

# NetXPTO - LinkPlanner

June 5, 2019

---

## Contents

<b>1</b>	<b>Library</b>	<b>4</b>
1.1	ADC	5
1.2	Add	8
1.3	Arithmetic Encoder	9
1.4	Arithmetic Decoder	11
1.5	Alice QKD	13
1.6	Alice QKD	15
1.7	Ascii Source	18
1.8	Ascii To Binary	20
1.9	Balanced Beam Splitter	22
1.10	Bit Error Rate	23
	Bibliography	29
1.11	Binary Source	29
1.12	Binary To Ascii	33
1.13	Bob QKD	35
1.14	BobQuantumRx	36
1.15	Bit Decider	38
1.16	Clock	39
1.17	Clock_20171219	41
1.18	Complex To Real	44
1.19	Coupler 2 by 2	46
1.20	Carrier Phase Compensation	47
1.21	Decision Circuit	51
1.22	Decoder	53
1.23	Discrete To Continuous Time	55
1.24	DownSampling	57
1.25	DSP	59
1.26	EDFA	61
1.27	Electrical Signal Generator	64
	1.27.1 ContinuousWave	64

1.28 Entropy Estimator . . . . .	66
1.29 Entropy Estimator . . . . .	67
1.30 Fork . . . . .	69
1.31 Gaussian Source . . . . .	70
1.32 Hamming Decoder . . . . .	72
1.33 Hamming Encoder . . . . .	73
1.34 MQAM Receiver . . . . .	74
1.35 Huffman Decoder . . . . .	78
1.36 Huffman Encoder . . . . .	79
1.37 Ideal Amplifier . . . . .	80
1.38 IQ Modulator . . . . .	82
Bibliography . . . . .	87
1.39 IIR Filter . . . . .	87
Bibliography . . . . .	88
1.40 Local Oscillator . . . . .	88
1.41 Local Oscillator . . . . .	90
1.42 MS Windows IP Tunnel . . . . .	93
1.43 Mutual Information Estimator . . . . .	95
Bibliography . . . . .	98
1.44 MQAM Mapper . . . . .	98
1.45 M-QAM Receiver . . . . .	101
1.46 MQAM Transmitter . . . . .	108
1.47 Netxpto . . . . .	118
1.47.1 Version 20180118 . . . . .	118
1.47.2 Version 20180418 . . . . .	118
1.48 Alice QKD . . . . .	119
1.49 Polarizer . . . . .	121
1.50 Probability Estimator . . . . .	122
1.51 Bob QKD . . . . .	125
1.52 Eve QKD . . . . .	126
1.53 Rotator Linear Polarizer . . . . .	127
1.54 Mutual Information Estimator . . . . .	129
Bibliography . . . . .	132
1.55 Optical Switch . . . . .	132
1.56 Optical Hybrid . . . . .	133
1.57 Photodiode pair . . . . .	135
1.58 Photoelectron Generator . . . . .	138
Bibliography . . . . .	143
1.59 Power Spectral Density Estimator . . . . .	143
Bibliography . . . . .	149
1.60 Pulse Shaper . . . . .	149
1.61 Quantizer . . . . .	151

1.62 Resample . . . . .	154
1.63 SNR Estimator . . . . .	156
Bibliography . . . . .	161
1.64 Sampler . . . . .	161
1.65 SNR of the Photoelectron Generator . . . . .	163
Bibliography . . . . .	168
1.66 Sink . . . . .	168
1.67 SNR Estimator . . . . .	170
Bibliography . . . . .	175
1.68 Single Photon Receiver . . . . .	175
1.69 SOP Modulator . . . . .	179
Bibliography . . . . .	183
1.70 Source Code Efficiency . . . . .	183
1.71 White Noise . . . . .	184
1.72 Ideal Amplifier . . . . .	187
1.73 Phase Mismatch Compensation . . . . .	189
Bibliography . . . . .	191
1.74 Frequency Mismatch Compensation . . . . .	191
Bibliography . . . . .	194
1.75 Cloner . . . . .	194
Bibliography . . . . .	195
1.76 Error Vector Magnitude . . . . .	195
Bibliography . . . . .	197
1.77 Load Ascii . . . . .	197
1.78 Load Signal . . . . .	199
1.79 Matched Filter . . . . .	200
1.80 Timing Deskew . . . . .	201
1.81 DC component removal . . . . .	203
1.82 Orthonormalization . . . . .	205
Bibliography . . . . .	207
1.83 Matched Filter . . . . .	207
1.84 Upsampler . . . . .	208
1.85 Matched Filter . . . . .	210



## 1.1 ADC

<b>Header File</b>	: adc_*.h
<b>Source File</b>	: adc_*.cpp
<b>Version</b>	: 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	<i>inf</i>
maxValue	volts	double	any	1.5
minValue	volts	double	any	–1.5

Table 1.1: 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**

### **Suggestions for future improvement**



## 1.2 Add

<b>Header File</b>	:	add.h
<b>Source File</b>	:	add.cpp
<b>Version</b>	:	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)

### 1.3 Arithmetic Encoder

<b>Header File</b>	: arithmetic_encoder.h
<b>Source File</b>	: arithmetic_encoder.cpp
<b>Version</b>	: 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 1.2: 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**

**Suggestions for future improvement**

## 1.4 Arithmetic Decoder

<b>Header File</b>	: arithmetic_decoder.h
<b>Source File</b>	: arithmetic_decoder.cpp
<b>Version</b>	: 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 1.3: 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**

### **Suggestions for future improvement**

## 1.5 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**

**Suggestions for future improvement**

## 1.6 Alice QKD

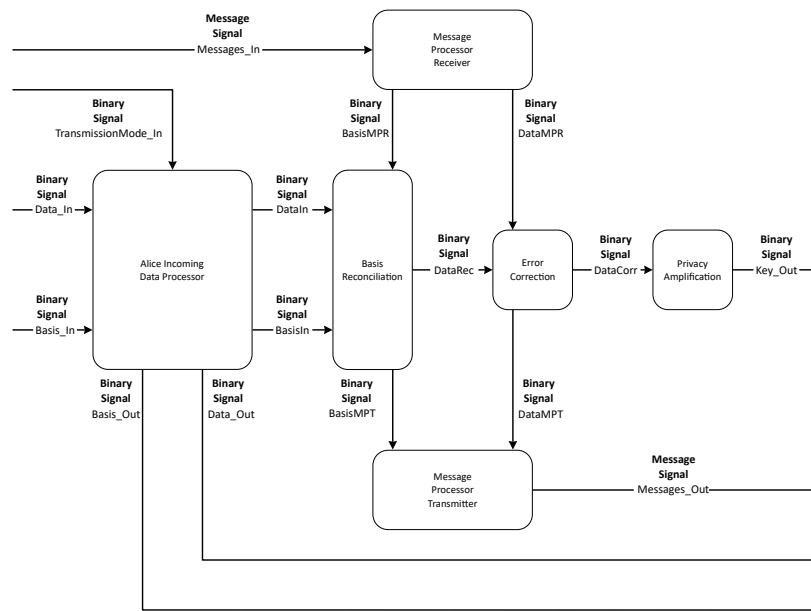
This block is the main processor for Alice.

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 *BasisReconciliation* 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**

### **Suggestions for future improvement**

## 1.7 Ascii Source

<b>Header File</b>	: ascii_source_*.h
<b>Source File</b>	: ascii_source_*.cpp
<b>Version</b>	: 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 1.4: 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

## 1.8 Ascii To Binary

<b>Header File</b>	: ascii_to_binary_*.h
<b>Source File</b>	: ascii_to_binary_*.cpp
<b>Version</b>	: 20180905 (André Mourato)

### Methods

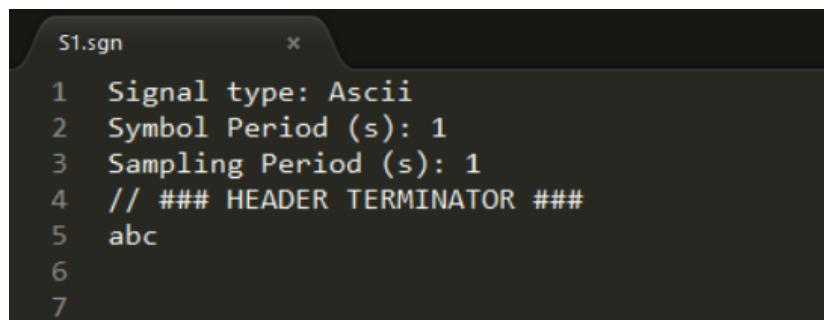
```
AsciiToBinary(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

```
void initialize(void);
```

```
bool runBlock(void);
```

### Functional description

Figure 1.6 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 1.5, is the concatenation of the previous binary codes. The full binary sequence is 0110000111000101100011.



```

S1.sgn
1  Signal type: Ascii
2  Symbol Period (s): 1
3  Sampling Period (s): 1
4  // ### HEADER TERMINATOR ###
5  abc
6
7

```

Figure 1.1: Ascii signal passed as input to the AsciiToBinary block

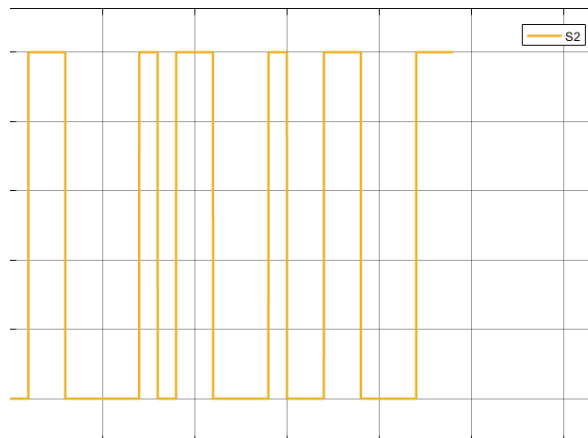


Figure 1.2: Resulting Binary signal from the output of the AsciiToBinary block

### Input Signals

**Number:** 1

**Type:** Ascii

### Output Signals

**Number:** 1

**Type:** Ascii

## 1.9 Balanced Beam Splitter

<b>Header File</b>	: balanced_beam_splitter.h
<b>Source File</b>	: balanced_beam_splitter.cpp
<b>Version</b>	: 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,

$$M_{BS} = \frac{\sqrt{2}}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (1.1)$$

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \begin{bmatrix} A \\ B \end{bmatrix} \quad (1.2)$$

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

## 1.10 Bit Error Rate

<b>Header File</b>	:	bit_error_rate_*.h
<b>Source File</b>	:	bit_error_rate_*.cpp
<b>Version</b>	:	20180815
<b>Contributors</b>	:	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	$1 \times 10^{-10}$	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.

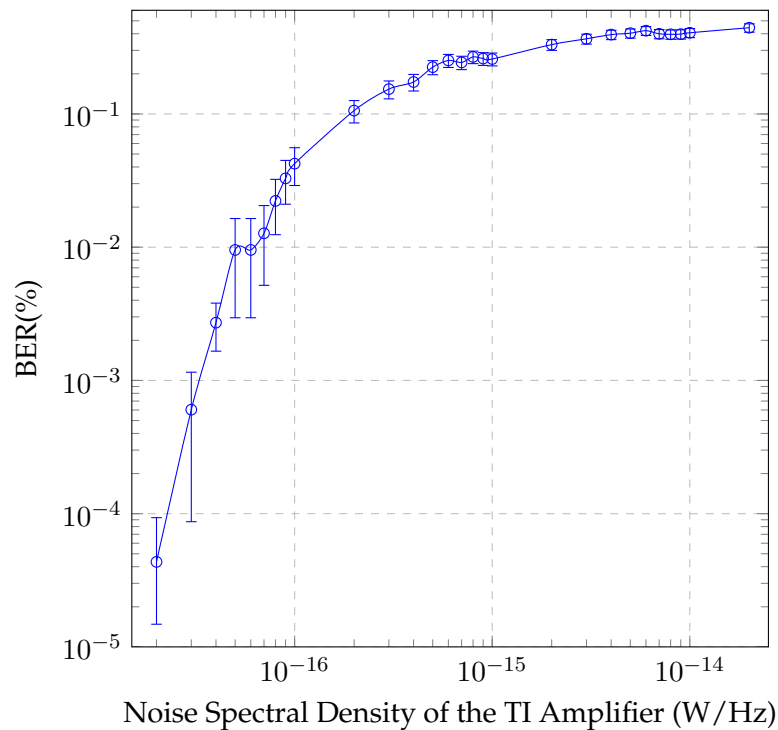
## Methods

- 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

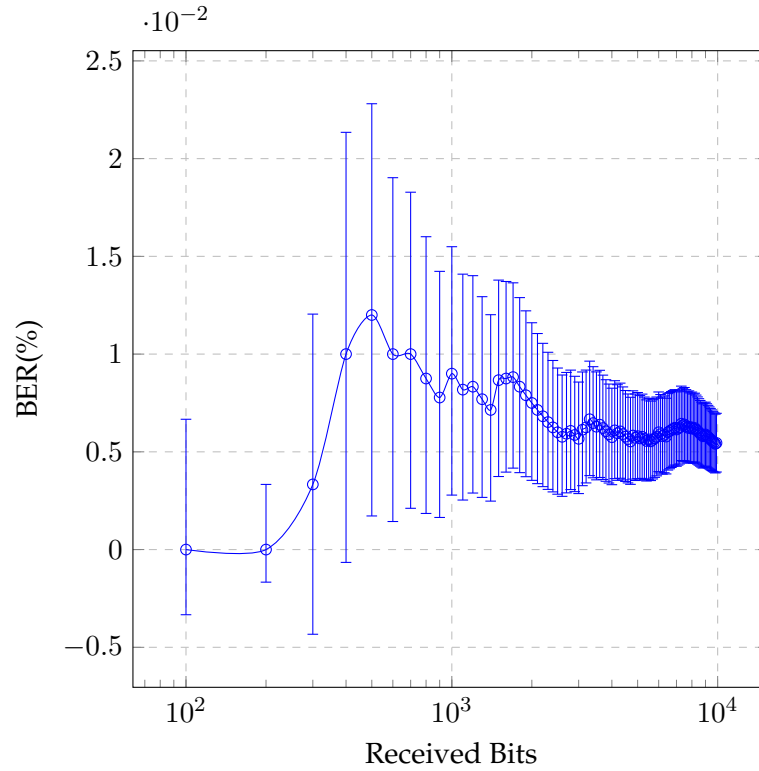
- One interesting test we can do with this block, is analyse how a certain parameter of the system affects de BER. In the following graph we demonstrate an example of how the Noise Spectral Density of the TI Amplifier on the receiver affects the BER:



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:

$$\widehat{\text{BER}} = 1 - \frac{K}{N_{bits}}. \quad (1.3)$$

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

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

$$BER_{UB} = \widehat{BER} + \frac{z_{\alpha/2}}{\sqrt{N_{bits}}} \sqrt{\widehat{BER}(1 - \widehat{BER})} + \frac{1}{3N_{bits}} \left[ 2 \left( \frac{1}{2} - \widehat{BER} \right) z_{\alpha/2}^2 + 2 - \widehat{BER} \right] \quad (1.4)$$

$$BER_{LB} = \widehat{BER} - \frac{z_{\alpha/2}}{\sqrt{N_{bits}}} \sqrt{\widehat{BER}(1 - \widehat{BER})} + \frac{1}{3N_{bits}} \left[ 2 \left( \frac{1}{2} - \widehat{BER} \right) z_{\alpha/2}^2 - 1 - \widehat{BER} \right] \quad (1.5)$$

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  $\widehat{BER}$ , 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.

## 1.11 Binary Source

<b>Header File</b>	: binary_source_*.h
<b>Source File</b>	: binary_source_*.cpp
<b>Version</b>	: 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       | 3. DeterministicCyclic      | 5. AsciiFileAppendZeros |
| 2. PseudoRandom | 4. DeterministicAppendZeros | 6. AsciiFileCyclic      |

### Signals

<b>Number of Input Signals</b>	0
<b>Type of Input Signals</b>	-
<b>Number of Output Signals</b>	$\geq$
<b>Type of Output Signals</b>	Binary

Table 1.5: 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 1.6: 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  $1 - \text{probabilityOfZero}$ .

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

**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 1.3 numbered in this order). Some of these require wrap.

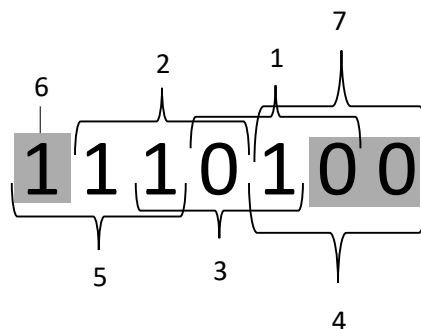


Figure 1.3: Example of a pseudorandom sequence with a pattern length equal to 3.



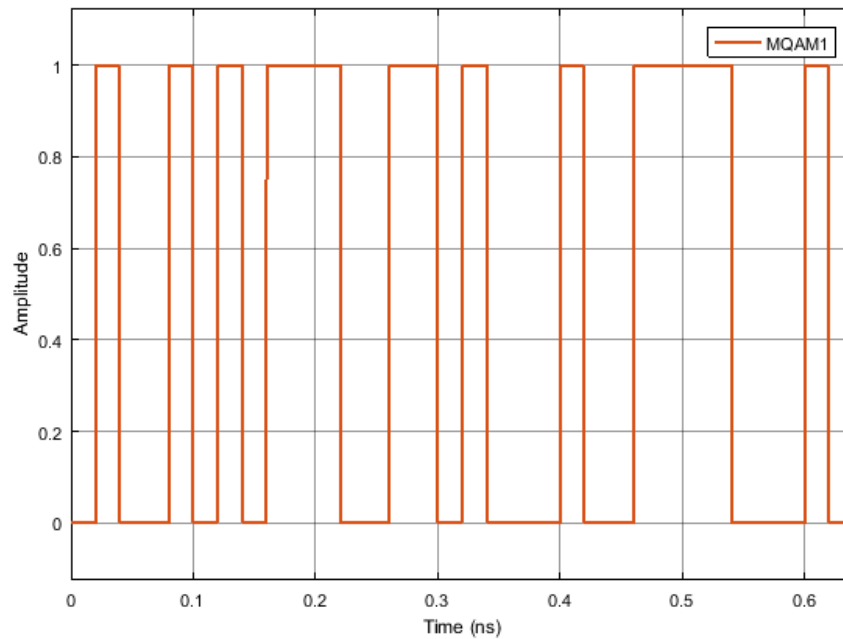


Figure 1.4: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

**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 1.4.

### Suggestions for future improvement

Implement an input signal that can work as trigger.

## 1.12 Binary To Ascii

<b>Header File</b>	: binary_to_ascii_.h
<b>Source File</b>	: binary_to_ascii_.cpp
<b>Version</b>	: 20180905 (André Mourato)

### Methods

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

