanning is first done with a large step to find a rough value for $\Delta\omega$, this is then repeated with a smaller step to find a more exact estimate [**zhou11**].

Spectral evaluation frequency mismatch compensation

In this method the frequency is estimated by evaluating the spectrum of the mth power of the input signal [**selmi09**], which exhibits a peak at the frequency $m\Delta\Omega$, this value is used as the estimate.

7.82 Cloner

Header File	:	cloner_*.h
Source File	:	cloner_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

This block takes no input parameters.

Methods

- Cloner(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);

Input Signals

Number: 1
Type: Real, Complex, Complex_XY, Binary

Output Signals

Number: Arbitrary
Type: Same as input

Functional Description

This block accepts a signal and outputs a number of copies of the input. The number of the copies is set by the number of output signals given to the block. The block adapts dynamically.

Theoretical Description

7.83 Error Vector Magnitude

Header File	:	error_vector_magnitude_*.h
Source File	:	error_vector_magnitude_*.cpp
Version	:	20190114 (Daniel Pereira)

Input Parameters

Name	Type	Default Value
referenceAmplitude	double	1.0
m	integer	0
midRepType	enum	Cumulative

Methods

- ErrorVectorMagnitude(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,OutputSig){};
- void initialize(void);
- bool runBlock(void);
- void setMidReportSize(int M) { m = M; }
- void setMidReportType(MidReportType mrt) { midRepType = mrt; }
- void setReferenceAmplitude(double rAmplitude) { referenceAmplitude = rAmplitude; }

Input Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Output Signals

Number: 1
Type: Complex (DiscreteTimeContinuousAmplitude)

Functional Description

This block accepts one binary string and outputs copy of the input binary string. This block also outputs .txt files with a report of the estimated Error Vector Magnitude (EVM), $\widehat{EVM}$.

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. This block can operate mid-reports using CUMULATIVE mode, in which $\widehat{EVM}$ is calculated taking into account all received points, or in a RESET mode, in which $\widehat{EVM}$ is computed from only the points after the previous mid-report.

Theoretical Description

The $\widehat{\text{EVM}}$ is obtained by evaluating the relation between the magnitude of the error vector $\vec{e}_v$ and the magnitude of the vector of the ideal symbol position $\vec{ref}_v$. This process is presented visually in Figure 7.55 and is described by

$$\widehat{\text{EVM}} = 100\sqrt{\frac{|\vec{e}_v|}{|\vec{ref}_v|}} = 100\sqrt{\frac{|\vec{m}_v - \vec{ref}_v|}{|\vec{ref}_v|}}. \tag{7.102}$$

Figure 7.55: Visual representation of the EVM method in a QPSK constellation. The igual constellation points are presented in blue, while an example for the actual measured symbol is presented in red.

7.84 Load Ascii

Header File	:	load_ascii.h
Source File	:	load_ascii.cpp
Version	:	20190205 (Daniel Pereira)

This block loads signals from any ascii file into the netxpto simulation environment.

Input Parameters

Parameter	Type	Values	Default
samplingPeriod	double	any	1.0
symbolPeriod	double	any	1.0
asciiFileName	string	any	InputFile.txt
delimiterType	delimiter_type	CommaSeperatedValues, ConcatenatedValues	CommaSeperatedValues
dataType	signal_value_type	BinaryValue, RealValue, ComplexValue, ComplexValueXY	BinaryValue

Table 7.50: Load ascii input parameters

Methods

LoadAscii()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setSymbolPeriod(double sPeriod);

void setDataType(signal_value_type dType);

void setAsciiFileName(string aFileName);

Functional description

This block loads signals from a .txt file and generates a signal with a specified data type given by the input parameter *dataType* and with a specified delimiter type given by the input parameter *delimiterType*.

Input Signals

Number: 0

Output Signals

Number: 1
Type: Optical signal, Electrical signal, Complex signal, Binary signal (user defined)

7.85 Load Signal

Header File	:	load_signal.h
Source File	:	load_signal.cpp
Version	:	20190205 (Daniel Pereira)

This block loads signals from a .sgn file into the netxpto simulation environment.