```
void initialize(void);
```

```
bool runBlock(void);
```

### Functional description

Figure 1.5 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: 0110000111000101100011. 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 1.6, contains the characters that can be represented with the previous binary codes.

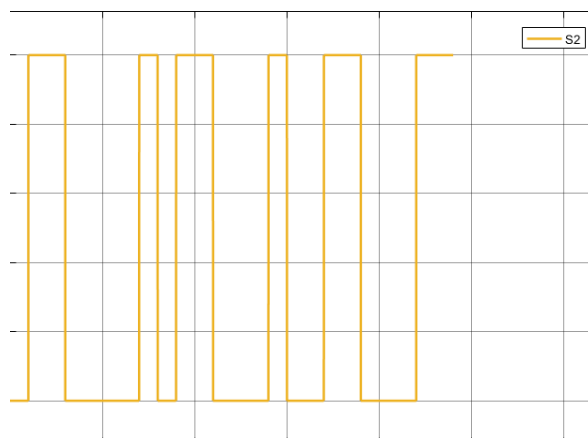
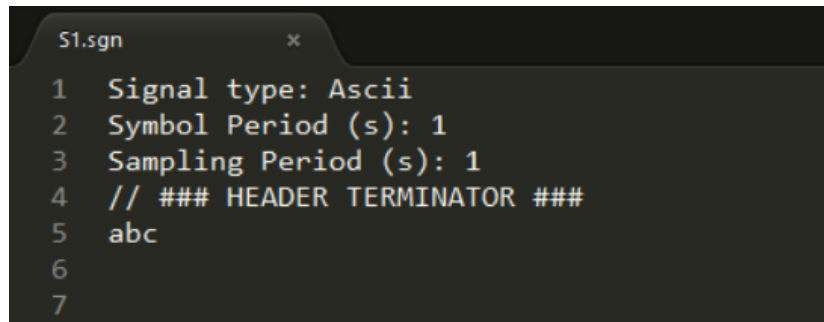


Figure 1.5: Binary signal passed as input to the BinaryToAscii block



```
1 Signal type: Ascii
2 Symbol Period (s): 1
3 Sampling Period (s): 1
4 // ### HEADER TERMINATOR ###
5 abc
6
7
```

Figure 1.6: Resulting Ascii signal from the output of the BinaryToAscii block

### Input Signals

**Number:** 1

**Type:** Ascii

### Output Signals

**Number:** 1

**Type:** Ascii

### 1.13 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

#### Suggestions for future improvement

This block is quantum channel Bob's receiver. This block accepts 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.

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

## Functional description



The mode selection signal that outputs this superblock will inform Alice about which mode she should select, 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** : 1

**Type** : Binary.

### **Examples**

### **Suggestions for future improvement**

## 1.15 Bit Decider

<b>Header File</b>	:	bit_decider.h
<b>Source File</b>	:	bit_decider.cpp
<b>Version</b>	:	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)

## 1.16 Clock

<b>Header File</b>	: clock.h
<b>Source File</b>	: 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 1.7: 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

## Suggestions for future improvement

## 1.17 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

### Suggestions for future improvement

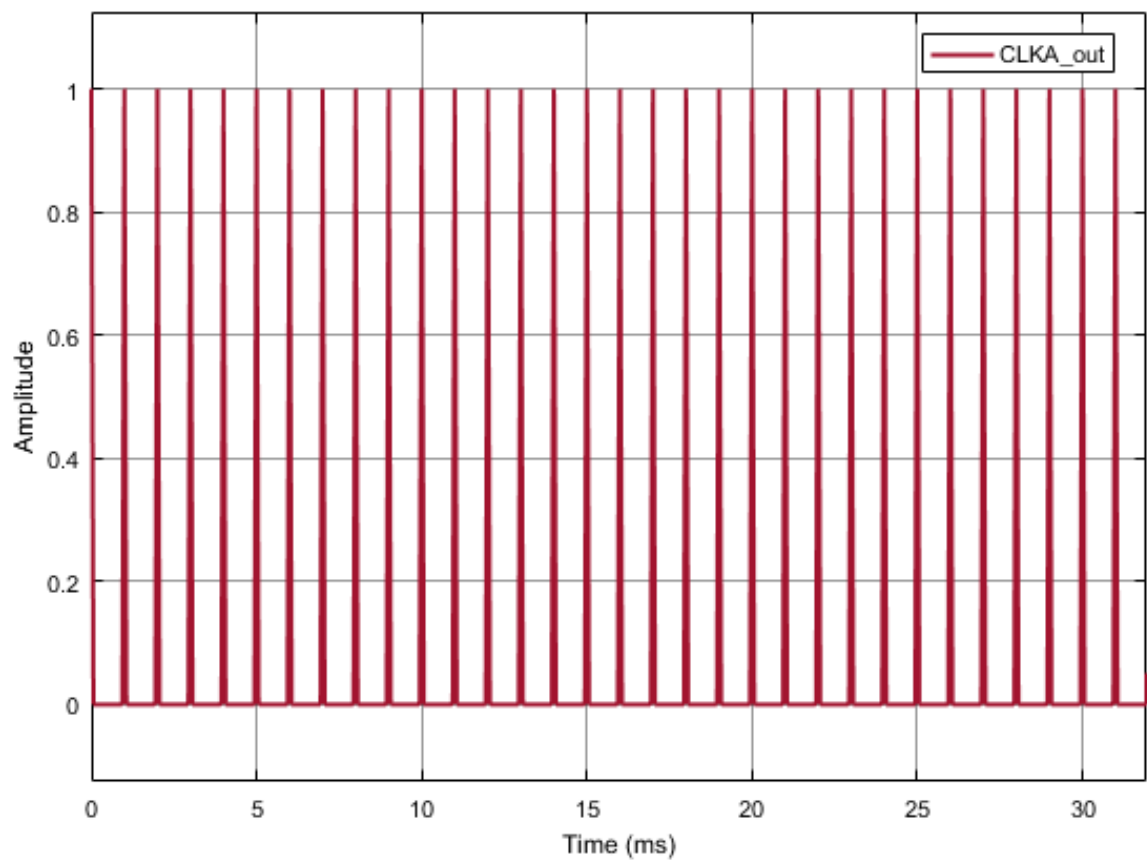


Figure 1.8: Example of the output signal of the clock without phase shift.

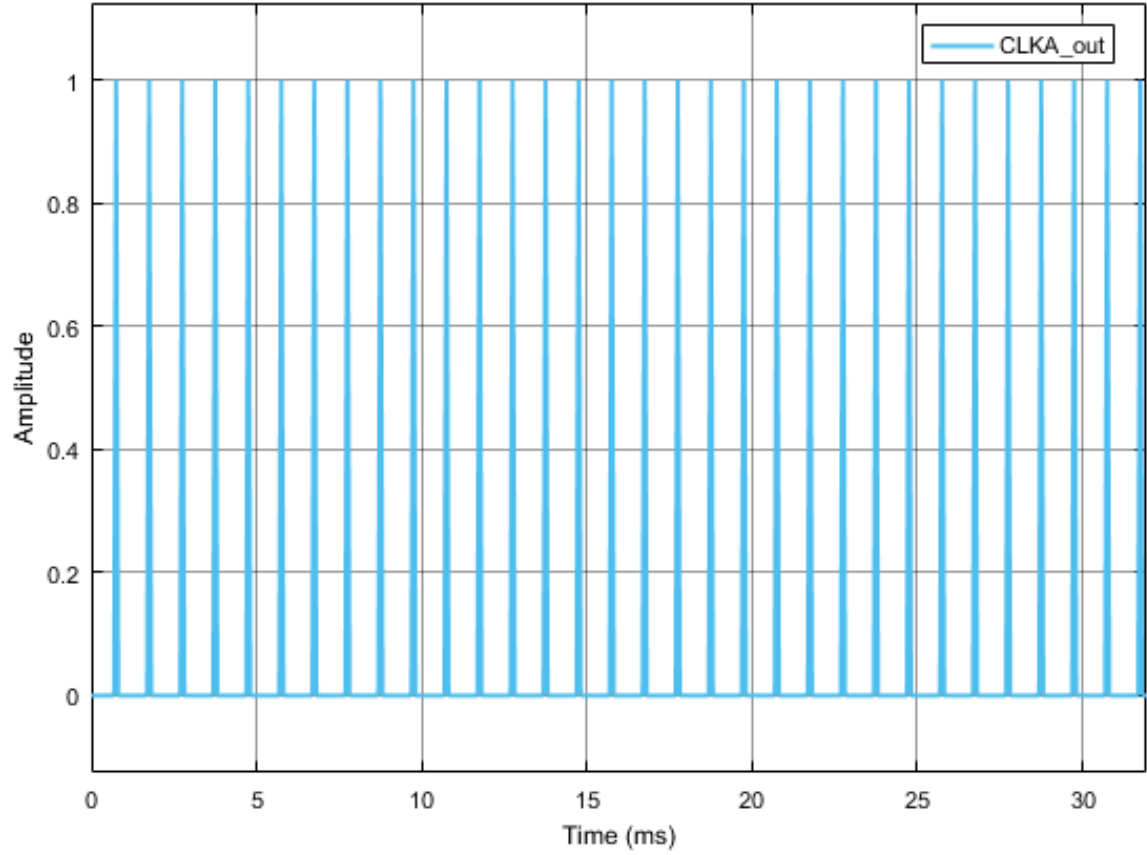


Figure 1.9: Example of the output signal of the clock with phase shift.

## 1.18 Complex To Real

<b>Header File</b>	:	complex_to_real_*.h
<b>Source File</b>	:	complex_to_real_*.cpp
<b>Version</b>	:	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**

### **Suggestions for future improvement**

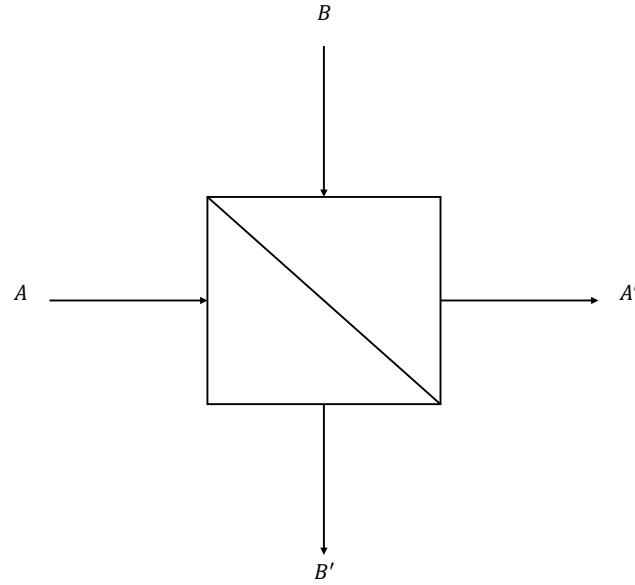


Figure 1.10: 2x2 coupler

### 1.19 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \quad (1.6)$$

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,

$$T = \sqrt{1 - \eta_R} \quad (1.7)$$

$$R = \sqrt{\eta_R} \quad (1.8)$$

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.

## 1.20 Carrier Phase Compensation

<b>Header File</b>	: carrier_phase_estimation_*.h
<b>Source File</b>	: carrier_phase_estimation_*.cpp
<b>Version</b>	: 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 1.8: 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 1.9: 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

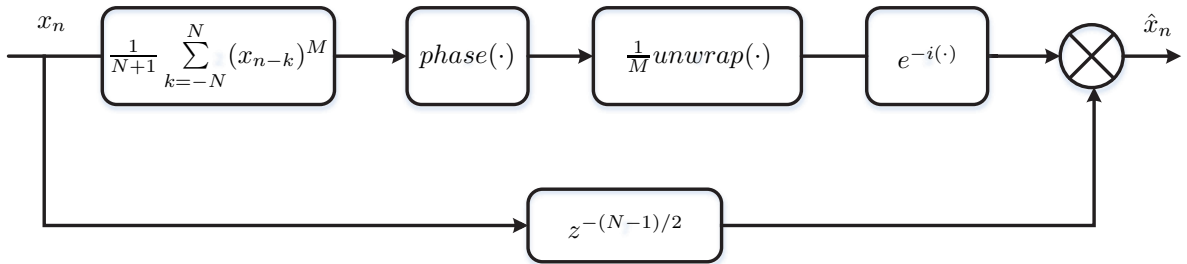


Figure 1.11: Block diagram of Viterbi-Viterbi algorithm for carrier phase recovery.

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 1.11, 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

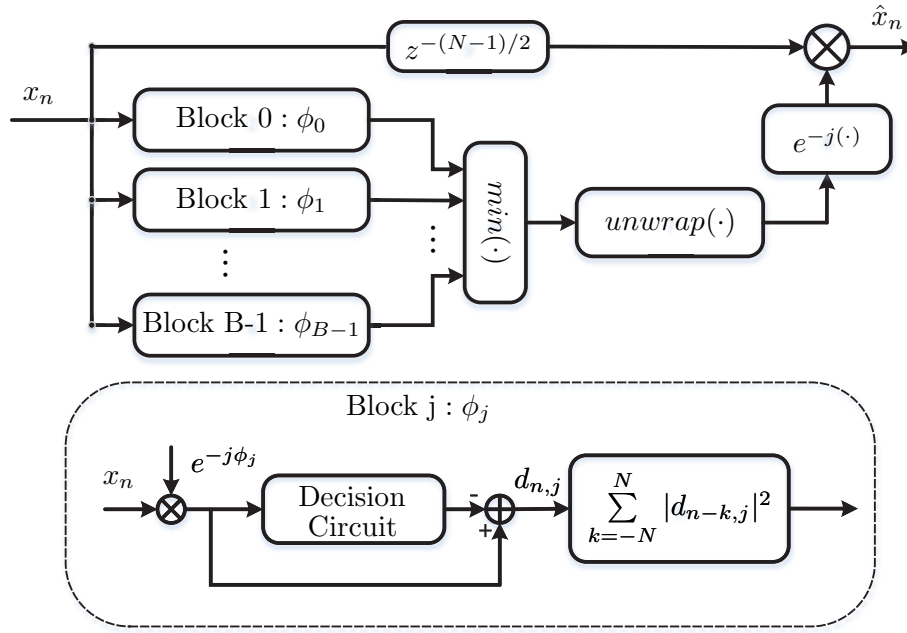


Figure 1.12: Block diagram of blind phase search algorithm for carrier phase recovery.

An alternative to the VV phase noise estimator is the so-called BPS algorithm, in which the operation principle is shown in the Figure 1.12. Firstly, a block of  $2N + 1$  consecutive received symbols is rotated by a number of  $B$  uniformly distributed test phases defined as,

$$\phi_b = \frac{b}{B} \pi, b \in \{0, 1, \dots, B - 1\}. \quad (1.9)$$

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**

### **Suggestions for future improvement**

## 1.21 Decision Circuit

<b>Header File</b>	: decision_circuit_*.h
<b>Source File</b>	: decision_circuit_*.cpp
<b>Version</b>	: 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 1.10: 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**

#### **Suggestions for future improvement**

Extend the decision to higher order modulation format.

## 1.22 Decoder

<b>Header File</b>	: decoder.h
<b>Source File</b>	: 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	$\geq 4$	4
iqAmplitudes	vector<t_complex>	—	{ { 1.0, 1.0 }, { -1.0, 1.0 }, { -1.0, -1.0 }, { 1.0, -1.0 } }

Table 1.11: 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.

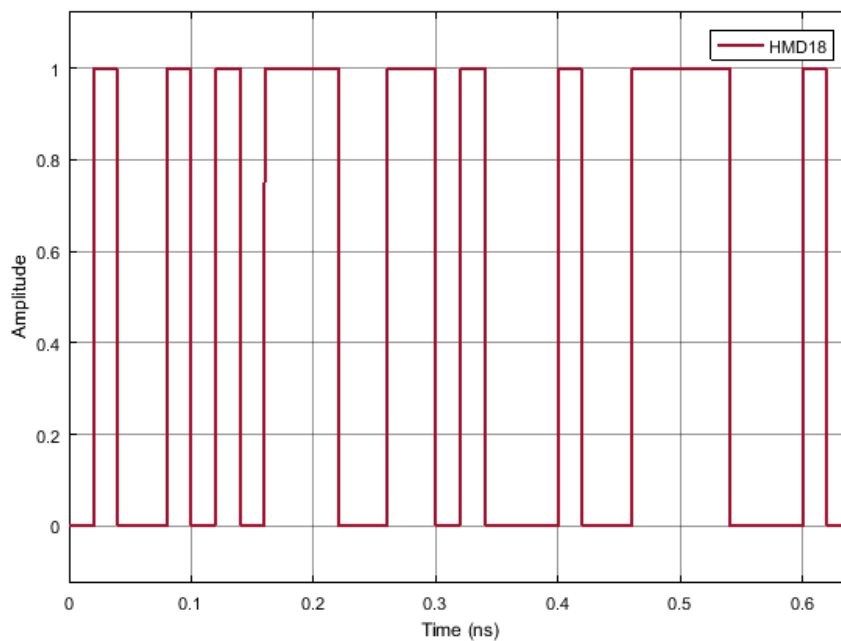


Figure 1.13: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

## Suggestions for future improvement

## 1.23 Discrete To Continuous Time

<b>Header File</b>	: discrete_to_continuous_time.h
<b>Source File</b>	: 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 1.12: 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

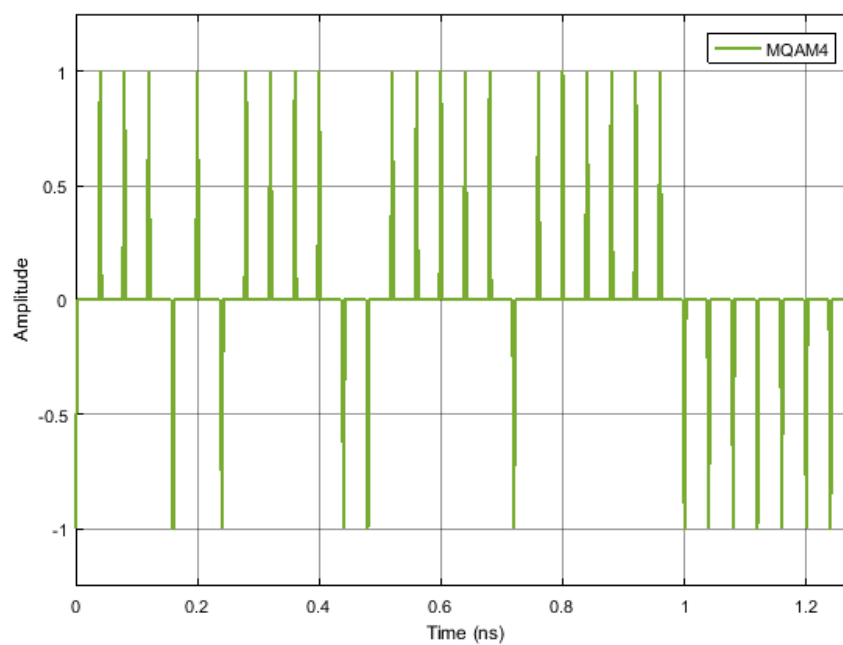


Figure 1.14: Example of the type of signal generated by this block for a binary sequence 0100...

## 1.24 DownSampling

<b>Header File</b>	: down_sampling_*.h
<b>Source File</b>	: down_sampling_*.cpp
<b>Version</b>	: 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 1.13: 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*<sup>th</sup> sample after the first.

### **Input Signals**

**Number:** 1

### **Output Signals**

**Number:** 1

**Type:** Electrical real signal

### **Examples**

### **Suggestions for future improvement**

## 1.25 DSP

<b>Header File</b>	: dsp_*.h
<b>Source File</b>	: dsp_*.cpp
<b>Version</b>	: 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 1.14: 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**

### **Suggestions for future improvement**

## 1.26 EDFA

<b>Header File</b>	: m_qam_receiver.h
<b>Source File</b>	: 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.

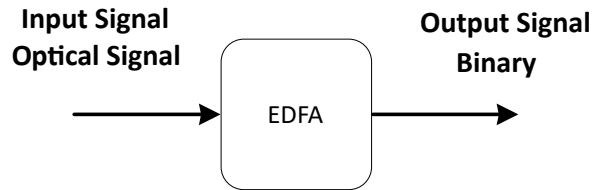


Figure 1.15: Basic configuration of the MQAM receiver

### 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 1.16.

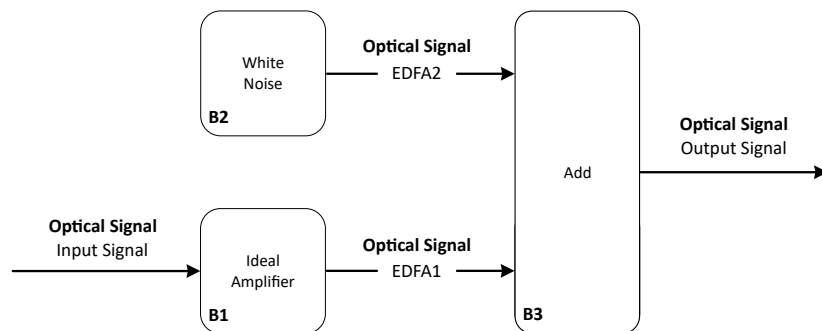


Figure 1.16: Schematic representation of the block homodyne receiver.

### 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 1.15: 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**

**Suggestions for future improvement**



## 1.27 Electrical Signal Generator

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

### 1.27.1 ContinuousWave

Continuous Wave the function of the desired signal. This must be introduced 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  $1 \times \text{gain}$ .

#### Input Parameters

- ElectricalSignalFunction (ContinuousWave) signalFunction
- 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**

#### **Suggestions for future improvement**

Implement other functions according to the needs.

## 1.28 Entropy Estimator

<b>Header File</b>	:	entropy_estimator_*.h
<b>Source File</b>	:	entropy_estimator_*.cpp
<b>Version</b>	:	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)

## 1.29 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

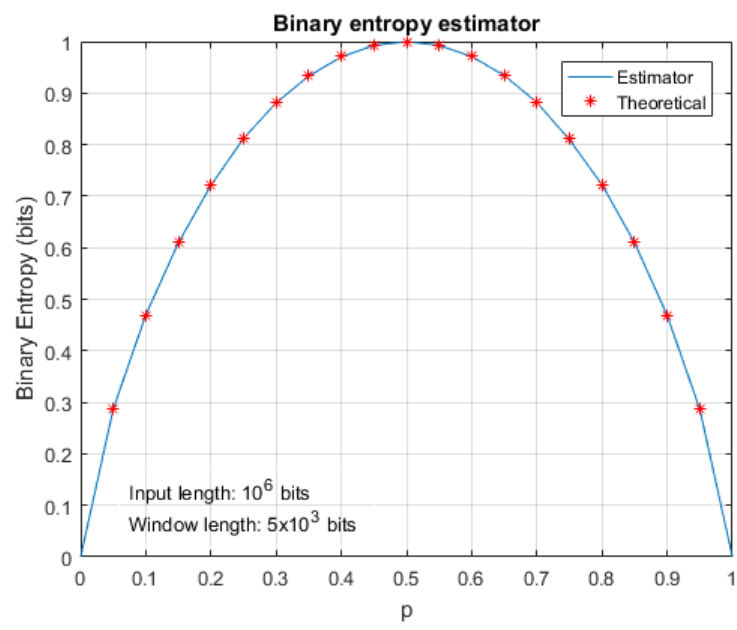
**Type:** Binary Stream

### 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



## 1.30 Fork

<b>Header File</b>	:	fork_20171119.h
<b>Source File</b>	:	fork_20171119.cpp
<b>Version</b>	:	20171119 ( <b>Student Name:</b> 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.

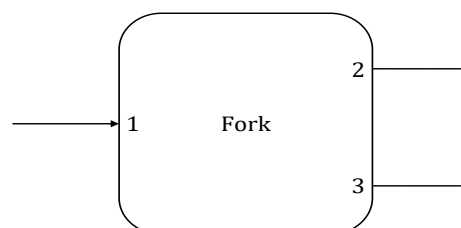


Figure 1.17: Fork

### 1.31 Gaussian Source

<b>Header File</b>	: gaussian_source.h
<b>Source File</b>	: 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 1.16: 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**

## **Suggestions for future improvement**



## 1.32 Hamming Decoder

<b>Header File</b>	: hamming_decoder_*.h
<b>Source File</b>	: hamming_decoder_*.cpp
<b>Version</b>	: 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

### 1.33 Hamming Encoder

<b>Header File</b>	: hamming_encoder_*.h
<b>Source File</b>	: hamming_encoder_*.cpp
<b>Version</b>	: 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

### 1.34 MQAM Receiver

<b>Header File</b>	: m_qam_receiver.h
<b>Source File</b>	: 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) outputting a binary signal. A simplified schematic representation of this block is shown in figure 1.18.

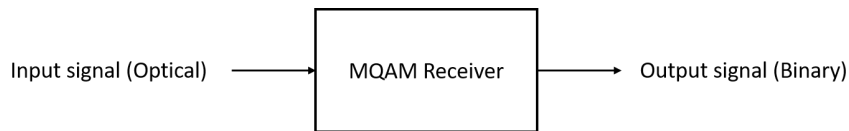


Figure 1.18: Basic configuration of the MQAM receiver

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

It can also be seen from figure 1.19 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*.

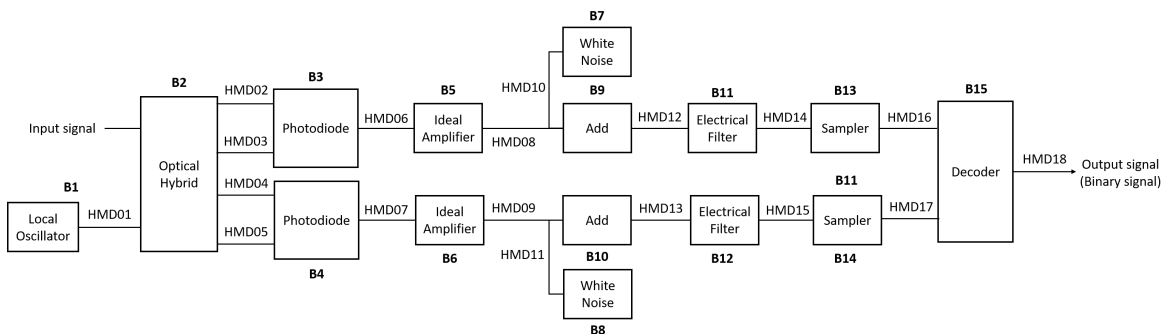


Figure 1.19: Schematic representation of the block homodyne receiver.

### Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 1.35) 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	<b>Example</b> 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 skip	setSamplesToSkip	int(t_integer)	
Save internal signals	setSaveInternalSignals	bool	True or False
Sampling period	setSamplingPeriod	double	Given by $symbolPeriod / samplesPerSymbol$

Table 1.17: 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**

**Suggestions for future improvement**

### 1.35 Huffman Decoder

<b>Header File</b>	:	huffman_decoder_*.h
<b>Source File</b>	:	huffman_decoder_*.cpp
<b>Version</b>	:	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 1.18: 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

## 1.36 Huffman Encoder

<b>Header File</b>	:	huffman_encoder_*.h
<b>Source File</b>	:	huffman_encoder_*.cpp
<b>Version</b>	:	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 1.19: 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



### 1.37 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	$1 \times 10^4$

Table 1.20: 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**