Input Parameters

Parameter	Type	Values	Default
sgnFileName	string	any	InputFile.sgn

Table 7.51: Load signal input parameters

Methods

LoadSignal()

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

void initialize(void);

bool runBlock(void);

void setSgnFileName(string sFileName);

Functional description

This block loads signals from a .sgn file into a signal with symbol and sampling period set by the input .sgn file and type set by the output signal (make sure that the output signal has the correct format, otherwise it will be corrupted).

Input Signals

Number: 0

Output Signals

Number: 1
Type: Binary signal, Integer signal, Complex signal, Complex XY signal, Photon signal, Photon Multipath signal, Photon Multipath XY signal

7.86 Matched Filter

Header File	:	matched_filter.h
Source File	:	matched_filter.cpp
Version	:	20190205 (Daniel Pereira)

This block applies a matched filter to the input signal.

Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	64
rollOffFactor	double	any	0.5

Table 7.52: Matched filter input parameters

Methods

MatchedFilter(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setNumberOfTaps(int numberOfTaps);

void setRollOffFactor(double rollOffFactor);

Functional description

This block implements a matched filter, that coincides with the modulation applied to the signal at the transmitter stage. The filter is applied in frequency domain, where it takes the form of a simple multiplication.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

7.87 Timing Deskew

Header File	:	timing_deskew.h
Source File	:	timing_deskew.cpp
Version	:	20190225 (Daniel Pereira)

This block allows for the removal of the timing mismatches between the real and imaginary components of a complex signal.

Input Parameters

Parameter	Type	Values	Default
skew	vector<`double`>	any	[0, 0]

Table 7.53: Timing deskew input parameters

Methods

TimingDeskew(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setSkew(vector<double> s) { skew.resize(s.size()); skew[0] = s[0]; skew[1] = s[1]; }

Functional description

This DSP step takes a complex input signal and removes a given amount of timing skew between its real (in-phase) and imaginary (quadrature) parts. This DSP step follows the topology presented in Figure 7.56, in which the input signal $S_{\text{in}}(n)$ is separated into its in-

Figure 7.56: Block diagram representation of the deskew procedure.

phase and in-quadrature components, $I(n)$ and $Q(n)$ and are each sent through independent delay line filters. In turn, the delay line filters function as shown in Figure 7.57. The signal

Figure 7.57: Block diagram representation of the implemented delay line filter.

at the input, $S_{\text{in}}(t_n)$, is transformed to Fourier space, where it is multiplied by a deskew element τ

$$\hat{S}'(f_n) = \hat{S}(f_n)e^{i2\pi f_n \tau}. \tag{7.103}$$

The value of τ here is set by trial and error and is characteristic of a certain receiver. For these results the timing skew was set at 20 ps.

This signal is then transformed back to time domain and the imaginary part is discarded. After both components are passed through the delay line, they are combined into a complex signal as the output

$$S_{\text{out}}(t_n) = \text{real}(I''(t_n)) + i\text{real}(Q''(t_n)). \tag{7.104}$$

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

7.88 DC component removal

Header File	:	dc_component_removal.h
Source File	:	dc_component_removal.cpp
Version	:	20190225 (Daniel Pereira)

This block removes the DC/low frequency component of a complex time continuous signal.

Input Parameters

This block has no input parameters.

Methods

DCComponentRemoval(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

Functional description

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

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n) - \text{mean}(S_{\text{in}}(t_n)) \tag{7.105}$$

The effect of this DSP step is presented in Figure **??**. To better show the effect of this DSP, an exaggerated DC removal is presented in Figure 7.58, where it can be seen that the DC component is removed without introducing obvious discontinuities in the signal.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

Figure 7.58: Exaggerated DC component removal.

7.89 Orthonormalization

Header File	:	orthonormalization.h
Source File	:	orthonormalization.cpp
Version	:	20190225 (Daniel Pereira)

This block applies a Gram-Schmidt orthonormalization procedure to the input signal.

Input Parameters

This block has no input parameters.

Methods

Orthonormalization(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

Functional description

This block orthonormalizes the data by implementing a Gram-Schmidt algorithm [**arfken99**]. This is implementation follows the topology presented in Figure 7.59. The input signal $S_{\text{in}}(t_n)$ is

Figure 7.59: Block diagram representation of the Gram-Schmidt orthonormalization procedure.

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

$$P_I = \text{mean}(I^2(t_n)), \tag{7.106}$$

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

$$I_{\text{on}}(t_n) = \frac{I(t_n)}{\sqrt{P_I}}. \tag{7.107}$$

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

$$Q_{\text{og}}(t_n) = Q(t_n) - \frac{I(t_n)\text{mean}(I(t_n)Q(t_n))}{P_I}, \tag{7.108}$$

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

$$P_Q = \text{mean}(Q_{\text{og}}^2(t_n)), \tag{7.109}$$

$$Q_{\text{on}}(t_n) = \frac{Q_{\text{og}}(t_n)}{\sqrt{P_Q}}. \tag{7.110}$$

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

$$S_{\text{out}}(t_n) = I_{\text{on}}(t_n) + iQ_{\text{on}}(t_n) \tag{7.111}$$

Input Signals

Number: 1 **Type:** Complex signal

Output Signals

Number: 1
Type: Complex signal

Sugestions for future improvement

- Add the optimized recursive Gram-Schmidt algorithm as an option.

7.90 Matched Filter

Header File	:	matched_filter.h
Source File	:	matched_filter.cpp
Version	:	20190205 (Daniel Pereira)

This block applies a matched filter to the input signal.

Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	64
rollOffFactor	double	any	0.5

Table 7.54: Matched filter input parameters

Methods

MatchedFilter(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setNumberOfTaps(int numberOfTaps);

void setRollOffFactor(double rollOffFactor);

Functional description

This block implements a matched filter, that coincides with the modulation applied to the signal at the transmitter stage. The filter is applied in frequency domain, where it takes the form of a simple multiplication.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

7.91 Upsampler

Header File	:	upsampler.h
Source File	:	upsampler.cpp
Version	:	20190319 (Daniel Pereira)

This block applies an interpolation filter.

Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	8
upSamplingFactor	int	any	2

Table 7.55: Matched filter input parameters

Methods

Upsampler(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig) :Block(InputSig, OutputSig){};

void initialize(void);

bool runBlock(void);

void setNumberOfTaps(int numberOfTaps);

void setUpsamplingFactor(int upSamplingFactor);

Functional description

This block applies the FIR interpolation filter presented in Figure 7.60.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

Figure 7.60: Impulse response of a 4× upsampling filter using 16 taps.

7.92 Matched Filter

Header File	:	resampler.h
Source File	:	resampler.cpp
Version	:	20190319 (Daniel Pereira)

This block upsamples the input signal by a user defined factor.

Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	64
upSamplingFactor	int	any	0.5

Table 7.56: Matched filter input parameters

Methods

Resampler(initializer_list<Signal *> &InputSig, initializer_list<Signal *> &OutputSig);

void initialize(void);

bool runBlock(void);

void setNumberOfTaps(int numberOfTaps);

void setUpsamplingFactor(int upSamplingFactor);

Functional description

This block condenses the blocks interspace0s and upsampler into a single. A complex signal at its input is upsampled by using a FIR interpolation filter.

Input Signals

Number: 1
Type: Complex signal

Output Signals

Number: 1
Type: Complex signal

Suggestions for future improvement

Have this block also be able to downsample signals.

Chapter 8

Mathlab Tools

8.1 Generation of AWG Compatible Signals

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

This section shows how to convert a simulation signal into an AWG compatible waveform file through the use of a matlab function called sgnToWfm. This allows the application of simulated signals into real world systems.

8.1.1 sgnToWfm.m

Structure of a function

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

Inputs

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

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

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

Outputs

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

data: A vector with the signal data.

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

samplingPeriod: Sampling period of the signal.

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

numberOfSymbols: Number of symbols retrieved from the signal.

samplingRate: Sampling rate of the signal.

Functional Description

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

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

Using one argument:

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

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

Using two arguments:

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

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

Using three arguments:

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

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

8.1.2 sgnToWfm_20171121.m

Structure of a function

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

Inputs

Same as discussed above in the file sgnToWfm.m.