### **Suggestions for future improvement**

## 1.38 IQ Modulator

<b>Header File</b>	: iq_modulator.h
<b>Source File</b>	: iq_modulator.cpp
<b>Source File</b>	: 20180130
<b>Source File</b>	: 20180828 (Romil Patel)

### Version 20180130

This block accepts one input 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 1.21: 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  $\frac{1}{2}\sqrt{\text{outputOpticalPower}}$  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 transmitted to the receptor. The binary signal is sent to the Bit Error Rate (BER) measurement block.

## Input Signals

**Number** : 2

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

## Output Signals

**Number** : 1 or 2

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

## Example

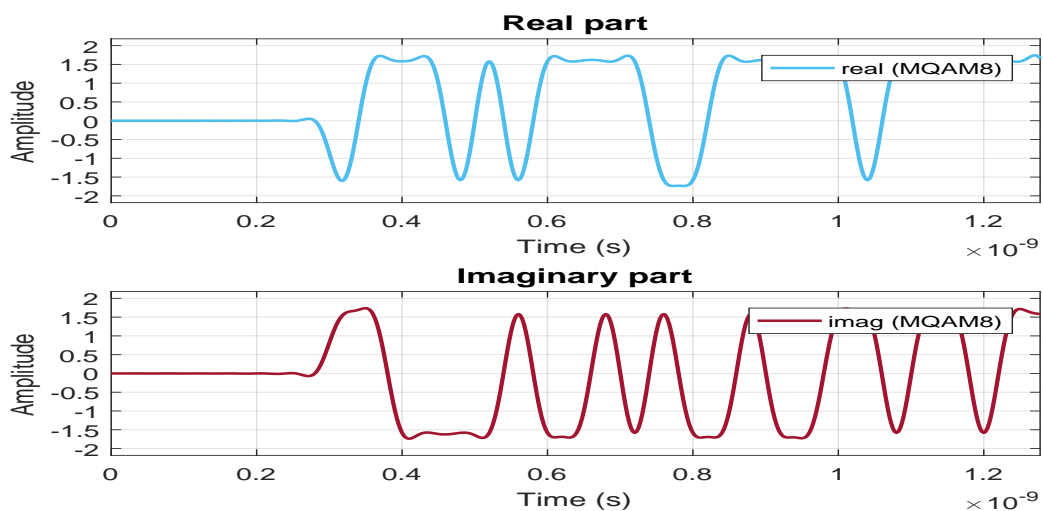


Figure 1.20: Example of a signal generated by this block for the initial binary signal 0100...

**Version 20180828**

**Input Parameters:**

—NA—

**Input Signals:**

**Number:** 1, 2, 3

**Type:** RealValue

**Output Signals:**

**Number:** 4

**Type:** RealValue

### Functional Description

This block 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.

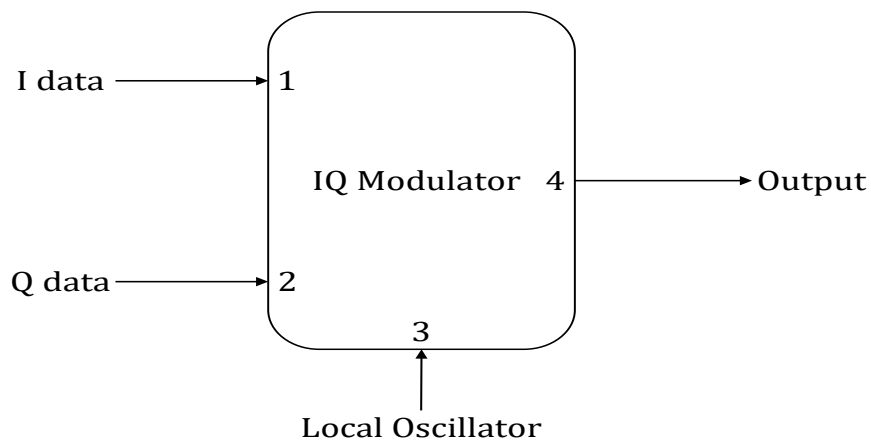


Figure 1.21: IQ Modulator block

### 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,

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

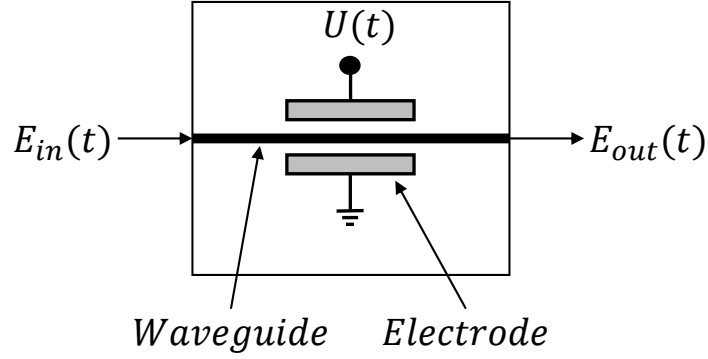


Figure 1.22: Phase Modulator

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  $\pi$ ). The transfer function of the structure can be given as,

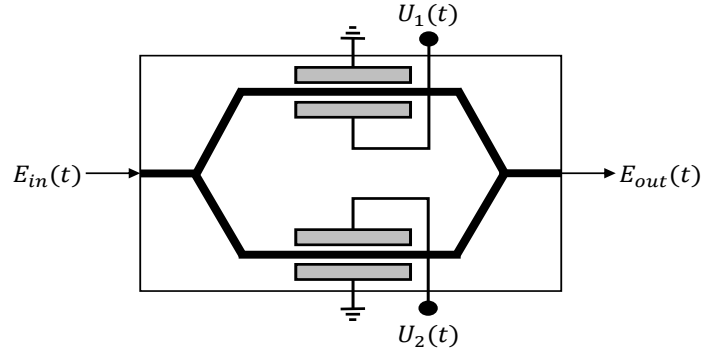


Figure 1.23: Mach-Zehnder Modulator

$$\frac{E_{out}(t)}{E_{in}(t)} = \frac{1}{2} \cdot (e^{j\phi_1(t)} + e^{j\phi_2(t)}) \quad (1.10)$$

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

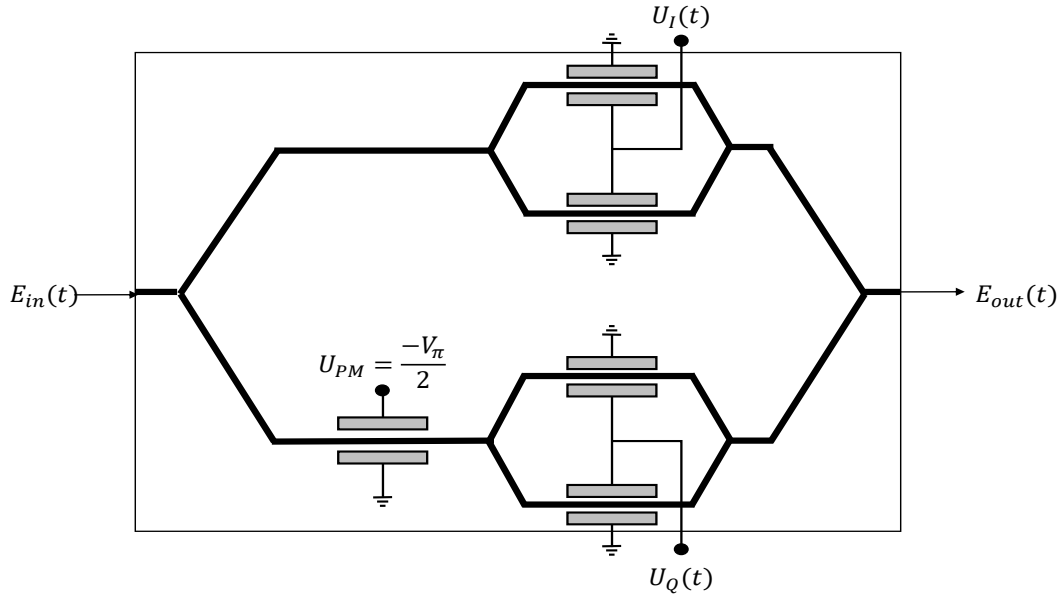


Figure 1.24: IQ Mach-Zehnder Modulator

function of the IQ MZM can be written as,

$$E_{out}(t) = \frac{1}{2} E_{in}(t) \left[ \cos\left(\frac{\pi U_I(t)}{2V_\pi}\right) + j \cdot \cos\left(\frac{\pi U_Q(t)}{2V_\pi}\right) \right] \quad (1.11)$$

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

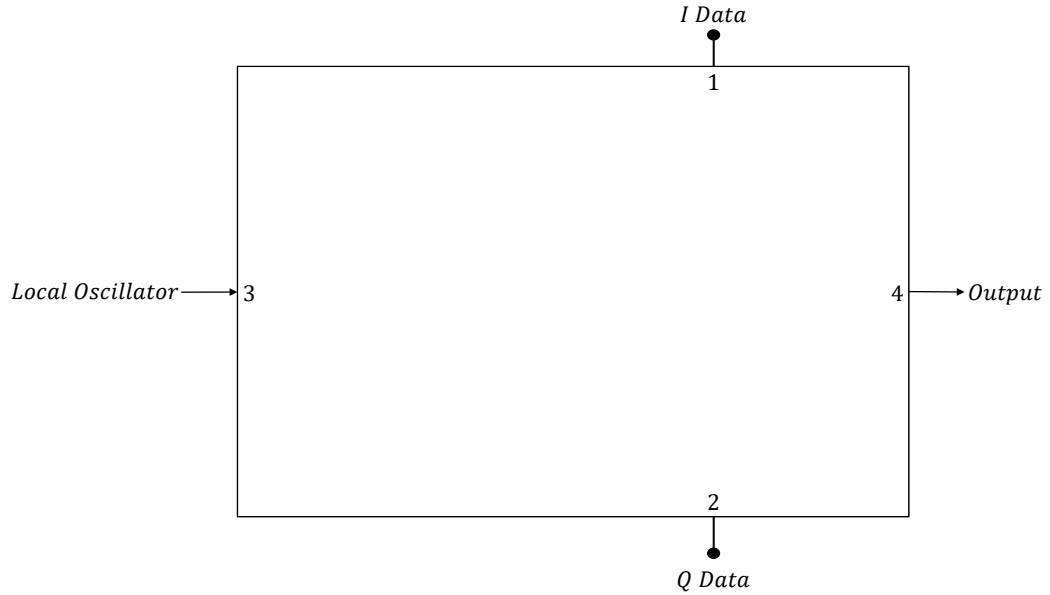


Figure 1.25: Simulation model of the IQ Mach-Zehnder Modulator

### 1.39 IIR Filter

<b>Header File</b>	: iir_filter_*.h
<b>Source File</b>	: iir_filter_*.cpp
<b>Version</b>	: 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



## 1.40 Local Oscillator

<b>Header File</b>	: local_oscillator.h
<b>Source File</b>	: local_oscillator.cpp
<b>Version</b>	: 20180130
<b>Version</b>	: 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	$\text{SPEED\_OF\_LIGHT} / \text{outputOpticalWavelength}$
phase	double	$\in [0, \frac{\pi}{2}]$	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 1.22: 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 wavelength);  
  
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

## 1.41 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	$\text{SPEED\_OF\_LIGHT} / \text{outputOpticalWavelength}$
phase0	double	$\in [0, \frac{\pi}{2}]$	0
samplingPeriod	double	any	0.0
laserLW	double	any	0.0
laserRIN	double	any	0.0

Table 1.23: 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 wavelength);
```

```
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**

**Suggestions for future improvement**

## 1.42 MS Windows IP Tunnel

<b>Header File</b>	: ms_windows_ip_tunnel_*.h
<b>Source File</b>	: ms_windows_ip_tunnel_*.cpp
<b>Version</b>	: 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.

### 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.
signalType	int	any	0	Type of signal.
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 1.24: MS Windows IP Tunnel input parameters

### Methods

IP Tunnel(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.

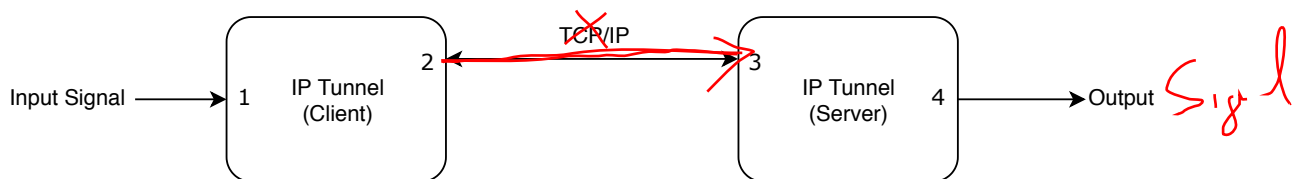


Figure 1.26: MS Windows IP Tunnel block

### 1.43 Mutual Information Estimator

<b>Header File</b>	: mutual_information_estimator_20180723.h
<b>Source File</b>	: 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  $x_j$  ( $j = 1, 2, \dots, J$ ) 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:

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^K \sum_{j=1}^J P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \quad (1.12)$$

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:

$$I(X; Y) = H(Y) - H(Y|X). \quad (1.13)$$

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  $\alpha$  probability calculated in this block. Furthermore, another probability of interest is  $p$  which corresponds to the error probability of the channel. Both  $\alpha$  and  $p$  are estimated in this block.

In order to calculate the mutual information, from equation 1.31 we should calculate the conditional entropy  $H(Y|X)$  and the entropy of the channel outputs  $H(Y)$ . First, lets calculate  $H(Y|X)$ :

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^K \sum_{j=1}^J P(y_k|x_j) P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) + \bar{\alpha}(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= H(p), \end{aligned} \quad (1.14)$$



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  $P(Y = 1)$ .

$$\begin{aligned} P(Y = 0) &= P(Y = 0|X = 0)P(X = 0) + P(Y = 0|X = 1)P(X = 1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \quad (1.15)$$

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

$$\begin{aligned} P(Y = 1) &= P(Y = 1|X = 1)P(X = 1) + P(Y = 1|X = 0)P(X = 0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y = 0). \end{aligned} \quad (1.16)$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \quad (1.17)$$

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

$$I(X; Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \quad (1.18)$$

The upper and lower bounds,  $I(X; Y)_{UB}$  and  $I(X; Y)_{LB}$  respectively, are calculated using the method of Coppler-Pearson as described in section 1.10 for bit error rate calculation. This way, it returns the simplified expression:

$$I(X; Y)_{UB} = I(X; Y) + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[ 2 \left( \frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 + (2 - I(X; Y)) \right] \quad (1.19)$$

$$I(X; Y)_{LB} = I(X; Y) - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[ 2 \left( \frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 - (1 + I(X; Y)) \right], \quad (1.20)$$

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  $X = 0$ ,  $\alpha$ . 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,  $\alpha$ , 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  $\alpha$  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.

## 1.44 MQAM Mapper

<b>Header File</b>	: m_qam_mapper.h
<b>Source File</b>	: 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 1.25: 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 attributes 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 illustrated in figure 1.27.

### Input Signals

**Number** : 1

**Type** : Binary (DiscreteTimeDiscreteAmplitude)

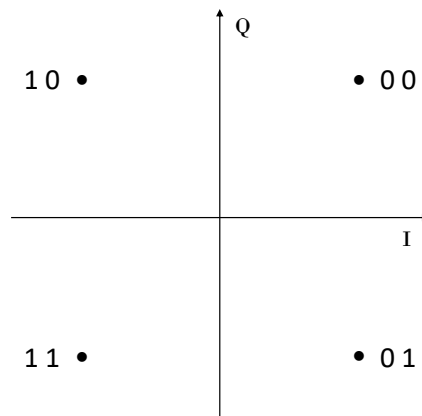


Figure 1.27: Constellation used to map the signal for  $m=4$

### Output Signals

**Number** : 2

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

### Example

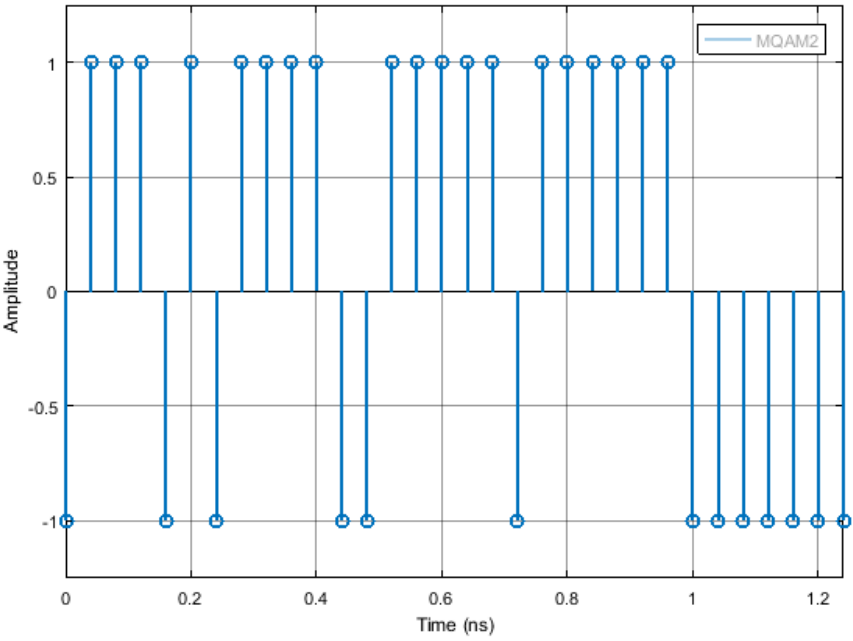


Figure 1.28: Example of the type of signal generated by this block for the initial binary signal 0100...

## 1.45 M-QAM Receiver

<b>Header File</b>	: m_qam_receiver.h
<b>Source File</b>	: m_qam_receiver.cpp
<b>Version</b>	: 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 1.29.

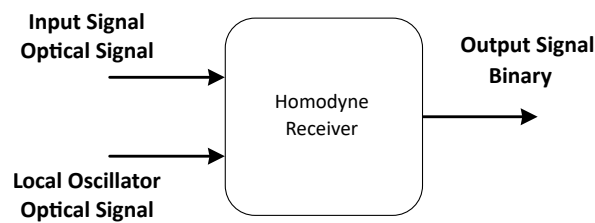


Figure 1.29: Simplified model of the MQAM receiver

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 sBeginningOfImpulseResponse);
- double const **getMFSeeBeginningOfImpulseResponse**(void);

Methods to configure the Samplers:

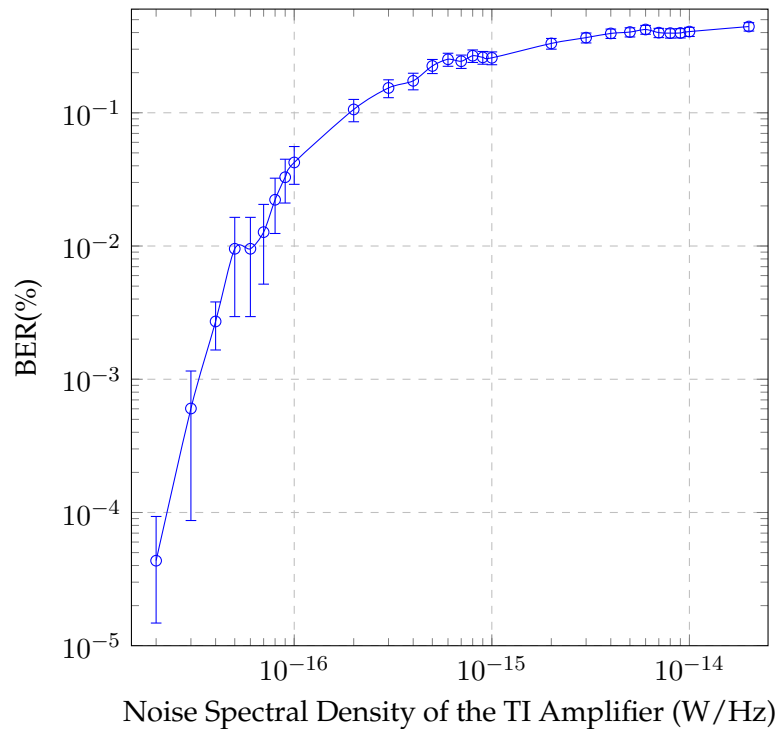
- void **setSamplesToSkip**(int sToSkip);

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:



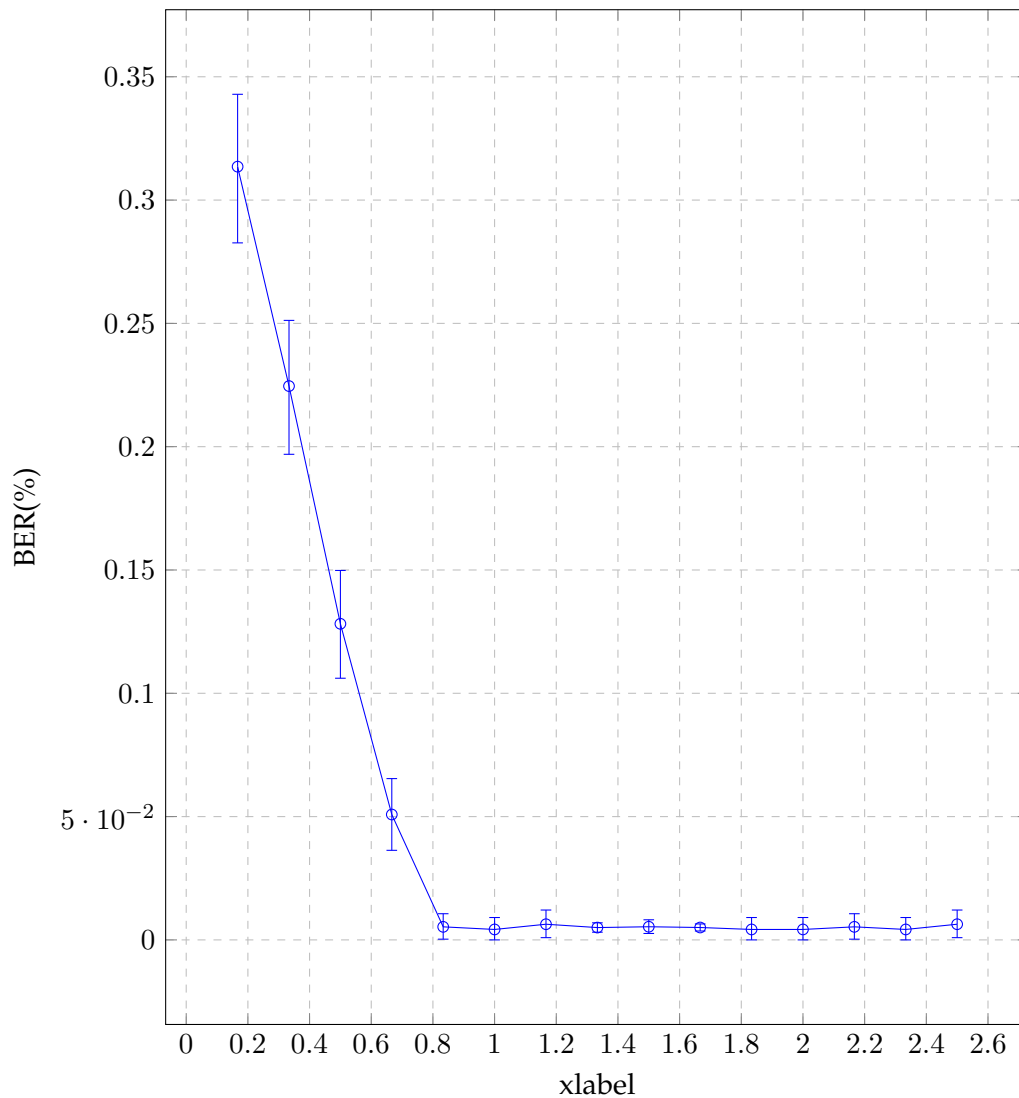
We can observe that the BER decreases at an enlarging rate, as we augment the spectral density of the noise on the TI Amplifier. Giving us an estimate of the sensitivity of the receiver to thermal noise.

– Impact of the bandwidth of the TI Amplifier:

On the following graph we can see the impact of the bandwidth 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 figure 1.30, 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.

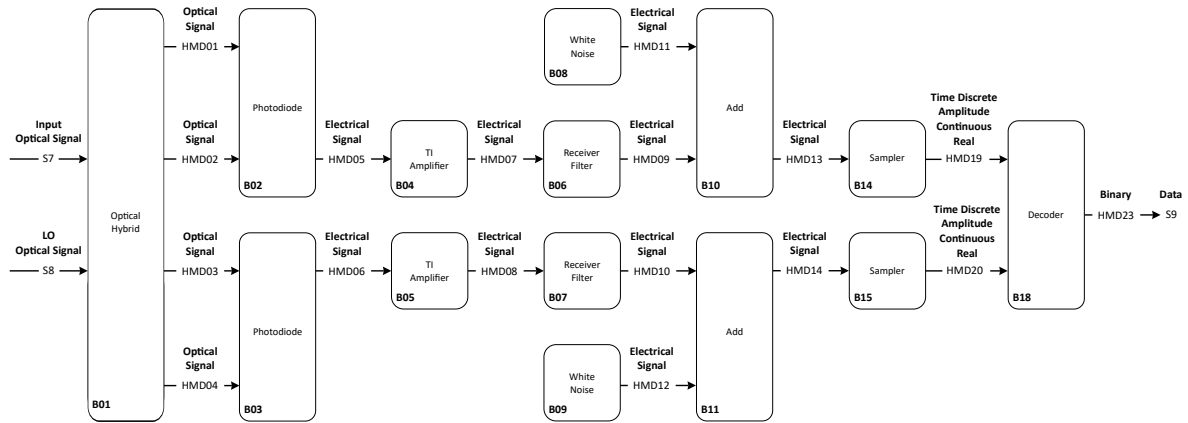


Figure 1.30: Schematic representation of the block homodyne receiver.

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 it 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 to 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 accessed/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.

## 1.46 MQAM Transmitter

<b>Header File</b>	: m_qam_transmitter.h
<b>Source File</b>	: m_qam_transmitter.cpp
<b>Contributors</b>	: 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 1.31.

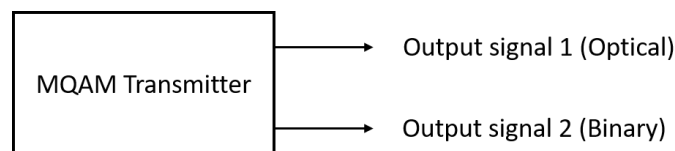


Figure 1.31: Basic configuration of the MQAM transmitter

### 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 1.36). The binary signal generated in the internal block Binary Source (block B1 in figure 1.36) 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 1.36).

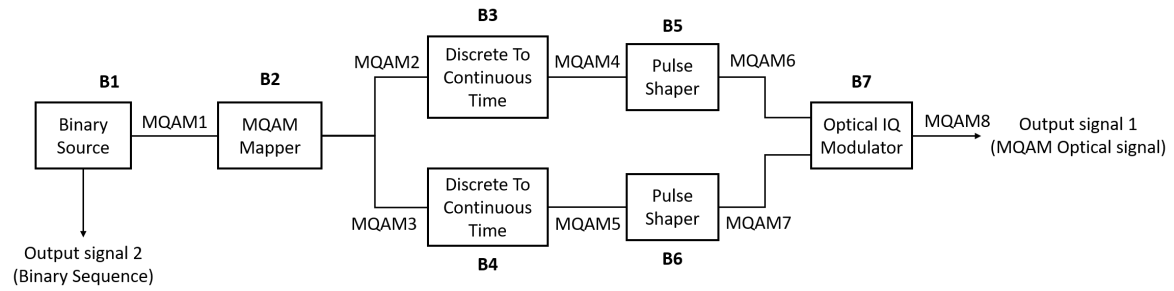


Figure 1.32: Schematic representation of the block MQAM transmitter.

### 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 1.26: 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 1.27: 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 impResponseTimeLength);
- 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:

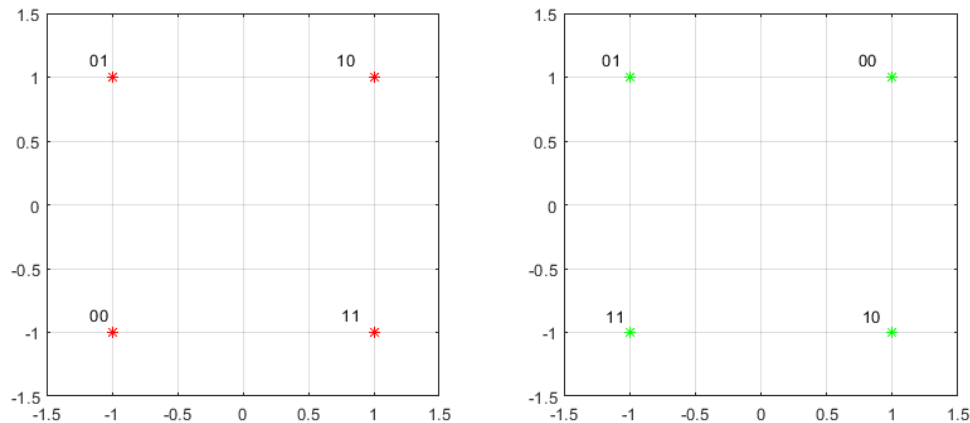


Figure 1.33: Two different constellations for QPSK.

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.

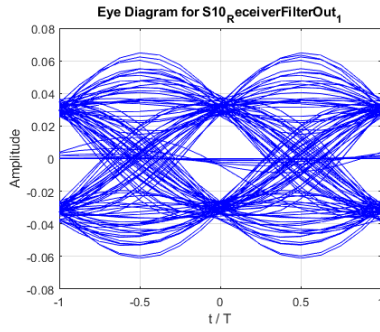
This happens because, in the presence of noise, the symbols can be decoded to adjacent symbols. And that obviously has an effect on the BER. What we can see here is how the Symbol Error Rate (SER) affects the BER. It's obvious that, once the only configuration that was changed was the constellation, the SER is constant for both cases.

So, what do these two constellations have different to cause different measured BER?

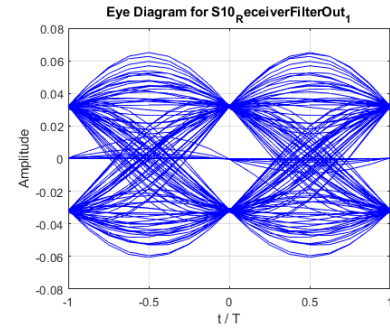
The difference is that, the constellation on the right has Grey encoding, meaning that adjacent symbols only differ in one bit. In this way, a misevaluated symbol will only result in one out of two bits wrong, thus minimizing the BER of a system.

- 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



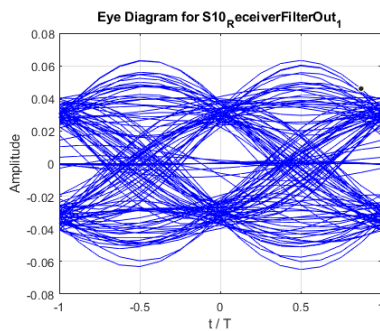
(b) Eye diagram - Root Raised Cosine

Figure 1.34: Eye diagrams without noise

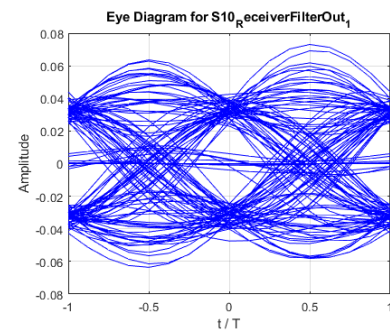
In the figure 1.34a we have the eye diagram of the received signal, whose pulse was shaped with a Raised Cosine and the matched filter at the receiver had the same filter shape. The result is not what we are expecting, because the Raised Cosine pulse shape is known for having null ISI, but we can clearly 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 1.34b.

Even though that it is clear to see the differences between figures 1.34a and 1.34b, 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 1.35b.



(a) Eye diagram - Raised Cosine with noise



(b) Eye diagram - Root Raised Cosine with noise

Figure 1.35: 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 1.36.

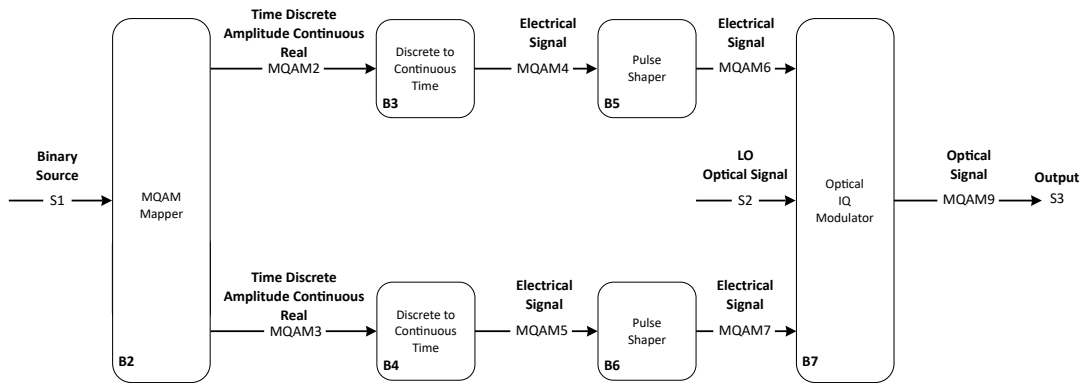


Figure 1.36: Schematic representation of the block MQAM transmitter.

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.

## Suggestions 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.

## 1.47 Netxpto

<b>Header File</b>	:	netxpto.h
	:	netxpto_20180118.h
	:	netxpto_20180418.h
<b>Source File</b>	:	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	

### 1.47.1 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.

### 1.47.2 Version 20180418

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

### Suggestions for future improvement

## 1.48 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**

**Suggestions for future improvement**

## 1.49 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 attributes for each pair of bits a point in space. In this case, it is be considered four possible polarization states:  $0^\circ$ ,  $45^\circ$ ,  $90^\circ$  and  $135^\circ$ .

### 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

## 1.50 Probability Estimator

This block 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  $\Omega$  associated with a random experience and  $A$  an event such that  $P(A) = p \in ]0, 1[$ . Lets  $X : \Omega \rightarrow \mathbb{R}$  such that

$$\begin{aligned} X(\omega) &= 1 && \text{,if } \omega \in A \\ X(\omega) &= 0 && \text{,if } \omega \in \bar{A} \end{aligned} \quad (1.21)$$

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  $P(X = 0)$ ,

$$\begin{aligned} P(X = 1) &= P(A) = p \\ P(X = 0) &= P(\bar{A}) = 1 - p \end{aligned} \quad (1.22)$$

$X$  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  $\text{VAR}(X) = p(1 - p)$  [**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  $np(1 - p)$ .

$$\frac{X - np}{\sqrt{np(1 - p)}} \sim N(0, 1). \quad (1.23)$$

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:

$$\frac{X - np}{\sqrt{n\hat{p}(1 - \hat{p})}} \sim N(0, 1) \quad (1.24)$$

$$\begin{aligned} P\{-z_{\alpha/2} < \frac{X - np}{\sqrt{n\hat{p}(1 - \hat{p})}} < z_{\alpha/2}\} &\approx 1 - \alpha \\ P\{-z_{\alpha/2}\sqrt{n\hat{p}(1 - \hat{p})} < np - X < z_{\alpha/2}\sqrt{n\hat{p}(1 - \hat{p})}\} &\approx 1 - \alpha \\ P\{\hat{p} - z_{\alpha/2}\sqrt{\hat{p}(1 - \hat{p})/n} < p < \hat{p} + z_{\alpha/2}\sqrt{\hat{p}(1 - \hat{p})/n}\} &\approx 1 - \alpha \end{aligned} \quad (1.25)$$

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:

$$\text{Probability}_1 = \frac{\text{Number of 1's}}{\text{Number of Received Bits}} \quad (1.26)$$

$$\text{Probability}_0 = \frac{\text{Number of 0's}}{\text{Number of Received Bits}}. \quad (1.27)$$

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

$$\text{ME} = z_{\text{score}} \times \sqrt{\frac{\hat{p}(1 - \hat{p})}{N}} \quad (1.28)$$

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,

$$\begin{aligned} \text{ME} &= \pm z_{\alpha/2} \frac{\sigma}{\sqrt{N}} \\ \text{ME} &= \pm z_{\alpha/2} \sqrt{\frac{\hat{p}(1 - \hat{p})}{N}}, \end{aligned} \tag{1.29}$$

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**

## 1.51 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

### Suggestions for future improvement

## 1.52 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

### Suggestions for future improvement

### 1.53 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  $0^\circ$ , otherwise, if the value is 1, the rotator is set to rotate the input photon stream by an angle of  $45^\circ$ .

#### Input Signals

**Number** : 2

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

#### Output Signals

**Number** : 1

**Type** : Photon Stream



**Examples**

**Suggestions for future improvement**

## 1.54 Mutual Information Estimator

<b>Header File</b>	: mutual_information_estimator_20180723.h
<b>Source File</b>	: 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  $x_j$  ( $j = 1, 2, \dots, J$ ) 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:

$$\begin{aligned} I(X; Y) &= H(X) - H(X|Y) \\ &= \sum_{k=1}^K \sum_{j=1}^J P(x_j, y_k) \log \frac{P(x_j|y_k)}{P(x_j)}, \end{aligned} \quad (1.30)$$

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:

$$I(X; Y) = H(Y) - H(Y|X). \quad (1.31)$$

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  $\alpha$  probability calculated in this block. Furthermore, another probability of interest is  $p$  which corresponds to the error probability of the channel. Both  $\alpha$  and  $p$  are estimated in this block.

In order to calculate the mutual information, from equation 1.31 we should calculate the conditional entropy  $H(Y|X)$  and the entropy of the channel outputs  $H(Y)$ . First, let's calculate  $H(Y|X)$ :

$$\begin{aligned} H(Y|X) &= \sum_{k=1}^K \sum_{j=1}^J P(y_k|x_j) P(x_j) \log \frac{1}{P(y_k|x_j)} \\ &= \alpha(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) + \bar{\alpha}(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= (\alpha + \bar{\alpha})(\bar{p} \log \frac{1}{\bar{p}} + p \log \frac{1}{p}) \\ &= H(p), \end{aligned} \quad (1.32)$$

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  $P(Y = 1)$ .

$$\begin{aligned} P(Y = 0) &= P(Y = 0|X = 0)P(X = 0) + P(Y = 0|X = 1)P(X = 1) \\ &= \bar{p}\alpha + p\bar{\alpha}. \end{aligned} \quad (1.33)$$

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

$$\begin{aligned} P(Y = 1) &= P(Y = 1|X = 1)P(X = 1) + P(Y = 1|X = 0)P(X = 0) \\ &= p\alpha + \bar{p}\bar{\alpha} \\ &= 1 - P(Y = 0). \end{aligned} \quad (1.34)$$

As a result:

$$H(Y) = H(\bar{p}\alpha + p\bar{\alpha}) \quad (1.35)$$

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

$$I(X; Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p) \quad (1.36)$$

The upper and lower bounds,  $I(X; Y)_{UB}$  and  $I(X; Y)_{LB}$  respectively, are calculated using the method of Coppler-Pearson as described in section 1.10 for bit error rate calculation. This way, it returns the simplified expression:

$$I(X; Y)_{UB} = I(X; Y) + \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[ 2 \left( \frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 + (2 - I(X; Y)) \right] \quad (1.37)$$

$$I(X; Y)_{LB} = I(X; Y) - \frac{1}{\sqrt{N_T}} z_{\alpha/2} \sqrt{I(X; Y)(1 - I(X; Y))} + \frac{1}{3N_T} \left[ 2 \left( \frac{1}{2} - I(X; Y) \right) z_{\alpha/2}^2 - (1 + I(X; Y)) \right], \quad (1.38)$$

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  $X = 0$ ,  $\alpha$ . 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,  $\alpha$ , 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  $\alpha$  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.

## 1.55 Optical Switch

This block has one input signal and two output 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

### Suggestions for future improvement

## 1.56 Optical Hybrid

<b>Header File</b>	: optical_hybrid.h
<b>Source File</b>	: 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  $90^\circ$  in the complex plane. Figure 1.37 shows a schematic representation of this block.

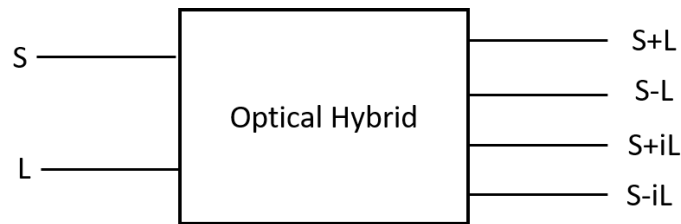


Figure 1.37: Schematic representation of an optical hybrid.

### Input Parameters

Parameter	Type	Values	Default