Outputs

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

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

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

Functional Description

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

Outputs

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

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

Decimation factor calculation

The flowchart for calculating the decimation factor is as follows:

Figure 8.1: Flowchart to calculate decimation factor

»»»> Develop.Romil

8.1.3 Loading a signal to the Tekatronix AWG70002A

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

Sampling rate up to 16 GS/s: This is the most important characteristic because it determines the maximum sampling rate that your signal can have. It must not be over 16 GS/s or else the AWG will not be able to recreate it appropriately.

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

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

1. Using the function sgnToWfm: Start up Matlab and change your current folder to mtools and add the signals folder that you want to convert to the Matlab search path. Use the function accordingly, putting as the input parameter the signal file name you want to convert.

2. AWG sampling rate: After calling the function there should be waveform file in the mtools folder, as well as a variable called samplingRate in the Matlab workspace. Make sure this is equal or bellow the maximum sampling frequency of the AWG (16 GS/s), or else the waveform can not be equal to the original signal. If it is higher you have to adjust the parameters in the simulation in order to decrease the sampling frequency of the signal(i.e. decreasing the bit period or reducing the samples per symbol).

3. Loading the waveform file to the AWG: Copy the waveform file to your pen drive and connect it to the AWG. With the software of the awg open,go to browse for waveform on the channel you want to use, and select the waveform file you created (Figure 7.1).

Figure 8.2: Selecting your waveform in the AWG

Now you should have the waveform displayed on the screen. Although it has the same shape, the waveform might not match the signal timing wise due to an incorrect sampling rate configured in the AWG. In this example (Figure 7.2), the original signal has a sample rate of 8 GS/s and the AWG is configured to 1.25 GS/s. Therefore it must be changed to the correct value. To do this go to the settings tab, clock settings, and change the sampling rate to be equal to the one of the original signal, 8 GS/s (Figure 7.3). Compare the waveform in the AWG with the original signal, they should be identical (Figure 7.4).

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

Figure 8.4: Configuring the right sampling rate

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

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

8.2 Polarization Analysis Signals

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

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

8.2.1 jonesToStokes.m

Structure of a function

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

Inputs

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

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

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

NumberOfSamples: Number of Samples to plot in Poincare sphere.

Outputs

Two graphics are plotted from the photon stream input signal:

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

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

Functional Description

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

Students Name	:	Mariana Ramos
Version	:	ACF_20180614.m

8.2.2 ACF.m

Structure of a function

[] = ACF(fname,deltaP,N)

Inputs

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

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

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

Outputs

This matlab function has two outputs:

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

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

Functional Description

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

Students Name	:	Mariana Ramos
Version	:	plotPhotonStream_20180102.m

8.2.3 plotPhotonStream_20180102.m

Structure of a function

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

Inputs

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

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

h: Number of the figure to plot.

Outputs

This matlab function has four outputs:

mag_x: Magnitude of the complex number X.

mag_y: Magnitude of the complex number Y.

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

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

Functional Description

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

Students Name	:	Mariana Ramos
Version	:	plot_sphere.m

8.2.4 plot_sphere.m

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

Chapter 9

Algorithms

9.1 Fast Fourier Transform

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

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 [**Rao2010a**]. The computational complexity of Radix-2 and Bluestein algorithm is $O(Nlog_2N)$, however, the computation of Bluestein algorithm involves the circular convolution which increases the number of computations. Therefore, to reduce the computational time it is advisable to work with the vectors of length 2^N [**Chu2000**].

Figure 9.1: Top level architecture of FFT algorithm

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.

```
%
    %%%%
```

```
        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

%%
%
    %%%%
```

```
%%%
```

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

```
# include "fft_20180208.h"
# include <complex>
# include <fstream>
# include <iostream>
# include <math.h>
# include <stdio.h>
# include <string>
# include <strstream>
# include <algorithm>
# include <vector>
#include <iomanip>

using namespace std;

int main()
{
  ////////////////////////////////// Section 1 //////////////////////////////////////
  /////////// Read the input text file (import "time_function.txt" ) ///////////
  ////////////////////////////////////////////////////////////////////////////////
  ifstream inFile;
  inFile.precision(15);
  double ch;
  vector <double> inTimeDomain;
  inFile.open("time_function.txt");
  // First data (at 0th position) appled to the ch it is similar to the "cin".
  inFile >> ch;
  // It'll count the length of the vector to verify with the MATLAB
  int count=0;
  while (!inFile.eof()){
    // push data one by one into the vector
    inTimeDomain.push_back(ch);
    // it'll increase the position of the data vector by 1 and read full vector.
    inFile >> ch;
    count++;
  }
  inFile.close(); // It is mandatory to close the file at the end.

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

  vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
```

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

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

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

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

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

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

Listing 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

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 [**Smith1999**, **Blahut1985**] :
Step 1 : Determine the length M of impulse response, $h(n)$.