outputOpticalPower	double	any	$1e - 3$
outputOpticalWavelength	double	any	$1550e - 9$
outputOpticalFrequency	double	any	$\text{SPEED\_OF\_LIGHT} / \text{outputOpticalWavelength}$
powerFactor	double	$\leq 1$	0.5

Table 1.28: 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****Suggestions for future improvement**

## 1.57 Photodiode pair

<b>Header File</b>	: photodiode_old.h
<b>Source File</b>	: photodiode_old.cpp

This block simulates a block of two photodiodes assembled like in figure 1.38. 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 signal corresponds to the output signal of the block.

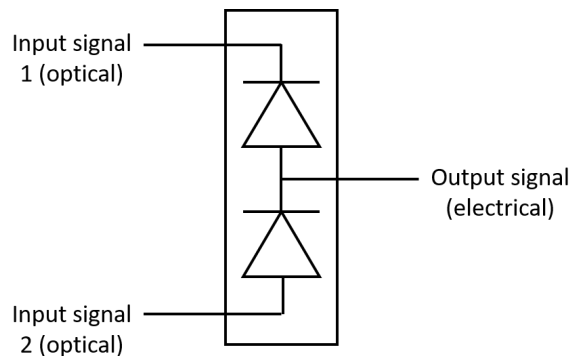


Figure 1.38: Schematic representation of the physical equivalent of the photodiode code block.

### 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

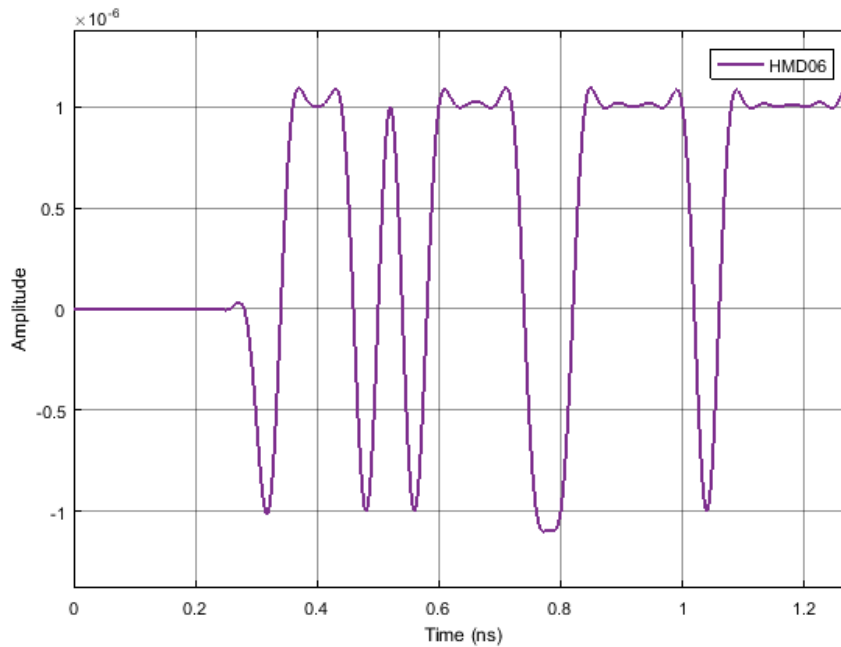


Figure 1.39: Example of the output signal of the photodiode block for a binary sequence 01

**Suggestions for future improvement**

## 1.58 Photoelectron Generator

<b>Header File</b>	: photoelectron_generator_*.h
<b>Source File</b>	: photoelectron_generator_*.cpp
<b>Version</b>	: 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.

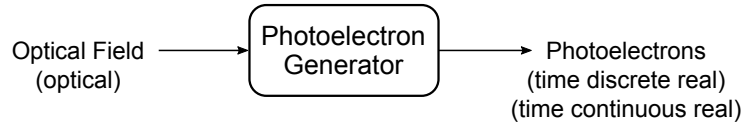


Figure 1.40: Schematic representation of the photoelectron generator code block

### 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  $\eta$ . Given an input beam with an optical power  $P(t)$  in which the photons are around the wavelength  $\lambda$ , the flux of photons  $\phi(t)$  is calculated as [saleh1991]

$$\phi(t) = \frac{P(t)\lambda}{hc} \quad (1.39)$$

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

$$\bar{n}(t) = \int_t^{t+T} \phi(\tau) d\tau \quad (1.40)$$

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

$$p(n) = \frac{\bar{n}^n \exp(-\bar{n})}{n!} \quad (1.41)$$

where  $n = n(t)$  and  $\bar{n} = \bar{n}(t)$ .

For each incident photon, there is a probability  $\eta$  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  $\eta$ . 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]

$$m \sim \text{Poisson}(\eta\bar{n}) \quad (1.42)$$

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 neglected, then the output number of photoelectrons,  $n_e(t)$  in the interval, will be equal to

$$m(t) = \overline{\eta n(t)} \quad (1.43)$$

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  $\overline{\eta n(t)}$

$$m(t) \sim \text{Poisson}(\overline{\eta n(t)}) \quad (1.44)$$

### 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)

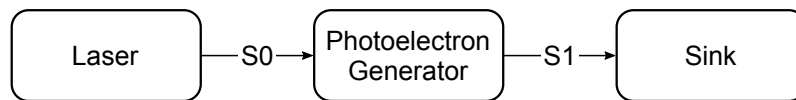
**Examples**

Figure 1.41: 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

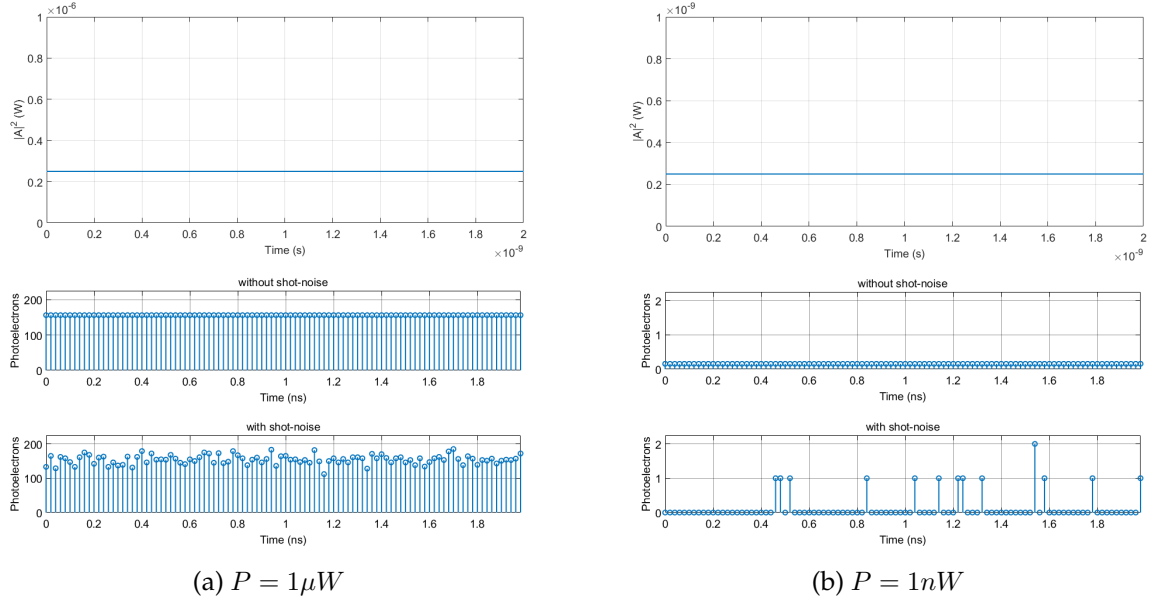


Figure 1.42: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 1.42 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 contrast with the reference result, where only single events are represented, the  $P = 1\mu W$  case shows that all samples account many photoelectrons. Given its 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.

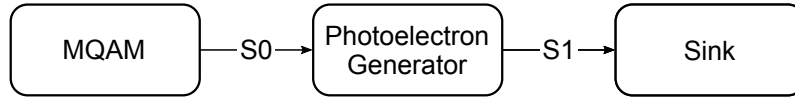


Figure 1.43: Variable power simulation setup

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`.

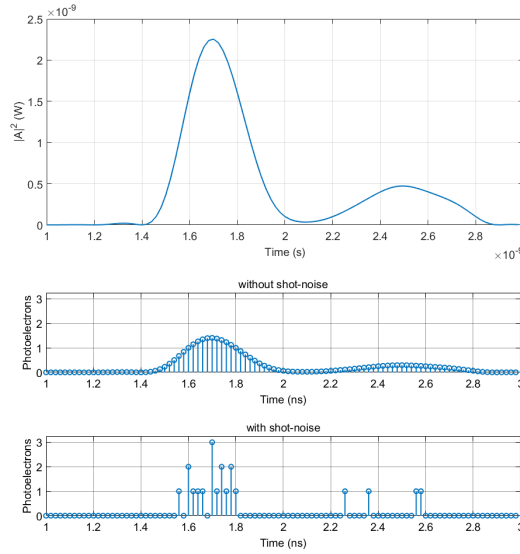


Figure 1.44: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 1.44 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 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.

### Suggestions for future improvement

## 1.59 Power Spectral Density Estimator

<b>Header File</b>	: power_spectral_density_estimator_*.h
<b>Source File</b>	: power_spectral_density_estimator_*.cpp
<b>Version</b>	: 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 TimeContinuousAmplitudeContinuousReal 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

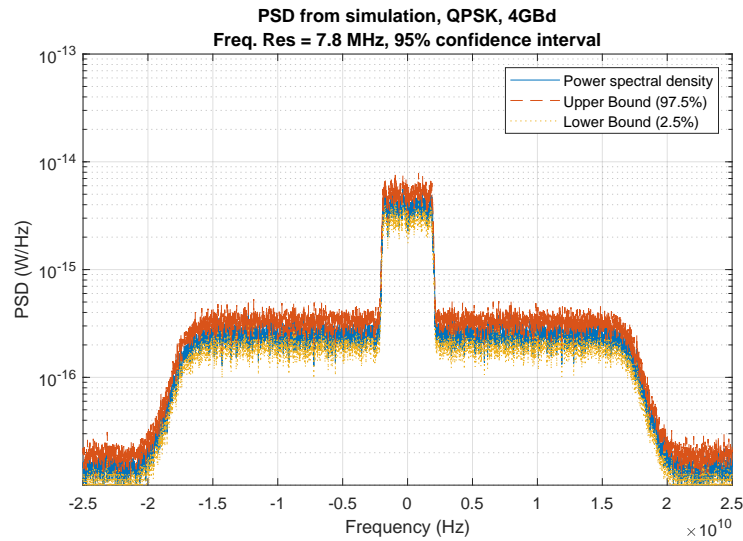


Figure 1.45: Estimated power spectral density for a QPSK signal.

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]:

$$S_{XX}(f) = \lim_{T \rightarrow \infty} E \left[ \hat{S}_{XX}(f, T) \right] \quad (1.45)$$

with

$$\hat{S}_{XX}(f, T) = \frac{|X_T(f)|^2}{T} \quad (1.46)$$

and

$$X_T(f) = \int_0^T x(t) \exp(-j2\pi ft) dt \quad (1.47)$$

$\hat{S}_{XX}(f, T)$  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:

$$\tilde{P}(f) = \frac{1}{NT_s} |X_N(f)|^2 \quad (1.48)$$

with

$$X_N(f) = T_s \sum_{n=0}^{N-1} X(n) \exp(-j2\pi fnT_s) \quad (1.49)$$

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

To obtain the periodogram, the data needs to be windowed. Windowing implies a function that selects a certain portion 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,  $|W(f)|^2 \rightarrow \delta(f)$  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 turns 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 an estimated Power Spectral Density value  $\hat{S}(f)$  is defined by [nsapplication255]

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

where A and B are the bounds of the confidence interval, and  $\alpha$  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

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

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

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

$$\text{CDF}(\chi^2) = \frac{\gamma(\nu/2, \chi^2/2)}{\Gamma(\nu/2)} \quad (1.50)$$

where  $\gamma(\nu/2, \chi^2/2)$  is an incomplete gamma function and  $\Gamma(\nu/2)$  is the gamma function. The values of  $\chi^2$  needed for calculating A and B can be found by numerically evaluating 1.50 to find the values which fit the desired probabilities. This is currently done using the bisection 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 1.59 and 1.59.

### 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].

## 1.60 Pulse Shaper

<b>Header File</b>	: pulse_shaper.h
<b>Source File</b>	: 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
rollOffFactor	real	$\in [0, 1]$	0.9

Table 1.29: 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 through 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 *rollOffFactor* 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

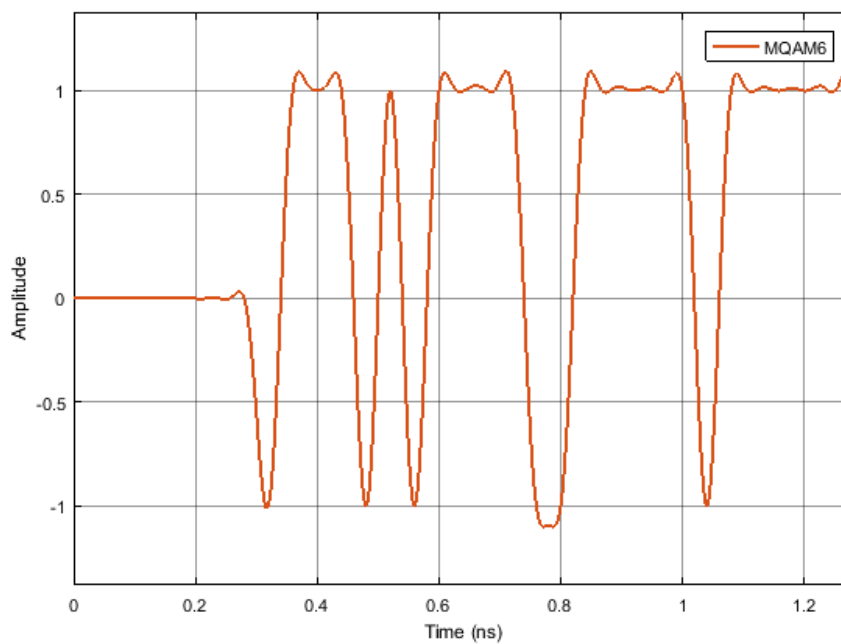


Figure 1.46: Example of a signal generated by this block for the initial binary signal 0100...

## Suggestions for future improvement

Include other types of filters.

## 1.61 Quantizer

<b>Header File</b>	: quantizer_*.h
<b>Source File</b>	: quantizer_*.cpp
<b>Version</b>	: 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	<i>inf</i>
maxValue	volts	double	any	1.5
minValue	volts	double	any	-1.5

Table 1.30: 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) max\_value = maxvalue;

double getMinValue() return max\_value;

void setMaxValue(double maxvalue) max\_value = maxvalue;

double getMaxValue() return max\_value;



**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**

### **Suggestions for future improvement**

## 1.62 Resample

<b>Header File</b>	: resample_*.h
<b>Source File</b>	: resample_*.cpp
<b>Version</b>	: 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 `samplingRate`, which is externally configured.

### Input Parameters

Parameter	Type	Values	Default
rFactor	double	any	<i>inf</i>
samplingPeriod	double	any	0.0
symbolPeriod	double	any	1.5

Table 1.31: 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**

### **Suggestions for future improvement**

### 1.63 SNR Estimator

<b>Header File</b>	: snr_estimator_*.h
<b>Source File</b>	: snr_estimator_*.cpp
<b>Version</b>	: 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 `TimeContinuousAmplitudeContinuousReal` 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 one of several methods to estimate the signal's SNR. Any of them can be useful 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]:

$$P = \frac{1}{N} \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N^2} \sum_{k=0}^{N-1} |X(k)|^2 \quad (1.51)$$

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 spectrum in the area without signal. This is 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.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \quad (1.52)$$

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 if 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 to 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.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (1.53)$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{I_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4P_S P_N + k_n P_N^2 \quad (1.54)$$

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:

$$P_S = \sqrt{2M_2^2 - M_4} \quad (1.55)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (1.56)$$

and therefore

$$\text{SNR} = \frac{\sqrt{2M_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \quad (1.57)$$

### 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].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (1.58)$$

Thus,  $P_S$ ,  $P_N$  and the SNR become:



$$P'_S = \sqrt{M_2^2 - M_4'} \quad (1.59)$$

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (1.60)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (1.61)$$

### 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).

## 1.64 Sampler

<b>Header File</b>	: sampler.h
<b>Source File</b>	: 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 1.32: 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  $2 * 8 * \text{samplesPerSymbol}$ .

## Input Signals

**Number:** 1

**Type:** Electrical real (TimeContinuousAmplitudeContinuousReal)

## Output Signals

**Number:** 1

**Type:** Electrical real (TimeDiscreteAmplitudeContinuousReal)

## Examples

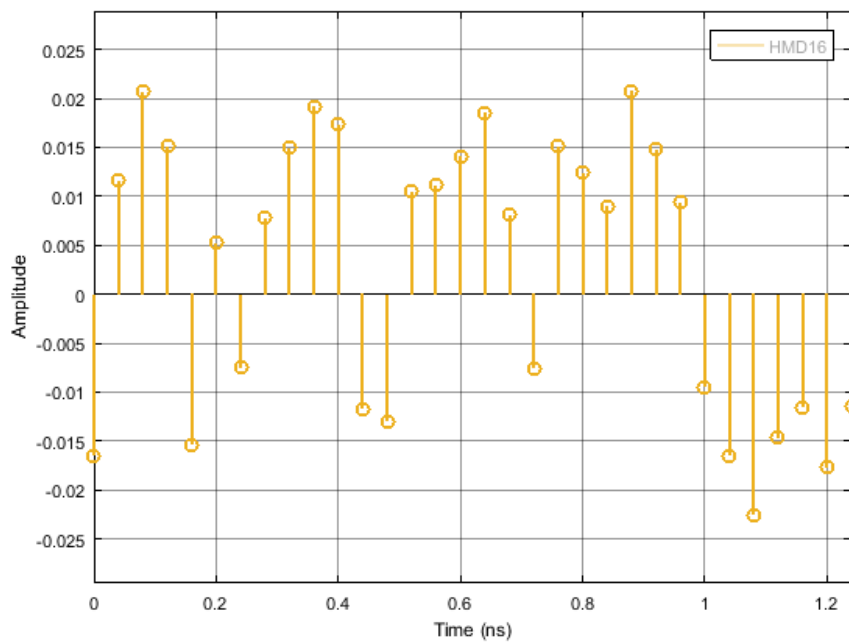


Figure 1.47: Example of the output signal of the sampler

## Suggestions for future improvement

## 1.65 SNR of the Photoelectron Generator

<b>Header File</b>	: <code>srn_photoelectron_generator_*.h</code>
<b>Source File</b>	: <code>srn_photoelectron_generator_*.cpp</code>
<b>Version</b>	: 20180309 (Diamantino Silva)

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

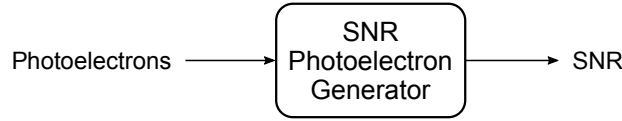


Figure 1.48: Schematic representation of the SNR of the Photoelectron Generator code block

### 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  $\Delta t$ . 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]

$$\text{SNR} = \frac{\bar{n}^2}{\sigma_n^2} \quad (1.62)$$

in which  $\bar{n}$  is the mean value and  $\sigma_n^2$  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,  $\mu_k$ , and variance,  $s_k^2$  [smith1997]

$$\mu_k = \langle y \rangle_k \quad s_k^2 = \frac{N}{N-1} \left( \langle y^2 \rangle_k - \langle y \rangle_k^2 \right) \quad (1.63)$$

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

$$\langle y^n \rangle_k = \frac{1}{N_k} \sum_{j=j_{\min}(k)}^{j_{\max}(k)} y_j^n \quad (1.64)$$

in which

$$j_{min}(k) = \lceil t_k / \Delta t \rceil \quad (1.65)$$

$$j_{max}(k) = \lceil t_{k+1} / \Delta t \rceil - 1 \quad (1.66)$$

$$N_k = j_{max}(k) - j_{min}(k) + 1 \quad (1.67)$$

$$t_k = kT \quad (1.68)$$

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

$$S_1(k) = N_k \langle y \rangle_k \quad (1.69)$$

$$S_2(k) = N_k \langle y^2 \rangle_k \quad (1.70)$$

Using these two variables, we can rewrite  $\mu_k$  and  $s_k^2$  as

$$\mu_k = \frac{S_1(k)}{N_k} \quad s_k^2 = \frac{1}{N_k - 1} \left( S_2(k) - \frac{1}{N_k} (S_1(k))^2 \right) \quad (1.71)$$

The signal to noise ratio of the time interval  $k$ ,  $\text{SNR}_k$ , can be expressed as

$$\text{SNR}_k = \frac{\mu_k^2}{\sigma_k^2} = \frac{N_k - 1}{N_k} \frac{(S_1(k))^2}{N_k S_2(k) - (S_1(k))^2} \quad (1.72)$$

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]

$$\text{SNR} = \eta \bar{n} \quad (1.73)$$

in which  $\eta$  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  $\text{tWindow} = T$ . 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 1.66, 1.67 and 1.68. 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  $S_1(k)$  and  $S_2(k)$  from the

state variables as  $S_1(k) = \text{aux\_sum1}$  and  $S_2(k) = \text{aux\_sum2}$ , and proceed to the calculation of the  $\text{SNR}_k$ , using equation 1.72.

If the simulation ends before reaching the end of the current time window, we calculate the  $\text{SNR}_k$ , 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
<code>windowTime</code>	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

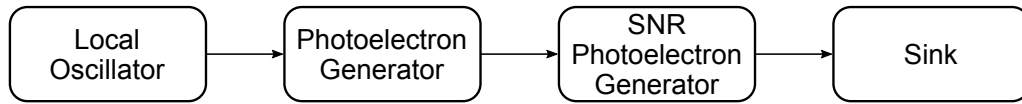
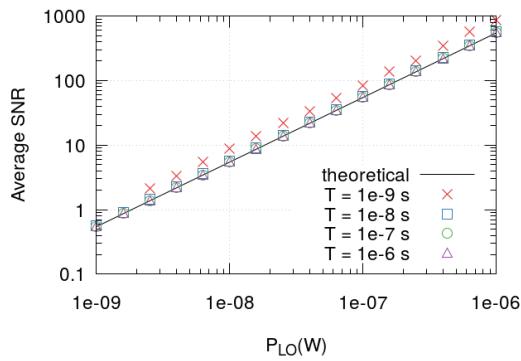


Figure 1.49: Simulation setup

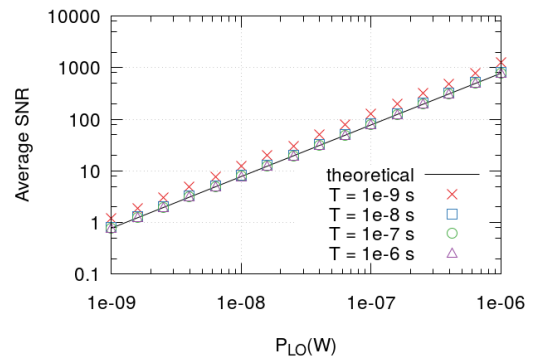
To confirm the block's correct output, we have designed a simulation setup which calculates the SNR of a stream of photoelectrons generated by the detection of a laser photon stream by a photodiode.

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,  $\eta$ . 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 1.50: Theoretical and simulated results of the average SNR, for two photodiode efficiencies.

The plots in 1.50 show the comparison between the theoretical result 1.73 and the simulation results. We see that for low values of  $T$ , the average SNR shows a systematic deviation from the theoretical result, 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 1.50a). 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

$$p_{ave} = (1 - \exp(-N\bar{m}))^M \quad (1.74)$$

## 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 distribution 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.



## 1.66 Sink

<b>Header File</b>	: sink_*.h
<b>Source File</b>	: sink_*.cpp
<b>Version</b>	: 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 1.33: 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.

## 1.67 SNR Estimator

<b>Header File</b>	: snr_estimator_*.h
<b>Source File</b>	: snr_estimator_*.cpp
<b>Version</b>	: 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 `TimeContinuousAmplitudeContinuousReal` 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 one of several methods to estimate the signal's SNR. Any of them can be useful 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]:

$$P = \frac{1}{N} \sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N^2} \sum_{k=0}^{N-1} |X(k)|^2 \quad (1.75)$$

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 spectrum in the area without signal. This is 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.

$$\text{SNR} = \frac{P_s}{P_n} = \frac{P_{s+n} - P_n}{P_n} = \frac{P_{s+n-n_0} R_{sym}}{n_0 R_{sam}} \quad (1.76)$$

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 if 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 to 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.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (1.77)$$

$$M_4 = E\{|S_k|^4\} = E\{(S_{I_k}^2 + S_{Q_k}^2)^2\} = k_e P_S^2 + 4P_S P_N + k_n P_N^2 \quad (1.78)$$

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:

$$P_S = \sqrt{2M_2^2 - M_4} \quad (1.79)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (1.80)$$

and therefore

$$\text{SNR} = \frac{\sqrt{2M_2^2 - M_4}}{M_2 - \sqrt{2M_2^2 - M_4}} \quad (1.81)$$

### 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].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (1.82)$$

Thus,  $P_S$ ,  $P_N$  and the SNR become:

$$P'_S = \sqrt{M_2^2 - M_4'} \quad (1.83)$$

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (1.84)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (1.85)$$

### 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).

## 1.68 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 1.51.

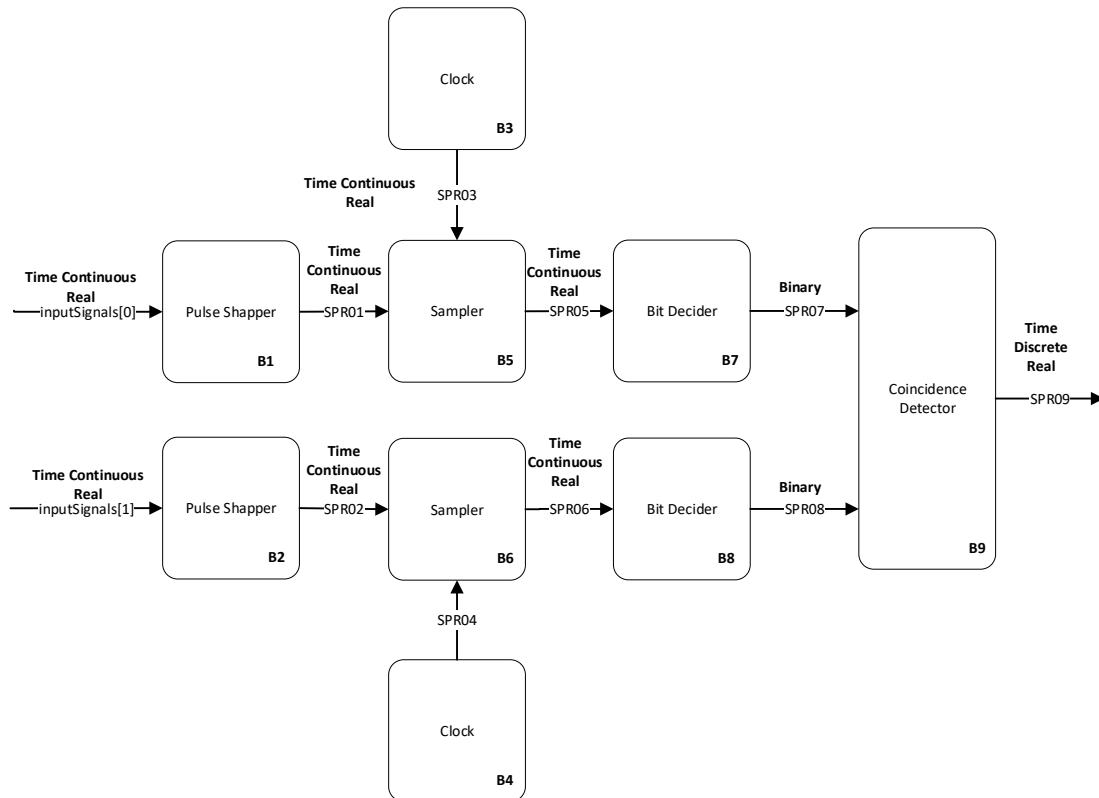


Figure 1.51: Basic configuration of the SPR receiver

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

It can also be seen from figure 1.51 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 1.35) 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 1.34: 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**

**Suggestions for future improvement**

## 1.69 SOP Modulator

<b>Header File</b>	: sop_modulator_*.h
<b>Source File</b>	: sop_modulator_*.cpp
<b>Version</b>	: 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 to 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  $\theta$  and  $\phi$  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.

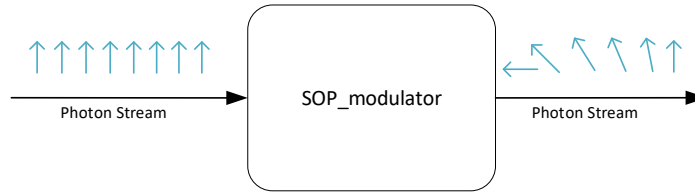


Figure 1.52: Diagram block of SOP block input and outputs.

The block in figure 1.52 was implemented based on a continuous matrix calculation. Let's assume that the input photon stream can be represented by the matrix:

$$S_{in} = \begin{bmatrix} A_x \\ A_y \end{bmatrix}, \quad (1.86)$$

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:

$$S_{out} = S_{in} \mathbf{J}_k, \quad (1.87)$$

where,

$$\mathbf{J}_k = J(\dot{\mathbf{a}}) \mathbf{J}_{k-1}. \quad (1.88)$$

$J(\boldsymbol{\alpha})$  is a random matrix which can be expressed using the matrix exponential parameterized by  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)$ :

$$\begin{aligned} J(\boldsymbol{\alpha}) &= e^{-i\boldsymbol{\alpha} \cdot \vec{\sigma}} \\ &= \mathbf{I}_2 \cos(\theta) - i\mathbf{a} \cdot \vec{\sigma} \sin(\theta), \end{aligned} \quad (1.89)$$

where  $\vec{\sigma} = (\sigma_1, \sigma_2, \sigma_3)$  is the tensor of Pauli Matrices:

$$\sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}; \sigma_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \sigma_3 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \quad (1.90)$$

$\mathbf{I}_2$  is the  $2 \times 2$  identity matrix. In addition,  $\boldsymbol{\alpha} = \theta \mathbf{a}$ , having the length  $\theta = \|\boldsymbol{\alpha}\|$  and the direction on the unit sphere  $\mathbf{a} = (a_1, a_2, a_3)$ , where  $\|\cdot\|$  denotes the Euclidian norm. Furthermore, the randomness is achieved by  $\boldsymbol{\alpha}$  parameters:

$$\dot{\boldsymbol{\alpha}} \sim \mathcal{N}(0, \sigma_p^2 \mathbf{I}_3), \quad (1.91)$$

where  $\sigma_p^2 = 2\pi\Delta pT$ , being  $T$  the symbol period and  $\Delta p$  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 1.35) the input parameters and corresponding functions are summarized.

Input parameters	Function	Accepted values
sopType	Simulation type that the user intends to simulate.	Deterministic OR Stochastic
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 1.35: 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**

**Suggestions for future improvement**

## 1.70 Source Code Efficiency

<b>Header File</b>	:	source_code_efficiency_*.h
<b>Source File</b>	:	source_code_efficiency_*.cpp
<b>Version</b>	:	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 sourceOrder.