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

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

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

Step 5 : Make the segments of the input sequences of length L, $x_i(n)$, where index i correspond to the ith 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().

```
%
    %%%%
```

```
        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)')

%%
%
      %%%%
```

```
%%
%
    %%%%
```

```
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
%%
%
    %%%%
```

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

```
# include "overlap_save_20180208.h"

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

using namespace std;

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

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

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

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

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

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

```
ifstream inFileFilter;
double chFilter;
vector <double> inTimeDomainFilter;
inFileFilter.open("time_domain_filter.txt");
inFileFilter >> chFilter;
int countFilter = 0;

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

//////////////////////////////// Section 2 ////////////////////////////////////
//////////////////////// Real to complex conversion ////////////////////////
////////////////////////////////////////////////////////////////////////////////
/////////////// For signal data ///////////////
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
  im[i] = 0;               // Imaginary data of the signal
}
// Next, Real and Imaginary vector to complex vector conversion
inTimeDomainComplex = reImVect2ComplexVector(re, im);

/////////////// For filter data ///////////////
vector <complex<double>> inTimeDomainFilterComplex(inTimeDomainFilter.size());
vector <double> reFilter(inTimeDomainFilter.size());
vector <double> imFilter(inTimeDomainFilter.size());

for (unsigned int i = 0; i < inTimeDomainFilter.size(); i++)
{
  reFilter[i] = inTimeDomainFilter[i];
  imFilter[i] = 0;
}
inTimeDomainFilterComplex = reImVect2ComplexVector(reFilter, imFilter);

//////////////////////////////// Section 3 ////////////////////////////////////
////////////////////////////// Overlap & save ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
vector <complex<double>> y;
y = overlapSave(inTimeDomainComplex, inTimeDomainFilterComplex);

//////////////////////////////// Section 4 ////////////////////////////////////
```

```
    ////////////////////////////////// Save data ////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    ofstream outFile;
    complex<double> outFileData;
    outFile.precision(20);
    outFile.open("overlap_save_data.txt");

    for (unsigned int i = 0; i <y.size(); i++)
    {
      outFile << y[i].real() << endl;
      outFile << y[i].imag() << endl;
    }
    outFile.close();

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

3. Sinusoid with exponent

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

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

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

This section explains the steps of comparing real-time overlap-save method with the time-domain filtering. The structure of the real-time overlap-save function

Figure 9.24: MATLAB and C++ comparison

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

$overlapSave(x_{m-1}(n), x_m(n), h(n))$ requires an impulse response $h(n)$ of the filter. There are two methods to feed the impulse response to the real-time overlap-save function:
Method 1. The impulse response $h(n)$ of the filter can be fed using the time-domain impulse 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.

```
//////////////////////////////////////////////////////////////
//////////////////////////// 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 ////////////////////
  ////////////////////////////////////////////////////////////////
  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;
}


///// Overlap save::Transfer Function : 2018 - 11 - 10 //////
//////////////////////////////////////////////////////////////
////////////////////// FD_Filter_20181110 ////////////////////
```

```
/////////////////////////////////////////////////////////////////
void FD_Filter_20181110::initializeFD_Filter_20181110(void)
{
  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

  if (!getSeeBeginningOfTransferFunction()) {
    int aux = (int) (((double)transferFunctionLength) / 4) + 1;
    outputSignals[0]->setFirstValueToBeSaved(aux);
  }

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

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

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

bool FD_Filter_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]);
      }
    }

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

    currentCopy = cc;

  }

  vector<t_complex> out;
    out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(transferFunction));

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

  K = K + 1;
  previousCopy = currentCopy;

  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.

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

Figure 9.26: Real-time overlap-save example setup

Figure 9.27: Comparison of signal S6 and S7

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());

  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;
}



///// Overlap save::Transfer Function : 2018 - 11 - 10 //////
////////////////////////////////////////////////////////////////
////////////////////// FD_Filter_20181110 ////////////////////
```

```
////////////////////////////////////////////////////////////////
void FD_Filter_20181110::initializeFD_Filter_20181110(void)
{
  outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
  outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
  outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

  if (!getSeeBeginningOfTransferFunction()) {
    int aux = (int) (((double)transferFunctionLength) / 4) + 1;
    outputSignals[0]->setFirstValueToBeSaved(aux);
  }

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

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

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

bool FD_Filter_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]);
      }
    }

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

    currentCopy = cc;

  }

  vector<t_complex> out;
    out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(transferFunction));

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

  K = K + 1;
  previousCopy = currentCopy;

  return true;
}
```