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

Table 1.36: 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)



## 1.71 White Noise

<b>Header File</b>	: white_noise_20180420.h
<b>Source File</b>	: 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	$> 0$	$1.5 \times 10^{-17}$
seed	int	$\in [1, 2^{32} - 1]$	1
samplingPeriod	double	$> 0$	1.0

Table 1.37: 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  $T$ .

$$N = n_0 B = n_0 \frac{2}{T} \quad (1.92)$$

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**

## 1.72 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	$1 \times 10^4$

Table 1.38: 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**

### **Suggestions for future improvement**

## 1.73 Phase Mismatch Compensation

<b>Header File</b>	: phase_mismatch_compensation_*.h
<b>Source File</b>	: phase_mismatch_compensation_*.cpp
<b>Version</b>	: 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 1.53. An unmodulated

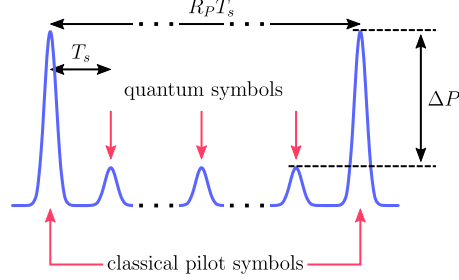


Figure 1.53: Representative time dependence of a pilot assisted phase mismatch compensation signal.

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

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\epsilon(t_n)]}, & n = mR_P \end{cases}, \quad n, m \in \mathbb{N}. \quad (1.93)$$

The phase mismatch for the  $n$ th pulse is estimated from an average of the phase of the previous and the later pilots, resulting in

$$\hat{\epsilon}(t_n) = \frac{\epsilon(t_{(m+1)R_P}) + \epsilon(t_{mR_P})}{2}, \quad mR_P < n < (m+1)R_P, \quad (1.94)$$

with the mismatch compensation being accomplished by multiplying the constellation by  $e^{-i(\hat{\epsilon}(t_n))}$ , resulting in

$$x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)-\hat{\epsilon}(t_n)]}. \quad (1.95)$$

## 1.74 Frequency Mismatch Compensation

<b>Header File</b>	: frequency_mismatch_compensation_*.h
<b>Source File</b>	: frequency_mismatch_compensation_*.cpp
<b>Version</b>	: 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

$$y(t) = |y(t)|e^{i[\Delta\omega t + \theta(t) + \epsilon(t)]}, \quad (1.96)$$

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

$$y(t)' = y(t) * e^{-i\Delta\hat{\omega}t}. \quad (1.97)$$

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 1.54. At the receiver stage, after coherent detection,

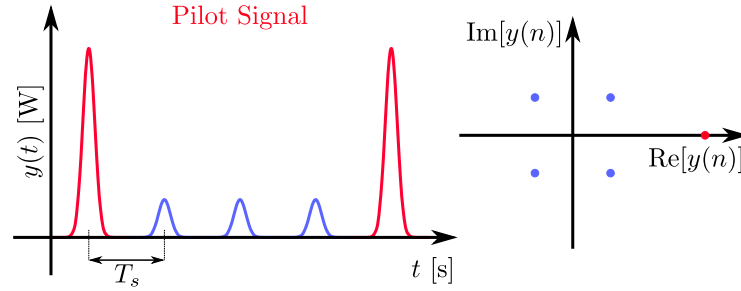


Figure 1.54: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted scheme. Pilot pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

the signal is described by

$$y(t_n) = \begin{cases} x_s(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n + \epsilon(t_n)]}, & n = mR_P \end{cases}, \quad (1.98)$$

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

$$\begin{aligned} x_f(m) &= y_r^*(t_{mR_P})y_r(t_{(m+1)R_P}) \\ &= x_p(t_{mR_P})x_p(t_{(m+1)R_P})e^{i(\Delta\omega R_P T_s + \Delta\epsilon(m))}, \end{aligned} \quad (1.99)$$

where

$$\Delta\epsilon(m) = \epsilon(t_{mR_P}) - \epsilon(t_{(m+1)R_P}). \quad (1.100)$$

The frequency estimation is accomplished by taking the expected value of the complex argument of (1.99) and dividing it by  $R_P T_s$ . This returns

$$\begin{aligned} \Delta\hat{\omega} &= \text{E} \left[ \frac{1}{R_P T_s} \arg(x_f(m)) \right] \\ &= \text{E} \left[ \Delta\omega + 2k\pi(R_P T_s)^{-1} + \frac{\Delta\epsilon(m)}{R_P T_s} \right] \\ &= \Delta\omega + 2k\pi(R_P T_s)^{-1}. \end{aligned} \quad (1.101)$$

### 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  $m$ th power of the input signal [selmi09], which exhibits a peak at the frequency  $m\Delta\Omega$ , this value is used as the estimate.

## 1.75 Cloner

<b>Header File</b>	: cloner_*.h
<b>Source File</b>	: cloner_*.cpp
<b>Version</b>	: 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

## 1.76 Error Vector Magnitude

<b>Header File</b>	: error_vector_magnitude_*.h
<b>Source File</b>	: error_vector_magnitude_*.cpp
<b>Version</b>	: 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  $\widehat{EVM}$  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  $ref_v$ . This process is presented visually in Figure 1.55 and is described by

$$\widehat{EVM} = 100 \sqrt{\frac{|\vec{e}_v|}{|ref_v|}} = 100 \sqrt{\frac{|\vec{m}_v - ref_v|}{|ref_v|}}. \quad (1.102)$$

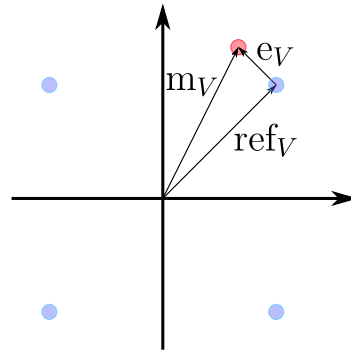


Figure 1.55: Visual representation of the EVM method in a QPSK constellation. The ideal constellation points are presented in blue, while an example for the actual measured symbol is presented in red.

## 1.77 Load Ascii

<b>Header File</b>	: load_ascii.h
<b>Source File</b>	: load_ascii.cpp
<b>Version</b>	: 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 1.39: 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)

## 1.78 Load Signal

<b>Header File</b>	: load_signal.h
<b>Source File</b>	: load_signal.cpp
<b>Version</b>	: 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 1.40: 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



## 1.79 Matched Filter

<b>Header File</b>	: matched_filter.h
<b>Source File</b>	: matched_filter.cpp
<b>Version</b>	: 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 1.41: 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

## 1.80 Timing Deskew

<b>Header File</b>	: timing_deskew.h
<b>Source File</b>	: timing_deskew.cpp
<b>Version</b>	: 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 1.42: 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 1.56, in which the input signal  $S_{in}(n)$  is separated into its in-

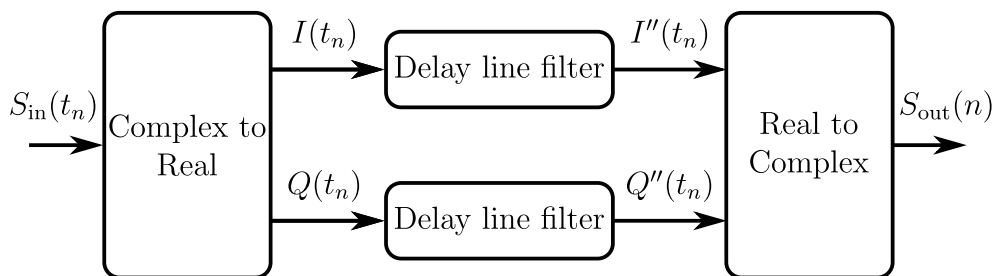


Figure 1.56: Block diagram representation of the deskew procedure.

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 1.57. The signal

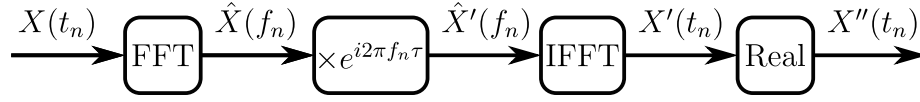


Figure 1.57: Block diagram representation of the implemented delay line filter.

at the input,  $S_{\text{in}}(t_n)$ , is transformed to Fourier space, where it is multiplied by a deskew element  $\tau$

$$\hat{S}'(f_n) = \hat{S}(f_n)e^{i2\pi f_n \tau}. \quad (1.103)$$

The value of  $\tau$  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

$$S_{\text{out}}(t_n) = \text{real}(I''(t_n)) + i\text{real}(Q''(t_n)). \quad (1.104)$$

## Input Signals

**Number:** 1

**Type:** Complex signal

## Output Signals

**Number:** 1

**Type:** Complex signal

## 1.81 DC component removal

<b>Header File</b>	: dc_component_removal.h
<b>Source File</b>	: dc_component_removal.cpp
<b>Version</b>	: 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  $S_{in}(t_n)$ . 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

$$S_{out}(t_n) = S_{in}(t_n) - \text{mean}(S_{in}(t_n)) \quad (1.105)$$

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 1.58, 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

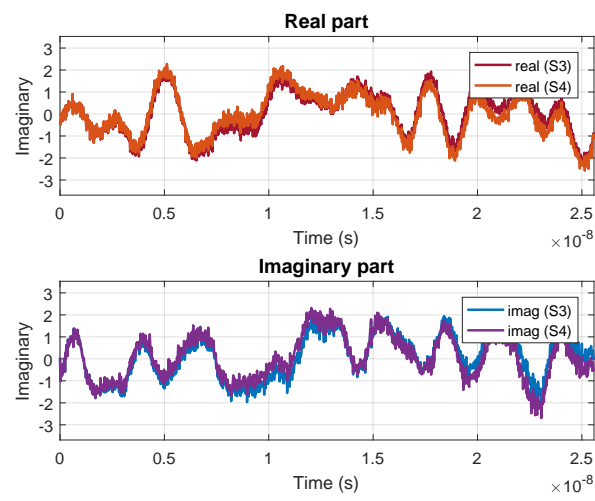


Figure 1.58: Exaggerated DC component removal.

## 1.82 Orthonormalization

<b>Header File</b>	: orthonormalization.h
<b>Source File</b>	: orthonormalization.cpp
<b>Version</b>	: 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 implementation follows the topology presented in Figure 1.59. The input signal  $S_{\text{in}}(t_n)$  is

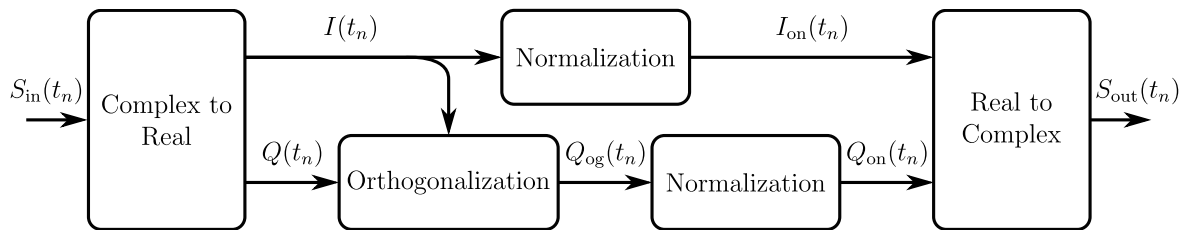


Figure 1.59: Block diagram representation of the Gram-Schmidt orthonormalization procedure.

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.:

$$P_I = \text{mean}(I^2(t_n)), \quad (1.106)$$

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

$$I_{\text{on}}(t_n) = \frac{I(t_n)}{\sqrt{P_I}}. \quad (1.107)$$

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

$$Q_{\text{og}}(t_n) = Q(t_n) - \frac{I(t_n)\text{mean}(I(t_n)Q(t_n))}{P_I}, \quad (1.108)$$

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

$$P_Q = \text{mean}(Q_{\text{og}}^2(t_n)), \quad (1.109)$$

$$Q_{\text{on}}(t_n) = \frac{Q_{\text{og}}(t_n)}{\sqrt{P_Q}}. \quad (1.110)$$

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

$$S_{\text{out}}(t_n) = I_{\text{on}}(t_n) + iQ_{\text{on}}(t_n) \quad (1.111)$$

### Input Signals

**Number:** 1 **Type:** Complex signal

### Output Signals

**Number:** 1

**Type:** Complex signal

### Suggestions for future improvement

- Add the optimized recursive Gram-Schmidt algorithm as an option.

## 1.83 Matched Filter

<b>Header File</b>	: matched_filter.h
<b>Source File</b>	: matched_filter.cpp
<b>Version</b>	: 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 1.43: 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



## 1.84 Upsampler

<b>Header File</b>	: upsampler.h
<b>Source File</b>	: upsampler.cpp
<b>Version</b>	: 20190319 (Daniel Pereira)

This block applies an interpolation filter.

### Input Parameters

Parameter	Type	Values	Default
numberOfTaps	int	any	8
upSamplingFactor	int	any	2

Table 1.44: 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 1.60.

### Input Signals

**Number:** 1

**Type:** Complex signal

### Output Signals

**Number:** 1

**Type:** Complex signal

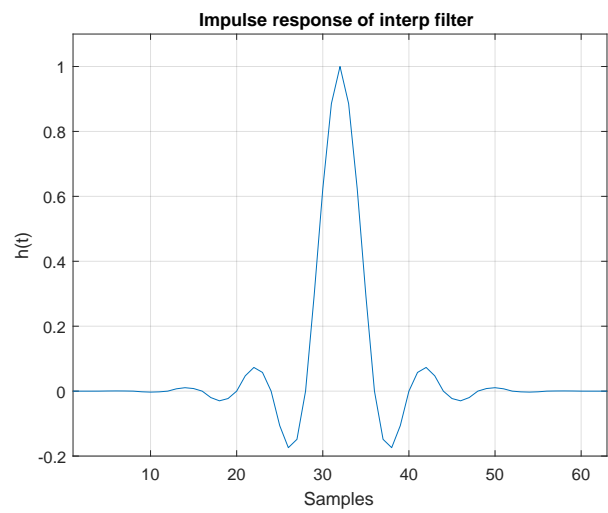


Figure 1.60: Impulse response of a  $4\times$  upsampling filter using 16 taps.

## 1.85 Matched Filter

<b>Header File</b>	: resampler.h
<b>Source File</b>	: resampler.cpp
<b>Version</b>	: 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 1.45: 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.