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.

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

Figure 9.29: Real-time overlap-save example setup

Figure 9.30: Comparison of signal S6 and S7

9.3 Filter

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

In order to filter any signal, a new generalized version of the filter namely $filter_*.h$ & $filter_*.cpp$ is programmed which facilitate to filtering in both time and frequency domain. Basically, $filter_*.h$ file contains the declaration 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. [**Kuo**, **Rappaport2002**, **Oppenheim1999**] .
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())
    {
      for (int i = 0; i < currentCopy.size(); i++)
      {
        currentCopyAux.push_back(currentCopy[i]);
      }
    }

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

    currentCopy = cc;

  }

  vector<t_complex> out;
    out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(transferFunction));

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

  K = K + 1;
  previousCopy = currentCopy;

  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.

Figure 9.33: Filter test setup

Step 4 : Run the *filter_test.cpp* code and compare the signals **S6.sgn** and **S7.sgn** using visualizer.
Here, we have used three different types of pulse shaping filter namely, raised cosine, root raised cosine and Gaussian pulse shaper. The following Figure 9.34, 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.

9.4 Hilbert Transform

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

What is the purpose of Hilbert transform?

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

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

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

Transfer function for the discrete Hilbert transform

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

Method 1 :

The discrete time analytical signal $S_a(t)$ corresponding to $s(t)$ is defined in the frequency domain as **[Marple]** (This method requires MATLAB Hilbert transform definition)

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

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

Method 2 :

The discrete time Hilbert transformed signal $\hat{s}(t)$ corresponding to $s(t)$ is defined in the frequency domain as **[Oppenheim1999]**

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

which is inverse transformed to obtain a Hilbert transformed signal $\hat{S}(t)$. To generate an analytical signal, $\hat{S}(t)$ is added to the $S(t)$ to get the equation 9.15.

Real-time Hilbert transform : Proposed logical flow

To understand the new proposed method, consider that the signal consists of 2048 samples and the **bufferLength** is 512. Therefore, by considering the **bufferLength**, we will process the whole signal in four consecutive blocks namely A, B, C and D; each with the length of 512 samples as shown in Figure 9.37. 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.

Chapter 10

Code Development Guidelines

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

10.0.1 Integrated Development Environment

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

Visual Studio Community 2017 - version 15.7.6

10.0.2 Compiler Switches

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

Chapter 11

Building C++ Projects Without Visual Studio

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

11.1 Installing Microsoft Visual C++ Build Tools

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

11.2 Adding Path To System Variables

Please follow this step-by-step tutorial carefully.

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

11.3 How To Use MSBuild To Build Your Projects

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

1. Open the **Command Line** and navigate to your project folder (where the .vcxproj file is located).
2. Enter the command:
   msbuild <filename> /tv:14.0 /p:PlatformToolset=v140,TargetPlatformVersion=8.1,OutDir=".\"
   , where <filename> is your .vcxproj file.

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

11.4 Known Issues

11.4.1 Missing ucrtbased.dll

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

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

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

Chapter 12

Git Helper

12.1 Starting with Git

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

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

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

git init

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

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

12.2 Data Model

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

Git manipulates the following objects:

- commits - text files that store a description of the repository;
- trees - text files that store a description of a folder;
- blobs - the files that exist in your repository;
- tags - text files that store information about commits.

The objects are stored in the folder *.git/objects*. Each store object is identified by its SHA1 hash value, i.e. 20 bytes which identifies unequivocally the object. The SHA1 is just an algorithm

Figure 12.1: Git data model.

that accepts some binary information and generates 20 bytes which are ideally unique for that information. The probability of collisions is extremely low, i.e. the probability that different information generates the same hash value is extremely low. Note that 20 bytes can be represented by a 40 characters hexadecimal string. The identifier of each object is the 40 characters hexadecimal string. Each particular object is stored in a sub-folder inside the *.git/objects*. The name of the sub-folder is the two most significative characters of the SHA1 hash value. The name of the file that is inside the sub-folder is the remanning thirty eight characters of the SHA1 hash value. The Git stores all committed versions of a file. The Git maintains a contend-addressable file systems, i.e. a file system in which the files can be accessed based on its contend. Lets look more carefully in each stored object.

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

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

Here goes the commit message.

A **tree** object is identified by a SHA1 hash value, and has a list of blobs and trees that

are inside that tree. A tree object identifies a folder and its contend. Example of a tree file contend:

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

A **blob** is identified by a SHA1 hash value, and has the file contend compressed. A git header and tailor is added to each file and the file is compressed using the zlib library. The git header is just the object type, a space character, the file size in bytes and the \NUL caracter, for instance "blob 13\NUL", the tailor is just the \n caracter. The compressed blob (header+file contend+tailer) is stored as a binary file.

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

12.2.1 Objects Folder

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

The folder *.git/objects* stores information about all objects (commits, trees, blobs and annotated tags). The objects are stored in files inside folders. The name of the folders are the 2 first characters of the SHA1 40 characters hexadecimal string. The name of the files are the other 38 hexadecimal characters of the SHA1. The information is compressed to save space.

12.3 Refs

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

name of the file is the name of the ref and inside the file is a string with the SHA1 hash value. Tags and branches are example of refs. Tags are static references, i.e. tags are never updated, and branches are dynamic references, i.e. branches are always automatically updated.

12.3.1 Refs Folder

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

12.3.2 Branch

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

12.3.3 Heads

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

12.4 Git Logical Areas

Git uses several spaces.

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

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

12.5 Merge

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

12.5.1 Fast-Forward Merge

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

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

12.5.2 Three-Way Merge

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

12.6 Remotes

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

12.6.1 GitHub

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

Figure 12.3: Git remotes.

12.7 Commands

12.7.1 Porcelain Commands

Porcelain commends are high-level commands.

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

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

git cat-file

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

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

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

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

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

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

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

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

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

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

git ls-files

git ls-remote

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

git pull

git pull, downloads remote branch history and incorporates changes.

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

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

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

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

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

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

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

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

git stash drop, discards the most recently stashed changeset.

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

12.7.2 Pluming Commands

Pluming commands are low-level commands.

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

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

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

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

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

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

git ls-files
git ls-files –stage, shows all files that you are tracking.

git write-tree

git commit-tree

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

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

git update-ref
git update-ref refs/heads/<branch name> <commit sha1 value>, creates a branch that points to the <commit sha1 value>.

git verify-pack

12.8 Navigation Helpers

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

12.9 Configuration Files

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

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

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

git config –global -e

12.10 Pack Files

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

12.11 Applications

12.11.1 Meld

12.11.2 GitKraken

12.12 Error Messages

12.12.1 Large files detected

Clean the repository with the BFG Repo-Cleaner.

Run the Java program:

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

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

git push −−force.

12.13 Git with Overleaf

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

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

In this case the repository address is https://git.overleaf.com/12925162jkwbhrdkwrfm
The only change was the replacement of **www** by **git**.

Now you can just do
git clone https://git.overleaf.com/12925162jkwbhrdkwrfm
and clone your repository.
You can also do
git push https://git.overleaf.com/12925162jkwbhrdkwrfm

Chapter 13

Simulating VHDL Programs with GHDL

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

13.1 Adding Path To System Variables

Please follow this step-by-step tutorial:

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

13.2 Using GHDL To Simulate VHDL Programs

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

13.2.1 Simulation Input

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

13.2.2 Executing Testbench

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

13.2.3 Simulation Output

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